SLAQ: Quality-Driven Scheduling for Distributed Machine Learning

Haoyu Zhang*, Logan Stafman*, Andrew Or, Michael J. Freedman
Princeton University

Abstract

Training machine learning (ML) models with large datasets can incur significant resource contention on shared clusters. This training typically involves many iterations that continually improve the quality of the model. Yet in exploratory settings, better models can be obtained faster by directing resources to jobs with the most potential for improvement. We describe SLAQ, a cluster scheduling system for approximate ML training jobs that aims to maximize the overall job quality.

When allocating cluster resources, SLAQ explores the quality-runtime trade-offs across multiple jobs to maximize system-wide quality improvement. To do so, SLAQ leverages the iterative nature of ML training algorithms, by collecting quality and resource usage information from concurrent jobs, and then generating highly-tailored quality-improvement predictions for future iterations. Experiments show that SLAQ achieves an average quality improvement of up to 73% and an average delay reduction of up to 44% on a large set of ML training jobs, compared to resource fairness schedulers.

Categories and Subject Descriptors

[*Computer systems organization*]: Distributed architectures; [*Computing methodologies*]: Distributed artificial intelligence; [*Theory of computation*]: Approximation algorithms analysis

General Terms

Design, Experimentation, Performance

Keywords

Scheduling, Machine Learning, Approximate Computing, Resource Management, Quality

*indicates equal contribution by authors.

SoCC '17, September 24–27, 2017, Santa Clara, CA, USA

ACM ISBN 978-1-4503-5028-0/17/09.
https://doi.org/10.1145/3127479.3127490

1 Introduction

Machine learning (ML) is an increasingly important tool for large-scale data analytics, including online search, marketing, healthcare, and information security. A key challenge in analyzing massive amounts of data with ML arises from the fact that model complexity and data volume is growing much faster than hardware speed improvements. Thus, time-sensitive machine learning on large datasets necessitates the use and efficient management of cluster resources. Three key features of ML are particularly relevant to resource management.

ML algorithms are intrinsically approximate. ML algorithms generally consist of two stages: *training* and *inference*. The training stage builds a model from a training dataset (e.g., images with labeled objects), and the inference stage uses the model to make predictions on new inputs (e.g., recognizing objects in a photo). ML models are intrinsically approximate functions for input-output mapping. We use *quality* to measure how well the model maps input to the correct output.

ML training is typically iterative with diminishing returns. While the inference stage is often lightweight and can run in real-time, the training stage is computationally expensive and usually requires multiple passes over large datasets. It generates a low-quality model at the beginning and improves the model's quality through a sequence of training iterations until it converges. In general, the quality improvement diminishes as more iterations are completed.

Training ML is an exploratory process. ML practitioners retrain their models repeatedly to explore feature validity [1], tune hyperparameters [2, 3], and adjust model structures [4] before they operationalize their final model, which is deployed for performing inference on individual inputs. The goal of retraining is to get the final model with the best quality. Since ML training jobs are expensive, practitioners in experimental environments often prefer to work with more approximate models trained within a short period of time for preliminary validation and testing, rather than wait a significant amount of time for a better trained model with poorly tuned configurations. In fact, algorithm tuning is an empirical process of trial and error that can take significant

effort, both human and machine. With the exponential growth of data volume, the cost of decision making on model configurations will likely continue to increase.

Many ML frameworks have been developed [5, 6, 7, 8] to run large-scale training jobs in clusters with shared resources. Existing schedulers primarily focus on *resource fairness* [9, 10, 11, 12, 13, 14], but are agnostic to model quality and job runtime. During a burst of job submissions, equal resources will be allocated to jobs that are in their early stages and could benefit significantly from extra resources as those that have nearly converged and cannot improve much further. This is not efficient.

We present SLAQ, a cluster scheduling system for ML training jobs that aims to maximize the overall job quality. SLAQ dynamically allocates resources based on job resource demands, intermediate model quality, and the system's workload. The intuition behind SLAQ is that in the context of approximate ML training, more resources should be allocated to jobs that have the most potential for quality improvement.

SLAQ leverages the fact that most ML training algorithms are implemented as an iterative optimization process. By continually monitoring the history of quality improvement and runtime, SLAQ generates highly-tailored and accurate quality predictions for future training iterations. SLAQ estimates the impact of resource allocation on model quality, and explores the quality-runtime trade-offs across multiple jobs. Based on this information, SLAQ adjusts their resource allocations of all running jobs to best utilize the limited cluster resources. The SLAQ scheduler is designed to be dynamic and fine-grained, so that resource allocations can adapt quickly to jobs' quality and the system's workload changes.

Challenges and solutions. In designing SLAQ, we had to overcome several technical challenges.

First, ML training algorithms measure the quality of models with tens of different metrics, which makes it difficult to compare the training progress of different jobs. SLAQ normalizes these metrics using the reduction of loss values. These intermediate quality measures are reported directly by the application APIs. Our normalization effectively unifies the quality measures for a broad set of ML algorithms.

Second, SLAQ should be able to precisely predict the impact that an extra unit of resources would have on the quality and runtime of ML training jobs. Previous work [15] predicts a job's runtime based on its computation and communication structure, but it requires that the job be analyzed or profiled offline. Unfortunately, the significant overhead of this offline analysis is prohibitive for our exploratory setting. SLAQ uses online prediction: it predicts the time and quality of the coming iterations based on statistics collected from previous iterations.

SLAQ supports configurable high-level goals when scheduling jobs. When maximizing the aggregate quality improvement, it can best utilize the cluster resources and achieve a higher total quality gain across all jobs. When maximizing the minimum quality, SLAQ can achieve the equivalent of max-min fairness applied to quality (rather than resource allocation).

While we designed our scheduler for ML training applications, SLAQ can schedule many applications with approximate intermediate results. Some approximate jobs produce partial results at intermediate points of the application's run [16], while others generate approximate results from samples to avoid scanning entire datasets [17]. Improvement in the quality of these systems' results also diminishes with more processing time [18]. To that end, SLAQ's techniques are broadly applicable to other data analytics systems that employ iterative approximation approaches.

On the other hand, while SLAQ works with a large class of important ML algorithms, some non-convex ML algorithms are not currently supported. The convergence properties and optimization of these algorithms are being actively studied, and we leave scheduling support for these algorithms to future work.

We implemented SLAQ as a new scheduler within the Apache Spark framework [19]. SLAQ can use its quality-driven scheduling for many of the ML algorithms available in MLlib [5], Spark's machine learning package. In fact, SLAQ supports unmodified ML applications using existing MLlib optimizers, as well as applications using new optimization algorithms with only minor modifications. We evaluate various distinct ML training algorithms on datasets collected from various online sources. We found that SLAQ improves the average quality by up to 73% and reduces the average delay by up to 44% compared to fair resource scheduling.

2 Background and Motivation

The past several years has seen a rapid increase in both the volume of data used to train ML models and the size and complexity of these models. Growth in the performance of the underlying hardware, however, has not caught up, thus placing higher demands on the computational resources used for this purpose.

An important way that data scientists cope with these demands is to leverage more approximate models for preliminary testing, in order to exclude bad trials and iterate to the right configuration. A significant amount of time and resource usage can potentially be saved because of the iterative nature of ML optimization algorithms, and the diminishing returns of quality improvements during the training iterations. Today's schedulers, however, do not provide a ready means to follow this strategy; a traditional max-min fair scheduler (similarly, the domi-

nant resource fair scheduler [11]) ensures fair *resource* allocation without considering the potential of these resources to improve model quality.

This section motivates and provides background for SLAQ. §2.1 describes the iterative nature of the ML training process and how it is characterized by diminished returns. We introduce the exploratory training process in §2.2 and describe current practices in §2.3. We discuss the problems with existing cluster schedulers and propose our quality-aware scheduler in §2.4.

2.1 ML Training: Iterative Optimization Process

The algorithms used for the ML training process typically include a dataset specification, a loss function, an optimization procedure, and a model [20]. A machine learning model is a parametric transformation $f_\theta : X \mapsto Y$ that maps input variables to output variables, and it typically contains a set of parameters θ which will be regularly adjusted during the training process. The loss function represents how well the model maps training examples to correct output, and is often combined with a regularization term to incorporate generalizability. Training machine learning models can be summarized as optimizing the model parameters to minimize the loss function when applying the model on a dataset.

When the machine learning model is nonlinear, most loss functions can no longer be optimized in closed form. Algorithms such as Gradient Descent, L-BFGS and Expectation Maximization (EM) are widely used in practice to *iteratively* solve the numerical optimization of the loss function. As the sizes of the dataset and model grow, the batch algorithms can no longer solve the optimization problem efficiently. Instead, various new algorithms have been proposed to improve the efficiency of the optimization process in an iterative and distributed fashion. For example, stochastic gradient descent (SGD) [21] reduces computationally complexity by evaluating the loss function and gradient on a randomly drawn subset of the overall dataset in each iteration.

The training process with the iterative optimization algorithms can be viewed as a refinement loop of the model. After initializing the parameter values (e.g., with random values), the optimization algorithms calculate changes on parameters in order to reduce the loss function, and update the model with new parameter values. This process continues until the decrease in the loss function falls below a certain threshold, or until a preset number of iterations have elapsed.

Another approach that some ML algorithms take is ensemble learning. Instead of training a complicated model with a large number of parameters, these algorithms focus on aggregating results from multiple diverse but small *submodels*. For example, boosting algorithms

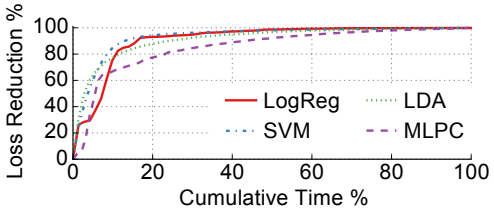


Figure 1: Cumulative time to achieve different percentages of loss reduction with four jobs: Logistic Regression (LogReg), Support Vector Machine (SVM), Latent Dirichlet Allocation (LDA) and Multi-Layer Perceptron Classifier (MLPC). Job convergence is defined to be $1/10000$ of initial loss reduction.

improve the accuracy of the model classifier by examining the errors in results, adding new submodels to the ensemble, and adjusting the weights of the set of submodels. Boost aggregating (bagging) algorithms train multiple submodels on different subsets of the training data by sampling with replacement. The training process of the ensemble models involves both iteratively refining each submodel, and iteratively adding new submodels or adjusting the weights of existing components.

When training a machine learning model, the first several iterations generally boost the quality very quickly. This is because the initial parameters of a model are generally set randomly. However, for most ML training algorithms, the quality improvements are subject to diminishing returns; iterations in later stages continue to cost the same amount of computational resources while making only marginal improvements on model quality as the results finally converge. For example, error in gradient descent algorithms on convex optimization problems often converges approximately as a geometric series [22]. Theoretically, at the kth iteration, the loss function reduction is $O(\mu^k)$, where μ is the convergence rate ($|\mu| < 1$). In general, loss reduction (quality improvement) diminishes as more iterations are completed.

Figure 1 plots the relative cumulative time to achieve different percentages of loss reduction. For example, it takes 20% time for the SVM job to reduce loss by 95%, and 80% time to further reduce it until convergence. Jobs for ML algorithm debugging and model tuning only require the training process to be almost completed to tell potentially good configurations from bad trials, and thus could save a lot of time and resources.

The law of diminishing returns applies in many other data analytics systems in addition to machine learning. Sampling-based approximate query processing systems compute approximate results by processing only a sample of the entire dataset in order to reduce resource usage and processing delay [17, 23, 24, 25]. Databases can also take advantage of online aggregation to incremen-

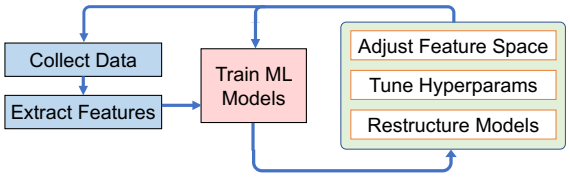


Figure 2: Retrain machine learning models.

tally refine the approximated results of SQL aggregate queries [16, 26, 27]. Using the *error* or *uncertainty* as a measurement of quality in these queries, we can observe that in most cases the convergence rate of these metrics are also monotonically decreasing.

2.2 Retraining Machine Learning Models

Training machine learning models is not a one-time effort. ML practitioners often train a model on the same dataset multiple times for exploratory purposes. This process provides early feedback to practitioners and helps direct their search for high quality models.

Feature engineering. Many ML algorithms require a featurized representation of the input data for efficient training and inference. For example, a speech recognition algorithm utilizes the discretized frequency features extracted from continuous sound signals with Fourier transforms and knowledge about the human ear [28]. Identifying exactly the useful features that yield the best quality relies on both domain knowledge and many training experiments.

Hyperparameter tuning. Many ML models expose *hyperparameters* that describe the high-level complexity or capacity of the models. Optimal values of these hyperparameters typically cannot be learned from the training data. Examples of hyperparameters include the number of hidden layers in a neural network, the number of clusters in a clustering algorithm, and the learning rate of mode parameters. It is desirable to explore different combinations of hyperparameter values, train multiple models, and use the one that gives the best result.

Model structure optimization. To ship ML models and run inference tasks on mobile and IoT devices, large models need to be compressed to reduce the energy consumption and accelerate the computation. Various model compression techniques have been developed [29, 4]. These methods usually prune the unnecessary parameters of the model, retrain the model with the modified structure, and then prune again. This requires training the same job multiple times to get the best compression without compromising the quality of the model.

In addition, the interactions between features, hyperparameters and model structures make it even harder to search for the best model configuration. For example, features are often correlated with one another, and modifying the set of features also requires recalibrating the hyperparameters (such as learning rate). Expensive model configuration decisions demand highly efficient resource management in shared clusters.

2.3 Current Practices in ML Training

When exploring the ML model configuration space, users often submit training jobs with either a time cutoff or a loss value cutoff. Both monitoring heuristics are widely used in practice but have significant drawbacks.

Training ML models within a fixed time frame often results in unpredictable quality. This is because it is often difficult to predict a priori what the loss values will be at the deadline. More importantly, when a training job shares cluster resources with other jobs, the number of iterations completed by the deadline also depends on the cluster's workload and the scheduler's decisions.

A fixed loss (or fixed Δloss) cutoff is also difficult to reason about. Loss values in different algorithms are different in magnitude and have completely different meanings (further explained in §4.1). Additionally, with more complicated model structures and training algorithms, it is not rare to see the convergence rate of loss function fluctuate due to stochastic methods and model staleness [30]. Fixed loss values also make users lose the potential to gain further improvement on the training.

Some users choose to manually monitor the loss function values during the training process and stop the job when they think the models are good enough. However, large-scale ML jobs could take hours or even days to complete, which makes the monitoring impractical.

In the context of exploratory ML training, it is desirable to explore the quality-runtime trade-off across multiple concurrent jobs. SLAQ automates this process and obviates the need for the user to reason about arbitrary trade-offs. SLAQ flexibly fulfills a broad range of requirements for quality and delay of ML trainings, from approximate but timely models, to more traditional accurate model training. It allows users to stop jobs early before perfect convergence, and obtain a model with a loss function converged enough with much shorter latency.

2.4 Cluster Scheduling Systems

A cluster scheduler is responsible for managing resource allocation across multiple jobs. Modern data analytics frameworks (such as Hadoop [31], Spark [19], etc.) typically have two layers of scheduling: the job-level scheduler allocates resources to concurrent jobs running on the workers, while the task-level scheduler focuses on assigning tasks within a job to the available workers.

Existing job-level schedulers (Yarn [9], Mesos [10], Apollo [32], Hadoop Capacity [13], Quincy [14], etc.) mostly allocate resources based on resource fairness or

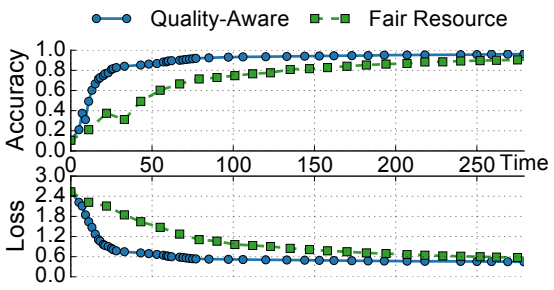


Figure 3: Accuracy (top) and loss function values (bottom) of a job with resources allocated by a quality-aware scheduler and a fair scheduler. Accuracy (percentage of correctly predicted data points) is evaluated on a testing dataset at the end of each training iteration. The more resources allocated to a job, the faster an iteration can be finished.

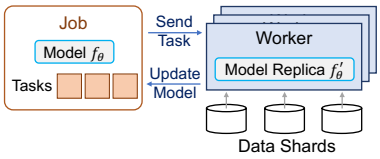
priorities. For ML training jobs, however, these schedulers often make suboptimal scheduling decisions because they are agnostic to the progress (quality improvement) within each job. We argue that the scheduler should collect quality and delay information from each job and dynamically adjust the resource allocation to optimize for cluster-wide quality improvement.

SLAQ is a *fine-grained* job-level scheduler: it focuses on the allocation of cluster resources between competing ML *jobs*, but does so over *short time intervals* (i.e., hundreds of milliseconds to a few seconds). Scheduling on short intervals ensures the continued rebalancing of resources across jobs, whose iteration time varies from tens to hundreds of milliseconds.

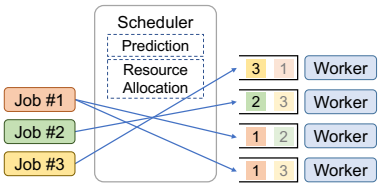
In a shared cluster with multiple users constantly submitting their training jobs, Figure 3 shows how the accuracy and loss values of one job change over time. With the fair scheduler, the job receives its fair share of cluster resources throughout its lifetime. A key observation here is that if we had given this job more resources in its early stages, its accuracy (loss) could have increased (decreased) much faster. SLAQ does exactly this, allocating more resources to the job when its potential improvement is large. In particular, the job was able to achieve 90% accuracy within a much shorter time frame (70s) with SLAQ than with the fair scheduler (230s). Especially for exploratory training jobs, this level of accuracy is frequently sufficient.

3 System Overview

SLAQ is a cluster management framework that hosts multi-tenant approximate ML training jobs running on shared resources. A centralized SLAQ scheduler coordinates the resource allocation of multiple ML training



(a) Distributed ML Training



(b) Scheduler Architecture

Figure 4: Running ML training jobs with SLAQ.

jobs. As shown in Figure 4(a), each job is composed of a set of tasks. Each task processes data based on the ML algorithm on a small partition of the dataset, and can be scheduled to run on any node. The driver program contains the iterative training logic, generates tasks for each iteration, and tracks the overall progress of the job. In the case of training ML models, a task generates an update to the model parameters based on a partition of the training dataset. The duration of a task typically ranges from tens of milliseconds to a few seconds. When the tasks finish processing the data, the updates from all tasks are aggregated and sent back to the job driver program to update the primary copy of the model.

Similar to many cluster management systems, SLAQ divides machines into smaller *workers*, which is the minimum unit of resource to run a task. Figure 4(b) shows that each job driver, at a certain time, can send tasks to the workers allocated to that job in the cluster.

The SLAQ scheduler directly communicates with the drivers of currently running jobs to track their progress and update their resource allocation periodically. At the beginning of each scheduling epoch, SLAQ allocates resources between all the jobs based on system workload, the demands, and progress of the jobs. The scheduler reclaims workers back from some job drivers, and reallocates them to other jobs for better system-wide performance goals. Note that this is very different from many of the existing cluster managers [9, 13] which only statically allocate resources to jobs before they get started.

We made this decision because of two reasons. First, unlike general batch processing, jobs that train ML models are typically iterative and usually need longer time to complete. Scheduling only at the start of the job is too coarse-grained and can easily lead to starvation or underutilization of system resources. Second, the quality improvement of the training jobs often changes rapidly (as described in §2.1). Fixed allocation makes the scheduler

unable to adapt to jobs' changes in quality improvement and resource demands.

4 Design

This section describes the mechanisms by which SLAQ addresses its key challenges. First, how to normalize quality measures between distinct jobs in order to determine how quickly they are increasing (or not) in quality relative to one another (§4.1). Second, how SLAQ uses jobs' resource usage and quality information to precisely predict the impact of resource allocation in an online fashion (§4.2). Third, how SLAQ allocates resources to maximize system-wide quality improvement (§4.3).

4.1 Normalizing Quality Metrics

As explained in §2.1, ML training algorithms are designed to be an optimization process which iteratively minimizes a loss function, and thus improves the model's quality. ML algorithms use various different measurement metrics to indicate the quality of model training. Though comparing a single job's quality improvement across iterations is simple, comparing these metrics across different jobs presents a challenge. To schedule for better overall quality, we need to compare the quality metrics across different jobs. This enables SLAQ to trade off resources and quality between jobs.

One straightforward solution is to use a universal metric such as *accuracy* to measure the model quality. Accuracy represents the percentage of correctly predicted data points, and the range is always from 0 to 1. Similarly, the F1 score, ROC curve, and confusion matrix also measure the model quality taking the false positive and false negative ratios and multi-class results into consideration [37]. While these metrics are intuitively understandable to classification algorithms, they are not applicable to non-classification algorithms such as regression or unsupervised learning. In addition, accuracy and similar metrics require constructing a model and evaluating that model against a labeled validation set, which introduces an additional overhead to the job.[1]

Loss normalization. In contrast to the accuracy metrics, the loss function is calculated by the algorithm itself in each iteration, incurring no additional overhead. However, each algorithm's loss function has a different real-world interpretation. The range, convexity, and monotonicity of the loss functions depend on both the models and the optimization algorithms [20]. Directly normalizing loss values requires a priori knowledge of the loss range, which is impractical in an online setting.

[1] Validation is commonly used in ML training to prevent overfitting. Due to the overhead, however, model evaluation on the validation set is usually performed once every several iterations, not every iteration.

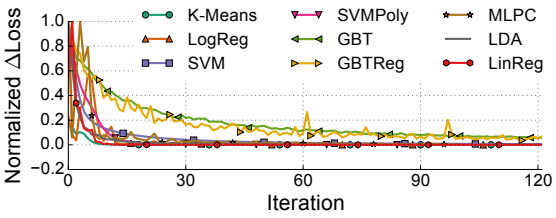


Figure 5: Normalized ΔLoss for ML algorithms.

For example, clustering algorithms (e.g., K-Means) use the *sum of squared distances to the cluster centroids* as the loss function. Classification and regression algorithms (e.g., SVM, Linear Regression, etc.) commonly use hinge or logistic gradient loss which represents *discrepancy of prediction* on the training data. The range of the measured values can vary by orders of magnitude: K-Means on our synthetic dataset reduces the loss from 300 down to 0, and the range highly depends on the absolute coordinates of the data points; on the other hand, SVM on a handwritten digit recognition dataset [34] reduces the loss from 1 down to 0.4. Unfortunately, there are no known analytical models to predict these ranges without actually running the training jobs.

Based on the convergence properties of loss functions (further explained in §4.2), we choose to normalize the *change* in loss values between iterations, as opposed to the loss values themselves. Most optimizers used in training algorithms try to reduce the values of loss functions, and for convex optimization problems, the values decrease monotonically [22]. The convergence rate, because of the diminishing returns, generally decreases in later iterations. So for a certain job, we normalize the change of loss values in the current iteration with respect to the largest change we have seen so far.

Figure 5 shows the normalized changes of loss values for common ML algorithms (summarized in Table 1). Because a loss function eventually converges to a certain value, the corresponding change of loss values always converges to 0. As a result, even though the set of algorithms have diverse loss ranges, we observe that they generally follow similar convergence properties, and can be normalized to decrease from 1 to 0. This helps SLAQ track the progress of different training jobs, and, for each job, correctly project the time to reach a certain loss reduction with a given resource allocation.

SLAQ supports a large class of important ML algorithms, but currently does not support some non-convex optimization algorithms due to the lack of convergence analytical models.

Algorithm	Acronym	Type	Optimization Algorithm	Dataset
K-Means	K-Means	Clustering	Lloyd Algorithm	Synthetic
Logistic Regression	LogReg	Classification	Gradient Descent	Epsilon [33]
Support Vector Machine	SVM	Classification	Gradient Descent	Epsilon
SVM (polynomial kernel)	SVMPoly	Classification	Gradient Descent	MNIST [34]
Gradient Boosted Tree	GBT	Classification	Gradient Boosting	Epsilon
GBT Regression	GBTReg	Regression	Gradient Boosting	YearPredictionMSD [35]
Multi-Layer Perceptron Classifier	MLPC	Classification	L-BFGS	Epsilon
Latent Dirichlet Allocation	LDA	Clustering	EM / Online Algorithm	Associated Press Corpus [36]
Linear Regression	LinReg	Regression	L-BFGS	YearPredictionMSD

Table 1: Summary of ML algorithms, types, and the optimizers and datasets we used for testing.

4.2 Measuring and Predicting Loss

After unifying the quality metrics for different jobs, we proceed to allocate resources for global quality improvement. When making a scheduling decision for a given job, SLAQ needs to know how much loss reduction the job would achieve by the next epoch if it was granted a certain amount of resources. We derive this information by predicting (i) how many iterations the job will have completed by the next epoch (§4.2.1), and (ii) how much progress (i.e., loss reduction) the job could make within these iterations (§4.2.2).

Prediction for iterative ML training jobs is different from general big-data analytics jobs. Previous work [15, 38] estimates job's runtime on some given cluster resources by analyzing the job computation and communication structure, using offline analysis or code profiling. As the computation and communication pattern changes during ML model configuration tuning, the process of offline analysis needs to be performed every time, thus incurring significant overhead. ML prediction is also different from the estimations to approximate analytical SQL queries [16, 17] where the resulting accuracy can be directly inferred with the sampling rate and analytics being performed. For iterative ML training jobs, we need to make *online* predictions for the runtime and intermediate quality changes for each iteration.

4.2.1 Runtime Prediction

SLAQ is designed to work with distributed ML training jobs running on batch-processing computational frameworks like Spark and MapReduce. The underlying frameworks help achieve *data parallelization* for training ML models: the training dataset is large and gets partitioned on multiple worker nodes, and the size of models (i.e., set of parameters) is comparably much smaller. The model parameters are updated by the workers, aggregated in the job driver, and disseminated back to the workers in the next iteration.

SLAQ's fine-grained scheduler resizes the set of workers for ML jobs frequently, and we need to predict the iteration of each job's iteration, even while the number and set of workers available to that job is dynamically changing. Fortunately, the runtime of ML training—at least for the set of ML algorithms and model sizes on which we focus—is dominated by the computation on the partitioned datasets. SLAQ considers the total CPU time of running each iteration as $c \cdot S$, where c is a constant determined by the algorithm complexity, and S is the size of data processed in an iteration. SLAQ collects the aggregate worker CPU time and data size information from the job driver, and it is easy to learn the constant c from a history of past iterations. SLAQ thus predicts an iteration's runtime simply by $c \cdot S/N$, where N is the number of worker CPUs allocated to the job.

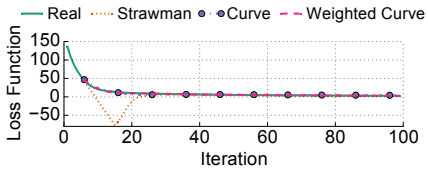
We use this heuristic for its simplicity and accuracy (validated through evaluation in §6.3), with the assumption that communicating updates and synchronizing models does not become a bottleneck. Even with models larger than hundreds of MBs (e.g., Deep Neural Networks), many ML frameworks could significantly reduce the network traffic with model parallelism [39] or by training with relaxed model consistency with bounded staleness [40], as discussed in §7. Advanced runtime prediction models [41] can also be plugged into SLAQ.

4.2.2 Loss Prediction

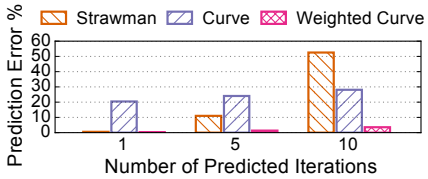
Iterations in some ML jobs may be on the order of 10s–100s of milliseconds, while SLAQ only schedules on the order of 100s of milliseconds to a few seconds. Performing scheduling on smaller intervals would be disproportionally expensive due to scheduling overhead and lack of meaningful quality changes. Further, as disparate jobs have different iteration periods, and these periods are not aligned, it does not make sense to try to schedule at "every" iteration of the jobs.

Instead, with runtime prediction, SLAQ knows how many iterations a job could complete in the given scheduling epoch. To understand how much quality improvement the job could get, we also need to predict the loss reduction in the following several iterations.

A strawman solution is to directly use the loss reduction obtained from the last iteration as the predicted loss reduction value for the following several iterations. This



(a) Predicting loss values 10 iterations in advance.



(b) Average loss prediction errors when predicting 1, 5 and 10 iterations in advance.

Figure 6: Predicting loss values with 3 methods.

method actually works reasonably well if we only need to predict one or two iterations. However, this could perform poorly in practice when the number of iterations per scheduling epoch is higher. This could be the case, for example, when the training dataset is small or an abundance of resources is allocated to the job.

We can improve the prediction accuracy by leveraging the convergence properties of the loss functions of different algorithms. Based on the optimizers used for minimizing the loss function, we can broadly categorize the algorithms by their convergence rate.

Algorithms with sublinear convergence rate. First-order algorithms in this category[2] have a convergence rate of $O(1/k)$, where k is the number of iterations [42]. For example, gradient descent is a first-order optimization method which is well-suited for large-scale and distributed computation. It can be used for SVM, Logistic Regression, K-Means, and many other commonly used machine learning algorithms. With optimized versions of gradient descent, the convergence rate could be improved to $O(1/k^2)$.

Algorithms with linear or superlinear convergence rates. Algorithms in this category[3] have a convergence rate of $O(\mu^k), |\mu| < 1$. For example, L-BFGS, which is a widely used Quasi-Newton Method, has a superlinear convergence rate which is between linear and quadratic. It can be used for SVM, Neural Networks, and others.

Distributed optimization algorithms. Optimization algorithms like gradient descent require a full pass through the complete dataset to update the model's parameters. This can be very expensive for large jobs which have data partitions stored on multiple nodes. Distributed ML training benefits from stochastic optimization algorithms. For example, stochastic gradient descent (SGD) processes a mini-batch (samples extracted from a subset of the training data) at a time and updates the parameters in each step. The significant efficiency improvement of SGD comes at the cost of slower convergence and fluctuation in loss functions. In terms of number of iterations, however, SGD still converges at a rate of $O(1/k)$ with properly randomized mini-batches.

With the assumptions of loss convergence rate, we use curve fitting to predict future loss reduction based on the history of loss values. For the set of machine learning algorithms we consider, we use the history of loss values at a certain time to fit a curve $f(k) = \frac{1}{ak^2+bk+c} + d$ for sublinear algorithms, or $f(k) = \mu^{k-b} + c$ for linear and superlinear algorithms.

We further improve the prediction accuracy using exponentially weighted loss values. Intuitively, loss values obtained in the near past are more informative for predicting the loss values in the near future. The weights assigned to loss values decay exponentially when new iterations finish, and the parameters of the curve equations get adjusted for each prediction.

Figure 6 shows the loss values predicted using the different methods described above. The strawman solution works well when predicting only one iteration in advance, but degrades quickly as the number of iterations to predict increases. The latter scenario is likely because SLAQ makes a scheduling decision once every epoch, which typically spans multiple iterations. In contrast, as shown in Figure 6(b), the weighted curve fitting method achieves a low average prediction error of 3.5% even when predicting up to 10 iterations in advance.

4.3 Scheduling Based on Quality Improvements

With accurate runtime and loss prediction, SLAQ allocates cluster CPUs to maximize the system-wide quality. SLAQ can flexibly support different optimization metrics, including both maximizing the total (sum) quality of all jobs, as well as maximizing the minimum quality (equivalent to max-min fairness) across jobs.

Maximizing the total quality. We schedule a set of J jobs running concurrently on the shared cluster for a fixed scheduling epoch T, i.e., a new scheduling decision can only be made after time T. The optimization problem for maximizing the total normalized loss reduction over a short time horizon T is as follows. Sum of allocated resources a_j cannot exceed the cluster resource capacity C.

$$\max_{j \in J} \quad \sum_j Loss_j(a_j, t) - Loss_j(a_j, t+T)$$
$$s.t. \quad \sum_j a_j \leq C$$

[2] Assume the loss function f is convex, differentiable, and ∇f is Lipschitz continuous.

[3] Assume the loss function f is convex and twice continuously differentiable, optimization algorithms can take advantage of the second-order derivative to get faster convergence.

Algorithm 1 Maximizing Total Loss Reduction

- *epoch*: scheduling time epoch
- *num_cores*: total number of cores available
- *alloc*: number of cores allocated to jobs
- *prior_q*: priority queue containing jobs and their loss reduction values if allocated with one extra core

```
1: function PredictLossReduction(job)
2:     pred_loss = PredictLoss(job, alloc[job], epoch)
3:     pred_loss_p1 = PredictLoss(job, alloc[job]+1, epoch)
4:     return pred_loss − pred_loss_p1
5: function AllocateResources(jobs)
6:     for all job in active jobs do
7:         alloc[job] = 1
8:         num_cores = num_cores − 1
9:         pred_loss_red = PredictLossReduction(job)
10:        prior_q.enqueue(job, pred_loss_red)
11:    while num_cores > 0 do
12:        job = prior_q.dequeue()
13:        alloc[job] = alloc[job] + 1
14:        num_cores = num_cores − 1
15:        pred_loss_red = PredictLossReduction(job)
16:        prior_q.enqueue(job, pred_loss_red)
17:    return alloc
```

When including job j at allocation a_j, we are paying cost of a_j and receiving value of $\Delta l_j = Loss_j(a_j,t) - Loss_j(a_j,t+T)$. The scheduler prefers jobs with highest value of $\Delta l_j/a_j$; i.e., we want to receive the largest gain in loss reduction normalized by resource spent.

Algorithm 1 shows the resource allocation logic of SLAQ. We start with $a_j = 1$ for each job to prevent starvation. At each step we consider increasing a_i (for all queries i) by one unit (in our implementation, one CPU core) and use our runtime and loss prediction logic to get the predicted loss reduction. Among these queries, we pick the job j that gives the most loss reduction, and increase a_j by one unit. We repeat this until we run out of available resources to schedule.

Maximizing the total loss reduction targets the cost-effectiveness of cluster resources. This is desirable not only on clusters used by a single company which may have high resource contention, but potentially even on multi-tenant clusters (clouds) in which revenue could be directly associated with the total quality progress (loss reduction) of ML jobs.

Maximizing the minimum quality. Below is the optimization problem to minimize the maximum loss value (or equivalently, maximizing the minimum quality) over time horizon T. With a set of J jobs running concurrently, this scheduling policy makes sure no one is *falling behind*. We require that all loss values be no bigger than l and we minimize l.

$$\min_{j \in J} \quad l$$
$$s.t. \quad \forall j : Loss_j(a_j, t+T) \le l$$
$$\sum_j a_j \le C$$

The system quality, in this case, is represented by the loss value l of the worst job j. The only way we can improve it is to reduce the loss value of j. Our heuristic is thus as follows. We start with $a_j = 1$, and at each step we pick job $i = \arg\min_j Loss_j(a_j, t+T)$. We increase its allocation a_i by one unit, recompute $Loss_i(a_i, t+T)$, and repeat this process until we run out of resources.

Maximizing the minimum quality achieves max-min fairness in model quality. It is especially useful for ML applications that include multiple collaborative models, and the overall quality is determined by the lowest quality of all the submodels. For example, a security application for network intrusion detection should train multiple collaborative models identifying distinct attacking patterns with max-min fairness in quality.

Prioritize jobs on shared clusters. The above scheduling policies are based on the assumptions that all the concurrently running jobs have equal importance, and thus they will be treated equally when comparing their quality. This can be easily adjusted to account for jobs with different importance by adding a weight multiplier to the jobs, identically to how max-min fairness can be easily changed to weighted max-min fairness.

For example, a cluster may host experiment jobs and production jobs for ML training, and a higher weight should be assigned to jobs for production uses. With the same training progress, a job with a higher weight will get its loss reduction proportionally amplified by the scheduler compared to a normal job. Thus, high-priority jobs generally get more iterations finished with SLAQ.

Mixing ML with other types of jobs. SLAQ can also run non-ML jobs sharing the same cluster with approximate ML jobs. For non-ML jobs, the scheduler falls back to fairness or reservation based resource allocation. This effectively reduces the total capacity C available to all approximate ML jobs. SLAQ follows the same algorithms to maximize the total or minimum quality under varying resource capacity C.

5 Implementation

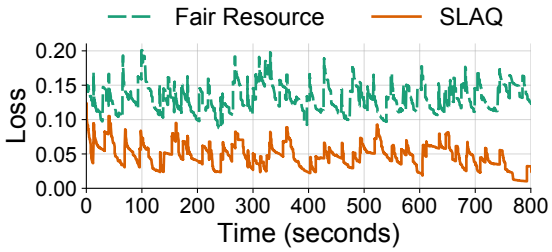
We implemented SLAQ within the popular Apache Spark framework [19], and utilize its accompaning MLlib machine learning library [5]. Spark MLlib describes ML workflow as a pipeline of *transformers*, and it provides a set of high-level APIs to help design ML algorithms on large datasets. Many commonly used ML algorithms are pre-built in MLlib, including feature extraction, classification, regression, clustering, collaborative filtering, and so on. These algorithms can easily be extended and modified for specific use cases.

The SLAQ prototype is implemented based on the Spark job scheduler. Multiple jobs place the ready tasks into task pools, which are then controlled and dispatched

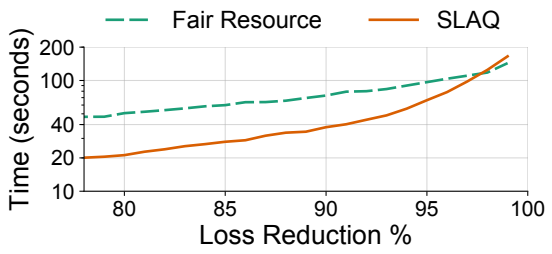
by SLAQ scheduler. The driver programs of ML jobs continually report their loss value information for each iteration they finish.

Token bucket. SLAQ uses a token bucket algorithm to implement the resource allocation policies described in §4.3. At each scheduling epoch, CPU time of all allocated cores is added to each job as *tokens*. SLAQ assigns tasks to available workers, and keeps track of how many tokens are consumed by those tasks by collecting Spark worker statistics. Tasks are throttled if the corresponding job has used up its tokens.

Running unmodified ML applications. ML *applications* written using Spark MLlib can directly run on SLAQ without any modifications. This is because SLAQ extends the underlying *optimizers* (e.g., SGD, L-BFGS, etc.) APIs to report loss values at each iteration. We cover most library algorithms provided in MLlib. Even when it is necessary to add new library algorithms, one can easily adopt SLAQ by reporting loss values using SLAQ's API. This is a one-line modification in most of the algorithms present in MLlib.



(a) Average of normalized loss values.



(b) Time to achieve loss reduction percentage.

Figure 7: Comparing loss improvement and runtime between SLAQ and fair scheduler.

6 Evaluation

In this section, we present evaluation results on SLAQ. We demonstrate that SLAQ (i) provides significant improvement on quality and runtime for approximate ML training jobs, (ii) is broadly applicable to a wide range of ML algorithms, and (iii) scales to run a large number of ML training algorithms on clusters.

6.1 Methodology

Testbed. Our testbed consists of a cluster of 20 instances of c3.8xlarge machines on Amazon EC2 Cloud. Each worker machine has 32 vCPUs (Intel Xeon E5-2680 v2 @ 2.80 GHz), 60GB RAM, and is connected with 10Gb Ethernet links.

Workload. We tested our system with the most common ML algorithms derived from MLlib with minor changes, including (i) classification algorithms: SVM, Neural Network (MLPC), Logistic Regression, GBT, and our extension to Spark MLlib with SVM polynomial kernels; (ii) regression algorithms: Linear Regression, GBT Regression; (iii) unsupervised learning algorithms: K-Means clustering, LDA. Each algorithm is further diversified to construct different models. For example, SVM with different kernels, and MLPC Neural Network with different numbers of hidden layers and perceptrons.

Datasets. With the algorithms, our models are trained on multiple datasets we collected from various online sources with modifications, as well as on our synthetic datasets. The datasets span a variety of types (plain texts [36], images [34], audio meta features [35], and so on [43]). The size of the distinct datasets we use in each run is more than 200GB. In the experiments, all the training datasets are cached as Spark Dataframes in cluster shared memory. We set the fraction of data sample processed at each iteration to be 100%, i.e., the entire training data is processed in every iteration.

Baseline. The baseline we compare against is a work-conserving fair scheduler. It is the widely-used scheduling policy for cluster computing frameworks [9, 10, 11, 13, 14]. The fair scheduler evenly divides available resources to all active jobs. It also dynamically adjusts resource allocations to fair share when new jobs join and old jobs leave the system.

6.2 System Performance

6.2.1 Scheduler Quality and Runtime Improvement

To evaluate job quality improvement, we first run a set of 160 ML training jobs with different algorithms, model sizes, and datasets on the shared cluster of 20 nodes. In the experiment, jobs are submitted to the cluster with their arrival time following a Poisson distribution (mean arrival time 15s). A job is considered fully converged when its normalized loss reduction is below a very small value, in this case, the loss reduction at the 100th iteration.[4] We compare the aggregate quality and runtime of these jobs between SLAQ and the fair scheduler.

[4]Recall that the loss reduction for each iteration is independent of the amount of resources the job is allocated; the resource allocation instead dictates the amount of *wall-clock time* each iteration takes.

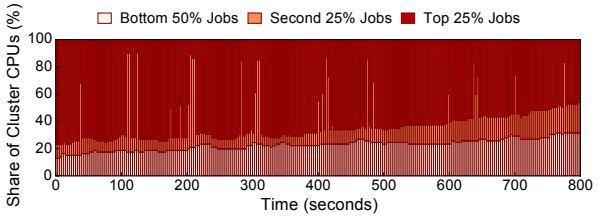


Figure 8: Resource allocation across jobs. At the beginning, jobs with the greatest 25% loss allocated vast majority of resources; towards the end, the difference in loss shrinks, the allocation is more spread out.

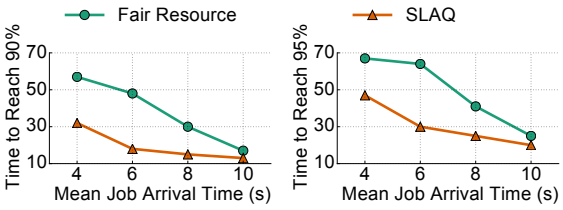


Figure 9: The performance difference between SLAQ and a fair resource scheduler is more significant under workloads with greater contention, e.g., jobs arriving with a mean arrival time of 4s compared to 10s.

Figure 7(a) shows the average normalized loss values across running jobs with SLAQ and the fair scheduler in an 800s time window of the experiment. When a new job arrives, its initial loss is 1.0, raising the average loss value of the active jobs; the spikes in the figure indicate new job arrivals. Yet because SLAQ allocates resources to maximize the total quality improvement (loss reduction), the average loss value of all active jobs using SLAQ is much lower than with the fair scheduler. In particular, SLAQ's average loss value is 0.49 at each scheduling epoch, which is 73% lower than that of the fair scheduler.

Figure 7(b) shows the average time it takes a job to achieve different loss values. As SLAQ allocates more resources to jobs that have the most potential for quality improvement, it reduces the average time to reach 90% (95%) loss reduction from 71s (98s) down to 39s (68s), 45% (30%) lower. At the very end of the job execution, further iterations take longer time as the job quality is less likely to be improved. Thus, in an environment where users submit exploratory ML training jobs, SLAQ could substantially reduce users' wait times.

Figure 8 explains SLAQ's benefits by plotting the allocation of CPU cores in the cluster over time. Here we group the active jobs at each scheduling epoch by their normalized loss: (i) 25% jobs with high loss values; (ii) 25% jobs with medium loss values; (iii) 50% jobs with low loss values (almost converged). With a fair scheduler, the cluster CPUs should be allocated to the three groups proportionally to the number of jobs. In contrast, SLAQ adapts to the job quality improvement, and allocates much more computation resource to (i) and (ii). In fact, jobs in group (i) take 60% of cluster CPUs, while jobs in group (iii), despite having 50% of the population, get only 22% of cluster CPUs on average. SLAQ transfers many resources from nearly converged jobs to the jobs that have the most potential for significant quality improvement, which is the underlying reason for the improvement in Figure 7.

6.2.2 Handling Different Workloads

The achieved qualities of training jobs strongly depend on the cluster workload. As the workload increases, it becomes more important to efficiently utilize the resources. In this experiment, we vary the mean arrival time of new jobs, which in turn varies the number of concurrent jobs, and observe how SLAQ and the fair scheduler handle resource contention under different workloads.

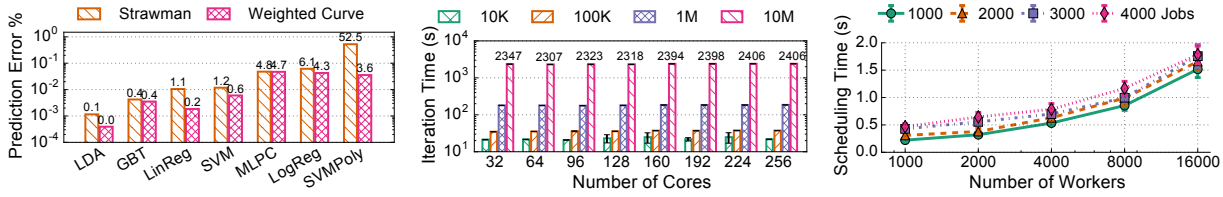
Figure 9 illustrates that SLAQ achieves a greater relative benefit over a fair schedule under more contentious or aggressive workloads. We start with a mean arrival time of 10s (or equivalently, 6 new jobs per minute). Under the light workload, the computation resources are relatively abundant for each job, so the time to reach 90% (95%) loss reduction is similar for both schedulers, with SLAQ performing 23% (20%) better.

As we increase the system workload with smaller mean job arrival times, cluster resource contention increases. SLAQ allocates resources to the jobs with the greatest potential. As a result, when the mean arrival time is 4s (15 new jobs per minute), SLAQ achieves an average time for jobs to reach 90% (95%) loss reduction that is 44% (30%) less than the fair scheduler.

6.3 Robustness of Prediction

SLAQ relies on an estimate of the expected loss reduction of a job, given a certain resource alloction (see §4.2). To ensure stability, SLAQ makes a reallocation decision only once per scheduling epoch. Thus, the scheduler requires (i) the loss predictor to precisely estimate the loss values at least a few iterations in advance, and (ii) the runtime predictor to accurately report how long each iteration takes with a certain number of allocated cores.

Figure 10(a) plots the loss prediction error for the types of ML algorithms we tested (Table 1). We compare the loss prediction error relative to the true values for 10 iterations, with both strawman and weighted curve fitting methods of §4.2. Our prediction achieves less than 5% prediction errors for all the algorithms.



(a) Predicting the next 10th iteration. (b) Average CPU time to finish each iteration. (c) Scheduling time.

Figure 10: SLAQ loss / runtime prediction and overhead.

Recall that SLAQ uses a simple heuristic to estimate the iteration runtime with N cores. To demonstrate that each iteration's CPU time is $c \cdot S$ (c as a constant), regardless of how many workers are allocated, we evaluate the total CPU time to complete an iteration with a fixed data size S. We vary the number of workers (32 cores each) between 1 and 8 and training neural network models of sizes from 10KB to 10MB. Figure 10(b) illustrates that, at least for ML models smaller than tens of MB, communication and model synchronization do not affect processing time. Therefore, when dynamically changing N, an iteration's time can simply be estimated as $c \cdot S/N$. We discuss extending SLAQ to large models in §7.

6.4 Scalability and Efficiency

Figure 10(c) plots the time taken by SLAQ to schedule tens of thousands of concurrent jobs on large clusters (simulating both the jobs and worker nodes). SLAQ makes its scheduling decisions in between hundreds of milliseconds to a few seconds, even when scheduling 4000 jobs across 16K worker cores. These decisions are made each scheduling epoch, a timeframe of a few seconds. As shown in Figure 6, the more iterations in advance SLAQ predicts, the larger potential error it will incur. The agility of SLAQ enables the scheduler to predict only a few iterations in advance for each ML training job, adjusting its resource allocation decisions frequenty to meet the jobs' quality goals. SLAQ's scheduling time is comparable to the scalability of schedulers in many big data clusters today, leading us to conclude that SLAQ is sufficiently fast and scalable for (rather aggressive) real-world needs.

7 Discussion

Communication overhead. SLAQ is tested with ML models that have a moderate number of parameters. Recent developments in distributed frameworks for training ML models, especially deep neural networks (DNN), incur more communication and synchronization overhead between the ML job driver and worker nodes. For example, with a large number of perceptrons and multiple layers, a DNN model can grow to tens of GBs [44, 45].

Since our current implementation is based on Spark, the driver essentially becomes a single-node parameter server [46], which is responsible for gathering, aggregating, and distributing the models in every iteration. This communication overhead—due to Spark's architecture—limits our ability to train large models.

Several solutions have been proposed to mitigate the communication overhead problem. Model parallelization using architectures based on parameter servers or graph computing proportionally scale the model serving nodes with the workers [7, 8, 39, 47]. With these optimized frameworks, SLAQ's performance improvement based on online prediction and scheduling heuristics should apply to large ML models.

Distributed ML training with relaxed consistency. Distributed ML frameworks used in practice leverage a relaxed consistency model with bounded staleness [40] to reduce the communication costs during model synchronization. The convergence progress of the underlying ML training algorithms is typically robust to a certain degree of fluctuation and slack, so the efficiency improvement obtained from the parallelism outweighs the staleness slowdown on convergence rate.

A commonly used execution model with bounded staleness is Bulk Synchronous Parallel (BSP), which allows multiple workers to individually update on partitioned training data and only synchronizes their models every several iterations [30, 47, 48]. We can extend SLAQ to support these frameworks by collecting the batch iteration time on each worker, and the model quality and communication time at each synchronization barrier to help estimate the loss reduction under the two levels of iterativeness. In fact, the convergence property of ML training is also studied in [48] with the BSP execution model under various conditions (e.g., varying communication latency and cluster sizes).

Non-convex optimization. SLAQ's loss prediction is based on the convergence property of the underlying optimizers and curve fitting with the loss history. Loss functions of non-convex optimization problems are not guaranteed to converge to global minima, nor do they

necessarily decrease monotonically. The lack of an analytical model of the convergence properties interferes with our prediction mechanism, causing SLAQ to underestimate or overestimate the potential loss reduction.

One solution to this problem is to let users provide the scheduler with hint of their target loss or performance, which could be acquired from state-of-the-art results on similar problems or previous training trials. The convergence properties and optimization of non-convex algorithms is being actively studied in the ML research community [49, 50]. We leave modeling the convergence of these algorithms to future work.

8 Related Work

Approximate computing systems. Many systems [23, 51, 52, 17, 24, 25] allow users to get approximate results with significantly reduced job completion time. Online aggregation databases [16, 26] generate approximate results and iteratively refine the quality. While we designed SLAQ for iterative ML training jobs, our techniques are broadly applicable to scheduling data analytics systems that iteratively refine their results.

Scheduling ML systems. Large-scale ML frameworks [5, 7, 8, 39, 53, 54, 55] optimize the computation and resource allocation for multi-dimensional matrix operators within a training job. These systems greatly accelerate the training process and reduce job's synchronization overhead. As a cluster scheduler, SLAQ could support different underlying ML frameworks (with modifications) in the future, and allocate resources at the job level to optimize across different ML training jobs.

ML model search. Several systems [2, 41] are designed to accelerate the model searching procedure. TuPAQ [41] uses a planning algorithm to discover hyperparameter settings and exclude bad trials automatically. SLAQ is designed for ML training in general exploratory settings on multi-tenant clusters. Automated model search systems could work in conjuction with SLAQ for faster decisions and better cluster utilization.

Cluster scheduling systems. Existing cluster schedulers [9, 10, 11, 12, 13, 14] primarily focus on resource fairness, job priorities, cluster utilization, or resource reservations, but do not take job quality into consideration. They mostly ignore the quality-time trade-off, and the quality trade-off between jobs. This trade-off space is crucial for ML training jobs to get approximate results with much less resource usage and lower latency.

Estimation of resource usage and runtime. Ernest [15] predicts job quality and runtime based on the internal computation and communication structures of large-scale data analytics jobs. CherryPick [38] improves cloud configuration selection process using Bayesian Optimization. Despite the generality, these systems require jobs to be analyzed offline. When users debug and adjust their models, the computation structure is likely to change very often, and thus the offline analysis will bring significant overhead. NearestFit [56] provides a progress indicator for a broad class of MapReduce applications with online prediction. SLAQ uses also online prediction to avoid offline overhead, and leverages the iterative nature of ML training jobs to improve the accuracy of prediction.

Deadline-based scheduling. Many systems [57, 58, 59, 60] utilize scheduling to meet deadlines for batch processing jobs or to reduce lag for streaming analytics jobs. Jockey [61] uses a combination of offline prediction and dynamic resource allocation to ensure batch processing queries meet their latency SLAs while minimizing their impact on other jobs sharing the cluster. Instead of hard deadlines, some real-time systems [62, 63] use soft deadlines and penalize additional delay beyond the deadlines. However, these systems mainly consider the quality-runtime trade-offs for a single job, instead of optimizing across multiple approximate jobs.

Utility scheduling. Utility functions have been widely studied in network traffic scheduling to encode the benefit of performance to users [64, 65, 66]. Recent work on live video analytics [67] leverages utility-based scheduling to provide a universal performance measurement to account for both quality and lag.

9 Conclusion

We present SLAQ, a quality-driven scheduling system designed for large-scale ML training jobs in shared clusters. SLAQ leverages the iterative nature of ML algorithms and obtains application-specific information to maximize the quality of models produced by a large class of ML training jobs. Our scheduler automatically infers the models' loss reduction rate from past iterations, and predicts future resource consumption and loss improvements online for subsequent allocation decisions. As a result, SLAQ improves the overall quality of executing ML jobs faster, particularly under resource contention.

Acknowledgments

We are grateful to Siddhartha Sen, Daniel Suo, Linpeng Tang, Marcela Melara, Amy Tai, Aaron Blankstein, and Elad Hazan for reading early versions of the draft and providing feedback. We also thank our shepherd Immanuel Trummer and the anonymous SoCC reviewers for their valuable and constructive feedback. This work was supported by NSF Awards CNS-0953197 and IIS-1250990.

References

[1] M. Anderson, D. Antenucci, V. Bittorf, M. Burgess, M. Cafarella, A. Kumar, F. Niu, Y. Park, C. Ré, and C. Zhang. Brainwash: A Data System for Feature Engineering. In *CIDR*, 2013.

[2] J. Snoek, H. Larochelle, and R. P. Adams. Practical Bayesian Optimization of Machine Learning Algorithms. In *NIPS*, 2012.

[3] D. Maclaurin, D. Duvenaud, and R. P. Adams. Gradient-based Hyperparameter Optimization through Reversible Learning. 2015.

[4] S. Han, H. Mao, and W. J. Dally. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. *CoRR*, abs/1510.00149, 2015.

[5] X. Meng, J. K. Bradley, B. Yavuz, E. R. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. MLlib: Machine Learning in Apache Spark. *CoRR*, abs/1505.06807, 2015.

[6] H2O: Open Source Platform for AI. Retrieved 04/20/2017, URL: https://docs.h2o.ai.

[7] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A System for Large-scale Machine Learning. In *USENIX OSDI*, 2016. ISBN 978-1-931971-33-1.

[8] Caffe2. Retrieved 04/20/2017, URL: https://github.com/caffe2/caffe2.

[9] Apache Hadoop YARN. Retrieved 02/08/2017, URL: http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[10] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *USENIX NSDI*, 2011.

[11] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *USENIX NSDI*, 2011.

[12] A. A. Bhattacharya, D. Culler, E. Friedman, A. Ghodsi, S. Shenker, and I. Stoica. Hierarchical Scheduling for Diverse Datacenter Workloads. In *ACM SoCC*, 2013.

[13] Capacity Scheduler. Retrieved 04/20/2017, URL: https://hadoop.apache.org/docs/r2.4.1/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html.

[14] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *ACM SOSP*, 2009.

[15] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *USENIX NSDI*, 2016.

[16] K. Zeng, S. Agarwal, and I. Stoica. iOLAP: Managing Uncertainty for Efficient Incremental OLAP. In *ACM SIGMOD*, 2016.

[17] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *ACM EuroSys*, 2013.

[18] S. Zilberstein. Using anytime algorithms in intelligent systems. *AI magazine*, 17(3):73, 1996.

[19] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *USENIX NSDI*, 2012.

[20] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[21] L. Bottou and O. Bousquet. The Tradeoffs of Large Scale Learning. In *NIPS*, 2008.

[22] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[23] B. Babcock, S. Chaudhuri, and G. Das. Dynamic Sample Selection for Approximate Query Processing. In *ACM SIGMOD*, 2003.

[24] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu. GRASS: Trimming Stragglers in Approximation Analytics. In *USENIX NSDI*, 2014.

[25] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica. The Power of Choice in Data-aware Cluster Scheduling. In *USENIX OSDI*, 2014.

[26] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. In *ACM SIGMOD*, 1997.

[27] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online Aggregation for Large MapReduce Jobs. *Proceedings of the VLDB Endowment*, 4(11), 2011.

[28] Y. Tohkura. A Weighted Cepstral Distance Measure for Speech Recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 35:1414–1422, 1987.

[29] Y. L. Cun, J. S. Denker, and S. A. Solla. Optimal Brain Damage. In *NIPS*. 1990.

[30] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *USENIX OSDI*, 2014.

[31] Apache Hadoop. Retrieved 02/08/2017, URL: http://hadoop.apache.org.

[32] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *USENIX OSDI*, 2014.

[33] PASCAL Challenge 2008. Retrieved 04/20/2017, URL: http://largescale.ml.tu-berlin.de/instructions/.

[34] MNIST Database. Retrieved 04/20/2017, URL: http://yann.lecun.com/exdb/mnist/.

[35] Million Song Dataset. Retrieved 04/20/2017, URL: https://labrosa.ee.columbia.edu/millionsong/.

[36] Associated Press Dataset - LDA. Retrieved 04/20/2017, URL: http://www.cs.columbia.edu/~blei/lda-c/.

[37] D. Powers. Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness & Correlation. *Journal of Machine Learning Technologies*, 2(1): 37–63, 2011.

[38] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *USENIX NSDI*, 2017.

[39] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling Distributed Machine Learning with the Parameter Server. In *USENIX OSDI*, 2014.

[40] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Exploiting Bounded Staleness to Speed Up Big Data Analytics. In *USENIX ATC*, 2014.

[41] E. R. Sparks, A. Talwalkar, D. Haas, M. J. Franklin, M. I. Jordan, and T. Kraska. Automating Model Search for Large Scale Machine Learning. In *ACM SoCC*, 2015.

[42] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference and Prediction*. Springer, 2nd edition, 2009.

[43] LibSVM Data. Retrieved 04/20/2017, URL: https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/.

[44] Q. V. Le, R. Monga, M. Devin, G. Corrado, K. Chen, M. Ranzato, J. Dean, and A. Y. Ng. Building High-Level Features Using Large Scale Unsupervised Learning. *CoRR*, abs/1112.6209, 2011.

[45] K. Ni, R. A. Pearce, K. Boakye, B. V. Essen, D. Borth, B. Chen, and E. X. Wang. Large-Scale Deep Learning on the YFCC100M Dataset. *CoRR*, abs/1502.03409, 2015.

[46] Arimo TensorSpark. Retrieved 04/20/2017, URL: https://goo.gl/SYPMIZ.

[47] W. Xiao, J. Xue, Y. Miao, Z. Li, C. Chen, M. Wu, W. Li, and L. Zhou. Tux²: Distributed Graph Computation for Machine Learning. In *USENIX NSDI*, 2017.

[48] P. Moritz, R. Nishihara, I. Stoica, and M. I. Jordan. SparkNet: Training Deep Networks in Spark. *CoRR*, abs/1511.06051, 2015.

[49] N. Boumal, P.-A. Absil, and C. Cartis. Global Rates of Convergence for Nonconvex Optimization on Manifolds. *ArXiv e-prints*, May 2016.

[50] S. Lacoste-Julien. Convergence Rate of Frank-Wolfe for Non-Convex Objectives. *ArXiv e-prints*, July 2016.

[51] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. In *ACM SIGMOD*, 1997.

[52] C. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable Approximate Query Processing with the DBO Engine. *ACM Transactions on Database Systems*, 33(4):23, 2008.

[53] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR*, abs/1512.01274, 2015.

[54] F. Seide and A. Agarwal. CNTK: Microsoft's Open-Source Deep-Learning Toolkit. In *KDD*, 2016.

[55] PyTorch. Retrieved 04/20/2017, URL: http://pytorch.org/.

[56] E. Coppa and I. Finocchi. On Data Skewness, Stragglers, and MapReduce Progress Indicators. In *ACM SoCC*, 2015.

[57] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure. Adaptive Control of Extreme-Scale Stream Processing Systems. In *IEEE ICDCS*, July 2006.

[58] A. Verma, L. Cherkasova, and R. H. Campbell. ARIA: Automatic Resource Inference and Allocation for Mapreduce Environments. In *ICAC*, 2011.

[59] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, Í. Goiri, S. Krishnan, J. Kulkarni, and S. Rao. Morpheus: towards automated SLOs for enterprise clusters. In *USENIX OSDI*, 2016.

[60] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao. Reservation-based Scheduling: If You're Late Don't Blame Us! In *ACM SoCC*, 2014.

[61] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *ACM EuroSys*, 2012.

[62] E. Wandeler and L. Thiele. Real-time Interfaces for Interface-based Design of Real-time Systems with Fixed Priority Scheduling. In *5th ACM International Conference on Embedded Software*, 2005.

[63] E. D. Jensen, P. Li, and B. Ravindran. On Recent Advances in Time/Utility Function Real-Time Scheduling and Resource Management. *IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, 2005.

[64] R. Johari and J. N. Tsitsiklis. Efficiency Loss in a Network Resource Allocation Game. *Math. Oper. Res.*, 29: 407–435, 2004.

[65] F. P. Kelly, A. K. Maulloo, and D. K. H. Tan. Rate Control for Communication Networks: Shadow Prices, Proportional Fairness and Stability. *The Journal of the Operational Research Society*, 49:237–252, 1998.

[66] S. H. Low and D. E. Lapsley. Optimization Flow Control—I: Basic Algorithm and Convergence. *IEEE/ACM Transactions on Networking*, 7(6):861–874.

[67] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman. Live Video Analytics at Scale with Approximation and Delay-Tolerance. In *USENIX NSDI*, 2017.