Automating Model Search for Large Scale Machine Learning

Evan R. Sparks
Computer Science Division
UC Berkeley
sparks@cs.berkeley.edu

Ameet Talwalkar
Computer Science Dept.
UCLA
ameet@cs.ucla.edu

Daniel Haas
Computer Science Division
UC Berkeley
dhaas@cs.berkeley.edu

Michael J. Franklin
Computer Science Division
UC Berkeley
franklin@cs.berkeley.edu

Michael I. Jordan
Computer Science Division
UC Berkeley
jordan@cs.berkeley.edu

Tim Kraska
Dept. of Computer Science
Brown University
tim_kraska@brown.edu

Abstract

The proliferation of massive datasets combined with the development of sophisticated analytical techniques has enabled a wide variety of novel applications such as improved product recommendations, automatic image tagging, and improved speech-driven interfaces. A major obstacle to supporting these *predictive applications* is the challenging and expensive process of identifying and training an appropriate predictive model. Recent efforts aiming to automate this process have focused on single node implementations and have assumed that model training itself is a black box, limiting their usefulness for applications driven by large-scale datasets. In this work, we build upon these recent efforts and propose an architecture for automatic machine learning at scale comprised of a cost-based cluster resource allocation estimator, advanced hyperparameter tuning techniques, bandit resource allocation via runtime algorithm introspection, and physical optimization via batching and optimal resource allocation. The result is TUPAQ, a component of the MLbase system that automatically finds and trains models for a user's predictive application with comparable quality to those found using exhaustive strategies, but an order of magnitude more efficiently than the standard baseline approach. TUPAQ scales to models trained on Terabytes of data across hundreds of machines.

1. Introduction

Rapidly growing data volumes coupled with the maturity of sophisticated statistical techniques have led to a new type of data-intensive workload: predictive analytics. In order to make useful predictions, a high-quality predictive model must be built, typically using sophisticated machine learning (ML) techniques that often rely on large-scale, distributed datasets to achieve high statistical performance.

SoCC '15, August 27-29, 2015, Kohala Coast, HI, USA.

ACM ACM 978-1-4503-3651-2/15/08…$15.00.
http://dx.doi.org/10.1145/2806777.2806945

To come up with high quality models that make accurate predictions, researchers, data scientists, and business analysts must make a number of important decisions. First, an input dataset must be transformed from a domain specific format to features which are predictive of the field of interest. Although this feature engineering task is challenging, it addresses only a portion of the design space of machine learning. Once features have been engineered, users must make several other important decisions. They must pick a learning setting appropriate to their problem—for example, regression, classification, or recommendation. Next, users must choose an appropriate model, such as Logistic Regression or a Kernel SVM. Each model family has a number of hyperparameters, such as degree of regularization or learning rate, and each of these must be tuned to an appropriate value. Finally, users must pick a software package that can train their model, choose to configure one or more machines to execute the training routine, and evaluate the resulting model's quality. In our experience, the initial model configuration selected by the user is almost always suboptimal, owing to the complexity and number of decisions that precede it. Identifying a high quality model thus typically involves a costly and often manual search process. The decision process and exponentially large space of candidate configurations is illustrated in Figure 1.

Distributed and cloud computing provide a compelling way to accelerate this process, but also present additional challenges. Though parallel storage and processing techniques enable users to train models on massive datasets and accelerate the search process by training multiple models at once, the distributed setting forces several more decisions upon users: what parallel execution strategy to use, how big a cluster to provision, how to efficiently distribute computation across it, and what machine learning framework to use. These decisions are onerous—particularly for users who are experts in their own field but inexperienced in machine learning and distributed systems.

Existing techniques to automate the search for a high-quality predictive model focus on the single node setting; extensions for large-scale problems or distributed settings are very basic [28]. Moreover, while many machine learning frameworks have been designed to train a single predictive model with fixed hyperparameter configurations efficiently, they provide at best rudimentary and inefficient tools to aid in the search *among* hyperparameter configurations for a high quality model.

To address these challenges, we present TUPAQ, a system designed to efficiently and scalably automate the process of training predictive models. Central to the system is a planning algorithm

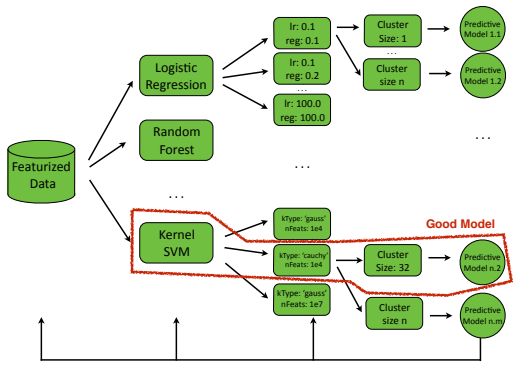

Figure 1: Finding an appropriate predictive model for a dataset is a process of continuous refinement. Each stage must be carefully tuned to ensure high quality. TUPAQ automates this process. Here 'lr' denotes a learning rate parameter, 'reg' the degree of regularization, 'kType' a kernel to use and 'nFeats' the number of random features to use.

which decides on an efficient parallel execution strategy during model training, and uses sophisticated techniques both to identify new hyperparameter configurations to try and to proactively eliminate models which are unlikely to provide good results.

In this paper, we make the following contributions:

- We introduce a simple, workload-driven cluster size estimator that determines the appropriate number of machines to use when fitting large-scale ML models.
- We describe the TUPAQ algorithm for large scale model search which combines advanced hyperparameter tuning techniques with physical optimization for efficient execution.
- We describe an implementation of the TUPAQ algorithm in Apache Spark, building on our earlier work on the MLbase architecture [29].
- We evaluate several points in the design space with respect to each of our logical and physical optimizations, and demonstrate that proper selection of each can dramatically improve both accuracy and efficiency.
- We present experimental results on large, distributed datasets up to Terabytes in size, demonstrating that TUPAQ's search techniques converge to high quality models an order of magnitude faster than a standard model search strategy.

In the remainder of this paper, we formally define the model search problem, and provide a high level overview of TUPAQ and its architecture. Next, we present details about TUPAQ's four main optimizations. Then, we present a large-scale evaluation of TUPAQ. We conclude with a discussion of related and future work.

2. Model Search and TUPAQ

In this section, we define the model search problem in more detail, and compare two approaches to solving it. The first, which we call the baseline approach, is inspired by common practice. The second approach, used by TUPAQ, allows us to take advantage of logical and physical optimizations in the model search process. TUPAQ has a rich design space, which we describe in further detail in Section 3. Finally, we describe the architecture of TUPAQ and how it fits into the broader MLbase architecture.

2.1 Defining Model Search

Given a dataset, an attribute of that dataset to predict, and a space of possible model configurations to consider, the goal of *model search* is to find a supervised learning model that will provide good predictions for the attribute of interest on unseen data—that is, a model with low generalization error. We focus specifically on the supervised learning setting, where each element of the training dataset has a label or score associated with it. In our environment, the user's dataset may consist of up to millions of training data points and hundreds of thousands of features. Datasets this size are commonly needed to build high quality models in domains such as computer vision, speech recognition, and natural language processing.

The model search procedure aims to find a model that maximizes some measure of quality (e.g., in terms of goodness of fit to held-out data) in a short amount of time, where learning resources are constrained by some budget in terms of the number of models considered, maximum execution time, number of scans over the training data, or even total money to spend with a cloud computing provider. The model search procedure thus takes as input a training dataset, a description of a space of models to search, and some budget or stopping criterion. The description of the space of models to search includes the set of model families to search over (e.g., SVM, decision tree, etc.) and reasonable ranges for their associated hyperparameters (e.g., regularization parameter for a regularized linear model or maximum depth of a decision tree). The output of the procedure is a model that can be applied to unlabeled data points to obtain a prediction for the desired attribute.

2.2 Our Setting

In this work, we operate in a scenario where individual models are of dimensionality d, where d is less than the total number of example data points n. In the binary classification setting this corresponds to d features in the training data. Note that, despite being smaller than n, d can nonetheless be quite large, e.g., $d = 200,000$ in our large-scale speech experiments and $d = 160,000$ in our large scale image experiments (see Section 5). We are also focused on the situation where the data size ($n * d$) is very large. We run experiments on up to terabytes of data and our system is designed to scale even further.

Moreover, we focus on the classification setting, and we consider a small number of model families, $f \in F$, each with several hyperparameters, $\lambda \in \Lambda$. These assumptions map well to reality, as there are a handful of general-purpose classification methods that are deployed in practice. Further, we expect that these techniques will naturally apply to other supervised learning tasks—such as regression and collaborative filtering.

We evaluate the quality of each model by computing accuracy on held-out datasets, and we measure search time as the amount of time required to explore a fixed number of models from some model space. This accuracy measure is our measure of model quality. In our large-scale distributed experiments (see Section 5) we report parallel run times.

Additionally, we focus on model families that are trained via multiple sequential scans of the training data as opposed to model families which require random access to training data to compute their updates. This iterative sequential access pattern encompasses a wide range of learning algorithms, especially in the large-scale distributed setting. For instance, efficient distributed implementations of linear regression [36], tree based models [38], Naive Bayes classifiers [36], and k-means clustering [36] all follow this same access pattern. In particular, we focus on three model families: linear Support Vector Machines (SVM), logistic regression trained via gradient descent, and nonlinear SVMs using random features [41] trained via block coordinate descent. These model families were

chosen primarily because of their practical popularity, as well as the ease with which the batching optimization described in Section 3 can be applied to them.

2.3 Connections to Query Optimization

Our view is that model search can be considered as a form of query optimization. If we view model search as a task which is specified declaratively in terms of a search space, data, and an objective function, this becomes more clear. Given this observation, it is natural to draw connections between automating the model search process and the decades worth of research in optimizing declarative relational database queries. Traditional database systems invest in the costly process of query planning to determine a good execution plan that can be reused repeatedly upon subsequent execution of similar queries. Similarly, model search involves the costly process of identifying a high quality predictive model in order to subsequently perform near real-time model evaluation. Indeed, relational query optimization is commonly viewed as a search problem, where the optimizer must find a good query plan in the large space of join orderings and access methods, just as model search must find a good model in the large space of potential model families and their configurations.

There are some notable differences between these two problems, however, leading to a novel set of challenges to address in the context of model search. First, unlike traditional database queries, due to the inherent uncertainty in predictive models learned from finite datasets, the model search process does not yield a unique answer. Hence, model search must focus on both quality and efficiency (traditional query planning need only consider efficiency), and must trade off between the two objectives when they conflict. Second, the search space for models is not endowed with well-defined algebraic properties, as it consists of possibly unrelated model families, each with its own access patterns and hyperparameters. Third, evaluating a candidate model is expensive and in this context involves learning the parameters of a statistical model. Learning the parameters of a single model can involve upwards of hundreds of passes over the input data, and there exist few heuristics to estimate the effectiveness of a model before this costly training process.

Now, we turn our attention to scalable model search strategies.

2.4 Baseline Model Search

The conventional approach to model search is sequential grid search [4, 31, 39], which divides the hyperparameter space into a regular grid and trains models at these grid points. For instance, consider a single ML model family with two hyperparameters. If the two hyperparameters are in the range 0 to 100 and the budget allows for 25 total model configurations, then each hyperparameter will be sampled at $(0, 25, 50, 75, 100)$.

In the cluster setting, in current systems users decide how to parallelize the execution of grid search. In cases where data is small enough to fit in memory of a single machine, often each machine is responsible for trying a set of grid points on a copy of the training data—we refer to this situation as *model parallel*. However, if data is too large to fit in memory on a single machine, *data parallel* strategies are employed, where partial statistics of a model update are computed on worker machines, communicated back to a master machine, combined to produce an updated model, and sent back out to each worker. In either setting, an important factor that impacts job completion time is the size of the cluster to use. This choice of *execution strategy* is typically made by the user or developer of the system, and is not dynamically made by the search procedure. Our intuition is that when data no longer fits in memory on a single node, it makes sense to operate in the data parallel setting. We validate this intuition in Section 4.

Sequential grid search has several shortcomings. First, it is not adaptive—the search plan is statically determined a priori and intermediate results do not inform subsequent model fittings. Second, the curse of dimensionality limits the usefulness of this method in high dimensional hyperparameter spaces. Third, grid points may not represent a good approximation of global minima—true global minima may be hidden between grid points, particularly in the case of a very coarse grid. Nonetheless, sequential grid search is commonly used in practice, and is a natural baseline against which we compare our system.

Algorithm 1 lists this grid search strategy. In this example, the budget is the total number of models to train.

```
input  : LabeledData, ModelSpace, Budget
output: BestModel
bestModel ← ∅;
grid ← gridPoints(ModelSpace, Budget);
while Budget > 0 do
    proposal ← nextPoint(grid);
    model ← train(proposal, LabeledData,);
    if quality(model) > quality(bestModel) then
        bestModel ← model;
    end
    Budget ← Budget − 1;
end
return bestModel;
```

Algorithm 1: A baseline model search procedure with conventional grid search. The function "gridPoints" returns a coarse grid over the dimensions of model space, where the total number of grid points is determined by the budget.

2.5 TuPAQ Model Search

As discussed in the previous section, grid search is a suboptimal search method despite its popularity. Moreover, from a systems perspective, a naive implementation of Algorithm 1 would have additional drawbacks beyond those of grid search. In particular, it would ignore several physical optimizations such as proper use of batching and optimal use of cluster resources.

In contrast, we propose the TuPAQ algorithm, described in Algorithm 2, to address these shortcomings via logical and physical optimizations. The TuPAQ algorithm automatically determines an *ideal execution strategy* based on properties of the data and the budget (Line 1). The TuPAQ algorithm also allows for more *sophisticated hyperparameter tuning strategies*. Line 7 shows that our model search procedure can now use training history as input. Here, "proposeModel" can be an arbitrary model search algorithm. Second, our algorithm performs *batching* to train multiple models simultaneously (Line 8). Third, our algorithm deploys *bandit resource allocation via runtime inspection* to make on-the-fly decisions. Specifically, the algorithm compares the quality of the models currently being trained with historical information about the training process, and determines which of the current models should be trained further (Line 10).

These four optimizations are discussed in detail in Section 3, with a focus on the design space for each of them. In Section 4, we evaluate the options in this design space experimentally, and then in Section 5 compare the baseline algorithm (Algorithm 1) to TuPAQ running with good choices for execution strategy, hyperparameter tuning method, batch size, and bandit allocation criterion, i.e., choices informed by the results of Section 4.

```
input  : LabeledData, ModelSpace, Budget, BatchSize
output: BestModel
(bestModel, history, activeProposals) ← ∅;
searcher.setSearchSpace(ModelSpace);
executor.determineExecStrategy(LabeledData, Budget)
    // Execution Strategy
while Budget > 0 do
    freeSlots ← BatchSize − size(activeProposals);
    activeProposals ← activeProposals ∪
        searcher.proposeModels(freeSlots, history)
        // Hyperparameter Tuning
    (models, budgetUsed) ←
        executor.trainPartial(activeProposals, LabeledData)
        // Batching
    (finishedModels, activeProposals) ←
        banditAllocation(models, history) // Bandits
    history ← history ∪ models;
    Budget ← Budget − budgetUsed;
end
bestModel ← getBestFromHistory(history);
return (bestModel);
```

Algorithm 2: The planning procedure used by TuPAQ. 'executor' and 'searcher' represent handles to execution engine and search procedure, respectively. 'trainPartial' returns a partially trained model, and the bandit allocation strategy decides which models to keep training. The algorithm returns the best model it has seen once the budget is exhausted.

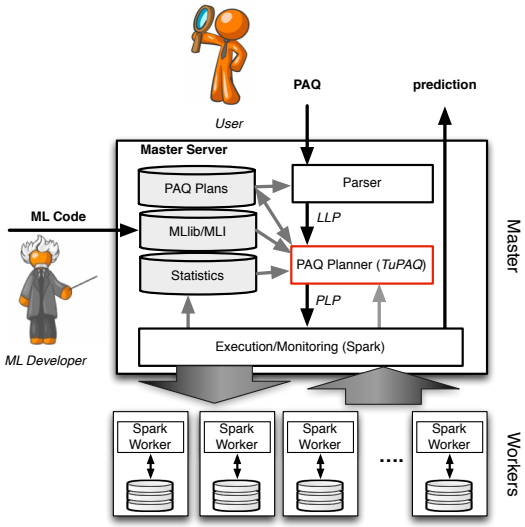

Figure 2: The TuPAQ planner is a critical component of the MLbase [29] system for simplified large scale machine learning. TuPAQ interacts with a distributed run-time and existing machine learning algorithms to efficiently find a model which yields high quality predictions.

Before exploring the design space for the TuPAQ algorithm, we first describe the architecture of TuPAQ and how it fits into a larger system to support large scale machine learning.

2.6 TuPAQ and the MLbase Architecture

TuPAQ lies at the heart of MLbase [29], a novel system designed to simplify the process of implementing and using scalable machine learning techniques. By giving users a declarative interface for machine learning tasks, the problem of hyperparameter tuning and feature selection can be pushed down into the system. The architecture of this system is shown in Figure 2.

At the center of the system, an optimizer or planner must be able to quickly identify a suitable model for supporting predictive queries. In the original presentation of MLbase, grid search was suggested as an implementation for the planner. However, while prototyping the system, it became clear that a great deal of time was spent in this component, so optimizing it was a crucial place to focus. Importantly, the declarative nature of MLbase allows for arbitrary procedures to be used at this level of the system, a choice that may not have been available otherwise. Our instantiation of this component is TuPAQ. The entire system is built upon Apache Spark, a cluster compute system designed for iterative computing [48], and we leverage MLlib and other components present in Apache Spark, as well as MLI [44]. Of course, TuPAQ itself has several moving parts, which we will now explore.

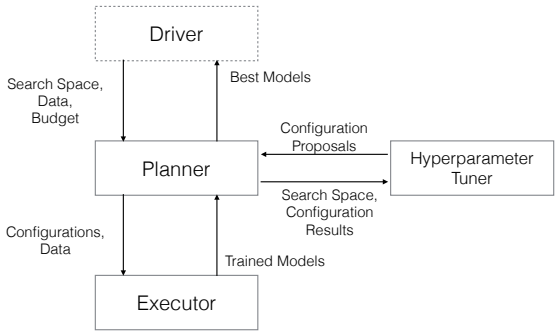

Figure 3: TuPAQ is composed of several components, each of which has a standard interface and may have multiple implementations.

Name	Type	Scale
IntegerP	Continuous	Uniform
FloatP	Continuous	Uniform/Log
DiscreteP	Discrete	Uniform
ChoiceP	Set[Parameter]	Uniform
SequenceP	Seq[Parameter]	N/A

Table 1: Users define parameter spaces by composing potentially nested parameters of various types.

2.6.1 TuPAQ Architecture

TuPAQ is designed according to the principles of modularization and reuse. In particular, it consists of several components which adhere to an abstract interface. The interaction of these components is illustrated in Figure 3.

The *driver* is some higher level component that calls into the *planner*. The driver is responsible for providing a model search space, and a budget. The planner passes this information on to the *hyperparameter tuner* whose job is to produce new model configurations to try. The planner passes these configurations to an *executor* which is responsible for actually training models given a handle to the user's dataset. The executor determines an appropriate execution strategy back to the planner, which in turn relays these to the hyperparameter tuner and polls it for new configurations to try. After the budget is exhausted, the planner returns the best models it has seen to the driver.

Users define their search space using an extensible API. Table 1 lists the data structures used to represent parameters, and Figure 4 shows an example search space defined using the API.

While we currently expect users to specify their search space, it is possible that developers can inform the system about reasonable options for hyperparameter settings (e.g. step size should be > 0.0),

```
val searchSpace = ChoiceP("family",
  Set(
    SeqP("RandomForest",
      Seq(
        DiscreteP("loss", Seq("gini","entr")),
        FloatP("fracFeatures", 0.0, 1.0),
        IntP("minSplit", 1, 20),
        IntP("maxDepth", 2, 30)
      )
    ),
    SeqP("LogisticRegression",
      Seq(
        FloatP("Reg", 1e-6, 1e6, scale=Log),
        FloatP("Step", 1e-6, 1e6, scale=Log)
      )
    )
  )
```

Figure 4: Using the small set of parameters from Table 1, users are able to construct rich parameter search spaces. Here, the user supplies parameter ranges to try for Random Forests or Logistic Regression models.

leaving the user oblivious to the methods and parameters used to fit their models.

Now that we have explored the architecture of the system, we turn our attention to its design space.

3. TUPAQ Design Choices

In this section, we examine the design choices available for the TUPAQ model search procedure. We target algorithms that run on tens to thousands of nodes in commodity computing clusters, and training datasets that fit comfortably into cluster memory—on the order of tens of gigabytes to terabytes. Training a single model to convergence on such a cluster is expected to require tens to hundreds of passes through the training data, and may take on the order of minutes. With a multi-terabyte dataset, performing a grid search involving even just 100 model configurations each with a budget of 100 scans of the training data could take hours to days of processing time, even assuming that the algorithm runs at memory speed. Hence, in this regime the baseline model search procedure is tremendously costly.

We now ask how the system and algorithms presented in Section 2 might be optimized to support fast, high quality search. In the remainder of this section, we present the following optimization four optimizations that in concert provide TUPAQ with an order-of-magnitude gain in performance over the baseline approach.

3.1 Cost-based Execution Strategy Selection

When data is too big to fit on a single node, data parallelism provides scalability that is linear with respect to cluster size. However, it presents its own set of drawbacks. In particular, it requires more coordination among workers than the model parallel strategy. Further, there are no formal procedures to estimate the optimal number of nodes to provision, and available guidance consists only of conventional wisdom and rules of thumb.

In this section, we present a simple cost-based model of estimated execution time in the data parallel setting that provides guidance for optimal cluster sizing based on the workload, hardware, and cluster management system used to execute parallel jobs.

3.1.1 Estimating Job Latency

Data parallel execution is the natural choice for model search if the CPU requirements of the model training become excessive or the data required to train a single model no longer fit in memory. Assuming we are in a data parallel setting, we must choose our cluster size to be large enough to take advantage of parallel data processing, but not so large that the speedup provided by parallelism is dominated by increased coordination (and therefore network bandwidth usage) between machines. Amdahl's law [9] provides an upper bound on the speedup we can hope to achieve: the maximum speedup due to parallelization is inversely proportional to the percentage of a job that is sequential. Commonly used cluster computing frameworks rely on a centralized master to manage distributed jobs, which increases the serial portion of execution time even for tasks that are embarrassingly parallel.

To capture these effects, we propose the following cost-based model of cluster job execution time:

$$t_{job}(w) = k_0 + k_1 w + \mathbb{I}_{cpu} k_2 \frac{c}{w} + \mathbb{I}_{mem} k_3 \frac{m}{w} + \mathbb{I}_{net} k_4 \frac{b}{w} \qquad (1)$$

Equation 1 describes our estimator, a function of w, the number of worker nodes in the cluster, which encapsulates several important insights. First, we explicitly separate the requirements of the job from the capacity of the cluster hardware. The constants k_2, k_3, and k_4 capture the (average) compute, memory, and network bandwidth available on each node in the cluster independently of c, m, and b, which are functions that describe the total compute, memory, and network requirements of the task given properties of the data in terms of FLOPS required, or bytes to be read from memory or across the network. While these functions must be defined ahead of time by an algorithm developer (or, alternatively, by sampling empirical job behavior), this makes our model easy to use in practice: users simply specify the hardware specifications of their cluster (readily available from cloud computing providers like Amazon EC2) and the resource requirements of the algorithm to be run. We will show an example of how these parameters may be estimated in Section 4.1.

In addition, we use a roofline [47] inspired model of the tradeoff between job-specific memory and compute requirements. That is, we ask ML algorithm developers to specify whether their job is memory, network or compute-bound, and assume that only the most constrained resource will affect overall job latency. The model captures this assumption with three indicator variables: $\mathbb{I}_{cpu}$, $\mathbb{I}_{mem}$, and $\mathbb{I}_{net}$. We estimate the execution time of memory-bound jobs based on the total memory requirements of the job divided by the number of parallel workers, and the execution time of compute-bound jobs based on the total compute requirements of the job divided by the number of parallel workers.

Finally, our model makes the assumption that overheads associated with the cluster compute framework itself can be completely separated from the job execution time. That is, all framework-specific overheads are captured in the variables k_0 and k_1. k_0 describes one-time static costs of using the framework to run a job, for example code compilation and DAG optimization, or serialization of code and (intermediate) results. k_1 describes framework costs that scale with the number of workers running the job, primarily the overhead of making scheduling decisions and associated queueing delays. Both of these parameters are framework-dependent.

Intuitively, we estimate that a data-parallel job will take time that is dictated by fixed cluster overheads, some marginal additional time for each node in a cluster, and the greatest of the parallel time devoted to moving data through the CPU, memory, or the network.

While this model is clearly a simplification of the nuances of cluster job performance, we have found that it estimates execution time for data-parallel iterative machine learning jobs reasonably well, making it useful for TUPAQ's model search task. We now describe the use of our model to determine the size of clusters, and will evaluate its use in Section 4.

3.1.2 Right Sizing the Cluster

The primary purpose of this estimator is to allow TUPAQ to determine a good choice for the number of nodes it should use to train a model in a data parallel environment. This optimum can be computed as the number of workers that should lead to the lowest possible job execution time, $w^* = \arg\min_w t_{job}(w)$. Since this model is simple and linear, we can easily find a closed-form solution by differentiating with respect to w and finding the roots:

$$w^* = \sqrt{\frac{\mathbb{I}_{cpu} k_2 c + \mathbb{I}_{mem} k_3 m + \mathbb{I}_{net} k_4 b}{k_1}}. \quad (2)$$

The solution in Equation 2 corresponds to intuitions about the tradeoffs of cluster computing. If the overheads of distributing the job to more workers (k_1) are as large as the resource requirements of the job ($\mathbb{I}_{cpu} k_2 c + \mathbb{I}_{mem} k_3 m + \mathbb{I}_{net} k_4 b$), then the model tells us not to distribute ($w^* = 1$).

While a simple application of Amdahl's law would prescribe an infinite number of workers in the presence of unlimited budget, the linear penalty applied to workers via k_1 in our model causes there to be a unique minimum in the above solution. To our knowledge, this type of penalty has not been explicitly modeled elsewhere in the cluster computing literature.

We will evaluate the effectiveness of this estimator in Section 4 but for now turn our attention to better search algorithms.

3.2 Advanced Hyperparameter Tuning

We can view hyperparameter tuning as an optimization problem over a potentially non-smooth, non-convex function in high dimensional space. This function is expensive to evaluate and we have no closed form expression for it (hence, we cannot compute derivatives). Although grid search remains the standard solution to this problem, various alternatives have been proposed for the general problem of derivative-free optimization, some of which are particularly well-suited for hyperparameter tuning. Each of these methods provides an opportunity to speed up TUPAQ's model search time, and in this section we provide a brief survey of the most commonly used methods.

Traditional methods for derivative-free optimization include grid search (the baseline choice for model search) as well as random search, Powell's method [40], and the Nelder-Mead method [37]. Given a hyperparameter space, grid search selects evenly spaced points (in linear or log space) from this space, while random search samples points uniformly at random from this space. Powell's method can be seen as a derivative-free analog to coordinate descent, while the Nelder-Mead method can be roughly interpreted as a derivative-free analog to gradient descent.

Both Powell's method and the Nelder-Mead method expect unconstrained search spaces, but function evaluations can be modified to severely penalize exploring out of the search space. However, both methods require some degree of smoothness in the hyperparameter space to work well, and can easily get stuck in local minima. Additionally, neither method lends itself well to categorical hyperparameters, since the function space is modeled as continuous. For these reasons, we are unsurprised that they are inappropriate methods to use in the model search problem where optimization is done over an unknown function that is likely non-smooth and not convex.

More recently, various methods specifically for hyperparameter tuning have been recently introduced in the ML community, including Tree-based Parzen Estimators (HyperOpt) [13], Sequential Model-based Algorithm Configuration (Auto-WEKA) [45] and Gaussian Process based methods, e.g., Spearmint [43]. These algorithms all share the property that they can search over spaces which are nested (e.g. multiple model families) and accept categorical hyperparameters (e.g. regularization method). HyperOpt begins with a random search and then probabilistically samples from points with more promising minima, Auto-WEKA builds a Random Forest model from observed hyperparameter results, and Spearmint implements a Bayesian method based on Gaussian Processes. In Section 4 we evaluate each method on several datasets to determine which method is most suitable for model search.

3.3 Bandit Resource Allocation

Models are not all created equal. In the context of model search, typically only a fraction of the models are of high-quality, with many of the remaining models performing drastically worse. Under certain assumptions, allocating resources among different model configurations can be naturally framed as a multi-armed bandit problem [16]. Indeed, assume we are given a *fixed set* of k model configurations to evaluate, as in the case of grid or random search, along with a fixed budget B. Then, each model can be viewed as an 'arm' and the model search problem can be cast as a k-armed bandit problem with T rounds. At each round we perform a single iteration of a particular model configuration, and return a reward indicating the quality of the updated model, e.g., validation accuracy. In such settings, multi-armed bandit algorithms can be used to determine a scheduling policy to efficiently allocate resources across the k model configurations. Typically, these algorithms keep a running score for each of the k arms, and at each iteration choose an arm as a function of the current scores.

```
input  : currentModels, history
output: finishedModels, activeProposals
(finishedModels, activeProposals) ← ∅[];
bestModel ← getBestFromHistory(history);
for m in currentModels do
    if fullyTrained(m) then
        finishedModels ← finishedModels ∪ m;
    else if quality(m) *(1+ε) > quality(bestModel) then
        activeProposals ← activeProposals ∪ m;
    end
end
return (finishedModels, activeProposals);
```

Algorithm 3: The bandit allocation strategy used by TUPAQ.

Our setting differs from this standard setting in two crucial ways. First, several of our search algorithms select model configurations to evaluate in an iterative fashion, so we do not have advanced access to a fixed set of k model configurations. Second, in addition to efficiently allocating resources, we aim to return a reasonable result to a user as quickly as possible, and hence there is a benefit to finish training promising model configurations once they have been identified.

Our bandit selection strategy is a variant of the action elimination algorithm of [19], and to our knowledge this is the first time this algorithm has been applied to hyperparameter tuning. Our strategy is detailed in Algorithm 3. This strategy preemptively prunes models that fail to show promise of converging. For each model (or batch of models), we first allocate a fixed number of iterations for training; in Algorithm 2 the trainPartial() function trains each model for PartialIters iterations. Partially trained models are fed into the bandit allocation algorithm, which determines whether to train the model to completion by comparing the quality of these models to the quality of the best model that has been trained to date. This procedure acts as a filter on whether models should be trained any further. Moreover, this comparison is performed using

a slack factor of $(1+\varepsilon)$; in our experiments we set $\varepsilon = .5$ and thus continue to train all models with quality within 50% of the best quality model observed so far. We chose this value for ε because it provided a good tradeoff between maintaining model quality and speeding up search in our experiments. The algorithm stops allocating further resources to models that fail this test, as well as to models that have already been trained to completion.

3.4 Batching

Batching is a natural system optimization in the context of training machine learning models, with applications for cross validation and ensembling [17, 32], however, it has not previously been applied to model search. For model search, we note that the access pattern over the training set is identical for many machine learning algorithms. Specifically, each algorithm takes multiple passes over the input data and updates some intermediate state (e.g., model weights) during each pass. As a result, it is possible to batch together the training of multiple models effectively sharing scans across multiple model estimations. In a data parallel distributed environment, this has several advantages:

1. Better CPU utilization by reducing wasted cycles.
2. Amortized task launching overhead across several models at once.
3. Amortized network latency across several models at once.

Ultimately, these three advantages lead to a significant reduction in learning time. We take advantage of this optimization in line 8 of Algorithm 2.

For concreteness and simplicity, we will focus on one algorithm—logistic regression trained via gradient descent—for the remainder of this section, but we note that these techniques apply to many model families and learning algorithms.

3.4.1 Logistic Regression

Logistic Regression is a widely used machine learning model for binary classification. The procedure estimates a set of model parameters, $w \in \mathbb{R}^d$, given a set of data features $X \in \mathbb{R}^{n \times d}$, and binary labels $y \in \{0,1\}^n$. The optimal model $w^* \in \mathbb{R}^d$ can be found by minimizing the negative likelihood function, $f(w) = -\log p(X|w)$. Taking the gradient of the negative log likelihood, we have:

$$\nabla f = \sum_{i=1}^{n} \left[\left(\sigma(w^\top x_i) - y_i \right) x_i \right], \tag{3}$$

where σ is the logistic function. The gradient descent algorithm (Algorithm 4) must evaluate this gradient function for all input data points, a task which can be easily performed in a data parallel fashion. Similarly, minibatch Stochastic Gradient Descent (SGD) has an identical access pattern and can be optimized in the same way by working with contiguous subsets of the input data on each partition.

```
input  : X, LearningRate, MaxIterations
output: Model
i ← 0;
Initialize Model;
while i < MaxIterations do
    read current;
    Model ← Model - LearningRate * Gradient(Model, X);
    i ← i + 1;
end
```

Algorithm 4: Pseudocode for convex optimization via gradient descent.

The above formulation represents the computation of the gradient by taking a single point and single model at a time. We can naturally extend this to multiple models simultaneously if we represent our models as a matrix $W \in \mathbb{R}^{d \times k}$, where k is the number of models we want to train simultaneously, i.e.,

$$\nabla f = \left[X^\top \left(\sigma(XW) - y \right) \right]. \tag{4}$$

In effect, by computing the gradient of several models simultaneously, we are able to compute several model updates simultaneously. By scaling each of these updates by the appropriate learning rate (an element-wise operation on the model), the procedure supports several learning rates.

This operation can be easily parallelized across data items with each worker in a distributed system computing the portion of the gradient for the data that it stores locally. Specifically, the portion of the gradient that is derived from the set of local data is computed independently at each machine, and these gradients are simply summed at the end of an iteration. The size of the partial gradients (in this case $O(d \times k)$) is much smaller than the actual data (which is $O(n \times d)$), so overheads of transferring these over the network is relatively small. For large datasets, the time spent performing this operation is almost completely determined by the cost of performing two matrix multiplications—the input to the σ function which takes $O(ndk)$ operations and requires a scan of the input data as well as the final multiply by $X^\top$ which also takes $O(ndk)$ operations and requires a scan of the data. This formulation allows us to leverage high performance linear algebra libraries that implement BLAS [33]—these libraries are tailored to execute exactly dense linear algebra operations as efficiently as possible and are automatically tuned to the architecture we are running on via [46].

3.4.2 Machine Balance

One obvious question the reader may ask is why implementing these algorithms via matrix-multiplication should offer speedup over vector/vector versions of the algorithms. After all, the runtime complexities of both algorithms are identical. However, modern x86 machines have been shown to have processor cores that significantly outperform their ability to read data from main memory [35]. In particular, on a typical x86 machine, the hardware is capable of reading 0.45B doubles/s from main memory per core, while the hardware is capable of executing 6.8B FLOPS in the same amount of time [34]. Specifically, on the machines we tested (Amazon `c3.8xlarge` EC2 instances), LINPACK reported peak GFLOPS of 110 GFLOPS/s when running on all cores, while the STREAM benchmark reported 60GB/s of throughput across 16 physical cores. This equates to a machine balance of approximately 15 FLOPS per double precision floating point number read from main memory if the machine is using both all available FLOPs and all available memory bandwidth solely for its core computation. This approximate value for the machine balance suggests an opportunity for optimization by reducing unused resources, i.e., wasted cycles. By performing more computation for every number read from memory, we can reduce this resource gap.

The Roofline model [47] offers a more formal way to study this effect. According to the model, total throughput of an algorithm is bounded by the smaller of 1) peak floating point performance of the machine, and 2) memory bandwidth times operational intensity of the algorithm, where operational intensity is a function of the number of FLOPs performed per byte read from memory. That is, for an efficiently implemented algorithm, the bottleneck is either I/O bandwidth from memory or CPU FLOPs.

Analysis of the unbatched gradient descent algorithm reveals that the number of FLOPs required per byte is quite small—just over 2 flops per number read from memory—a multiply and an

add—and since we represent our data as double-precision floating point numbers, this equates to 1/2 FLOP per byte. Batching allows us to move "up the roofline" by increasing algorithmic complexity by a factor of k, our batch size. The exact setting of k that achieves balance (and maximizes throughput) is hardware dependent, but we show in Section 5 that on modern machines, $k = 10$ is a reasonable choice.

3.4.3 Amortized Overheads

As discussed earlier, the cluster computing framework introduces overheads with each new job that is run. Batching multiple hyperparameter settings into the same Spark job allows us to share a single scan of the data across several hyperparameters and amortize these overheads to decrease the effective overhead per model trained.

4. Design Space Evaluation

Now that we have laid out the possible optimizations available to TUPAQ, we investigate the potential speedup offered by each in turn. In all experiments we split our base datasets into 70% training, 20% validation, and 10% testing. In all cases, models were fit to minimize classification error on the training set, while model search occurs based on classification error on the validation set (validation error).[1] We only report validation error numbers here, but test error was similar. TUPAQ is capable of optimizing for arbitrary performance metrics as long as they can be computed mid-flight, and extends to other supervised learning scenarios.

4.1 Execution Strategy Selection

To demonstrate the value of our estimator, we first investigated the consequences of choosing an inappropriate cluster size. We examined the overheads of data-parallel execution on a 16-node cluster running two jobs, one that operates on a 100 GB dataset and one that operates on only 1 GB of data. In the larger job (more appropriate for a 16-node cluster), only a small fraction ($< 10\%$) of execution was spent on cluster overheads. In the smaller job, however, virtually all ($> 90\%$) of the execution time was consumed by task scheduling and serialization/deserialization overheads.

Next, we evaluated the usefulness of our estimator by instrumenting both a high-quality multi-core machine learning framework [3] and our Scala/Spark codebase and executing SVM model training runs at data scales ranging from 100 MB to 100 GB and cluster sizes ranging from 1 to 64 Amazon `c3.8xlarge` EC2 instances (each having 60GB of RAM). In this case, the goal of the model is to predict the execution time for a single iteration of linear SVM for binary classification via gradient descent. As such, the input memory requirement (m in our estimator) is the size of the dataset in memory (the product of its dimensions and the size of a double-precision floating point number). The compute requirement, c, can be similarly defined. Because the CPU and memory requirements are the same, the task is memory-bound ($\mathbb{I}_{mem} = 1$) on this hardware. The network requirements are much smaller than the CPU or memory requirements, but are proportional to the number of columns in the input dataset per machine.

Given these job requirements, we estimated the parameters of our model (Equation 1) by fitting a regression to the observed data, and found that the model explains a significant portion of the variance in the training data (cross-validated $R^2 = 80.1\%$). We followed a standard 5-fold cross validation procedure where 5 separate models were trained on 80% of the training data and results are presented on the remaining held-out 20%. Figure 5 demonstrates the tightness of this fit visually, showing the estimated execution times overlaid on the times observed in the experimental data. In addition, the coefficients fit by regression matched our knowledge about the hardware the jobs were run on and our experience with the overheads associated with Apache Spark's scheduler. The k_0 term maps to 175ms of task overhead to launch a Spark job, the k_1 term indicates an additional 6ms of overhead per cluster node (penalizing very large clusters), while the k_3 term maps to 30GB/s of memory bandwidth on this memory-constrained job.

[1] While we have thus far discussed model quality, for the remainder of the paper we report validation error, i.e., the inverse of quality, because it is more commonly reported in practice.

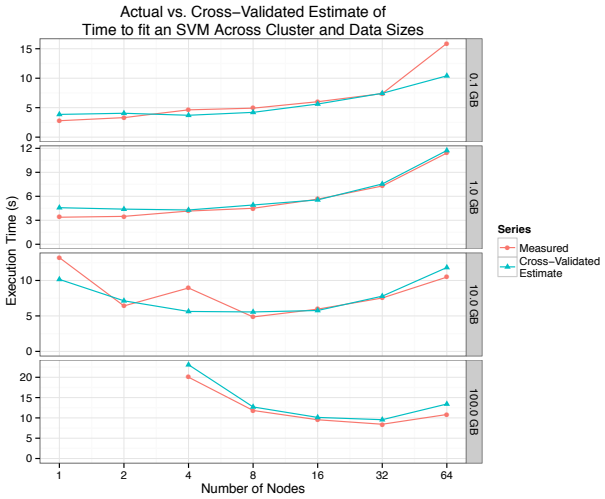

Figure 5: Measured time to run SVM for 20 iterations on small to larger datasets at various cluster sizes. In red, the observed time to run a particular configuration. In blue, cross-validated time predicted by our estimator. One and two machine configurations for the 100.0GB case not shown because the datasets do not fit in cluster memory.

	100 MB	1 GB	10 GB	100 GB
Vowpal Wabbit	0.4	2.2	19.1	1445.1
MLlib	2.9	3.5	38.4	268.8
MLLib Cluster Size	1	2	8	32

Table 2: A comparison of total machine×seconds spent on training an SVM in an optimized single-node framework (Vowpal Wabbit) and a data-parallel framework (MLlib). Once the problem can no longer be solved in-memory on a single node, it is better to use the data-parallel framework. The MLlib runs are the best times for the cluster sizes we tried – from 1 to 32 nodes.

Finally, we used our model to validate the intuition that model parallel execution only makes sense while the data fits in memory on a single node. Table 2 reports execution times (in machines × seconds) of the SVM model training runs described above. Note that the per machine execution time for Vowpal Wabbit is significantly lower than MLLib for datasets that fit comfortably in memory, indicating that a model parallel strategy is reasonable in this regime. However, once data does not fit into a single node's memory (i.e., 100GB datasets), Vowpal Wabbit must read from disk and data-parallel execution strategies on the cluster make more sense.

We use cluster sizes advised by this model for the remainder of the paper, rounding to the nearest power of two for simplicity of interpretation.

4.2 Hyperparameter Tuning

We evaluated the strategies for model search with a variable model fitting budget on five representative datasets for binary classification taken from the UCI Machine Learning Repository [10]. The

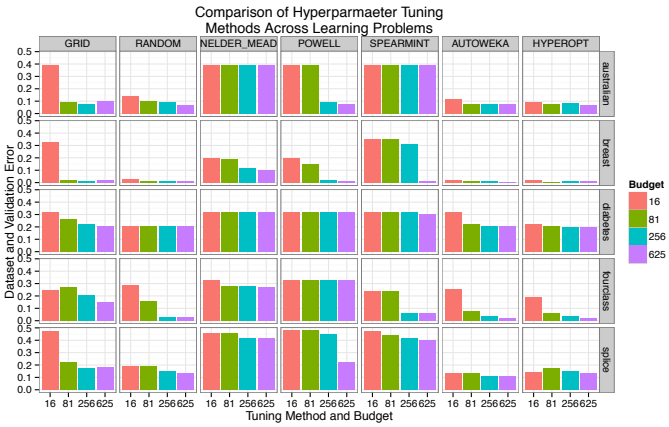

Figure 6: Hyperparameter tuning methods were compared across several datasets with a variable number of function evaluations. Classification error on a validation dataset is shown for each combination. HyperOpt and Auto-WEKA provide state of the art results, while random search performs best of the classic methods.

model search task involved tuning four hyperparameters—learning rate, L2 regularization parameter, size of a random projection matrix, and noise associated with the random feature matrix. The random features are constructed according to the procedure outlined in [41]. To accommodate the linear scale-up that comes with adding random features, we down sample the number of data points for each model training by the same proportion.

Our ranges for these hyperparameters were learning rate $\in (10^{-3}, 10^{1})$, regularization $\in (10^{-4}, 10^{2})$, projection size $\in (1 \times d, 10 \times d)$, and noise $\in (10^{-4}, 10^{2})$.

We evaluated seven tuning methods: grid search, random search, Powell's method, the Nelder-Mead method, Auto-WEKA, HyperOpt, and Spearmint.

Each dataset was processed with each search method with a varying number of model fittings, chosen to align well with a regular grid of n^4 points where we vary n from 2 to 5. This restriction on a regular grid is only necessary for grid search but included for comparability.

Results of the hyperparameter tuning experiments are presented in Figure 6. Each tile represents a different dataset/tuning method combination. Each bar within the tile represents a different budget in terms of models trained. The height of each bar represents classification error on the validation dataset.

With this experiment, we are looking for tuning methods that converge to good models in as small a budget as possible. Of all methods tried, HyperOpt and Auto-WEKA tend to achieve this criteria best, but random search is not far behind. This result has been noted by others [14], but intuitively this is because, absent other information, routines for hyperparameter tuning can do no better than random sampling at initialization. As with traditional statistical methods, one must collect a reasonable number of samples to get a good idea of the shape of "model space." In our regime, each model is expensive to compute, so the resources to try many models are limited. We chose to integrate HyperOpt into the larger experiments because it performed slightly better than Auto-WEKA. Our architecture fully supports additional search methods, and we expect to implement additional methods in our system over time.

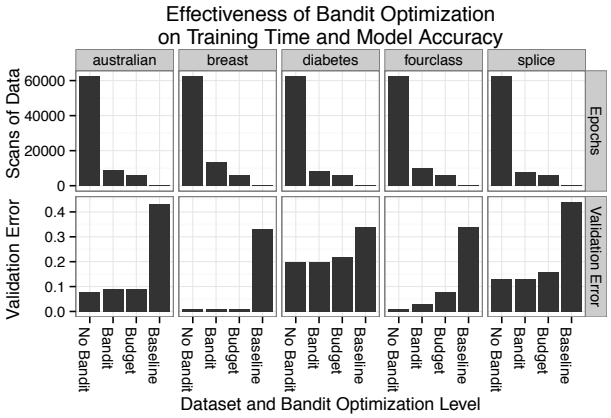

Figure 7: Here we show the effects of bandit resource allocation on trained model performance. Model search completed in an average of 84% fewer passes over the training data than without bandit allocation. Except in one case, validation error is nearly indistinguishable vs. the case where we do not employ the bandit strategy.

4.3 Bandit Resource Allocation

We evaluated the TUPAQ bandit resource allocation scheme on the same datasets with random search and 625 total function evaluations—the same as the maximum budget in the search experiments. The key question to answer here was whether we could identify and terminate poorly performing models early in the training process without significantly affecting overall search quality.

In Figure 7 we illustrate the effect that the TUPAQ bandit strategy has on validation error as well as on the number of total scans of the input dataset. Models were allocated 100 iterations to converge on the correct answer. After the first 10 iterations, models that were not within 50% of the classification error of the best model trained so far were preemptively terminated. A large percentage of models that show little or no promise of converging to a reasonable validation error were eliminated.

In the figure, the top set of bars represents the number of scans of the training data at the end of the entire search process. The bottom set of bars represent the validation error achieved at the end of the search procedure. The four scenarios evaluated—No Bandit, Bandit, Budget, and Baseline—represent the results of the search with no bandit allocation procedure (that is, each model is trained to completion), the algorithm the bandit allocation procedure enabled, a procedure with a limited budget (that is, exactly 10 scans per model evaluated), and the baseline error rate for each dataset. The Baseline scenario is a classifier that simply picks the most common class for each dataset, which is a natural naive baseline which can be trained by only looking at the training labels. Any reasonable machine learning method should improve on this result.

There was an 84% decrease in total epochs across these five datasets, and the validation error was roughly comparable to the unoptimized strategy. On average, this method achieves 97% reduction in model error vs. not stopping early when compared with validation error of a model which deterministically picks the most frequent class. By comparison, the Budget strategy only achieves an 89% reduction in model error with a 90% decrease in total epochs across these datasets, with higher variance. This relatively simple resource allocation method presents opportunities for dramatic reductions in runtime.

These results represent one point in the tradeoff space between training everything to full budget and training every model with only a very limited set of resources. While this heuristic seemed to work well for the datasets mentioned in TUPAQ, it has already

Batch Size	D		
	100	1000	10000
1	826.44	599.60	553.59
2	1521.23	1214.37	701.07
5	2411.53	3037.97	992.01
8	5557.69	3502.79	1243.79
10	7148.53	4216.44	1769.12
15	7874.01	6260.14	2485.15
20	11881.18	8248.36	2445.98

(a) Models trained per hour for varying batch sizes and model complexity. Data sizes ranged from 750MB (D=100) to 75GB (D=10000).

Batch Size	D		
	100	1000	10000
1	1.00	1.00	1.00
2	1.84	2.02	1.26
5	2.91	5.06	1.79
8	6.72	5.84	2.24
10	8.64	7.03	3.19
15	9.52	10.44	4.48
20	14.37	13.75	4.41

(b) Speedup factor vs fastest sequential unbatched method for varying batch size and model complexity.

Figure 8: Effect of batching is examined on 16 nodes with a synthetic dataset. Speedups diminish but remain significant as models increase in complexity.

inspired further study [27] and a more principled approach may be present in future incarnations of the system.

4.4 Batching

To evaluate the batching optimization, we used a synthetic dataset of $1,000,000$ data points in various dimensionality. To illustrate the effects of amortizing scheduler overheads vs. achieving machine balance, these datasets vary in size between 750MB and 75GB.

We trained these models on a 16-node cluster of `c3.8xlarge` nodes on Amazon EC2, running Apache Spark 1.1.0. We trained a logistic regression model on these data points via gradient descent with no batching (batch size = 1) and batching up to 20 models at once. We implemented both a naive version of this optimization—with while loops evaluating Equation 3 over each model in each task—as well as a more sophisticated version of this model which makes BLAS calls to perform the computation described in equation 4. For the batching experiments, we ran each algorithm for 10 iterations over the input data.

In Figure 8 we show the total throughput of the system in terms of models trained per hour varying the batch size and the model complexity. For models trained on the smaller dataset, we see the total number of models per hour can increase by up to a factor of 15 for large batch sizes. This effect should not be surprising, as the actual time spent computing is on the order of milliseconds and virtually all the time goes to scheduling task execution. In its current implementation, due to these scheduling overheads, this implementation of the algorithm under Spark will not outperform a single machine implementation for a dataset this small. As discussed earlier, data sets this small should likely be trained on a single node. We discuss an alternative execution strategy that would better utilize cluster resources for situations where the input dataset is small in Section 7.

At the other end of the spectrum in terms of data size and model complexity, we see the effects of scheduler delay start to lessen, and we maximize throughput in terms of models per hour at batch size 15. In Figure 9 we compare two different strategies of implementing batching—one via the naive method, and the other

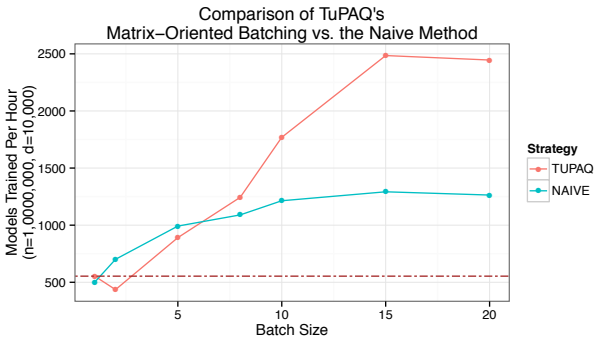

Figure 9: Leveraging high performance linear algebra libraries for batching leads to substantial speedups vs. naive methods. The dotted line shows models per hour via the fastest sequential strategy, and batching yields a 5x improvement.

via the more sophisticated method—computing gradient updates via BLAS matrix multiplication. For small batch sizes, the naive implementation actually performs faster than the BLAS optimized one. The matrix-based implementation easily dominates the naive implementation as batch size increases because the algorithm is slightly more cache efficient and requires only a single pass through the input data. The overall speedup due to batching with matrix multiplication is nearly a factor of 5.

A downside to batching in the context of model search is that the system may gain information by trying models sequentially that could inform subsequent models that is not incorporated in later runs. By fixing our batch size to a relatively small constant (O(10)) we are able to balance this tradeoff.

5. Putting It All Together

Now that we have examined each point in the model search design space individually, we evaluate the end-to-end performance of the TuPAQ procedure and show that the techniques evaluated in the previous section yield a 10x increase in raw throughput (models trained per unit time) while finding models that have as good or higher quality than those found with the baseline approach. We evaluate TuPAQ on very large scale data problems, at cluster sizes ranging from 16 to 128 nodes and datasets ranging from 30GB to over 3TB in size. These sizes represent the size of the actual features the model was trained on.

5.1 Platform Configuration

We evaluated TuPAQ on Linux machines running under Amazon EC2, instance type `c3.8xlarge`. These machines were configured with Redhat Enterprise Linux, Scala 2.10, version 1.9 of the Anaconda python distribution from Continuum Analytics[1], and Apache Spark 1.1.0. Additionally, we made use of Hadoop 1.0.4 configured on local disks as our data store for the large scale experiments. Total runtime would have been similar if we had used a cloud provider's file system infrastructure, because the datasets used fit comfortably into cluster memory and are cached after first use–that is, most of the time in model search goes into making hundreds or thousands of passes over the dataset in memory. Finally, we use MLI as of commit 3e164a2d8c as a basis for TuPAQ. As with any complex system, proper configuration of the platform to execute a given workload is necessary and Apache Spark is no exception. Specifically—choosing a correct BLAS implementation, configuring Spark to use it, and picking the right balance of executor threads per executor process took considerable effort. This configuration is generally applicable to BLAS-heavy, machine learning workloads and shouldn't need to change appreciably given a new

dataset. The complete system involves a Scala codebase built on top of Apache Spark, MLlib, and MLI.

Optimization \ Tuning Method	Grid	Random	HyperOpt
None	104.7	100.5	103.9
Bandits Only	31.3	29.7	50.5
Batching Only	31.3	32.1	31.8
All (TUPAQ)	11.5	10.4	15.8

Figure 10: Learning time in minutes for a 128-configuration budget across various optimization levels for ImageNet data. Unoptimized, sequential execution takes over 100 minutes regardless of search procedure used. Fully optimized execution can be an order of magnitude faster with TUPAQ.

5.2 Experimental Setup and Datasets

We used two datasets with two different learning objectives to evaluate our system at scale. The first dataset is a pre-featurized version of the ImageNet 2010 dataset [12], featurized using a procedure attributed to [18]. This process yields a dataset with $160,000$ features and approximately $1,200,000$ examples, or 1.4 TB of raw image features. In our 16-node experiments we downsample to the first $16,000$ of these features and use 20% of the base dataset for model training, which is approximately 30GB of data. In the 128-node experiments we train on the entire dataset. We explore five hyperparameters here—one parameter for the classifier we train—SVM or logistic regression, as well as learning rate and L2 Regularization parameters for each matching the above experiments. We allot a budget of 128 model fittings to the problem, each with 100 iterations over the training data. The cluster sizes are informed by the estimator presented in Section 3. While this process is not yet fully automated, future iterations of the system will use models similar to those described here to automatically determine the cluster size.

For this dataset, we search for a model capable of discriminating plants from non-plants given these image features. The images are generally in 1000 base classes, but these classes form a hierarchy and thus can be mapped into plant vs. non-plant categories. Baseline error for this modeling task is 14.2%, which is a bit more skewed than the previous examples. Our goal is to reduce validation error as much as possible, but our experience with this particular dataset has put a lower bound on validation error to around 9% accuracy with linear classification models.

The second dataset is a pre-featurized version of the TIMIT Acoustic-Phonetic continuous speech corpus [20], featurized according to the procedure described in [42]—yielding roughly $2,300,000$ examples each having 440 features. While this dataset is quite small, in order to achieve strong performance on this dataset, other researchers have noted that Kernel Methods offer the best performance [25]. Following the process of [41], this involves expanding the feature space of the dataset by nearly two orders of magnitude, yielding a dataset that has $204,800$ features, or approximately 3.4 TB. We explore five hyperparameters here—one parameter describing the distribution family of the random projection matrix—in this case Cauchy or Gaussian, the scale and skew of these distributions, as well as the L2 regularization parameter for this model, which will have a different setting for each distribution.

A necessary precondition to supporting speech-to-text systems, this dataset provides a examples of labeled phonemes, and our challenge is to find a model capable of labeling phonemes given some input audio. Baseline error for this modeling task is 95%, and state-of-the-art performance on this dataset is 35% error [25].

5.3 Optimization Effects

In Figure 10 we can see the effects of batching and bandit allocation on the throughput of the model search process for the ImageNet dataset. Specifically, though it takes nearly 2 hours to fit all 128 models on the 30GB dataset on the 16 node cluster without any optimizations, with the bandit rule and batching turned on, the system takes just 10 minutes to train 128 random search models to completion. This is a 10x speedup in the case of random search and a 7x speedup in the slightly slower case of HyperOpt. HyperOpt takes a bit longer because it does a good job of picking points that do not need to be terminated preemptively by the bandit strategy. That is, more of the models that HyperOpt selects are trained to completion than random search. Accordingly, HyperOpt arrives at a better model than random search given the same training budget.

Search Method	*Search Time (m)*	*Test Error (%)*
Grid (unoptimized)	104.7	11.05
Random (optimized)	10.4	11.41
HyperOpt (optimized)	15.8	10.38

Figure 11: Search time and best achieved model error for training 128 models on the 16-node ImageNet task. Both optimized HyperOpt and Random search perform significantly faster than unoptimized Grid search, while HyperOpt yields the best model for this image classification problem.

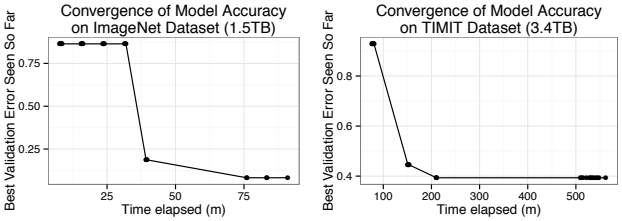

Figure 12: Performance of TUPAQ on large-scale problems using a 128-node cluster. (left) Training a model on a 1.2M $\times$ 160K dataset takes 90 minutes. (right) Training a multiclass phoneme classification model on 3.4 TB of TIMIT features yields accuracy approaching that of state of the art models in 3.5 hours.

Turning our attention to statistical performance illustrated in Figure 11, we can see that on this dataset HyperOpt converges to the best answer in just 15 minutes, while random search converges to within 5% of the best test error achieved by grid search a full order of magnitude faster than the baseline approach.

5.4 Large Scale Speech and Vision

Because we employ data-parallel versions of our learning algorithms, TUPAQ readily scales to multi-terabyte datasets that are an order of magnitude more complicated with respect to the feature space. For the ImageNet experiments, we used the same parameter search settings used with the smaller dataset, but we fixed the budget to 32 models to train. Our results are illustrated in Figure 12(top). Using the fully optimized HyperOpt based search method, we are able to search this space in under 90 minutes, and the method achieves a validation error of 8.2% in that time. In contrast, training all 32 models to completion using sequential grid search would have taken over 8 hours and cost upwards of \$2000.00—an expense we chose not to incur.

Turning our attention to an entirely different application area, we demonstrate the ability of the system to scale to a multi-terabyte, multi-class phoneme classification problem. Here, a multi-class kernel SVM was trained on $2,251,569$ data points with $204,800$ features, in 147 distinct classes. As shown in Figure 12(bottom), the system is capable of getting to a model with 39.5% test error—approaching that of state-of-the-art results in speech-to-text modeling—in just 3.5 hours. For this dataset, train-

ing the entire budget to completion without batching or bandit allocation would have taken 35 hours.

6. Related Work

There has been a recent proliferation of systems designed for low-level, ad-hoc distributed predictive analytics, e.g., Apache Spark [48], GraphLab [22], Stratosphere [8], but few provide tooling for searching over a large space of predictive models.

In terms of system-level optimization, both Kumar et. al. [32] and Canny et. al. [17] discuss batching as an optimization for speeding up machine learning systems. However, Kumar et. al. discuss this technique in the context of automatic feature selection, an important problem but distinct from model search, while Canny et. al. explore this technique in the context of parameter exploration, model tuning, ensemble methods and cross validation. We explore the impact of batching in a distributed setting at greater depth in this work, and present a novel application of this technique to the model search problem.

Herodotu et. al. [23] explore performance modeling for MapReduce jobs on Hadoop clusters in great depth, but their model requires extensive profiling in order to accurately estimate job completion time. In contrast, our estimator requires more input from an algorithm developer and is focused on predicting a reasonable cluster size for a given machine learning model.

In the data mining and machine learning communities, most related to TUPAQ is Auto-WEKA [45]. As the name suggests, Auto-WEKA aims to automate the use of Weka [5] by applying recent derivative-free optimization algorithms, in particular Sequential Model-based Algorithm Configuration (SMAC) [26], to the hyperparameter tuning problem. In fact, their proposed algorithm is one of the many optimization algorithms we use as part of TUPAQ. However, in contrast to TUPAQ, Auto-WEKA focuses on single node performance and does not optimize the parallel execution of algorithms. Moreover, Auto-WEKA treats algorithms as black boxes to be executed and observed, while our system takes advantage of knowledge of algorithm execution from both a statistical and physical perspective. Aside from SMAC, various methods for derivative-free optimization and hyperparameter tuning have been proposed. We discuss and evaluate several of these methods in Sections 3 and 4, and two of these methods are used in TUPAQ.

With respect to the bandit optimization discussed in Section 3, both Agarwal et. al. [6] and Jamieson et. al. [27] have discussed bandit-like techniques for pruning during model search. Agarwal et. al., however, require explicit forms of the convergence rate behavior of intermediate results, which may be difficult to calculate and thus make their work difficult to implement in practice. Jamieson et. al. drew inspiration from TUPAQ when formulating their theoretically principled algorithm, and versions of this may be included in future versions of the system.

Weka [5], MLlib [36], Vowpal Wabbit [3], Hyracks [15] and Mahout [2] are notable open-source ML libraries. These systems (all distributed with the exception of Weka), along with proprietary projects such as SystemML [21], all focus on training single models rather than model search.

While much recent work on image classification and speech recognition has been done in the context of "Deep Learning" [24, 30], our focus is on learning methods that scale horizontally for efficient use of cluster resources, which can be difficult with Deep Learning methods.

7. Future Work and Conclusions

In this work, we have described a system for large scale model search which leverages both logical and physical improvements to provide faster search over conventional methods. Specifically, by combining better model search methods, bandit methods, batching techniques, and a cost-based cluster sizing estimator, TUPAQ can find high quality models built on very large datasets an order of magnitude more efficiently than than the baseline approach.

Several avenues exist for further exploration, and we note two broad classes of natural extensions to TUPAQ.

Machine learning extensions. From an accuracy point of view, as additional model families are added to MLbase, TUPAQ could naturally lend itself to the construction of *ensemble models* at training time—effectively *for free*. That is, while TUPAQ discards all but the best model as part of its training process, the process of model training is expensive, and the results can potentially be reused for better performance. Ensembles over a diverse set of methods are particularly known to improve predictive performance, but as more models and more hyperparameter configurations are considered, model search systems run the risk of overfitting to the validation data, and accounting for this issue, e.g., by *controlling the false discovery rate* [11], would become especially important.

Systems extensions. Multi-stage *ML pipelines*, in which the initial data is transformed one or more times before being fed into a supervised learning algorithm, are common in most practical ML systems. Since each stage will likely introduce additional hyperparameters, model search becomes more challenging in the pipeline setting. In a regime where a dataset is relatively small but users still have access to cluster resources, there can be benefits (both in terms of simplicity and speed) to broadcast the data to each worker machine and train various models locally on each worker. Model search could be made more efficient by considering the tradeoffs between these regimes. Training models on *subsets of data* can efficiently yield noisy evaluations of candidate models, though careful subsampling is required to yield high-quality and trustworthy models [7]. Akin to traditional query planners, model search systems can learn from knowledge of the data they store and historical workloads. A model search system could store *search statistics* to tailor its search strategy to the types of models have been used for a user's data in the past. The evaluation of these techniques in TUPAQ will be natural once the system has been exposed to a larger set of workloads.

8. Acknowledgements

This research is supported in part by NSF CISE Expeditions award CCF-1139158 and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, SAP, Apple, Inc., Cisco, Clearstory Data, Cloudera, Ericsson, Facebook, GameOnTalis, General Electric, Hortonworks, Huawei, Intel, Microsoft, NetApp, Oracle, Samsung, Splunk, VMware, WANdisco and Yahoo!. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE 1106400.

Thanks to Trevor Darrell, Yangqing Jia, and Sergey Karayev who provided featurized ImageNet data, Ben Recht who provided valuable ideas about derivative-free optimization and feedback, and Shivaram Venkataraman, Peter Bailis, Alan Fekete, Dan Crankshaw, Sanjay Krishnan, Xinghao Pan, Kevin Jamieson, and our Shepherd, Siddhartha Sen, for helpful feedback.

References

[1] Anaconda python distribution. `http://docs.continuum.io/anaconda/`.

[2] Apache Mahout. `http://mahout.apache.org/`.

[3] Cluster parallel learning. [With Vowpal Wabbit]. `https://github.com/JohnLangford/vowpal_wabbit/wiki/Cluster_parallel.pdf`.

[4] GraphLab Create Documentation: model parameter search.

[5] WEKA. http://www.cs.waikato.ac.nz/ml/weka/.

[6] A. Agarwal, J. Duchi, and P. Bartlett. Oracle inequalities for computationally adaptive model selection. *arXiv.org*, Aug. 2012.

[7] S. Agarwal, H. Milner, A. Kleiner, A. Talwalkar, M. Jordan, S. Madden, B. Mozafari, and I. Stoica. Knowing when you're wrong: Building fast and reliable approximate query processing systems. *SIGMOD*, 2014.

[8] A. Alexandrov et al. The Stratosphere Platform for Big Data Analytics. *VLDB*, 2014.

[9] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, 1967.

[10] K. Bache and M. Lichman. UCI Machine Learning Repository, 2013.

[11] Y. Benjamini and Y. Hochberg. Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *JRSS B*, 1995.

[12] A. Berg, J. Deng, and F.-F. Li. ImageNet Large Scale Visual Recognition Challenge 2010 (ILSVRC2010), 2010.

[13] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for Hyper-Parameter Optimization. *NIPS*, 2011.

[14] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *JMLR*, 2012.

[15] V. R. Borkar et al. Hyracks: A Flexible and Extensible Foundation for Data-Intensive Computing. In *ICDE*, 2011.

[16] S. Bubeck and N. Cesa-Bianchi. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Foundations and Trends in Machine Learning*, 2012.

[17] J. Canny and H. Zhao. Big data analytics with small footprint: squaring the cloud. In *KDD*, 2013.

[18] J. Deng, J. Krause, A. C. Berg, and L. Fei-Fei. Hedging your bets: Optimizing accuracy-specificity trade-offs in large scale visual recognition. *CVPR*, 2012.

[19] E. Even-Dar, S. Mannor, and Y. Mansour. Action Elimination and Stopping Conditions for the Multi-Armed Bandit and Reinforcement Learning Problems. *JMLR*, 2006.

[20] J. Garofolo, L. Lamel, W. Fisher, J. Fiscus, D. Pallett, N. Dahlgren, and V. Zue. TIMIT Acoustic-Phonetic Continuous Speech Corpus. 1993.

[21] A. Ghoting et al. SystemML: Declarative machine learning on MapReduce. In *ICDE*, 2011.

[22] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*, 2012.

[23] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *VLDB*, 2011.

[24] G. Hinton et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *Signal Processing Magazine, IEEE*, 2012.

[25] P.-S. Huang, H. Avron, T. N. Sainath, V. Sindhwani, and B. Ramabhadran. Kernel methods match deep neural networks on TIMIT. IEEE, 2014.

[26] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential Model-Based Optimization for General Algorithm Configuration. pages 507–523, 2011.

[27] K. Jamieson and A. Talwalkar. Non-stochastic Best Arm Identification and Hyperparameter Optimization. *CoRR*, 2015.

[28] B. Komer et al. Hyperopt-sklearn: Automatic hyperparameter configuration for scikit-learn. In *ICML workshop on AutoML*, 2014.

[29] T. Kraska, A. Talwalkar, J. Duchi, R. Griffith, M. Franklin, and M. Jordan. MLbase: A Distributed Machine-learning System. In *CIDR*, 2013.

[30] A. Krizhevsky et al. Imagenet classification with deep convolutional neural networks. *NIPS*, 2012.

[31] M. Kuhn et al. *caret: Classification and Regression Training*, 2015. R package version 6.0-41.

[32] A. Kumar, P. Konda, and C. Ré. Feature Selection in Enterprise Analytics: A Demonstration using an R-based Data Analytics System. *VLDB Demo*, 2013.

[33] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Softw.*, 1979.

[34] J. D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. Technical report, University of Virginia, 1991-2007.

[35] J. D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *TCCA Newsletter*, 1995.

[36] X. Meng et al. MLlib: Machine Learning in Apache Spark. *CoRR*, 2015.

[37] J. A. Nelder and R. Mead. A Simplex Method for Function Minimization. *The computer journal*, 1965.

[38] B. Panda, J. S. Herbach, S. Basu, and R. J. Bayardo. Planet: Massively Parallel Learning of Tree Ensembles with MapReduce. *VLDB*, 2009.

[39] F. Pedregosa et al. Scikit-learn: Machine learning in Python. *JMLR*, 2011.

[40] M. J. Powell. An Efficient Method for Finding the Minimum of a Function of Several Variables Without Calculating Derivatives. *The computer journal*, 1964.

[41] A. Rahimi and B. Recht. Random Features for Large-Scale Kernel Machines. In *NIPS*, 2007.

[42] T. N. Sainath, B. Ramabhadran, M. Picheny, D. Nahamoo, and D. Kanevsky. Exemplar-based sparse representation features: From timit to lvcsr. *IEEE Transactions on Audio, Speech, and Language Processing*, 2011.

[43] J. Snoek, H. Larochelle, and R. P. Adams. Practical Bayesian Optimization of Machine Learning Algorithms. *arXiv.org*, June 2012.

[44] E. R. Sparks, A. Talwalkar, et al. MLI: An API for Distributed Machine Learning. In *ICDM*, 2013.

[45] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms. In *KDD*, 2013.

[46] R. C. Whaley and J. J. Dongarra. Automatically Tuned Linear Algebra Software. In *ACM/IEEE conference on Supercomputing*, 1998.

[47] S. Williams, A. Waterman, and D. Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *CACM*, 2009.

[48] M. Zaharia et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, 2012.