Scalable Automated Model Search

Evan Sparks

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2014-122
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-122.html

May 20, 2014

Copyright © 2014, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

Thanks to Ameet Talwalkar, Michael Franklin, Michael Jordan, and Tim Kraska who all contributed to this work.

Scalable Automated Model Search

by Evan R. Sparks

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Michael J. Franklin
Research Advisor

Date

* * * * * *

Benjamin Recht
Second Reader

May 14, 2014

Scalable Automated Model Search[*]

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

ABSTRACT

Model search is a crucial component of data analytics pipelines, and this laborious process of choosing an appropriate learning algorithm and tuning its parameters remains a major obstacle in the widespread adoption of machine learning techniques. Recent efforts aiming to automate this process have assumed model training itself to be a black-box, thus limiting the effectiveness of such approaches on large-scale problems. In this work, we build upon these recent efforts. By inspecting the inner workings of model training and framing model search as bandit-like resource allocation problem, we present an integrated distributed system for model search that targets large-scale learning applications. We study the impact of our approach on a variety of datasets and demonstrate that our system, named GHOSTFACE, solves the model search problem with comparable accuracy as basic strategies but an order of magnitude faster. We further demonstrate that GHOSTFACE can scale to models trained on terabytes of data across hundreds of machines.

Categories and Subject Descriptors

Big Data [**Distributed Computing**]: Large scale optimization

1. INTRODUCTION

Modern scientific and technological datasets are rapidly growing in size and complexity, and this wealth of data holds the promise for a variety of transformational applications. Machine learning (ML) and related fields are seemingly poised to deliver on this promise, having proposed and rigorously evaluated a wide range of data processing techniques over the past several decades. However, the challenge in effectively deploying these techniques in practice, and in particular at scale, remains a major bottleneck to the wider adoption of these statistical methods.

In this work, we explore the *model search* problem in a distributed learning environment. Specifically, how to best choose between model families for supervised learning problems and configure the hyperparameters for these algorithms *automatically* to models with high accuracy very quickly. This process of model family selection and hyperparameter tuning remains a largely ad-hoc task for ML experts, and a herculean undertaking for non-experts. In the best cases, tedious and often non-reproducible efforts are required to produce reasonable models, while in the worst cases, inaccurate or even faulty models, c.f., [23] are generated.

Recently there have been attempts to automate the process of model search, e.g., [38, 35, 14, 13]. These techniques show clear improvements over naive grid search. However, these techniques essentially ignore implementation details of the models being tuned, and instead assume that algorithm training is a black box. Specifically, model training is seen as an opaque, expensive function call, into which hyperparameter configurations are input, and an assessment of model performance is output. Such an approach limits the types of strategies available to automate and speed up the model search process. This assumption is particularly limiting in the distributed setting, where model training follows a similar access pattern for a wide range of learning algorithms.

In this paper, we take an integrated approach to the model search problem for large-scale distributed machine learning. Our strategy builds upon recent developments in model search by incorporating knowledge about the access patterns of model training and relating the subsequent model search problem to a multi-armed bandit resource allocation problem. Our system, which we call GHOSTFACE, is part of MLbase [26] and takes advantage of MLI [37], a new programming abstraction that simplifies the development of distributed ML algorithms. MLI partially unifies the implementation of ML algorithms, which allows GHOSTFACE to inspect them with respect to the physical process and order in which the algorithms access data (their access patterns) and their incremental progress (i.e., model performance after some number of iterations). This in turn enables unique optimizations in two ways: First, the data access by various algorithm instances can be combined, and second, cluster resources can be better allocated by providing more (less) resources to promising (poorly-performing) model configurations.

In summary, this paper makes the following contributions to the automation of model search at scale:

- We show how an integrated architecture can be leveraged to combine access patterns and thus increase data and instruction cache locality for parallel model training.
- We describe how search techniques can be combined with physical optimization and early stopping strategies.
- We describe the architecture of GHOSTFACE and how it works with MLbase to make ML easier for end users.
- We present our experimental results on various data sets, which show convergence to that is an order of magnitude faster than simple model search strategies while achieving comparable model accuracy.

2. RELATED WORK

Most related to GHOSTFACE is Auto-Weka [38]. As the name suggests, Auto-Weka aims to automate the use of Weka [10] by applying recent derivative-free optimization algorithms, in particular

[*]The majority of this work is under submission and appears under the title “Automating Model Search at Scale”.

Sequential Model-based Algorithm Configuration (SMAC) [24], to the model search problem. In fact, their proposed algorithm is one of the many optimization algorithms we use as part of GHOSTFACE. However, in contrast to GHOSTFACE, 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.

In addition to SMAC, other algorithms have been recently been proposed. In Bergstra et. al. [14], the effectiveness of random search for hyper-parameter tuning is established, while Bergstra et. al.[13] proposes a search method based on refinement on initially random search that is refined with new information, called Tree-structured Parzen Estimation (TPE). We make use of both methods in our system. Snoek et. al. [35] explore the use of Gaussian Processes for the model search problem, and propose a variety of search algorithms, including an algorithm that accounts for improvement per time-unit, and another extension targeting parallel implementations in which several new model configurations are proposed at each iteration. However, model training is nonetheless considered as black-box, and moreover, we found that their algorithms, collectively called Spearmint, often run for several minutes per iteration as the number of samples increases, which is too long to be practical in many scenarios.

In contrast to these recent works, the field of derivative free optimization has a long history of optimizing functions for which derivatives cannot be computed [18]. Our evaluation of these algorithms on the model search problem suggest that they are not well-suited for this task, potentially due to the lack of smoothness of the (unknown) model search function that we are optimizing.

In terms of system-level optimization, both Kumar et. al. [27] and Canny et. al. [16] discuss batching as an optimization for speeding up machine learning systems. However, [27] discusses this technique in the context of automatic feature selection, an important problem but distinct from model search, while [16] explores this technique in the context of parameter exploration and model tuning, as well as 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.

There are also several proprietary and open-source systems providing machine learning functionality with varying degrees of automation. Google Predict [4] is Google's proprietary web-service for prediction problems with some degree of automation, yet it restricts the maximum training data-size to 250MB and the internals of the system are largely unknown. Weka [10] and Mahout [2] are two notable open-source ML libraries, and in theory our proposed methods could work with them. In practice though, such integration would require some API restructuring, for example, to expose the access patterns of the algorithms available in these systems to GHOSTFACE. Similarly, there has been a lot of work on distributed run-times for ML such as Vowpal Wabbit [3], SystemML [22] and Hyracks [15]. GHOSTFACE is designed explicitly for model search and hyperparameter tuning at scale, while these systems focus on scalably training single models.

Finally, we note that GHOSTFACE is part of MLbase [26], a novel system to simplify the use of machine learning. MLbase, and hence GHOSTFACE, is built upon Apache Spark, a cluster compute system designed for iterative computing [39]. Further, GHOSTFACE leverages the low-level ML functionality in MLlib [21] (the lowest layer of MLbase that also serves as Spark's default ML library) and the MLI API [37].

3. OPTIMIZING LARGE SCALE MODEL SEARCH

One conventional approach to model search or hyperparameter tuning is sequential grid search. A space is defined over algorithm hyperparameters, and points are selected iteratively from a grid across these hyperparameters and model fitness at each of these points is evaluated iteratively.

This approach has several obvious drawbacks. First, no information about the results of previous iteration is incorporated into later search. 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. Still, it may be the most frequently used search method for hyperparameter tuning in practice, despite the existence of more effective search methods.

Although these more effective methods – most recently, Bayesian in variety – provide much better search performance than grid search, as we will see, they do not take into account the full picture of model search.

3.1 Background and Goals

Before presenting the optimizations that are key to achieving high performance in the model search problem, we first describe our problem setting and assumptions. Specifically, we are operating in a scenario where individual models are of dimensionality d, which is typically significantly less than the total number of example data points N. In this scenario, we are focusing on searching across a relatively small number of model families, $f \in F$, each with a relatively small number of hyperparameters, $\lambda \in \Lambda$. Moreover, we restrict our study to model families that are trained via a particular access pattern, namely multiple sequential scans of the training data. In particular, we focus in this paper on two model families: linear Support Vector Machines (SVM) and logistic regression, both trained via gradient descent. However, it is worth noting that the access pattern we consider encompasses a wide range of learning algorithms, especially in the large-scale distributed setting. For instance, efficient distributed implementations of linear regression [21], tree based models [32], Naive Bayes classifiers [21], and k-means clustering [21] all follow this same access pattern.

Our system is built on Apache Spark [39], and we are targeting algorithms that run on tens to thousands of nodes on commodity computing clusters, and datasets that fit comfortably into cluster memory - on the order of tens of gigabytes to terabytes. Training of a model to convergence on such a cluster is expected to require tens to hundreds of passes through the training data, and take on the order of minutes. Moreover, with a terabyte dataset, performing a naive model search using sequential grid search involving just 128 model configurations would take nearly a week, even given a compute cluster of 128 nodes. Hence, in this regime naive model search is tremendously costly, and our goal is to minimize total resource consumption as part of the model search process.

Existing algorithms for model search have considered model training to be a black box. The goal of these algorithms is to find model configurations with good statistical performance, which is usually measured in terms of validation error for classification models or validation RMSE for regression models. By observing that model training is not purely a black box and making some mild assumptions about how models are trained and what can be observed of their training, we validate and combine state of the art model search methods with additional techniques to reduce total runtime and total cost.

We reduce the resources required through three basic techniques: 1) batching model training to improve resource utilization, 2) better search methods, and 3) bandit-like resource allocation to focus our resources on promising model configurations. Together, these optimizations lead to a system that is an order of magnitude faster than one based on a sequential grid search approach.

3.2 Batching

As noted by others, e.g., [27, 16] batching is a natural system optimization in the context machine learning, with applications for cross validation and ensembling. In the context of model search, we note that the access pattern of gradient descent based algorithms is identical with respect to the input dataset. Specifically, each algorithm takes multiple passes over the input data and updates some intermediate state (model weights) during each pass. As a result, it is possible batch together the training of multiple models. In a distributed environment, this has several advantages:

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

Ultimately, these three advantages lead to a significant increase in aggregate throughput in terms of models trained per hour. For a typical distributed system, a model update requires at least two network round-trips to complete. One to launch a task on each worker node, and one to report the results of the task back to the master. By amortizing this cost across multiple models, we reduce total overhead due to network latency substantially.

In the context of a distributed machine learning system like GHOSTFACE, which runs on Spark, recent work [31] indicates that the startup time for a given task in Spark is roughly 5ms. By batching our updates into larger tasks, we are able to reduce the aggregate overhead of launching new tasks substantially. For example, if we have a batch size of 10 models, the average task overhead per model iteration drops to 0.5ms. Over the course of hundreds of iterations for hundreds of models, the savings can be substantial. This possibly comes at the expense of introducing stragglers to our jobs, but on balance we see faster throughput through batching.

Finally, modern x86 machines have been shown to have processor cores which significantly outperform their ability to read data from main memory [29]. In particular, on a typical x86 machine, the hardware is capable of reading 0.6B doubles/sec from main memory, while the hardware is capable of executing 15B FLOPS in the same amount of time [28]. This imbalance provides an opportunity for optimization by reducing unused resources, i.e., wasted cycles. By performing more computation for every double read from memory, we can reduce this resource gap. A typical gradient update in a sufficiently large model takes 2-4 FLOPs per feature read from memory – one multiply and one addition, plus some constant time for subtracting off the observation and performing a transformation. If models are small enough to fit in the CPU's cache lines, then we should expect to see a roughly 7-10x speedup in execution time by batching multiple models together. While we fall short of this theoretical limit in our evaluation of this optimization, it still provides a up to a 5x improvement on our synthetic evaluations and a 1.6x improvement on our larger scale experiment in which model complexity is on the order of $10,000$ parameters.

3.3 Better Search

We have validated existing methods of hyperparameter search for use in our scenario. We compare traditional methods with more recent methods in hyperparameter search, and present the results in Section 5. Our experiments confirm that two recent hyperparameter tuning algorithms – TPE and SMAC – provide better convergence properties than traditional methods, though surprisingly only slightly better than random search.

3.3.1 Traditional Methods

We explored a number of traditional methods for derivative-free optimization, including grid search, random search, Powell's method [34], and the Nelder-Mead method [30]. Our baseline method is grid search, whereby grid points are chosen from a uniform distribution or loguniform distribution from a grid of all hyperparameters for a given model.

Random search selects points at random from the same distributions as grid search. For the sake of comparability, we use the same number of random points as grid points in all of our experiments. We note here that random search works surprisingly well compared to the more advanced methods we studied. This has also been noted and studied by others [14].

Powell's method can be seen as a derivative free analog to coordinate descent. Given a starting point, each dimension in the hyperparameter space is searched for a local minimum. The vectors pointing to these local minima are summed and added to the starting point. The algorithm proceeds iteratively from this new point until convergence is reached or some maximum number of function evaluations have been made.

The Nelder-Mead method repeatedly samples from a polytope with $K + 1$ vertices in K dimensions on the hyperparameter space. This polytope is used to approximate a gradient, and the algorithm chooses a new point to sample opposite the direction of the gradient and proceeds with a new polytope formed with this new point.

Both Powell's 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 hyperaparamter 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 we are optimizing over an unknown function that is likely non-smooth and not convex.

3.3.2 TPE

Tree-based Parzen Estimators, first presented in [13], begin with a random search, and then probabilistically sample from points with more promising minima. More formally, TPE models $p(\lambda|c)$ based on:

$$p(\lambda|c) = \begin{cases} l(\lambda) & \text{if } c < c* \\ g(\lambda) & \text{if } c \geq c* \end{cases},$$

where c* represents the point at the γ quantile of points sampled so far, and $l(\lambda)$ and $g(\lambda)$ represent density estimates of the hyperparameters below and above the $c*$ threshold, respectively. The algorithm considers new candidates at random, and among these random candidates, selects the one with the lowest value of $g(\lambda)/l(\lambda)$, or, intuitively, the point most likely to be in the "good" region.

3.3.3 SMAC

The SMAC algorithm works by building a Random Forest model from observed hyperparameter results. The algorithm first uses Random Forests to estimate $p(c|\lambda)$, which models the dependence of the loss function c on the model hyper-parameters λ. By utilizing the Random Forests to obtain an estimate of both mean and variance for points in the hyperparameter space, they are able to find areas of highest expected improvement, as in more traditional Gaussian methods. Importantly, the authors of [38] point out that this model can capture complex and conditional hyperparameters, and SMAC was recently used for model search as part of of Auto-Weka.

3.3.4 GP Methods

[35] have proposed a Bayesian method for hyperparameter optimization based on Gaussian Processes. Their choice of kernel under this model allows for closed form solutions to expected improvement at each model iteration. Additionally, they present an optimization which speeds up operation of this algorithm on multicore machines. Evaluation of their acquisition function is cubic in the number of models trained, and candidate models may take minutes to be proposed after a few hundred models have been trained. Additionally, there is an exponential initialization phase at the beginning of model training which can be quite expensive to compute as the hyperparameter space grows.[1]

3.4 Bandit-like Resource Allocation

We observe that not all models are created equal. In the context of model search, only one model will converge to the "best" answer. Many of the remaining models perform drastically worse than this best model, and under certain assumptions, allocating resources among different model configurations can be naturally framed as a multi-armed bandit problem.

Given a fixed set of k model configurations to evaluate, where k is the product of distinct model families and the number of hyperparameter configurations considered for each family, the model search problem can naturally be cast as a repeated game with a fixed number of rounds T. At each round we perform a single iteration of a particular model configuration, and return a score (or reward) indicating the quality of the updated model, e.g., validation error. In such a 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 choosing an arm as a function of the current scores.

However, 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.

We thus propose a bandit-like heuristic that takes into account our problem-specific constraints while allocating resources. Our heuristic preemptively prunes models that fail to show promise of converging. Specifically, for each model we initially allocate a fixed number of iterations, and based on its quality we decide whether to train the model to completion. In our experiments, our initial allocation is 10 iterations (or passes through the data), and a model's quality is assessed by comparing its validation error with the smallest validation error we have already seen with previous model configurations trained on 10 iterations.

[1]In preparation for this paper, we contacted the authors about some deficiencies of their model in terms of computational and statistical performance, and they redirected us to very recent work [36] which can help these models achieve better statistical performance, potentially reducing the need for hundreds of function evaluations in low dimensional hyperparameter space. However, the code associated with this work was not available in time to be evaluated in this submission.

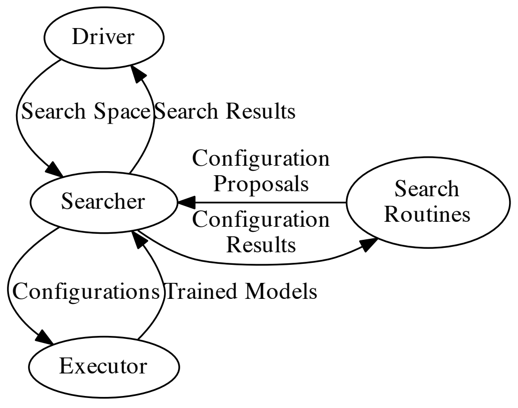

Figure 1: The Driver is responsible for defining a search space over which models are to be trained. The Searcher deterimines which configurations to try and, based on the results which ones to try next. It may interact with external search routines to accomplish this. The Executor is responsible for evaluating model configurations and reporting results - it may evaluate a particular configuration over a cluster of machines.

4. ARCHITECTURE

GHOSTFACE is built on top of Apache Spark and MLI. These systems have been designed to support high performance iterative MapReduce algorithms on working sets that fit into cluster memory, and simplified development of machine learning algorithms, respectively. It has been shown that these systems can outperform conventional MapReduce-based systems for machine learning like [2], often by an order of magnitude. We leverage features of both systems in our design and experiments. Since both Spark and MLI are written in Scala, we chose to write GHOSTFACE in Scala as well to leverage tight integration with the existing systems and libraries available in the JVM ecosystem. Our system is a component of MLbase [26], and an important component of MLbase is the MLbase Optimizer, designed to automate the process of constructing and refining machine learning pipelines. GHOSTFACE represents a first effort at building such an optimizer and is a central component to the design of this system.

The architecture of GHOSTFACE consists of three abstract entities: A *Driver*, one or more *Searchers*, and zero or more *Executors*. These entities are implemented as `akka` Actors, which enables a concurrent programming model that supports asynchronous communication between these entities. A graphical depiction of these entities is shown in Figure 1 and a summary of the protocol between these entities is captured in Figure 2. Additionally, models are trained in a *parallel dataflow* system, whose computational model – iterative MapReduce – lends itself well to expressing a large class of machine learning algorithms. The responsibilities of each entity are described below.

4.1 Driver

The driver is the main entry point for a GHOSTFACE model search. The Driver's primary responsibilities include instantiating one or more Searchers and collecting results from the search process. End users typically interact with the system through the Driver, which is instantiated with a dataset, a search space which describes the space of model configurations to explore, and information about how to configure the searcher and what type of searcher to instantiate (e.g. Grid, Random, TPE). The driver communicates asynchronously with the searcher and collects newly trained models as they become available. It can also be polled by user programs for new information about search progress and models that have been trained.

4.2 Searcher

A Searcher is responsible for instantiating Executors, deciding on model configurations for evaluation, farming that work out to its Executors, and relaying results back to the Driver. Importantly, we also centralize the logical and physical optimizations in the implementation of the searcher. Namely, it is the searcher's job to figure out how best to implement batching (by sending jobs to a Batch Executor), resource allocation (by evaluating results from Executors before letting work continue), and to implement the search method it is responsible for.

The architecture is such that we may use external libraries or service calls to implement the searching logic. For example, in implementing a TPE based searcher, the TPESearcher code makes external service calls to a web service that is simply a thin wrapper around the existing Hyperopt library [5].

By implementing a few concrete methods for suggesting points in the hyperparameter space and ingesting new results, our system allows easy addition of new algorithms for hyperparameter optimization as they become available. Additionally, new algorithms will benefit from the systems level optimizations made for all algorithms.

4.3 Executor

A Executor in GHOSTFACE is an abstract entity capable of evaluating model configurations. A GHOSTFACE Executor may launch jobs that are executed on hundreds of nodes. In practice, our executors usually have a handle to an entire cluster, which allows them to execute their jobs in a data parallel fashion. As such, we are often in a situation where only a single Executor needs to be created. While this detail need not be evident to the searchers, in the case of batching, for example, the *type* of Executor (that is, a Batch Executor vs. a Sequential Executor) can play a role in how tasks are allocated to it.

4.4 Hybrid Parallel Dataflow and Asynchronous Control for Model Search

In basing our system on Spark, we are explicitly employing a parallel dataflow engine to train our models. At the same time, all of the hyperparameter tuning happens in an asynchronous fashion with Actors handing out jobs and receiving results asynchronously. As such, our system can be viewed as a hybrid of a traditional BSP computation environment with an asynchronous control plane. By employing this approach, we are able to isolate the portions of our system that need to be high performance, fault tolerant, and data parallel from those that need to be asynchronous and reactive, leading to a system that achieves both goals.

5. EVALUATION

We evaluated GHOSTFACE in two ways. First, we validated our ideas about model search and preemptive pruning on five representative datasets across a variable number of function evaluations. In particular, we used these results to motivate which hyperparameter search strategy to incorporate into a larger system. We also used a small scale infrastructure to experiment with hyperparameter tuning in slightly higher dimensions and refine our preemptive pruning heuristic. Once these ideas were validated, we implemented them in a large scale system, designed with end-to-end performance in mind and better suited for a cluster environment. Next, we evaluated this approach on very large scale data problems, at cluster sizes ranging from 16 to 128 nodes and datasets ranging from 40GB to over 1TB in size. These numbers represent actual features the model was trained on, *not* the raw data from which these features were derived - that is, these numbers are the dataset size that the model was trained on - not some pre-aggregated dataset.

Before training, we split our base datasets into 70% training, 20% validation, and 10% testing. In all cases, models are fit to minimize classification error on the training set, while hyperparameter selection occurs based on classification error on the validation set (validation error). Since the hyperparameter tuning algorithms only have access to validation error, we report those numbers here, but test error was similar. GHOSTFACE is capable of optimizing arbitrary performance metrics as long as they can be computed mid-flight, and should naturally extend to other supervised learning scenarios such as multinomial classification problems or regression problems.

By using a state of the art search method, employing batching, and preemptively terminating non-promising models, we are able to see a 7x increase in raw throughput of the system in terms of models trained per unit time, while seeing an effective 19x speedup in terms of convergence to a reasonable model configuration versus the conventional approach.

5.1 Platform Configuration

We tested GHOSTFACE on Linux machines running under Amazon EC2, instance type `m2.4xlarge`, while the smaller scale experiments were performed on a single machine. These machines were configured with Redhat Enterprise Linux, version 1.9 of the Anaconda python distribution from Continuum Analytics[1], and Apache Spark 0.8.1. Additionally, we made use of Hadoop 1.0.4 configured on local disks as our data store for the large scale experiments. Finally, we use MLI as of commit 3e164a2d8c as a basis for GHOSTFACE.

5.2 Small Scale Experiments

We validated our ideas about model training on a series of small datasets with well-formed binary classification problems embedded in them. These datasets come from the UCI Machine Learning Repository [11]. The model search task involved tuning four hyperparameters - learning rate, L1 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 [7]. These last two parameters illustrate that these techniques apply to feature selection as well as model hyperparameters, but more importantly illustrate that 4 or 5 dimensional hyperparameter space is reasonable to search given a budget of a few hundred model trainings. To accomodate for the linear scaleup that comes with adding random features, we down sample the number of data points for each model training by the same proportion, leaving a learning problem that is equivalent in terms of computational resources required to train given our fixed iteration budget. We note that Snoek et. al. [35] introduce a method of selecting new hyperparameter configurations based on expected improvement *per second*, which would obviate the need for this adjustment, but because such a metric is not universally available we omit it here.

Message	Result	Meaning
RunSearch(space: SearchSpace, data: Table)		Begin Search Process
TopModel()	Model	Return best model after search.
AllBuilt()	Boolean	Is search process finished?
ModelsBuilt()	Int	How many models have been built?
Models()	Seq[Model]	Return all models that have been built.

(a) Driver/Searcher Protocol

Message	Result	Meaning
BuildModel(config: ModelConfig)		Add model with this configuration to work queue.
DoWork()		Begin consuming work queue.
StopWork()		Stop consuming work queue.
QueueSize()	Int	How big is your work queue?

(b) Searcher/Executor Protocol

Message	Meaning
ModelBuilt(model: Model)	A model has been built with a particular configuration and error.
WorkDone()	My work queue is empty.

(c) Executor/Searcher Protocol

Figure 2: Summary of protocol between Driver, Searcher, and Executor.

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 $\in (10^{-4}, 10^{2})$. For these experiments model family was fixed to linear SVM, however we should note that the random features introduce nonlinearity into the models.

A prototype built to run these experiments was constructed in Python, using the scikit.learn [33] library and numpy [6].

5.2.1 Search

We evaluated seven search methods: grid search, random search, Powell's method, the Nelder-Mead method, SMAC, TPE, and a Gaussian Process-based optimization method. We implemented random and grid search directly, and used implementations of Powell's method and the Nelder-Mead method from scipy [25]. For the other three methods, we used off-the-shelf software from the inventors of those methods, graciously made available to us freely online [5][9][8].

Results of the search experiments are presented in Figure 3. Each dataset was run through each search method with a varying number of function calls, chosen to align well with a regular grid of n^4 points where we vary n from 2 to 5. Of course, this restriction on a regular grid is only necessary for grid search, but we wanted to make sure that the experiments were comparable.

With this experiment, we are looking for methods that converge to good models in as few function calls as possible. We note that of all methods tried, TPE and SMAC tend to achieve this criteria best, but random search is not far behind. That said, given the expense of running many experiments, and the principled nature of TPE and SMAC, we decided it was best to integrate one of these state of the art methods into GHOSTFACE for the larger scale experiments.

We chose to integrate TPE into the larger experiments because it seemed to perform slightly better than SMAC on some experiments and because it was fairly easy to wrap in a simple web service and query for new points to try, while integrating with SMAC would have been a bit more onerous. That said, our architecture fully supports additional search methods, and we expect to see additional methods implemented over time.

5.2.2 Preemptive Pruning

We evaluated our preemptive pruning heuristic on the same datasets with random search and 625 total function evaluations. The key question to answer here was whether we could terminate poorly

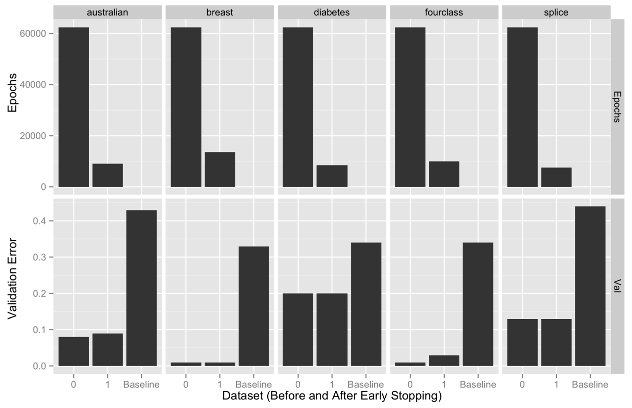

Figure 4: Here we show the effects of preemptive pruning on trained model performance. Model search completes in an average of 83% fewer passes over the training data than without preemptive pruning. Validation error is nearly indistinguishable vs. the case where we do not preemptively terminate model training.

performing models early in the training process without significantly affecting overall training error. In Figure 4 we illustrate the affect that early stopping has on number of total epochs (that is, passes over the dataset), which is a good proxy for speed of training, as well as validation error. We note a mean 86% decrease in total epochs across these five datasets, and note that validation error only gets slightly worse. On average, this method achieves 93% reduction in model error vs. not stopping early when compared with validation error of a simple baseline model.

Models were allocated 100 iterations to converge on the correct answer. After the first 10 iterations, models that were not within 5% classification error of the best model trained so far (after 10 iterations), were preemptively terminated. As a result, a large percentage of models which show little or no promise of converging to a reasonable validation error are eliminated.

As mentioned previously, we consider this heuristic a special case of a bandit algorithm where resources are explicitly allocated toward models that are progressing acceptably and not at all toward models that have not progressed acceptably or models that have exceeded their maximum number of iterations.

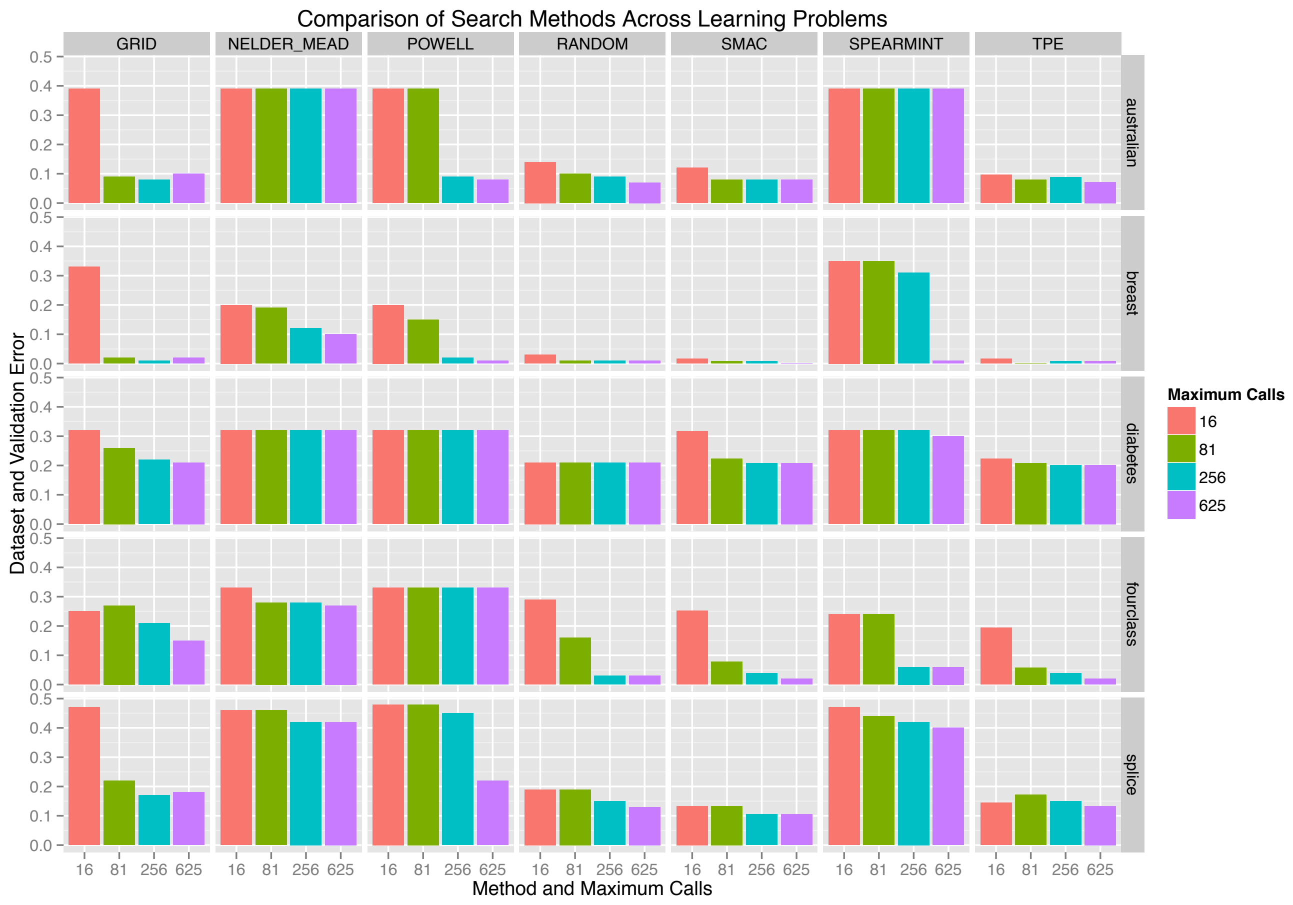

Figure 3: Search methods were compared across several datasets with a variable number of function evaluations. Classification error on a validation dataset is shown for each combination. TPE and SMAC provide state of the art results, while random search performs best of the classic methods.

5.3 Batching

We evaluated the effectiveness of batching at small scale using a synthetic dataset of $1,000,000$ data points in various dimensionality. We trained these models on a 16-node cluster of `m2.4xlarge` nodes on Amazon EC2, running Spark 0.8.1. We trained a linear SVM on these data points via gradient descent with no batching (batch size = 1) and batching up to 100 models at once. In Figure 5 we show the total throughput of the system in terms of models trained per hour varying the batch size and the model complexity. For small models, we see the total number of models per hour can increase by up to a factor of 6 for large batch sizes. As models grow in complexity, the effectiveness of batching deteriorates but remains significant. The downside to batching in the context of model search is that you may gain information by running models sequentially that could inform subsequent models that is not incorporated in later runs. By fixing our batch size to a relatively small constant (10) we are able to balance this tradeoff. Chen, et. al. [17] demonstrate that encouraging model diversity in this type of batching setting can be important, but we do not incorporate this insight in GHOSTFACE yet.

5.4 Large Scale Experiments

Our large scale experiments involve a Scala code base built according to the principles laid out in Section 4. Here, we ran experiments on 16 and 128 machines. Our dataset of reference is

Batch Size \ D	100	500	1000	10000
1	136.80	114.71	98.14	22.18
2	260.64	171.79	147.79	30.59
5	554.32	290.46	189.87	30.19
10	726.77	390.04	232.23	25.31

(a) Models trained per hour for varying batch sizes and model complexity.

Batch Size \ D	100	500	1000	10000
1	1.00	1.00	1.00	1.00
2	1.91	1.50	1.51	1.38
5	4.05	2.53	1.93	1.36
10	5.31	3.40	2.37	1.14

(b) Speedup factor vs baseline for varying batch size and model complexity.

Figure 5: Effect of batching is examined on 16 nodes with a synthetic dataset. Speedups diminish but remain significant as models increase in complexity.

pre-featurized version of the ImageNet Large Scale Visual Recognition Challenge 2010 (ILSVRC2010) dataset [12], featurized using a procedure attributed to [19]. This process yields a dataset with $160,000$ features and approximately $1,200,000$ examples, or 1.5 TB of raw image features. In our 16-node experiments we down sample 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 3 hyperparameters here - learning rate and L1 Regularization parameter matching the above experiments, with an additional parameter for which classifier we train - SVM or Logistic Regression. We allot a budget of 128 model fittings to the problem.

We search for a binary classification model that discriminates 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 accuracy 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 8% accuracy with linear classification models.

5.4.1 *Optimization Effects*

In Figure 6 we can see the effects of batching and preemptive pruning on the model training process. Specifically, given that we want to evaluate the fitness of 128 models, it takes nearly 13 hours to fit all 128 models on the 30GB dataset of data on the 16 node cluster. By comparison, with the preemptive pruning rule and batching turned on, the system takes just two hours to train a random search model to completion and a bit longer to train a TPE model to completion, a 6.3x speedup in the case of random search and a 4.2x speedup in the case of TPE. TPE takes slightly longer because it does a good job of picking points that do not need to be terminated preemptively. That is, more of the models that TPE selects are trained to completion than random search. Accordingly, TPE arrives at a better model than random search given the same training budget.

Turning our attention to model convergence illustrated in Figure 7, we can see that TPE converges to a good answer much faster than random or grid search and even better than random search, as we expected given the results of our smaller scale experiments. Specifically, TPE gets within 10% of the lowest error achieved after just 40 minutes. By contrast, the conventional approach with grid search and no optimization takes 4 hours and 41 minutes - thus, given this particular problem, better search, preemptive pruning, and batching yield a total speedup of 7x. Moreover, in practice, we would typically allow grid search to run to completion, because we have no expectation that grid search has found a reasonable answer partway through execution. Thus, if we compare the time that grid search takes to execute according to the conventional approach to the time it takes TPE to get to a reasonable answer, we compare 13 hours to 40 minutes, a 19x speedup.

5.5 Scalability

We now demonstrate GHOSTFACE's ability to scale to massive dataset sizes. Because we employ data-parallel versions of our learning algorithms, achieving horizontal scalability with additional compute resources is trivial. GHOSTFACE scales to a 1.5TB dataset that is 10-times more complicated with respect to model space easily. For these experiments, we ran with the same parameter search

[1]Note that grid search at middle of Figure 6 did not finish, but we expect it would have received a similar improvement from preemptive pruning. We will have these results shortly.

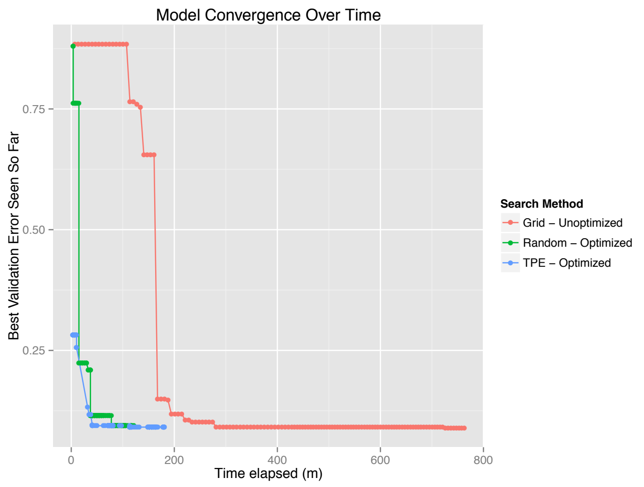

Figure 7: We compare the effect of various optimization methods on model convergence. We compare unoptimized grid search with fully optimized TPE search and Random search and see that TPE converges most quickly.

settings as the smaller dataset but this time with a fixed budget of 40 function evaluations. It should be noted that with this level of complexity in models, the batching optimization is likely not helping much due to the complexity of the models, but early stopping and better search certainly do. Our results are illustrated in Figure 8. Using the fully optimized TPE based search method, we are able to search this space in 13 hours, and the method is able to achieve a validation error of 8.2% for this dataset with 11 hours of processing.

6. FUTURE WORK

We note that these optimizations are just the tip of the iceberg in solving this problem faster. Advanced model training methods such as L-BFGS and ADAGRAD can speed convergence of individual models by requiring fewer passes over the training data [20].

More precise heuristics for preemptive pruning and better modeling of these heuristics may allow for better pruning of the search space mid-flight.

Search methods which take into account that multiple model configurations are being sampled may improve convergence rates [17].

Additionally, this method of multi-model training naturally lends itself to the construction of ensemble models at training time - effectively *for free*. However, there may be better search strategies for ensemble methods that encourage heterogeneity among ensemble methods, which we do not consider here.

6.1 Pipelines

The long term goal of this work is to provide a platform for the automatic construction and tuning of end-to-end machine learning pipelines. Here we describe how the model search procedure described above can be extended to the problem of pipeline construction, explore some potential pitfalls, and describe some preliminary thoughts about how these might be addressed.

6.1.1 *Pipeline Structure*

Real world machine learning pipelines can be viewed as data flow graphs with two related but separate components. The first component ingests and processes training data, performs feature extraction, and trains a statistical model. The second component

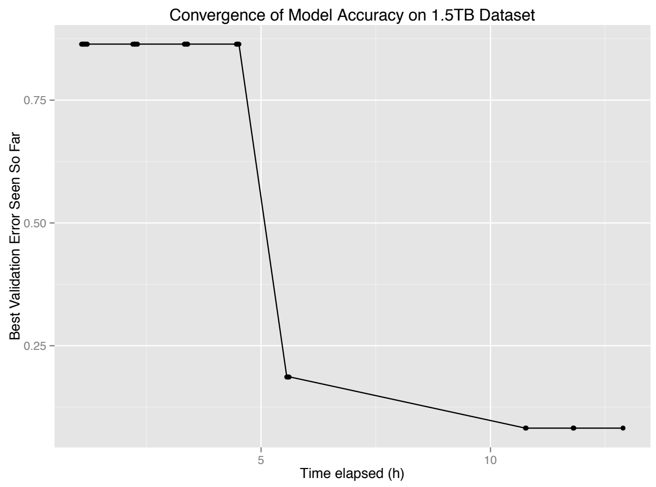

Figure 8: Training a model with a budget of 32 function evaluations on a 1.2m × 160k dataset takes 13 hours on a 128-node cluster with GHOSTFACE.

uses this model to make predictions on live data - possibly updating the model in the process. While these descriptions are simple, the internals of these components can be arbitrarily complex. For example, feature extraction, a common task, may be comprised of several steps - some of which are domain specific and others more general.

6.1.2 *Pipeline Operators*

While we can view machine learning pipelines as arbitrary data flow graphs that satisfy the above specification, in practice, machine learning pipelines consist of a set of fairly standard components - many of which are amenable to efficient execution in a distributed data flow environment. For concreteness, we enumerate some such operators which may be chained together to form a complete machine learning pipeline.

Domain Specific Operators are operators used to take data from a particular data domain (e.g. text, images, speech) and transform it into a form that is more amenable to use in a machine learning algorithm. Often these techniques have been developed by domain experts in the field over the course of years of research, and they may not be useful for other types of pipelines. The simplest example of a domain-specific operator would be a data loader - for

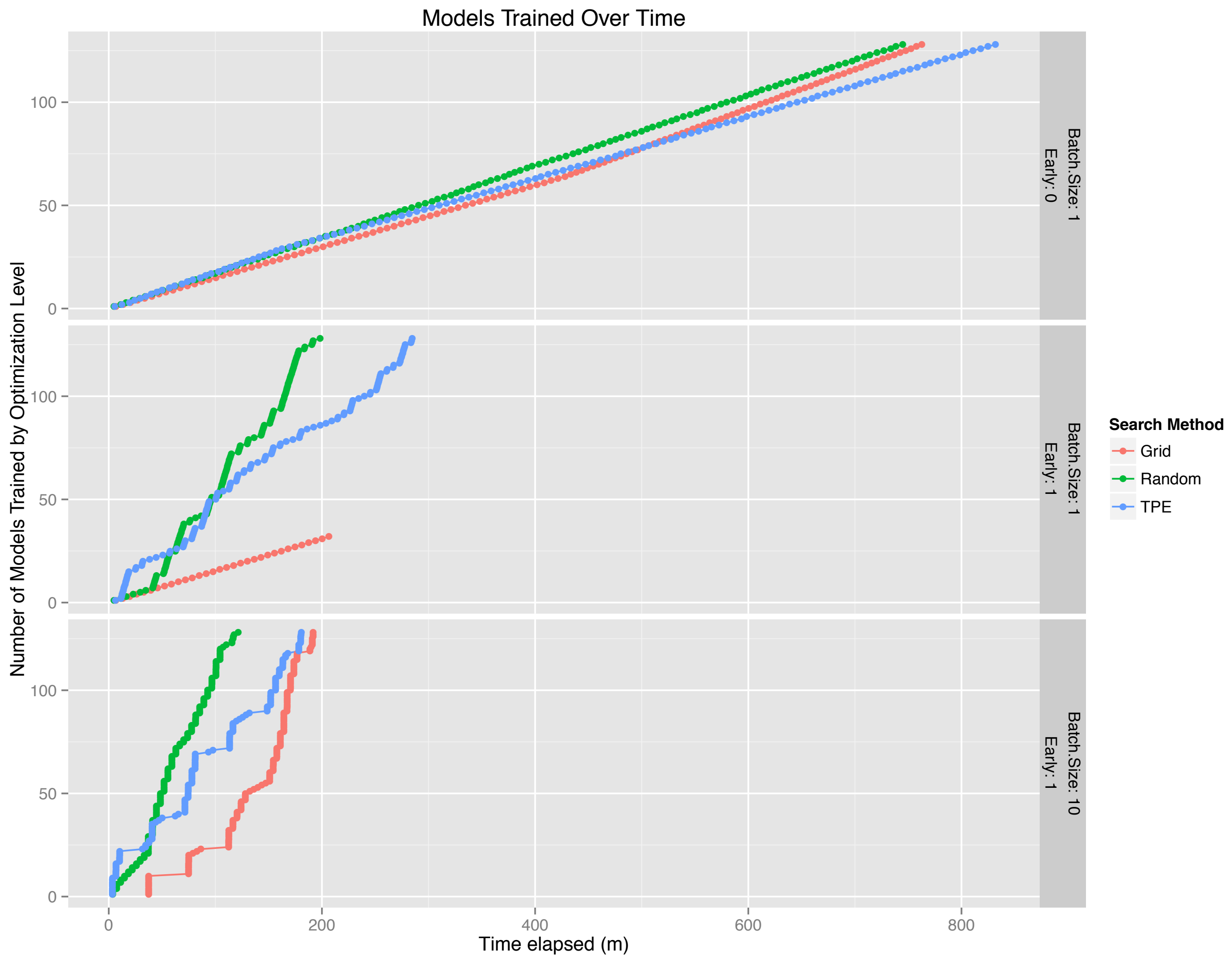

Figure 6: We compare the effect of various optimization methods on time to completion of a 128 model budget. Top: unoptimized. Middle: preemptive pruning turned on. Bottom: preemptive pruning and batching. Each optimization buys us nearly a factor of two in raw throughput.

text data we might need to parse XML into a document tree, while for image data we may need to load data from a compressed binary form on disk to a 3-dimensional array of pixels. More advanced operators in this space might include convolutions and edge detectors for images and speech, and tokenizers for text.

General Purpose Operators can be used once data has been loaded and domain-specific transformations have been applied. These general purpose transformations give data physical or statistical properties that are amenable for ingestion to a learning operator. Vectorization, one-hot-encoding, and normalization are all examples of general purpose operators that might need to be applied to such data. More sophisticated methods such as fourier transforms, random feature projection, ZCA whitening, and forms of dimensionality reduction also fall into this class of operators.

Learning Operators may be used to train a statistical model capable of performing inference over the featurized data - e.g. to classify an image or recommend a product to an end user. These primitives include algorithms for classification, regression, clustering and collaborative filtering. We note that these primitives can exist just about anywhere in a machine learning pipeline. For example - we might use K-Means clustering to learn convolution filters for image recognition, before training a linear model on the convolved data. Additionally, the data flow architecture is highly amenable to training ensemble models - instead featurized data flowing into a single training node, it can flow into many training nodes whose predictions can later be averaged by a later node.

6.1.3 Putting It All Together

We point out that given a learning pipeline, the pipeline itself has several hyperparameters to tune in addition to the model training hyperparameters before it can be used effectively. For example - how many convolutions need to be applied to an image for the purposes of image classification, how many principal components should we reduce to in the case of dimensionality reduction, how many random features should be produced. For a sufficiently complicated (and realistic) pipeline, we can easily end up with tens of hyperparameters, and we believe that while several of the methods in this paper may apply to this setting, optimizing over this many hyperparameters for learning problems is not a well explored space.

The very structure of the pipeline itself can also be viewed as a hyperparameter. While fully automating the construction of the data flow graph is likely beyond practical possibility, there may be hints we can take from the data and the user's problem that will automate much of the model construction. For example - we can eliminate domain specific operators at the beginning of our training pipeline that are irrelevant to the training domain. Further possibilities include finding "acceptable" configurations for early stages of the pipeline, fixing them, and determining later stages given these fixed choices. This heuristic may prevent finding the globally optimal pipeline structure, but may allow us to construct a pipeline that is good enough much more quickly.

7. CONCLUSION

We have demonstrated that by combining fast hyperparameter tuning methods, batching techniques, and a preemptive pruning heuristic, we can deliver a system capable of automating model search on very expensive models trained on very large datasets that is an order of magnitude faster than the conventional approach. GHOSTFACE is a component of the MLbase project, and will serve as a foundation for the automated construction of end-to-end pipelines for machine learning.

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!.

Thanks to Trevor Darrell, Yangqing Jia, and Sergey Karayev who provided featurized imagenet deta, and Ben Recht who provided valuable ideas about derivative-free optimization and feedback. Special thanks to Ameet Talwalkar, Michael Franklin, Michael Jordan, and Tim Kraska who were collaborators on this work.

9. 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] Google Prediction API. `https://developers.google.com/prediction/`.

[5] Hyperopt: Distributed asynchronous hyperparameter optimization in python. `http://hyperopt.github.io/hyperopt/`.

[6] Numpy. `http://www.numpy.org/`.

[7] Random features for large scale machine learning. `http://www.keysduplicated.com/~ali/random-features/`.

[8] Smac: Sequential model-based algorithm configuration. `http://www.cs.ubc.ca/labs/beta/Projects/SMAC/`.

[9] Spearmint (bayesian optimization). `http://people.seas.harvard.edu/~jsnoek/software.html`.

[10] Weka. `http://www.cs.waikato.ac.nz/ml/weka/`.

[11] K. Bache and M. Lichman. UCI machine learning repository, 2013.

[12] A. Berg, J. Deng, and F.-F. Li. Imagenet large scale visual recognition challenge 2010 (ilsvrc2010). `http://www.image-net.org/challenges/LSVRC/2010/`, 2010.

[13] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. 2011.

[14] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, 13:281–305, 2012.

[15] V. R. Borkar et al. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, 2011.

[16] J. Canny and H. Zhao. Big data analytics with small footprint: squaring the cloud. . . . *conference on Knowledge discovery and data mining*, 2013.

[17] Y. Chen and A. Krause. Near-optimal batch mode active learning and adaptive submodular optimization. pages 160–168, 2013.

[18] A. R. Conn, K. Scheinberg, and L. N. Vicente. *Introduction to Derivative-free Optimization*. SIAM, 2009.

[19] J. Deng, J. Krause, A. C. Berg, and L. Fei-Fei. Hedging your bets: Optimizing accuracy-specificity trade-offs in large scale visual recognition. pages 3450–3457, 2012.

[20] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient

methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011.

[21] M. Franklin et al. Mllib: A distributed machine learning library. In *NIPS Machine Learning Open Source Software*, 2013.

[22] A. Ghoting et al. Systemml: Declarative machine learning on mapreduce. In *ICDE*, pages 231–242, 2011.

[23] E. Hurwitz and T. Marwala. Common mistakes when applying computational intelligence and machine learning to stock market modelling. *CoRR*, abs/1208.4429, 2012.

[24] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. pages 507–523, 2011.

[25] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001–.

[26] T. Kraska, A. Talwalkar, J. Duchi, R. Griffith, M. Franklin, and M. Jordan. Mlbase: A distributed machine-learning system. In *CIDR*, 2013.

[27] A. Kumar, P. Konda, and C. Ré. COLUMBUS: Feature Selection on Data Analytics Systems. *algorithms*, 2013.

[28] J. D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[29] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, Dec. 1995.

[30] J. A. Nelder and R. Mead. A simplex method for function minimization. *The computer journal*, 7(4):308–313, 1965.

[31] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica. The case for tiny tasks in compute clusters.

[32] B. Panda, J. S. Herbach, S. Basu, and R. J. Bayardo. Planet: massively parallel learning of tree ensembles with mapreduce. *Proceedings of the VLDB Endowment*, 2(2):1426–1437, 2009.

[33] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[34] M. J. Powell. An efficient method for finding the minimum of a function of several variables without calculating derivatives. *The computer journal*, 7(2):155–162, 1964.

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

[36] J. Snoek, K. Swersky, R. S. Zemel, and R. P. Adams. Input warping for bayesian optimization of non-stationary functions, 2014.

[37] E. R. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. E. Gonzalez, M. J. Franklin, M. I. Jordan, and T. Kraska. Mli: An api for distributed machine learning. In *ICDM*, pages 1187–1192, 2013.

[38] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms. In *KDD '13: Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1–9. ACM Request Permissions, Aug. 2013.

[39] M. Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.