End-to-End Large Scale Machine Learning with KeystoneML

by

Evan Randall Sparks

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Michael J. Franklin, Co-chair
Professor Benjamin Recht, Co-chair
Professor Ion Stoica
Professor Joshua S. Bloom

Fall 2016

End-to-End Large Scale Machine Learning with KeystoneML

Abstract

End-to-End Large Scale Machine Learning with KeystoneML

by

Evan Randall Sparks

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Michael J. Franklin, Co-chair

Professor Benjamin Recht, Co-chair

The rise of data center computing and Internet-connected devices has led to an unparalleled explosion in the volumes of data collected across a multitude of industries and academic disciplines. This data serves as fuel for statistical machine learning techniques that in turn enable some of today's most advanced applications including those powered by image classification, speech recognition, and natural language understanding, which we broadly term *machine learning applications*.

Unfortunately, until recently the tools and techniques used to leverage recent advances in machine learning at the scales demanded by modern datasets, and thus develop these applications, have been available only to experts in fields such as distributed computing, statistics, and optimization.

I describe my efforts to render these tools accessible to a broader audience of application developers, and further demonstrate that by taking a holistic approach and capturing end-to-end high level specifications of machine learning applications the systems I present here can make novel, high impact optimizations to decrease resource consumption while simultaneously increasing throughput. These improvements are designed to decrease ML application development time, increase quality, and increase machine learning application developer productivity. I demonstrate the viability of these optimizations via experiments on a number of real-world applications in domains such as collaborative filtering, computer vision, and natural language processing.

Many of the ideas presented in this thesis have already had practical impact as embodied in the open source software packages KeystoneML and Apache Spark MLlib.

For my family.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to acknowledge the people who have supported me throughout graduate school.

My advisor, Mike Franklin, has been a perpetual source of inspiration, and encouraged me to dig deeper with every new finding. Professors Ameet Talwalkar and Benjamin Recht been close collaborators and worked with me through some of the hardest technical and academic challenges of my career. Professors Tim Kraska, Michael Jordan, Matei Zaharia, and Joseph Gonzalez have been supportive co-authors, and Joe Hellerstein, Alan Fekete, Ion Stoica, James Demmel, David Patterson, John Canny and Eugene Wu all provided invaluable feedback. Special thanks to Professors Franklin, Recht, Stoica, and Joshua Bloom for sitting on my dissertation committee.

My friends and colleagues in the AMPLab and at Berkeley have been numerous and have been fantastically helpful. Shivaram Venkataraman has been an amazingly supportive collaborator and friend. Daniel Haas has been pretty cool, too. Tomer Kaftan, Xinghao Pan, Ginger Smith, Jey Kottalam, Lisha Li, Kevin Jamieson, Eric Jonas, Zhao Zhang, Frank Nothaft, Hang Qi, Andre Wibisono, Sanjay Krishnan, Kay Ousterhout, Daniel Bruckner, Joshua Rosen, Vaishaal Shankar, Zongheng Yang, Gylfi Gudmundsson, Daniel Langkilde and Francois Belletti have been collaborators and all were fun to work with. Peter Bailis, Neil Conway, Peter Alvaro, Beth Trushkowsky, Michael Armbrust, Kristal Curtis, Sara Alspaugh, Dan Crankshaw, Stephen Tu, Ross Boczar all provided invaluable feedback, friendship, and conversations during my time here. Thanks to Brian Greenberg, Bill Ladd, Sean Smith, and Chris Bailey-Kellogg for helping me get here in the first place. I'd like to thank Pedro Rodriguez and Noah Golmant for allowing me to practice mentorship.

The AMPLab and CS department administrative and support staff has been a tremendous resource: Kattt Atchley, Carlyn Chinen, Boban Zarkovich, Shane Knapp, Matt Massie, Audrey Sillers and Jon Kuroda have all made my graduate career better.

This research has been supported in part by DHS Award HSHQDC-16-3-00083, NSF CISE Expeditions Award CCF-1139158, DOE Award SN10040 DE-SC0012463, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, IBM, SAP, The Thomas and Stacey Siebel Foundation, Apple Inc., Arimo, Blue Goji, Bosch, Cisco, Cray, Cloudera, Ericsson, Facebook, Fujitsu, HP, Huawei, Intel, Microsoft, Mitre, Pivotal, Samsung, Schlumberger, Splunk, State Farm and VMware.

Finally, my family has provided an endless well of support. They have endured late nights, awkward travel schedules, ups and downs, a cross-country move and tremendous opportunity cost, all for me to pursue my dreams. My parents, Alyse and Bill Sparks, my siblings, Emily and Kyle Martin, Jan and Andrew Sparks, and Emily and Mike Daigle, my grandparents Polly and Jack Connolly, Eric Galbraith, and Betty Lu and Robert Sparks, and the rest of my family have been tremendously supportive and set me up to achieve.

It will surprise nobody who knows her that my wife, Katie, is the kindest, most thoughtful, caring, supportive, and intelligent person I have ever known. She deserves credit for everything in this document. I hope that Charlotte and all of our future children one day read this and realize how awesome a person their mother is.

Chapter 1

Introduction

In recent years, computer applications that had been relegated to the realm of science fiction have become everyday technologies used in important industrial, commercial, scientific and consumer products. We can speak out loud to our telephones or speaker systems and have them follow our commands thanks to products such as Amazon's Alexa or Apple's Siri. Machines automatically categorize news stories and deliver customized reading experiences [46]. Recently, Google's DeepMind beat a top human player at the game Go, a feat that some experts had previously deemed impossible [12]. Self-driving cars with advanced obstacle and pedestrian avoidance systems have been deployed by some of the largest companies on the planet including Google, Uber, and Tesla. We owe many of these recent successes to a data revolution, and the use of advanced Machine Learning (ML) models fueled by the collection of unparalleled data volumes.

Unfortunately, each of the achievements above has come at the cost of heavy multi-year Research and Development investments, with teams of experts in ML and distributed computing required to build each application, often investing substantial resources in building custom software for each new challenge. Further complicating matters, the very data volumes that enable successful ML applications demand extensive use of storage, compute, and communication infrastructure when building these applications. It is not uncommon for state of the art models to require days or weeks of training in distributed computing environments, and an order of magnitude or more longer on single node systems. Even with the explosion of computational resources driven by cloud computing [11], mastery of the mathematical and computational techniques to build ML Applications at scale remains out of reach for all but the most savvy and well-funded organizations.

In this thesis I study techniques designed to lower this barrier to entry for large scale ML. Further, I have implemented several practical systems that exploit these techniques to decrease time to develop and deploy large large scale ML applications [142, 140, 141]. I now review the important economic, intellectual, and social trends that have led to the need for this work, and present an overview of the central abstractions developed in this work that drive the key results presented here.

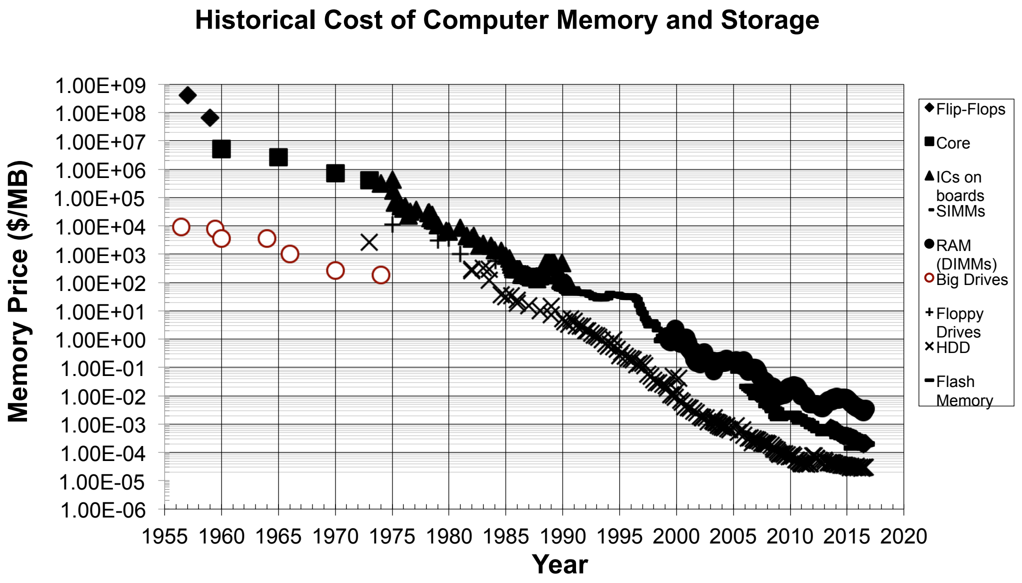


Figure 1.1: Cost of storage over time [109].

1.1 Datacenter Explosion

Economies of scale and other economic forces have caused industry to converge on a small number of dominant computer architectures.

Moore's Law, which served us well for nearly 50 years, has finally run its course [136]. Processing capacity measured by the maximum number of transistors per square inch on integrated circuits is no longer growing at an exponential rate. This fact, combined with the ever growing thirst for processing power has forced practitioners to explore both scale-out solutions (e.g. those that employ increased number of processing units) and scale-up (e.g. those that employ faster individual processing units) to support their applications.

A dominant scale-out solution is the *data center architecture*. This architecture is characterized by a number of commodity computers, each with multiple processing cores, fast transient memory (RAM), and a number of large, persistent storage devices (i.e. HDDs or SSDs), connected moderately fast networking in in a three tier or fat tree topology.

Coupled with the slow progress of Moore's law, the cost of persistent storage has plummeted over the last several decades, as shown in Figure 1.1.

It has never been easier or cheaper to store information on an unprecedented scale.

Despite these falling costs, growth in the bandwidth between compute nodes and their attached persistent storage has not kept up with growth in capacity, and has remained relatively constant over the last decade. This has led to the increasing use of distributed file systems such as the Hadoop Distributed File System (HDFS) [135], and shared-nothing distributed database systems [50] to support scale-out data processing.

Each of these trends has combined to make the datacenter architecture a predominant computer architecture inside major organizations in many industries, and it is the dominant architecture supported by cloud computing providers such as Amazon Web Services (AWS), Google Compute Engine (GCE) and Microsoft Azure.

Consequently, organizations have increasing access to distributed computing resources for their scaling needs, and this work focuses on scaling the training of ML applications using this architecture.

1.2 The Promise of Machine Learning

While collecting and storing data has never been easier, an interesting question is: How can this data can be used to enable scientific insight, better experiences for end users, or better decision making for businesses?

Indeed, today's advanced analytics applications increasingly use ML as a core technique in areas ranging from business intelligence to recommendation to natural language processing [104], speech recognition [77] and image classification [131].

ML methods take *training data* as input, and use that data to *learn* or *train* a *model* that maps data in the input space to predictions in some desired output space–text categories, phonemes, and image classes in the examples above. All else being equal, there is a direct relationship between the volume of input data and model quality, and an order of magnitude input data may be the difference between a model that produces acceptable predictions and one that does not [68].

The promise of machine learning methods is clear: given enough training data, which an organization may even collect as a byproduct of its usual business activities, machine learning promises to automatically infer a program that translates previously unseen data into *predictions* for some value of interest.

1.3 Machine Learning Pipelines

Simply stated, the promise of ML is that given enough training data and sophisticated enough learning algorithms, a computer should be able to automatically infer arbitrary functions mapping inputs of interest into a useful output space. The reality of using ML techniques in practice is substantially more complicated.

In practice, real ML applications can be broken down into multi-stage data processing *pipelines* involving feature extraction, dimensionality reduction, data transformations, and the training of supervised and unsupervised learning models to achieve high accuracy. An example pipeline for image classification based on the work presented by Coates, Lee, and Ng in 2011 [41] that includes model validation is shown in Figure 1.2.

In the Figure, training data, which has been collected and labeled by the developer, is first passed into several stages of data preprocessing, where data is mean-centered (Scale Estimator, Normalizer), *whitened* using a matrix factorization technique (ZCA, Whitener),

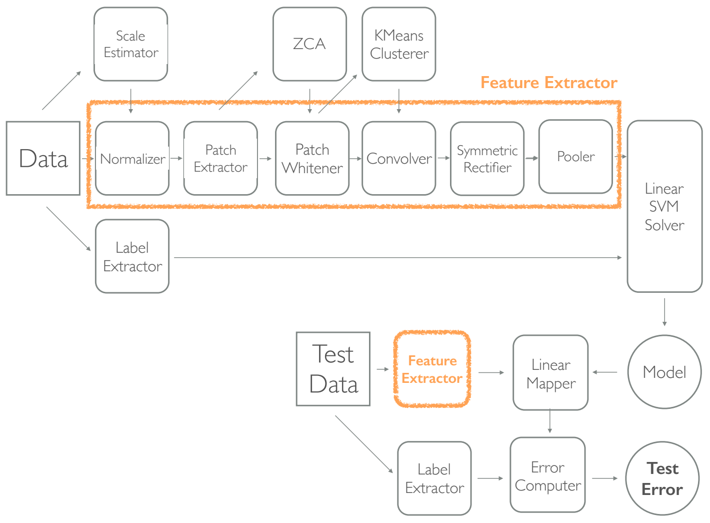


Figure 1.2: A “simple” image classification pipeline.

convolved with representative data samples (KMeans, Convolver), and has a non-linearity applied to it (Symmetric Rectifier) before being spatially aggregated (Pooler). Finally, the aggregated data is used as input to a linear Support Vector Machine (SVM) classifier.

Graphical depictions of the data flow such as those given in the Figure may be a convenient way for developers used to working with Extract, Transform, and Load (ETL) systems to understand the desired application behavior, but such a data flow can equivalently be written as the composition of mathematical functions. For example, let feats(Data), the output of the Pooler, denote the first argument to the SVM solver, and let labels(Data) denote the labels of the training data, which is the second argument to the solver. The `Model` that is emitted by the SVM solver, $x^\star$, can then be written as:

$$x^\star = \text{SVM-Solve}(\text{feats}(\text{Data}), \text{labels}(\text{Data})) \tag{1.1}$$

This equivalence between pipelines and functional expressions allows for whole-program analysis and optimization, which we will study in the context of ML in Chapter 5.

1.4 The Challenges of Large Scale ML

As data has become less expensive to collect and store, the size of training datasets available for ML applications has also grown. Given that ML applications perform better as their training sets grow, it is natural to want to *re-train* existing ML applications using these ever bigger datasets. Unfortunately, conventional solutions based on R, Python, or MATLAB are typically built to operate only on a single computer machine than on a scale-out architecture as described in Section 1.1. If training or intermediate data grows beyond available memory

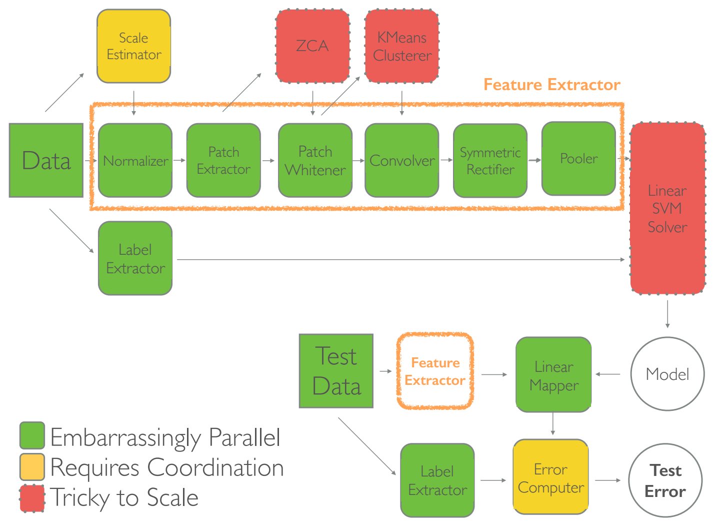


Figure 1.3: A "simple" image classification pipeline annotated by ease of scalability.

on such a machine, these environments may crash or slow down significantly as underlying virtual memory systems struggle to cope with the demanding load. Even well designed single-node systems will be fundamentally limited by the throughput capacity of secondary storage once training data reaches a certain scale. Consequently, I define *large scale learning* as a machine learning problem where time to solution benefits significantly from the use of resources beyond what is available on a single commodity workstation. This definition is more subtle than it appears at first glance, and I study the question of *when* to scale up in Chapter 4.

Earlier efforts at large scale ML have taken a straightforward approach [113, 73, 62, 153, 33]. Assuming that training an ML model is the main computational bottleneck in developing an ML application, a natural strategy is to examine the ML training algorithm in isolation and work to make it amenable to execution in a scale-out environment. As I discuss in further detail in the next chapter, this is the approach taken by earlier works, and it has been largely successful. However, these solutions are deficient for two main reasons. First, these existing solutions in general do not allow the developer to describe an end-to-end learning pipeline, instead offering scalability on a small handful of learning components. Second, as I show in Chapters 4 and 5, simply implementing scalable learning algorithms in isolation ignores several important avenues for performance improvement.

In Figure 1.3, the pipeline that was presented in Figure 1.2 has been annotated by highlighting the operators in the pipeline that require coordination among workers and those that can benefit from distributed execution in a scale-out setting. In the figure, operators that can be executed in a *data parallel* manner (that is, independently per input record) are highlighted in green. Those that require some minimal coordination (e.g. to compute simple sums or sufficient statistics of the data) are yellow. The operators that require significant coordination and possibly multiple passes of communication (e.g. estimating model parameters) are highlighted in red with dotted outlines.

Each stage may operate on some or all of the training or test data. Some stages are trivial to scale up in a datacenter environment. In general, such stages involve only operations that can work in parallel on individual data items in the training data set. Other operators may fundamentally require coordination. Few machine learning models, for example, can be accurately estimated on only subsets of the training data.

While scaling out ML operators is an important first step, a central argument in this work is that merely scaling out individual operators is not enough to scale out the training of ML applications, and that the strategy of scaling out ML algorithms in isolation potentially ignores significant opportunities for performance optimization.

1.5 Decisions, Decisions

Performance optimizations geared at accelerating the training of a *single* ML application also ignore an important practicality: the need for hyperparameter tuning. Informally, each operator in a machine-learning pipeline may have application-specific parameters that, in conventional systems, need to be set by the application developer. Such *hyperparameters* might include regularization penalties in regression or classification operators, strides or widths for convolution operators, and target dimensionality for dimensionality reduction operators. Figure 1.4 highlights the operators with hyperparameters in the simple image classification pipeline. I further note that even the *choice* of learning algorithm, say, KMeans vs. Gaussian Mixture Model (GMM) or SVM vs. Random Forest can be considered a hyperparameter, so the design space is even richer than what is illustrated here.

Little guidance exists for setting these parameters, and in practice *hyperparameter tuning* requires a search process where many instances of an ML application all with slightly different hyperparameter settings may be trained, and then the best one selected. The space of hyperparameter configurations grows exponentially with pipeline complexity, and so the practical limit to the number of applications that can be evaluated is often a developer's time or financial budget, or both.

In one major thrust of this thesis, I reduce the budget required for hyperparameter tuning by capturing the semantics of end-to-end ML applications and exploiting key properties of the application structure. For example, the system presented in Chapter 4 employs batching and early stopping techniques to make hyperparameter tuning up to 10× more efficient than sequentially executed pipelines. I also identify and exploit the fact that much of the computation across different hyperparameter settings of ML applications is redundant, and that by capturing end-to-end pipeline definitions for each *application instance*–a single application hypeparameter configuration–the system can capture this redundancy and eliminate it. I explore this optimization in depth in Chapter 6.

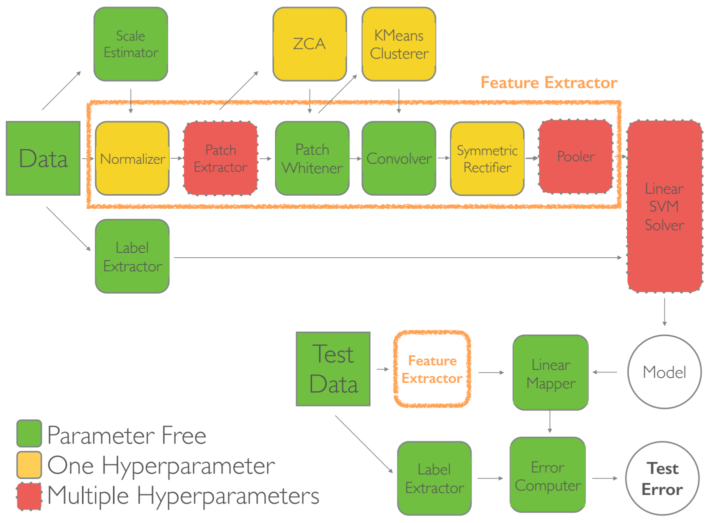


Figure 1.4: A "simple" image classification pipeline annotated with hyperparameters.

1.6 Thesis Overview and Contributions

Given the rise of the scale-out datacenter architecture, the promise of ML techniques on massive modern datasets, and the complexity of building ML applications, a natural question to ask is whether an ML application development environment can be constructed to simplify and expedite the process of large-scale ML application development. This question is the central topic of this thesis.

In this thesis, I make the following contributions [142, 140, 141]:

- I explore the challenges associated with building high quality ML applications on massive datasets in a conventional data center environment.
- I propose and evaluate novel declarative programming interfaces designed to hide the complexity of distributed computing from ML *algorithm* developers, and the complexity of both distributed computing and ML from ML *application* developers.
- By hiding this complexity, I open the door for extensive performance modeling and automatic optimization by the execution system at runtime that relies on knowledge of the end-to-end application. I propose performance models and optimization strategies for various aspects of these applications.
- I evaluate these proposals extensively via three main mechanisms: (1) micro-benchmarks designed to measure the performance of the system vs. theoretically justified limits, (2) ablation studies comparing the system with and without certain optimizations enabled, and (3) comparison with existing state of the art systems for large scale learning.

The ultimate aim of my work has been to lower the barrier of entry to building large scale ML applications. By separating ML application definition from its physical execution, the work of figuring out how to run ML at scale is offloaded from the ML application developers and pushed into the system. By capturing the end-to-end application specification, many sophisticated inter- and intra-application optimizations can be leveraged in the context of large scale learning.

1.7 Organization

The remainder of this thesis is organized as follows. Chapter 2 provides background on large scale ML application development and existing systems for large scale machine learning and automatic hyperparameter tuning.

Chapter 3 describes a programming interface designed to simplify the development and use of Machine Learning algorithms on Apache Spark. This interface combines ideas from relational database systems, popular data science platforms, and statistical programming environments. Using this interface, high quality algorithms that are competitive with existing systems for scalable ML can be written in tens of lines of code. Many of the ideas in this chapter formed the basis for Apache Spark MLlib, the standard library for machine learning in Apache Spark, and the implementations I describe served as the initial implementations in MLlib. Further, the concept of a distributed Table or Data Frame that is a key component of this interface has become a central developer-facing abstraction in Spark in more recent versions of the system.

Chapter 4 describes my work on large scale hyperparameter search for single-stage learning applications. I evaluate several state of the art methods for hyperparameter search, and draw the surprising conclusion that random search seems to work about as well as many more sophisticated methods on this problem. This finding has been echoed in subsequent works [98, 82]. This chapter further explores optimal compute cluster configuration and the use of early stopping and batching techniques to speed up this important step in the development of learning applications. By combining high speed search, early stopping, batching, and optimal cluster resource allocation the system achieves an order-of-magnitude improvement in throughput over a baseline that reflects standard practice.

Chapter 5 presents my work on combining several stages of data preprocessing, featurization, and model training into pipelines. I describe a system, KeystoneML that allows ML application developers to concisely specify learning pipelines using natural syntax at a logical level. Given such a specification, the system optimizes the execution of each operator in the pipeline and chooses among different physical implementations for each logical operator. Finally, the system performs end-to-end optimization of pipeline training to increase total throughput. I evaluate KeystoneML on several real-world workloads and demonstrate its ability to capture recent academic workloads concisely while delivering performance that exceeds the performance of existing state-of-the-art systems. I also study the impact of the optimizations on training throughput.

In Chapter 6 I explore the opportunities and challenges that arise when combining the hyperparameter tuning techniques from Chapter 4 with the pipeline designs of Chapter 5. In particular, I examine the opportunities for computational reuse that present themselves when certain hyperparameter tuning algorithms are employed. I then discuss the limits of these opportunities in the context of memory constraints, before describing the important problem of optimal cache scheduling in distributed dataflow systems. I propose a solution to this problem that employs mixed-integer linear programming before evaluating the effectiveness of greedy heuristics on finding an optimal solution.

Finally, Chapter 7 remarks on the successes and limitations of the current state of this work and discuss future directions for research on large-scale machine learning application development. It concludes with a summary of the main results of the thesis.

Chapter 2

Background

In this chapter I provide an introduction to the large scale ML application development process. I use this background to identify key challenges in this process and delineate the scope of this thesis, before discussing how my work relates to other research in the area of systems for large scale ML.

The central focus of this thesis is the study of systems for scalable ML that allow developers to describe machine learning applications using high level semantics without direct concern for how the underlying primitives supporting their applications are implemented. The separation of the description of the operators and application pipelines from their underlying implementation opens the door for optimizations such as those discussed in later chapters.

In this chapter I review the basics of applied machine learning, and describe how ML applications can be built using modular operators, along with some of the difficulties involved in doing so. I then discuss the computational challenges associated with scaling machine learning across commodity clusters. Finally I describe related academic and industrial efforts towards systems support for large scale ML application development.

2.1 Machine Learning Application Development

As with traditional software application development, the ML application development life cycle starts with a goal, called the *learning goal* in the context of ML. Example learning goals might include: a system that automatically categorizes text into a set of predetermined categories, a system that automatically recognizes pedestrians in video from a dashboard mounted camera, a system that finds high quality content recommendations, or a system that serves advertisements with the highest likelihood of generating revenue.

Given such a goal, the ML application developer then begins the ML application development process illustrated in Figure 2.1. At a high level, this process consists of four main stages: data collection, preprocessing-processing and featurization, model building, and deployment. Each stage may have feedback loops returning to any previous stage. For

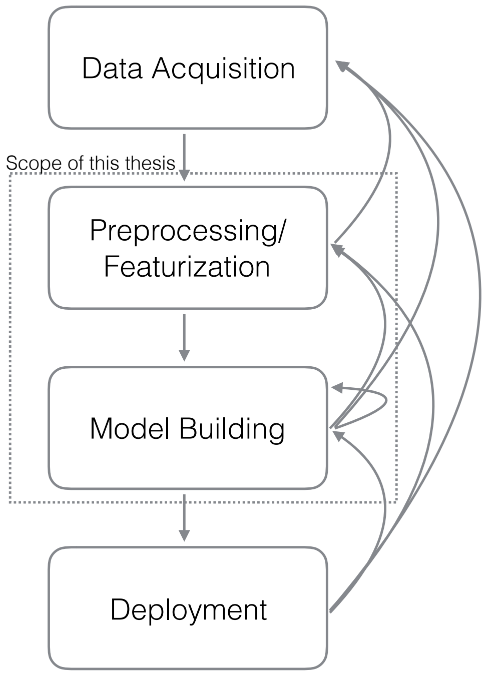


Figure 2.1: A high level view of the ML application development cycle.

example, results obtained during the model training process may indicate that more data needs to be collected. I next review each of the components in this process as they relate to the work presented in later chapters.

2.1.1 Data Collection

The first step in this process is the identification of a collection of datasets that provide suitable *training data* for the task at hand. In an ideal case, this training data is exhaustive, is perfectly suited to the task at hand, and is something the ML developer already has access to. In reality, this data may come from a number of possible sources: It may be manually collected from the Web and tagged by human workers (such as the ImageNet [18] dataset). It may be generated from simulation, as is the case in several autonomous vehicle applications [128]. It may also be a byproduct of existing work. For example, a search engine operator likely already has a large history of billions of mappings between search term and the resulting page a user clicked on–information that may be used to determine the best ads to serve to a particular user. Further, this data should be similar to what can be obtained when the application is deployed. For example, if the developer is building a content recommendation system based on a user's taste preferences, there must be some method of acquiring (or proxying) a *new* user's taste preferences when they first use the system.

In this thesis, I assume that the learning goal has already been identified and *training* *data* has been collected and put into a suitable format by the ML application developer. That is, if building an image classification system, the developer needs to have collected training images and associated labels and stored them in a database or file system. While this is a crucial step in the development of an ML application, its study is beyond the scope of this thesis.

2.1.2 Data Preprocessing and Featurization

Once data has been collected, and before data can be used as input to an algorithm that trains an ML model, it must be transformed from its native format (e.g. an image bitmap) into a *feature vector*, which is the conventional input to a learning algorithm. One natural question is why the raw bytes of an input file or database record are not a sufficient representation of such a feature vector.

First, per file in an input collection, feature vectors generated by such a process would likely be of different length because files are generally variable sized. This violates an important assumption that all the training features come from the same space. Second, the individual bytes may not be informative on their own. For example, it is unlikely that knowing that byte 334 of an image is "red" tells an algorithm very much about the content of an image. It is the "spatial context" of this byte that is important [115]. Thus, interesting combinations of the bytes in the file must be extracted and fed to the learning algorithm in order to give it this context, or knowledge of how to extract context must be encoded into the learning algorithm itself, as is the case in convolutional neural networks [97]. Further, many learning algorithms make other assumptions about their inputs. For example, SVMs typically assume that each input feature is on the same *scale* and thus it is common practice to make sure that each input feature in the training dataset has zero mean and unit variance.

As such, many domain-specific [102, 149, 59, 51] and general-purpose [156, 58, 119] feature extraction techniques have been developed to take raw data from variable-length and spatially-sensitive formats such as text, speech, and images and convert them into features amenable to training via a learning algorithm. Note, however, that one stage of featurization may not be enough–image features may be too big to feed to a conventional algorithm or may violate the same-scale assumption mentioned above.

Further, feature extractors may be combined. As a simple example, the popular SIFT descriptor assumes grayscale images are input and provides only information about edge data in the image. Most people would agree that color is also important information in many human visual tasks, and so color information, such as from the Local Color Statistic (LCS) descriptor, may also provide useful features.

As such, in may real world ML applications, developers may *compose* multiple feature extractors and data transformation steps in a row to form inputs to the learning algorithm, and they may *synthesize* features from multiple data sources or run multiple parallel feature extraction steps on the same input data.

To give a simple example, consider a very simple pipeline for image regression. Imagine a developer has n training images with associated real valued labels $b \in \mathbb{R}^n$. The data, $R \in$

$\{\text{images}\}^n$. Further, imagine the existence of a featurization function $f(.) \in \text{images} \Rightarrow \mathbb{R}^d$, and a standardization function, $s(.) \in \mathbb{R}^d \Rightarrow \mathbb{R}^d$, then featurized, standardized images can be expressed as:

$$A = s(f(R)) \tag{2.1}$$

The matrix, $A \in \mathbb{R}^{n \times d}$, is a collection of training features to be used as input to a machine learning algorithm.

2.1.3 Machine Learning Algorithms

Machine Learning, as a discipline, is at a basic level concerned with how, given *training data*, a machine can build a *model* to make better predictions or decisions about the world. While there exist a multitude of communities, algorithms, and techniques concerned with this goal in the fields of Computer Science and Statistics, all techniques share the property that the model is derived from training data.

Within the scope of machine learning, there are two important sub types: *supervised learning*, and *unsupervised learning*.

Supervised Learning

In supervised learning, a *learning algorithm* is given a training data set of *features* or *attributes* (typically $\in \mathbb{R}^{n \times d}$) and *training labels* (typically $\in 0, 1^{n \times k}$ in the case of multiclass classification or $\in \mathbb{R}^n$ in the case of regression) and asked to learn a *model* or *function* that will map new items from the same *feature space* to a prediction in the label space. n denotes the number of training examples, and d to denote the number of training features. For multi-class or multi-label problems k to denotes the number of distinct label values. A learning algorithm will generally identify the *model parameters* that minimize some *loss function* between the training examples and training labels. Common types of supervised learning tasks include classification tasks such as text categorization, image recognition, speech detection and regression that attempts to learn the mapping between input data and a real-valued number. While there are a multitude of models and algorithms available for these problems, common examples include linear regression and classification, Naive Bayes, Support Vector Machines, tree-based methods such as Decision Trees and Random Forests, and Neural Networks. The reader is referred to several resources on these methods [81, 115].

Perhaps the simplest model, linear regression, models the data in the following form:

$$b = Ax + \epsilon \tag{2.2}$$

Where $b \in \mathbb{R}^n$ are training labels, $A \in \mathbb{R}^{n \times d}$ are training features, $x \in \mathbb{R}^d$ are the *model parameters*, and $\epsilon \in \mathbb{R}^n$ are the *errors*–that is, the data that is unexplained by the model. Rearranging terms, it is simple to see that:

$$\epsilon = b - Ax \tag{2.3}$$

In Linear Regression, the loss function is least squares loss, that is, the objective is to find the $x^\star$ that minimizes squared error. Formally:

$$x^\star = \underset{x}{\text{minimize}} \quad l(x) = \underset{x}{\text{minimize}} \|b - Ax\|_2^2 \tag{2.4}$$

Using the notation for feature extraction above, the training objective of an end-to-end model for simple image regression might be specified as:

$$x^\star = \underset{x}{minimize} \quad \|b - s(f(R))x\|_2^2 \tag{2.5}$$

The conditions under which a linear model is an appropriate model choice are out of scope of this thesis, but the reader is referred to the discussion in [71] for more. I continue to use this simple model family throughout this chapter to illustrate the key issues explored in this thesis.

Unsupervised Learning

In unsupervised learning, a learning algorithm is given training data *without* training labels, and asked to find patterns among the data. This is a much less well-defined task as there may not be an obvious metric to use when evaluating the effectiveness of such algorithms (e.g. accuracy at predicting training labels). Common unsupervised learning tasks include clustering, dimensionality reduction, and low-rank factorization. Techniques are various and include K-Means, Principal Component Analysis (PCA), Gaussian Mixture Models, and Latent Dirichlet Allocation, among others.

Compound Learning Applications

In machine learning applications, developers frequently mix techniques at various stages of the learning pipeline. I call such applications *compound learning applications* For example, in the image classification pipeline described in the previous chapter uses unsupervised learning at two stages–for dimensionality reduction/whitening of image patches via ZCA, and later for soft-assignment clustering of these features to Fisher Vectors learned via Gaussian Mixture Models. Finally, the pipeline uses a linear SVM classifier to predict labels from the Fisher Vector features.

Other Kinds of Learning

There are, of course, further types of machine learning to consider, *semi-supervised learning* works on data that is only partially labeled and is often used in the important task of knowledge base construction [117]. *Reinforcement learning* typically comes up in the study of robotics and addresses how to learn a model when rewards or punishments are only occasionally observed. Closely related is the field of *active learning*, which analyzes explore-exploit tradeoffs in the context of collecting training data to train higher fidelity models, hyperparameter tuning, and other settings where data is ingested incrementally.

These last three types of machine learning are less widely used in practice than supervised and unsupervised learning techniques and are out of the scope of this thesis, but their incorporation into pipeline systems like KeystoneML (which I present in Chapter 5) is exciting future work. I instead focus on compound learning applications that use supervised and unsupervised techniques.

2.1.4 Deployment

Once an application is developed and the developer is comfortable that it will provide robust performance, it is *deployed*. In ML applications, this means that when raw, unlabeled data enters the system as input, features must be extracted from it in the same way they were extracted at training time. These features are then fed into the trained model to render predictions.

In the image regression example above, at test time, the system may output a prediction $\hat{b}_{\text{test}} \in \mathbb{R}$ based on test data $r_{\text{test}} \in \{\text{images}\}$, using the following:

$$\hat{b}_{\text{test}} = s(f(r_{\text{test}}))^\top x^\star \tag{2.6}$$

The KeystoneML system described in Chapter 5 allows developers to export their trained application and load it in a serving system at deployment time. This support is at present basic and provides no guarantees that the model will execute fast enough, e.g., meet service level agreements (SLAs), or degrade gracefully, or adapt to new data. These concerns present several interesting research challenges and are explored in related work [43].

2.1.5 Overfitting and Hyperparameter Tuning

Two common issues arise when developing machine learning applications in practice: *overfitting*, and the need for *hyperparameter tuning*. These issues represent the first set of *feedback loops*, as illustrated in Figure 2.1, that I study.

Overfitting

An important goal in building machine learning applications is to find models that *generalize* to previously unseen data. That is, models should make good predictions on *test data*. To give an extreme example, imagine that the "model" is simply a look-up table that has perfectly memorized every input example in the training set and its associated output label. If an item is not in the lookup table the system puts out a random answer. The accuracy of this model will be 100% on inputs that are in the training set, but the model will do no better than random guessing on test data. Conventionally, a model is said to be *overfit* if it does not generalize to test data.

A common way to estimate whether a model will generalize to unseen data is a procedure called *cross validation*. Subsets of the training data are input to the learning model, a model is trained, and then the model is evaluated on a held-out subset of the training data. This procedure is repeated until a sample of held out evaluations are collected. If the evaluations are sufficiently better than random, then a final model on all the training data is trained and that model is used going forward.

One common way to combat overfitting is via a technique called *regularization*. To provide a concrete example, imagine that the developer wants to predict a real value b from training features A using the model described above in Equation 2.2.

In situations where d (the dimensionality of the data) is very big relative to the size of an input dataset (e.g. a regression problem is *over-determined*), it is possible to find a map x such that each data example maps perfectly to an output, b, but without any guarantee that the x applied to new data items will correctly map to unseen data.

When regularizing the model, the loss function is modified by adding a penalty term, for example, Lasso regression minimizes the loss function:

$$x^\star = \underset{x}{\text{minimize}} \|Ax - b\|_2^2 + \lambda \|x\|_1 \tag{2.7}$$

Where $\lambda \in \mathbb{R} > 0$ is a *regularization parameter*. This new term penalizes $x^\star$ for having too many terms that are non-zero, and produces models that are simpler and generalize better despite having training loss that is at least as bad as the loss minimized in Equation 2.4. The reader is referred to [71] for a detailed discussion of this point. However, one obvious question, comes up: how does the developer set this new magic value, λ?

Hyperparameter Tuning

In the setting above, λ is a *hyperparameter*–a parameter to the model that is not inferred automatically from the data and instead must be set by the ML application developer. Given the above description of cross-validation, a natural approach would be to just try out a number of different settings for λ and inspect their cross-validated performance and pick the best one. This is common practice.

Unfortunately, a given learning pipeline may contain many such hyperparameters. In fact, a given ML model may have anywhere from 0 (e.g. Naive Bayes, Ordinary Least Squares) or a small handful (Lasso, Ridge, ElasticNet, Decision Trees) to tens of hyperparameters (e.g. architecture of a convolutional neural network) [19]. Further, each preprocessing stage may contain its own set of hyperparameters that must be set via trial and error. For example, the width of a Gaussian kernel in random feature projection, or the number of n-grams to use in text processing. Even discretized into a small number of potential values, the number of possible hyperparameter configurations grows exponentially with the number of parameters and trying them all out sequentially is infeasible.

Further complicating matters, the cost of evaluating a *single* hyperparameter configuration may be very expensive (up to days for modern neural network architectures), and so minimizing the raw number of hyperparameter configurations evaluated (and the time to evaluate each one) can provide substantial benefits in terms of both time to build an ML application and in terms of the cost of building such an application.

A central focus of this thesis is to determine how to solve the hyperparameter tuning problem in a resource-efficient way, which I consider in detail in Chapters 4 and 6.

2.1.6 Scaling Up Machine Learning

At the center of ML's recent success is the observation that, all else being equal, more training data makes models better [68]. A natural consequence of this fact is that ML application developers want to train models on ever increasing amounts of training data.

As stated in the introduction, a learning problem becomes *large scale* when it can benefit significantly from the use of distributed computing resources. I now motivate this definition with a brief discussion of large scale linear solvers.

Exact Solvers

So far, I have described machine learning applications in declarative mathematical terms. I have paid no attention to the problem of *finding* the optimal parameters ($x^\star$ in the running example). A machine learning algorithm or *solver* is responsible for finding such parameters. In some cases, such as the case of linear regression (Equation 2.4), the optimal value can be found via a closed form solution:

$$x^\star = (A^\top A)^{-1} A^\top b \tag{2.8}$$

For a matrix $A \in \mathbb{R}^{n \times d}$, the algorithmic complexity of this operation is $O(nd^2 + d^3)$. While this computation can be parallelized, for example, via Tall and Skinny QR decomposition (TSQR) [47], for sufficiently large values of n or d, the computational overheads of such an algorithm may be prohibitive.

Approximate Solvers

To cope with these problems, modern machine learning applications tend to use *approximate solvers* that find some approximate solution, in the example $\tilde{x}^\star$, where $|l(\tilde{x}^\star) - l(x^\star)| < \epsilon_{\text{approx}}$, where $l(.)$ denotes the loss function and $\epsilon_{\text{approx}} \in \mathbb{R}$ is the approximation error.

Fortunately, if the loss function is convex with respect to the model parameters, there are a host of methods [26, 22] that offer approximate solutions to these loss functions in time that is linear with the dataset size.

Perhaps the simplest such method is *Gradient Descent*, which operates by taking gradients of the loss function and iteratively updating an estimate of the model parameters. The gradient update is:

$$\tilde{x}^\star_{t+1} = \tilde{x}^\star_t - \eta \nabla l(\tilde{x}^\star_t) \tag{2.9}$$

This step is run iteratively for some fixed number of iterations or until $\tilde{x}^\star_{t+1}$ is not much different from its predecessor (i.e., it has *converged*). In the running example $\nabla l(\tilde{x}^\star_t) =$

$A^\top Ax - A^\top b$ can be computed in $O(nd)$ time, and so the total runtime is proportional to $O(ind)$ where i is the number of iterations of the algorithm. In the large scale learning regime this number will typically be much less than n or d unless the optimization problem is very poorly conditioned.

The iterative nature of solvers is something that is fundamental and common to many approximate solvers, and opens the door for computational reuse, a property that I exploit to increase throughput in subsequent chapters.

Distributed Solvers

While there is ample exciting work to be done in optimizing implementations of machine learning algorithms on single node machines [30, 127, 162], these solutions are naturally limited by the computational capacity and bandwidth of the machine they run on.

Distributed computing, and in particular computing over clusters of commodity machines in a datacenter environment, provides an exciting way to move beyond the limitations imposed by single node systems. Extensive study has been made of communication-avoiding distributed exact solvers [158] and more recent work has been done on analyzing properties of distributed approximate solvers [150].

In the big data regime, one common strategy for solving very large optimization problems, is to make use of *data parallelism* and use distributed processing technologies (e.g. Apache Spark [160]) to compute answers to sub problems individually. For example, the calculation of $\nabla l(x_t^\star)$ of the problem above can trivially be parallelized across w workers, leading to the computation of an update in $O(\frac{nd}{w})$ time, and workers must communicate $O(d)$ weights to a central master where they are aggregated, the gradient update is performed, and then they are sent out again. This communication may take $O(d \log_2 w)$ time to perform.

The overheads of both this communication and in coordinating the distributed computation may be non-trivial, but for large enough problems this strategy may provide near-linear speedups over running the same algorithm on a single machine.

In Chapter 3 we describe interfaces that can be used to describe distributed solvers and other kinds of learning algorithms using a distributed execution engine such as Apache Spark.

Picking a solver

Given the array of options for approximate, distributed, and local algorithms for solving machine learning problems, a natural question is: What is the right algorithm to use? The answer is: it depends. It depends strongly on data properties such as number of examples and features, but also on *sparsity.* In a distributed environment, it depends on the data's initial placement and partitioning, as well as the size of the compute cluster and relative speed of each node vs. their interconnect.

In Chapter 5 I explore this problem in detail and build a cost model not only for solvers but for other components of ML pipelines.

2.2 Related Work

There is a significant body of work related to large scale learning systems. Many of the foundational sources have been covered earlier in this section, but I review the most closely related work here.

2.2.1 Learning Systems

Some of the earliest efforts in computer science and numerical computing can be viewed as the development of computing environments designed to aid in the practice of statistical inference. For example, systems for high performance linear algebra [96, 9] and programming environments built on top of them such as MATLAB and R date back to the 1970s. While still popular for small scale problems, these systems have not been sufficiently adapted to make use of distributed computing resources, and do not provide sufficient tools to manage the end-to-end ML application development process.

More recent efforts include support for graphical models [100, 101], approximate solvers [64], and deep learning [83, 105]. While many of these systems provide support for learning models in a distributed fashion, they are in general focused on learning a particular class of models and support a single step in the ML application development process, and support for hyperparameter tuning is at best very basic.

Several systems [120, 21] are built to facilitate the construction of end-to-end learning applications, but so far few are designed to scale in a cluster environment.

2.2.2 Large Scale Learning

Several systems designed for large scale learning exist, and I subdivide them into two categories: *scale up* and *scale out.*

Scale up systems

Scale up systems attempt to increase the amount of data processed by machine learning algorithms by taking advantage of key trends in modern hardware: many core processing, large main memory systems, and general-purpose graphics processing units (GPUs). Such systems include: Bismarck [56], DimmWitted [162], TupleWare [44], BIDMach [30], Vowpal Wabbit [153], Shogun [134], LibLinear [55], the original GraphLab [100],and a host of GPU-backed learning systems including TensorFlow [105] and Caffe [83].

These systems have some attractive properties when compared with distributed systems. In particular, local interconnects are an order of magnitude or more faster than the fastest network interconnects, and so algorithms that are communication bottlenecked on distributed systems may not be on local systems.

On the other hand, fundamentally these systems are limited by their local physical resources, whether compute or I/O.

For example, suppose a developer wants to run a gradient step on of the linear regression problem discussed above. Fundamentally, this calculation comes down to running matrix-vector multiply between the data A and the current model $\tilde{x}_t^\star$. For a dense A matrix that fits in memory, on conventional architectures (e.g. x86) this operation is well known to be bottlenecked by the speed of main memory [111]. If A is partitioned across w servers, then each server can perform $|\frac{A}{w}|$ work while a single node system is forced to scan the entirety of A. In many practical cases, this leads to a speedup that is linear with the number of workers. Further, if the single node does not have enough memory to hold A in its entirety, then A must be read from secondary storage in a streaming fashion, which may make the algorithm orders of magnitude slower.

Scale out systems

Scale out systems, on the other hand, are designed such that they scale linearly with the amount of data by adding additional nodes in a distributed processing environment. Such systems have been studied in the database community since the 1980s [50, 24].

Many machine learning systems from the database community have been developed [73, 61, 123] according to shared-nothing principles, and more recent systems such as Mahout [10] and PowerGraph [101] are also built around this principle.

Spark ML [112] and MLlib [113] were directly influenced by the work done in this thesis, with the author contributing design, code, training and support to the Apache Spark project. However, both efforts represent an earlier snapshot of the ideas in this work as they developed. In particular, the focus on dynamic optimization of end-to-end learning is absent from either system, and support for hyperparameter tuning is nascent in both systems.

Existing systems generally constrain themselves to optimizations that are *local* to individual machine learning methods. That is–they do not take a holistic approach to optimizing logical large-scale ML applications as I do in this work.

The work in this thesis is primarily done in the scale out setting. However, I note that the scale-out and scale-up settings are not mutually exclusive. For example, the algorithms developed for this thesis make use of all machine cores, and make heavy use of large memories available on modern hardware. Careful attention was paid in this work to ensure that the per-node performance of each algorithm was within an acceptable margin of machine peak. This was achieved by being wary of Java Virtual Machine overheads and leveraging high performance libraries under the systems wherever possible. Where such libraries didn't exist, extensive profiling and hand-tuning of time-dominant algorithms was conducted.

2.2.3 Hyperparameter Tuning

As indicated in Section 2.1.5, hyperparameter tuning is a field of study in its own right and has been explored by the ML community extensively. Four main strategies exist: *Grid and Random Search*, *Derivative Free Optimization*, *Bayesian Optimization* and *Active Learning.* In this section I assume that for a given problem, a *hyperparameter space* $H \in \mathbb{R}^{d\times 2}$ can be defined over the d hyperparameters of a given problem. That is, a given hyperparameter configuration h, $H_{:,0} \leqq h \leqq H_{:,1}$. Assuming that there are resources to train a budget of $b \in \mathbb{Z}$ hyperparameter configurations, I now provide a high level overview of the operation of each of these algorithms.

Grid and Random Search

As discussed previously, Grid Search and Random Search are perhaps the most obvious algorithms for hyperparameter search.

Grid Search proceeds as follows: given H, select $\sqrt[d]{b}$ equally spaced points along each dimension and compute their Cartesian product. Cross-validate all b resulting hyperparameter configurations and return the one that the works best as measured by cross-validated accuracy or some other metric.

Random Search proceeds as follows: select b hyperparameter configurations from H uniformly at random. Cross-validate each configuration and return the one that works the best.

While these methods are simple, they are by far the most commonly deployed methods in practice. As I show in Chapter 4, there is perhaps good reason for this.

Derivative-Free Optimization

Derivative-free optimization [42] attempts to optimize functions that produce a single value given a real-valued vector input. Methods such as Nelder-Mead [116] and Powell's Method [122] have existed since the 1960s.

If the function of interest is the mapping of hyperparameter configuration to some statistic of interest (say, cross-validated test error), these methods can be used for hyperparameter tuning. Each of these methods has the general property that it assumes that the function it is optimizing is smooth and attempt to approximate a gradient from it by computing several points on the surface, and these assumptions may be unsuitable for the hyperparameter tuning setting.

Bayesian Optimization

Similar to derivative-free optimization, Bayesian Optimization attempts to learn the shape of a function using Bayesian methods. Commonly deployed methods include those based on Upper Confidence Bounds [143], Gaussian Processes [138], or tree-based methods [20, 78].

These methods differ from classical derivative-free optimization techniques in that they allow for *priors* on the space of allowable inputs, which may allow for faster learning of the hyperparameter response function.

Active Learning

Inspired in part by the work in Chapter 4 recently there has been an increase in interest from the Active Learning community in using these methods for hyperparameter optimization. Jamieson et. al. [82] and Li et. al. [98] study the hyperparameter tuning problem in theoretical depth and conclude (as I find, empirically) that under reasonable assumptions that Random Search is in expectation an optimal solution to this problem.

In this thesis, I present several techniques to speed up hyperparameter tuning based on better resource utilization and modeling (as in Chapter 4) and optimal computational reuse (Chapter 6).

2.2.4 Declarative Programming and Database Optimization

Database query optimization is a highly influential technique in the data management literature. The use of this technique in data processing systems dates back to the 1970s [133]. In traditional relational database management systems, programmers specify *queries* (programs) to the system in a declarative language (e.g. SQL)

The MLbase paper [88] described an architecture for a declarative system for large scale ML. Many of the ideas in that work are central to the rest of this study, in particular, efficient distributed algorithms, automatic operator selection, and automated hyperparameter tuning.

Other works [162] explore optimization opportunities in ML systems, while others still address the construction ML algorithms using declarative programming environments [154, 49].

SystemML [61] is a closely related system that optimizes the execution of declarative ML programs specified in an R-like syntax. The level of abstraction presented to the programmer by SystemML is too low level and consequently leaves substantial opportunities for optimization on the table. As a concrete example, I refer the reader to the comparisons with SystemML in Chapter 5.

In this work, I consider in depth the optimization opportunities that arise in distributed environments and leverage tools from the distributed systems community [160] as an execution layer for the higher-level programming primitives presented here.

2.2.5 Non-Goals: Generative Models, Tiny Data, and Deep Learning

While the goal of this thesis is to investigate systems support for large scale machine learning, I cannot possibly hope to cover *all* of this incredibly diverse field in a single study.

In particular, while *generative models*, such as those trained via Expectation Maximization and Gibbs Sampling are implemented and supported in KeystoneML (see, e.g. the study of GMM in chapter 5), the formal analysis in this work focuses on support for discriminative models. Generative models capture a full probabilistic model of all the variables in is model, while a discriminative model provides a model for the labels conditioned on the training data. However, the cost modeling and performance optimizations we discuss in subsequent chapters can apply to distributed algorithms for learning generative models.

Also absent from this thesis is a detailed review of *when* to use large-scale methods vs. methods that fit on a single machine. While the cost models used by KeystoneML can be applied to the latter case, they omit factors that capture important effects on single machine systems (e.g. local cache hierarchies and diverse local compute capabilities such as GPUs). However, the cost modeling framework I propose is general and can be extended to handle distributed and local computation on such resources.

One particular area that is not directly addressed in detail is the recently popular *Deep Learning*, which rose to prominence following success on image classification tasks using *convolutional neural networks*.

Chapter 5 compares KeystoneML to one system for convolutional neural networks. While it is interesting to think about extending KeystoneML to support these types of models, this support has limited for a number of reasons. First, Deep Learning is a recent trend and the majority of industrial use of ML is still focused on traditional methods. Second, Deep Learning models are in general non-convex, and thus fitting them using convex optimization routines (while perhaps empirically justified) is more art than science. For example, sensitivities of deep models to factors like batch size in minibatch stochastic gradient descent (SGD) optimization have been documented and currently popular models for computer vision such those presented by Krizhevsky et. al. [89] are not amenable to distributed training in commodity cluster environments. Recent works [114, 80] suggest that scalability for these popular models are severely limited by the speed of commodity networks, and even when using state-of-the-art Infiniband networks on models that are *designed* for scale out performance, their scalability as measured by time to convergence is significantly sub-linear.

As such, in this work I focus on traditional discriminative learning models that optimize convex loss functions with predictable communication and computation patterns.

2.2.6 Other Related Work

Systems support for ML at large scale is a topic of increasing interest, and while other works discuss solutions to similar problems to those studied in this thesis, they are not directly related. Other works such as Starfish [74] and Ernest [152] have studied performance modeling of MapReduce and Spark jobs in depth. While these works apply broadly to programs written against MapReduce clusters, they use black-box strategies to infer job execution costs, while I build cost models explicitly based on developer knowledge of the algorithms in use. This is possible in the restricted environment of ML algorithms under the APIs provided by the systems I present, but may not be generally feasible.

The caching strategy described in Chapter 5 can be viewed as a form of view selection for materialized view maintenance over queries with expensive user-defined functions [38, 72], I focus on materialization for intra-query optimization, as opposed to inter-query optimization [70, 35, 164, 52, 121]. While much of the related work focuses on the challenging problem of view maintenance in the presence of updates, in this thesis I exploit the iterative nature and immutable properties of this state.

Other work [161, 2] has looked at optimizing caching strategies and operator selection in the regime of feature selection and feature generation workloads. This thesis considers similar problems in the context of distributed ML operators and end-to-end learning pipelines. Developed concurrently to this work is TensorFlow [105]. While designed to support different learning workloads the optimizations that are a part of this work can also be applied to systems like TensorFlow.

The concept of using a high-level programming model has been explored in a number of other contexts, including compilers [95] and networking [85]. In this thesis I focus on machine learning workloads and propose node-level and end-to-end optimizations.

Chapter 3

MLI: An API for Distributed Machine Learning

Having set the stage for the large scale ML problem, I now describe key components of a system for large scale ML application development: a set of learning algorithms designed to run in a distributed setting on large scale datasets. Large scale learning algorithms that work on distributed data sets often share a set of code patterns and can make use of a common set of underlying data structures. In this chapter, I introduce MLI, an API for distributed machine learning designed to capture these common patterns and data structures and offer a programming environment amenable to the rapid implementation of large scale ML algorithms, such as the ones that are foundational sub-routines in future chapters.

3.1 Introduction

As described in Chapter 1, the recent success stories of machine learning (ML) driven applications have created an increasing demand for scalable ML solutions. Nonetheless, ML researchers often prefer to code their solutions in statistical computing languages such as MATLAB or R, as these languages allow them to code in fewer lines using syntax that resembles high-level pseudocode. MATLAB and R allow researchers to avoid low-level implementation details, leading to quickly developed prototypes that are often sufficient for small scale exploration. However, these prototypes are typically ad-hoc, non-robust, and non-scalable implementations. In contrast, industrial implementations of these solutions often require a high degree of development effort and are difficult to change once implemented.

The disconnect between ad-hoc scripts and the growing need for scalable ML, in particular systems that leverage the increasingly pervasive datacenter computing architectures discussed in Chapter 1, has spurred the development of several distributed systems for ML. Initial attempts at developing such systems exposed a restricted set of low-level primitives for development, e.g., MapReduce [10] or graph-based [100, 62] interfaces. The systems developed using these interfaces are often significantly faster and more scalable than MATLAB or

R scripts. They also tend to be much less accessible to ML researchers, as ML algorithms do not always naturally fit into the exposed low-level primitives, and moreover, efficient use of these primitives requires a fairly advanced knowledge of the underlying distributed system.

Subsequent attempts at developing distributed systems for ML have exposed high-level interfaces that compile down to low-level primitives. These systems abstract away much of the communication and parallelization complexity inherent in distributed ML implementations [61, 27, 145]. Although these systems can in theory obtain excellent performance, they are quite difficult to implement in practice, as they either heavily rely on optimizers to effectively transform high-level code into efficient distributed implementations, or utilize pattern matching techniques to identify regions that can be replaced by low-level implementations. The need for fast ML algorithms has also led to the development of highly specialized systems for ML using a restricted set of algorithms [134, 153], with varying degrees of scalability.

Given the difficulty that developers face when using low-level systems for large scale ML and the complexity inherent in developing for the high-level systems, ML researchers have yet to widely adopt any of the existing systems. Indeed, ML researchers, both in academic and industrial environments, often rely on system programmers to translate the prototypes of their novel, and often subtle, algorithmic insights into scalable and robust implementations. Unfortunately, there is often a ‘loss in translation’ during this process; small misinterpretation and/or minor errors are unavoidable and can significantly impact the quality of the algorithm. Furthermore, due to the randomized nature of many ML algorithms, it is not always straightforward to construct appropriate test-cases and discover these bugs.

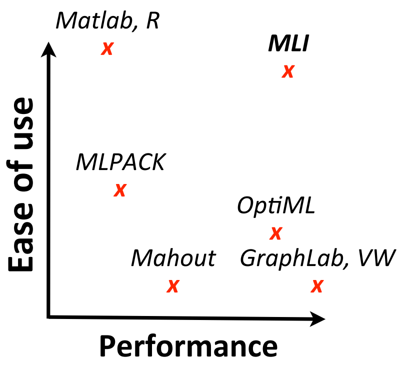


Figure 3.1: Landscape of existing development platforms for ML.

In this chapter, I present a novel API for ML, called MLI, to bridge this gap between prototypes and industry-grade ML software. I provide abstractions that simplify ML development in comparison to pure MapReduce and graph-based primitives, while nonetheless allowing developers control of the communication and parallelization patterns of their algorithms, thus obviating the need for a complex optimizer. MLI aims to be in the top right corner of Figure 3.1, by providing a development environment that is nearly on par with the usability of MATLAB or R, while matching the scalability of and approaching the walltime of low-level distributed systems. I make the following contributions in this chapter:

- **Syntax and Interfaces**: I show how MLI-supported high-level ML abstractions naturally target common ML problems related to data loading, feature extraction, model training and testing.
- **Usability**: I demonstrate that implementing ML algorithms written against MLI yields concise, readable code, comparable to MATLAB or R.
- **Scalability**: I describe an implementation of MLI on Apache Spark [160], a cluster computing system designed for iterative computation in a large scale distributed setting.

The results of performance experiments using logistic regression and matrix factorization in this chapter illustrate that MLI/Spark vastly outperforms Mahout and matches the scaling properties of specialized, low-level systems (Vowpal Wabbit, GraphLab), with performance within a small constant factor.

3.2 The MLI Interface

MLI promotes the design and implementation of developer-friendly, scalable algorithms. The interface consists of two fundamental objects – MLTable and LocalMatrix – each with its own API. These objects are used by developers to build Optimizers, which in turn are used by Algorithms to produce Models. It should be noted that these APIs are system-independent - that is, they can be implemented in local and distributed settings (Shared Memory, MPI, Spark, or Hadoop). The first implementation supporting this interface is built on the Spark platform.

These APIs help developers solve several common problems. First, MLTable assists in data loading and feature extraction. While other systems [153, 10, 62, 100] require data to be imported in custom formats, MLTable allows developers to load their data in an unstructured or semi-structured format, apply a series of transformations, and subsequently train a model. I address feature extraction more completely in Chapter 5.

LocalMatrix provides linear algebra operations on subsets of the fully featurized dataset. By breaking the full dataset into row-wise partitions of the original dataset and operating locally on those partitions, MLI gives the developer access to higher level programming primitives while still allowing them to control the communication that takes place between nodes and reason about the computational complexity of their algorithms.

As part of MLI, I also pre-define a set of common interfaces for Optimization, Algorithms, and Models to encourage code reuse and to ensure a consistent external system interface. In the remainder of this section I describe these abstractions in more detail.

3.2.1 MLTable

MLTable is an object that provides a familiar table-like interface to a developer, and is designed to mimic a SQL table, an R `data.frame`, or a MATLAB Dataset Array. The basic MLTable API is illustrated in Figure 3.2. An MLTable is a collection of *rows*, each of which conforms to the table's *column schema*. Each column is of a particular type, optionally has a name, and can be of the following basic types: String, Integer, Boolean, and Scalar (floating point numeric data). Importantly, any cell in the table can be "Empty" and this is represented with a special value. The table interface, which should be familiar to many developers, supports common operations like relational joins, unions, and projections - as well as map and reduce operations on rows that follow similar semantics to other MapReduce systems. Table 3.2 captures only core operations of the API and is not exhaustive.

Additionally, tables support batch operations on *partitions* of the data, which enable parallel data-local operation on multiple data items. While ML algorithms primarily expect numerical data as input, MLI exposes MLTable as an interface for processing the semi-structured, mixed type data that are present in real-world applications, and transforming this raw data into feature vectors for model training. Given this interface, developers are able to load structured data into an MLTable, and then apply a series of transformations to the data in parallel to produce input that is suitable for a ML algorithm. By supporting common data integration tasks out of the box in a straightforward and consistent manner, MLI significantly decreases the amount of time spent during data preparation and feature extraction.

Once data is featurized, it can be cast into an MLNumericTable, which is a convenience type that most ML algorithms will expect as input. The MLNumericTable interface is the same as MLTable, but it guarantees that all columns are numeric, and by convention each row will be treated as a single feature vector.

3.2.2 LocalMatrix

At their core, many ML algorithms are concisely expressed using linear algebra operations. For example, the update step in stochastic gradient descent for generalized linear models such as logistic regression, linear regression, etc. involves computing the gradient of a weight vector with respect to a test class and a training point. In the case of logistic regression, this is ultimately the dot product of two vectors, (or a matrix/vector multiplication in the case of mini-batch SGD), followed by a vector/vector subtraction.

LocalMatrix provides these linear algebra primitives but on *partitions* of data. The partitions of the data presented to the developer are typically automatically determined by the system. That is, MLI requires programmers to develop algorithms such that all operations can be performed locally and later combined via global `reduce` operations. This re-assembles to a large degree the shared nothing principle from distributed computing and often leads to highly scalable algorithms. I also considered exposing globally distributed linear algebra operations, but explicitly decided against it primarily because global operators

Operation	Arguments	Returns	Semantics
project	Seq[Index]	MLTable	Select a subset of columns from a table.
union	MLTable	MLTable	Concatenate two tables with identical schemas.
filter	MLRow ⇒ Bool	MLTable	Select a subset of rows from a data table given a functional predicate.
join	MLTable, Seq[Index]	MLTable	Inner join of two tables based on a sequence of shared columns.
map	MLRow ⇒ MLRow	MLTable	Apply a function to each row in the table.
flatMap	MLRow ⇒ TraversableOnce[MLRow]	MLTable	Apply a function to each row, producing 0 or more rows as output.
reduce	Seq[MLRow] ⇒ MLRow	MLTable	Combine all rows in a table using an associative, commutative reduce function.
reduceByKey	Int, Seq[MLRow] ⇒ MLRow	MLTable	Combine all rows in a table using an associative, commutative reduce function on a key-by-key basis where a key column is the first argument.
matrixBatchMap	LocalMatrix ⇒ LocalMatrix	MLNumericTable	Execute a batch function on a local partition of the data. Output matrices are concatenated to form a new table.
numRows	None	Long	Returns number of rows in the table.
numCols	None	Long	Returns the number of columns in the table.

Figure 3.2: MLTable API Illustration.

would hide the computational complexity and communication overhead of performing these operations. Instead, by offering linear algebra on subsets (i.e., partitions) of the data, MLI provides developers with a high level of abstraction while encouraging them to reason about efficiency.

Aside from the semantic difference that operations are performed on individual partitions, LocalMatrix is designed to resemble a `matrix` in MATLAB, R, or most other numerical programming environments. It supports indexing by rows, columns, or slices of each. A LocalMatrix also supports Matrix-Matrix and Matrix-Scalar algebraic operations, and common linear algebra routines like matrix inversion.

3.2.3 Optimization, Models, and Algorithms

In addition to MLTable and LocalMatrix, MLI provide additional interfaces called Optimizer, Algorithm, and Model.

Many models cannot be solved via closed form solutions, and even when closed-form solutions exist, the computational complexity of these solutions often increases super-linearly with data size, as in the case with basic linear regression. As a result, various optimization

Family	Example Uses	Returns	Semantics
Shape	dims(mat), mat.numRows, mat.numCols	Int or (Int,Int)	Matrix dimensions.
Composition	matA on matB, matA then matB	Matrix	Combine two matrices row-wise or column-wise.
Indexing	mat(0,??), mat(10,10), mat(Seq(2,4), 1)	Matrix or Scalar	Select elements or sub-matrices.
Reverse Indexing	mat(0,??).nonZeroIndices	Seq[Index]	Find indices of non-zero elements.
Updating	mat(1,2) = 5, mat(1, Seq(3,10)) = matB	None	Assign values to elements or sub-matrices.
Arithmetic	matA + matB, matA - 5, matA / matB	Matrix	Element-wise arithmetic between matrices and scalars or matrices.
Linear Algebra	matA times matB, matA dot matB, matA.transpose, matA.solve(v), matA.svd, matA.eigen, matA.rank	Matrix or Scalar	Basic and extended linear algebra support.

Figure 3.3: LocalMatrix API Illustration

techniques are used to converge to an approximate solution while iterating over the data. I treat optimization as a first class citizen in this API, and the system is built to support new optimizers. The reader is referred to the reference implementation for Stochastic Gradient Descent in Figure 3.6. For comparison, an implementation in MATLAB is shown in Figure 3.5.

Finally, MLI encourages developers to implement their algorithms using the Algorithm interface, which should return a model as specified by the Model interface. An algorithm implementing the Algorithm interface is a class with a `train()` method that accepts data and hyperparameters as input, and produces a Model. A Model is an object that makes predictions. In the case of a classification model, this would be a predicted class given a new example point. In the case of a collaborative filtering model, this might be recommendations for an existing user in the system. Both interfaces are rather simple, but crucially help to provide one common interface for developers (and to the MLBASE system as a whole).

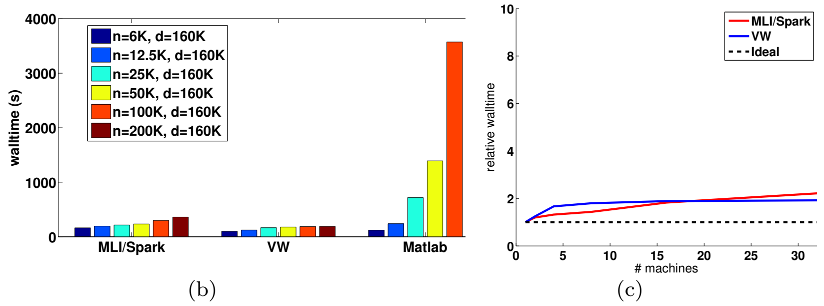
3.3 Examples

To evaluate the design claims made in the earlier sections, I evaluate MLI as well as competing ML systems on two representative real-world problems, namely binary classification and matrix factorization. When implementing algorithms against MLI, Spark was chosen as the first platform because it is well-suited for computationally intensive, iterative jobs on large datasets that are characteristic of large ML workloads. Moreover, many large-scale ML systems, e.g, [153, 62] do not emphasize fault tolerance. In contrast, Spark's resilience properties, due to automatic data replication and computation lineage, are quite attractive in a distributed environments where automatic recovery from node failure is a necessity. Given the choice to build on top of Spark, it was natural for the first implementation of the API to be in Scala.

The experiments illustrate three attractive features about MLI. First, I show that MLI yields concise and readable code. We compare the code length for comparable implementations of algorithms in MATLAB and MLI. Second, I argue that MLI supports a wide variety of algorithms. Although I focus on two problem settings, these examples demonstrate wide-ranging functionality of MLI and in fact naturally extend to a diverse group of ML algorithms, e.g., linear SVMs, linear regression, and (L1, L2, elastic net)-regularized variants therein, simply by changing the expression of the gradient function (and adding a proximal operator in the case of L1-regularization). Third, I demonstrate that the implementations written against MLI are performant and scalable. I present performance results comparing execution times of various systems on the two examples. I further present extensive strong and weak scalability results. Both sets of results show that the implementations in MLI match the scalability of low-level distributed systems with performance within a small constant factor.

System	Lines of Code
MLI	55
VW	721
MATLAB	11

(a)



(b) (c)

Figure 3.4: Logistic regression experiments. (a) Lines of code. (b) Execution time for weak scaling. (c) Weak scaling.

```
function w = log_reg(X, y, maxiter, learning_rate)
  [n, d] = size(X);
  w = zeros(d,1);
  for iter = 1:maxiter
    grad = X' * (sigmoid(X * w) - y);
    w = w - learning_rate * grad;
  end
end

% applies sigmoid function component-wise on the vector x
function s = sigmoid(x)
  s = 1 ./ (1 + exp(-1 .* x));
end
```

Figure 3.5: Logistic Regression Code in MATLAB.

```scala
object LRAlgorithm extends NumericAlgorithm[LRParameters] {
  def defaultParameters() = LRParameters()
  def sigmoid(z: Double): Double = 1.0/(1.0 + math.exp(-1.0*z))
  def train(data: MLNumericTable, params: LRParameters): LRModel = {
    val d = data.numCols-1

    def gradient(vec: MLVector, w: MLVector): MLVector = {
      val x = MLVector(vec.slice(1,vec.length))
      x times (sigmoid(x dot w) - vec(0))
    }

    //Run gradient descent on the data.
    val optParams = SGDParameters(
      wInit = MLVector.zeros(d),
      grad = gradient,
      learningRate = params.learningRate)
    val weights = SGDescent(data, optParams)

    new LRModel(data.toMLTable, params, weights)
  }
}
```

```scala
object SGD extends MLOpt with Serializable {

  def apply(data: MLNumericTable, params: SGDParameters): MLVector = {
    var weights = wInit
    var i = 0

    //Main loop of SGD. Calls local SGD and averages parameters.
    while(i < params.maxIter) {
      weights = data.matrixBatchMap(localSGD(_, weights, params.learningRate, params.grad)
          ).reduce(_ plus _) over data.partitions.length
      i+=1
    }
    weights
  }

  def localSGD(data: LocalMatrix, weights: MLVector, lambda: Double, gradientFunction: (
      MLVector, MLVector) => MLVector): LocalMatrix = {
    var localWeights = weights
    for (i <- data.toMLVectors) {
      //Compute the gradient and update the model.
      val grad = gradientFunction(i, loc)
      localWeights = localWeights minus (grad times lambda)
    }
    localWeights
  }
}
```

Figure 3.6: Logistic Regression Code in MLI (middle, bottom).

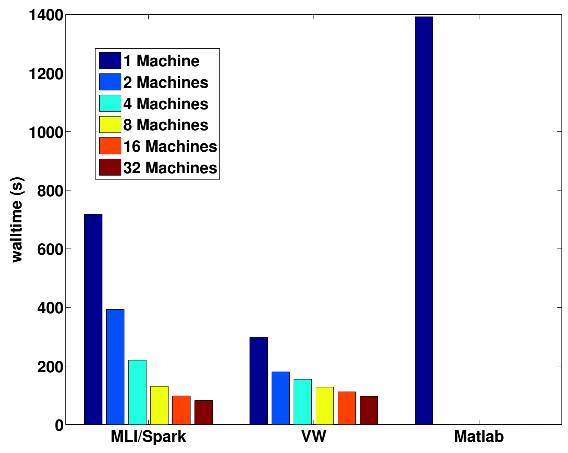


Figure 3.7: Execution time for strong scaling for logistic regression.

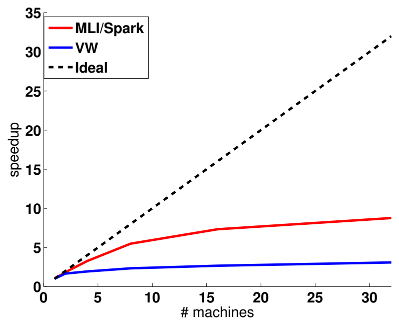


Figure 3.8: Strong scaling for logistic regression

3.3.1 Binary Classification: Logistic Regression

Let $X \in \mathbb{R}^{n \times d}$ be a dataset of n points with d features, $x_i \in \mathbb{R}^d$ be the ith data point, and define $y \in \{0,1\}^n$ as the corresponding set of binary labels. Logistic regression is a canonical classification algorithm. The optimal parameter vector $w^* \in \mathbb{R}^d$ 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:

$$\nabla f = \sum_{i=1}^{n} \left[\left(\sigma(w^\top x_i) - y_i \right) x_i \right], \tag{3.1}$$

where $\sigma(x) = 1/(1 + \exp -x)$ is the logistic sigmoid function. Gradient descent (GD) is a standard first-order iterative method to solve for w^*; at the tth iteration the algorithm moves in the direction of the negative gradient with step size controlled by a learning rate, η, i.e., set $w_{t+1} = w_t - \eta \nabla f$. Stochastic gradient descent (SGD) involves approximating the sum in

Equation 3.1 by a single summand.

Experimental Setup and Data

I ran both strong and weak scaling experiments on 1, 2, 4, 8, 16, and 32 machines. All are Amazon `m2.4xlarge` EC2 instances with 68GB of RAM and 8 virtual cores running in the `us1-east` region. They are configured using the default Spark 0.7.0 AMI and are running a recent version of Spark and Hadoop 1.0.4. I compare the system to version 7.2 of Vowpal Wabbit (VW) running on the same cluster, and MATLAB running on a similarly configured (single node) machine. I do not compare against Mahout for these experiments because its implementation of Logistic Regression via SGD is very communication intensive, and this implementation would not provide a fair comparison for Mahout.

I ran the weak scaling experiments on a training set of up to approximately 200GB of featurized ImageNet [18] data where each image is represented with 160K dense features, yielding approximately 200K images total for the 32-node experiment. The number of input points used is proportional to the number of nodes in the cluster for the experiment. I further note that this experiment only represents approximately 20% of the full ImageNet dataset. While I was able to train a full classifier using MLI in approximately 2.5 hours, the preprocessing required to prepare the data for VW on the full set of data was too onerous to complete the experiment. In the strong scaling experiments, I trained on 5% of this base data for the same number of nodes.

Implementation

I have implemented logistic regression via SGD. To approximate the algorithm used in VW [39] I ran SGD locally on each partition before averaging parameters globally. I note, however, that there are several alternative methods to implement SGD on top of MLI. Implementing Logistic Regression in MLI is as simple as defining the form of the gradient function and calling the SGD Optimizer with that function. Additionally, the code that implements *StochasticGradientDescent* is both short and fairly interpretable.

Algorithmically, the MLI implementation is identical to VW, with one meaningful difference, namely aggregating results across worker nodes after each round. VW uses an "AllReduce" communication primitive to build an aggregation tree when averaging together model parameters after each iteration. It then uses the same tree to broadcast these results back to workers. In MLI takes a more traditional MapReduce approach and average all parameters at the cluster's master node at each iteration, then broadcast the parameters to each node using a one-to-many broadcast. As the number of machines increases, VW's approach is theoretically more efficient from the perspective of communication and parallelizes better with respect to computation. In practice, scaling results are comparable as more machines are added.

In MATLAB, gradient descent was implemented instead of SGD, as gradient descent requires roughly the same number of numeric operations as SGD but does not require an inner loop to pass over the data. It can thus be implemented in a 'vectorized' fashion, which leads to a significantly more favorable runtime. I show MATLAB's performance here as a reference for training a model on a similarly sized dataset on a single multicore machine.

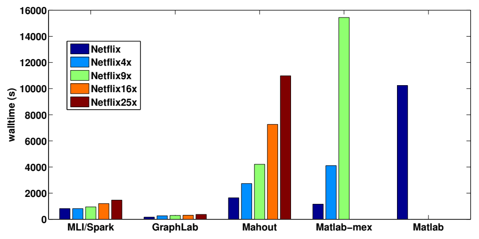
Results

In the weak scaling experiments (Figures 3.4b, 3.4c), one can see that the clustered system begins to outperform MATLAB at even moderate levels of data, and while MATLAB runs out of memory and cannot complete the experiment on the 200K point dataset, the system finishes in less than 10 minutes. Moreover, the highly specialized VW is on average 35% faster than MLI, and never twice as fast. These times *do not* include time spent preparing data for input input for VW, which was significant, but I expect that these would be a one-time cost in a production environment.

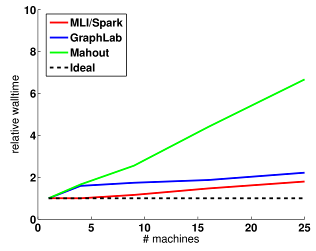
From the perspective of strong scaling the MLI actually outperforms VW in raw time to train a model on a fixed dataset size when using 16 and 32 machines, and exhibits better strong scaling properties, much closer to the gold standard of linear scaling for these algorithms. The cause is unclear.

System	Lines of Code
MLI	35
GraphLab	383
Mahout	865
MATLAB-Mex	124
MATLAB	20

(a)



(b)



(c)

Figure 3.9: ALS experiments. (a) Lines of code. (b) Execution time for weak scaling. (c) Weak scaling.

3.3.2 Collaborative Filtering: Alternating Least Squares

Matrix factorization is a technique used in recommender systems to predict user-product associations. Let $M \in \mathbb{R}^{m \times n}$ be some underlying matrix and suppose that only a small subset, $\Omega(M)$, of its entries are revealed. The goal of matrix factorization is to find low-rank matrices $U \in \mathbb{R}^{m \times k}$ and $V \in \mathbb{R}^{n \times k}$, where $k \ll n, m$, such that $M \approx UV^T$. Commonly, U and V are estimated using the following bi-convex objective:

$$\min_{U,V} \sum_{(i,j)\in\Omega(M)} (M_{ij} - U_i^T V_j)^2 + \lambda(||U||_F^2 + ||V||_F^2) \,. \tag{3.2}$$

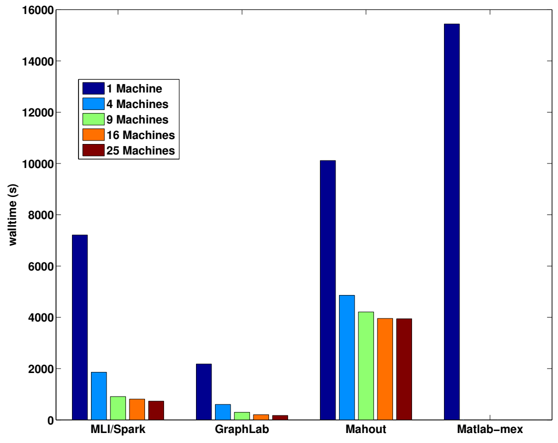


Figure 3.10: Execution time for strong scaling for ALS.

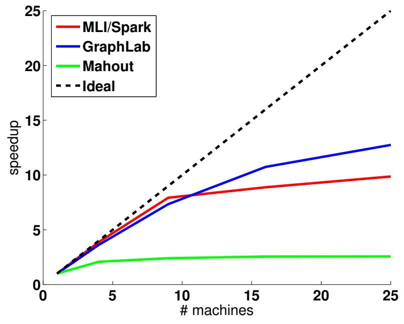


Figure 3.11: Strong scaling for ALS

Alternating least squares (ALS) is a widely used method for matrix factorization that solves (3.2) by alternating between optimizing U with V fixed, and V with U fixed, using a well-known closed-form solution at each step [87].

Experimental Setup and Data

I tested both strong and weak scaling experiments using 1, 4, 9, 16, and 25 machines with the same specifications as in the previous experiments. I ran the weak scaling experiments on a training set of up to approximately 50 GB of collaborative filtering data. This data is created by repeatedly tiling the Netflix collaborative filtering dataset. This allows us to maintain the sparsity structure of the dataset, and increase the number of parameters in a fixed manner. For weak scaling, the size of the dataset is proportional to the number of machines used in the cluster for the experiment. Thus, when running the largest experiment on 25 machines,

```
function [U, V] = ALS_matlab(M, U, V, k, lambda, maxiter)

  % Initialize variables
  [m,n] = size(M);
  lambI = lambda * eye(k);
  for q = 1:m
    Vinds{q} = find(M(q,:) ~= 0);
  end
  for q=1:n
    Uinds{q} = find(M(:,q) ~= 0);
  end

  % ALS main loop
  for iter=1:maxiter
    parfor q=1:m
      Vq = V(Vinds{q},:);
      U(q,:) = (Vq'*Vq + lambI) \ (Vq' * M(q,Vinds{q})');
    end
    parfor q=1:n
      Uq = U(Uinds{q},:);
      V(q,:) = (Uq'*Uq + lambI) \ (Uq' * M(Uinds{q},q));
    end
  end
end
```

Figure 3.12: Matrix Factorization via ALS code in MATLAB.

I used a dataset that is 25x the size of the Netflix dataset. In the strong scaling experiments, I trained on 9x the Netflix dataset, changing only the number of machines.

For both strong and weak scaling experiments, the following parameters are kept fixed. I ran ALS for 10 iterations, use a rank of 10, and set $\lambda = .01$. Training and testing error are not calculate training, but note that ALS methods from all systems achieved comparable error rates at the end of 10 iterations.

Implementation

ALS was implemented by updating the rows of U or V in parallel across machines, and then broadcasting the factors to each machine after each update. The system distribute both the matrix M and a transposed version of this matrix across machines in order to quickly access relevant ratings. The reference implementation makes use of several features of MLI, including support for CSR-compressed sparse representations of matrices, several linear algebra primitives, and heavy use of MLTable functionality. Linear algebra methods such as matrix transpose, matrix multiplication, and solving linear systems are supported. LocalMatrix also supports important access methods, such as the *nonZeroIndices*, which returns the nonzero column indices for a given row.

I compare MLI to the Mahout v0.6 and GraphLab v2.1 on the same cluster, and MATLAB running on a similarly configured machine. In addition, I tested a version of ALS in MATLAB using *mex*, an interface that allows MATLAB to call directly into C++/Fortran routines. Comparing MLI to these other implementations, one can see that the MLNumer-

```scala
object BroadcastALS {
  def train(trainData: MLTable, k: Int, lambda: Double,
        maxIter: Int): (LocalMatrix, LocalMatrix) = {
    val ctx = trainData.context
    val m = trainData.numRows
    val n = trainData.numCols
    val trainDataTrans = trainData.transpose
    val lambI = LocalMatrix.eye(k) * lambda
    // Initialize U and V matrices randomly
    val U0 = LocalMatrix.rand(m, k)
    val V0 = LocalMatrix.rand(n, k)
    (0 until maxIter).foldLeft((U0, V0))((UV, iterNum) => {
      val U = UV._1
      val V = UV._2
      // Broadcast V
      val V_b = ctx.broadcast(V)
      // Update U matrix
      val newU = computeFactor(trainData, V_b, lambI)
      // Broadcast U
      val U_b = ctx.broadcast(newU)
      // Update V matrix
      val newV = computeFactor(trainDataTrans, U_b, lambI)
      (newU, newV)
    })
  }

  def computeFactor(trainData: MLTable, fixedFactor: Broadcast[LocalMatrix],
        lambI: LocalMatrix): LocalMatrix = {
    trainData.map(localALS(_, fixedFactor.value, lambI)).toLocalMatrix
  }

  def localALS(trainDataPart: MLRow, Y: LocalMatrix, lambI: LocalMatrix) = {
    val tuple = trainDataPart.tuple
    val Yq = Y.getRows(tuple.nonZeroIndices)
    val resultMat = ((Yq.transpose times Yq) + lambI).solve(Yq.transpose times tuple.
        nonZeroProjection)
    resultMat.toVector
  }
}
```

Figure 3.13: Matrix Factorization via ALS MLI.

icTable and LocalMatrix objects provide convenient abstractions for patterns, thus resulting in concise code. Indeed, comparing Figure 3.12 and Figure 3.13 shows that the MLI implementation is about the same length as the MATLAB code, while Figure 3.9(a) shows the stark comparison in code length in comparison to Mahout and GraphLab.

Results

In the weak scaling experiments for ALS (Figures 3.9b, 3.9c), one can see that MLI outperforms MATLAB and the highly-optimized MATLAB-Mex, at even moderate levels of data. Both MATLAB and MATLAB-Mex run out of memory before successfully running the 16x or 25x Netflix datasets. MLI remains within 4x of the highly specialized system GraphLab, and maintain a similar scaling pattern. MLI outperform Mahout both in terms of total execution time for each run and scaling across cluster size.

MLI achieves similarly promising results with the strong scaling experiments, with MATLAB running out of memory before completing on the 9x Netflix dataset, and GraphLab outperforming MLI by less than a factor of 4x.

3.3.3 Configuration Considerations

Although I ran all of the experiments presented here on comparable or identical hardware, different software systems varied drastically in terms of ease of installation, configuration, and executing code.

Vowpal Wabbit

To use VW in cluster mode, developers must carefully partition their datasets into equally sized compressed files of training data, where the total number of files should equal the number of map tasks that the developer desires to use concurrently on the cluster. Although VW uses Hadoop Streaming to launch cluster tasks, it eschews the traditional MapReduce paradigm in favor of AllReduce. To support this new communication primitive, it must open a side-channel TCP socket between map tasks to communicate incremental results. The combination leads to a failure-prone system as well as difficulty in data preparation.

Mahout

Mahout is fairly easy to set up on an already existing Hadoop cluster, and its input file formats are reasonably close to those used in MLI. However, in order to run Mahout effectively on problems larger than the traditional Netflix dataset, the developer must take great care to tune job memory configuration parameters correctly to ensure that jobs complete in a performant manner.

GraphLab

While GraphLab performed very well in the speed and scalability tests, it was rather difficult to set up and integrate with an existing cluster with distributed data stored in HDFS. In order to set up GraphLab, developers must configure their clusters with MPI, download, build and install GraphLab and its required dependencies, and manually copy the software to each machine on the cluster. If a single input matrix will not fit into memory, it must be stored and loaded as multiple separate files. This complicates preprocessing and requires developers to take extra steps depending on their problem sizes.

MLI and Spark

Setting up and configuring the system is comparatively easy. Launching a well configured cluster required a single command, and the software ships with all its dependencies listed in

SBT, and can be compiled and run on a cluster simply by setting a few environment variables and running one Scala program. New algorithms can be easily added to the system as new Scala classes, and driver programs are easily generated based on examples in the existing library.

3.4 Related work

The widespread application of ML techniques has inspired the development of new ML platforms with focuses ranging from productivity to scalability. In this section we review a few of the representative projects. It is worth nothing that in many cases MLI is inspired and guided by these earlier efforts.

Systems like MATLAB and R pioneered high-productivity numerical computing. By combining high-level languages tailored to application domains with intuitive interfaces and interactive interpreted execution, these systems have redefined the way scientists and statisticians interact with data. From the database community, projects like MADLib [73] and Hazy [93] have tried to expose ML algorithms in the context of well established systems. Alternatively, projects like Weka [155], scikit-learn [120] and Google Predict [63] have sought to expose a library of ML tools in an intuitive interface. However, none of these systems focus on the challenges of scaling ML to the emerging distributed data setting.

High productivity tools have struggled to keep up with the growing computational, bandwidth, and storage demands of modern large-scale ML. Although both MATLAB and R now provide limited support for multi-core and distributed scaling, they adopt the traditional process centric approach and are not well suited for large-scale distributed data-centric workloads [151, 107]. In the R community, efforts have been made to run R on data-centric runtimes like Hadoop [124], but to my knowledge none have obtained widespread adoption.

Early efforts to develop more scalable tools and APIs for ML focused on specific applications. Systems like liblinear [55], Vowpal Wabbit [153], and Shogun [134] initially focused on linear models, online learning, and kernel methods, respectively. Others, like MLPack [45], started to develop entire collections of learning algorithms optimized for multicore architectures. These efforts lead to highly efficient systems for specialized tasks, but do not directly simplify the design and implementation of new scalable ML methods, and most are not well-suited to distributed learning.

Various methods have leveraged MapReduce platforms like Hadoop to develop distributed ML libraries. Mahout [10] does not simplify the design and development of new ML methods, and its reliance on HDFS to store and communicate intermediate state makes it poorly suited for iterative algorithms. SystemML [61] introduces a low-level algebra that it then compiles to MapReduce jobs. This algebra exposes the opportunity for advanced optimization, but also complicates the system, and SystemML also suffers from Hadoop's limitations on iterative computation.

Others have sought to generalize the MapReduce computational model. Systems like DryadLinq [159] and Hyracks [25] can efficiently execute complex distributed data-flow operations and express full relational algebras. However, these systems expose low-level APIs and require the ML expert to recast their algorithms as dataflow operators. In contrast, GraphLab [62] is well suited to certain types of ML tasks, its low-level API and focus on graphs makes it challenging to apply to more traditional ML problems. Alternatively, OptiML [145] and SEJITS [32] provide higher level embedded DSLs capable of expressing both graph and matrix operations and compiling those operations down to hardware accelerated routines. Both rely on pattern matching to find regions of code that can be mapped to efficient low-level implementations. Unfortunately, finding common patterns can be challenging in rapidly evolving disciplines like machine learning.

3.5 Conclusion

In this chapter, I have presented MLI, an API for building scalable distributed machine learning algorithms. I have shown that its components, MLTable and LocalMatrix, are useful primitives for data loading and transformation as well as data-local linear algebra operations. I have shown how these primitives can be used to code two fairly different but representative algorithms. I evaluated these algorithms in terms of both ease-of-development and computational performance, based on an implementation of MLI against Spark, comparing the system with several existing ones. The results show that MLI provides ML developers the tools to construct high performance distributed ML algorithms without onerous programming complexity. MLI is a foundational layer that informed the design and implementation of the other components of this thesis.

While this work was carried out early in the exploration of this thesis, it has had significant impact on the core abstractions for Machine Learning in Apache Spark. In particular, the Algorithm/Model abstractions proposed here were included in Apache Spark v0.8 as part of the initial release of MLlib, and the initial versions of both Generalized Linear Models (e.g. Logistic Regression) and Alternating Least Squares for Collaborative Filtering within MLlib were based on the implementations described here. Further, in subsequent releases of Apache Spark, MLlib was integrated with the DataFrame API–a declarative interface which allows programmers to use both relational operators, map-reduce operators, and statistical operators on collections of DenseVector or Matrix objects–which is identical in spirit to the MLTable API.

I next focus on the problem of efficiently hyperparameter tuning large-scale ML algorithms implemented using MLI as an underlying framework for algorithm development.

Chapter 4

TuPAQ: Automating Model Search for Large Scale Machine Learning

In the previous chapter, I presented a simplified programming interface for ML algorithm developers and ML application developers. This interface enables developer to write ML algorithms and also use off-the-shelf algorithms and run them scalably in a datacenter environment. However, little guidance has been proposed thus far on *which* model family to use or how to configure these models to produce a high quality models. Ultimately, decisions about choice of model family and algorithm configuration are *problem-dependent*, and in practice many developers simply take a pragmatic approach–that is, they use whatever combination of model family and settings works best on their problem. This pragmatic approach is often inefficient, and leaves substantial room for improvement in terms of decreased time to solution. In this chapter, I explore the process of hyperparameter tuning in the context of model training and describe several optimizations that, in concert, can speed up hyperparameter tuning by an order of magnitude compared to common practice.

4.1 Introduction

As I described in Section 1.5, to develop 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 that 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. I return to this challenge in Chapters 5 and 6.

Once features have been engineered, an ML application developer must make several other important decisions. They must pick a learning setting appropriate to their problem—for example, regression, classification, or recommendation. Next, the developer 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

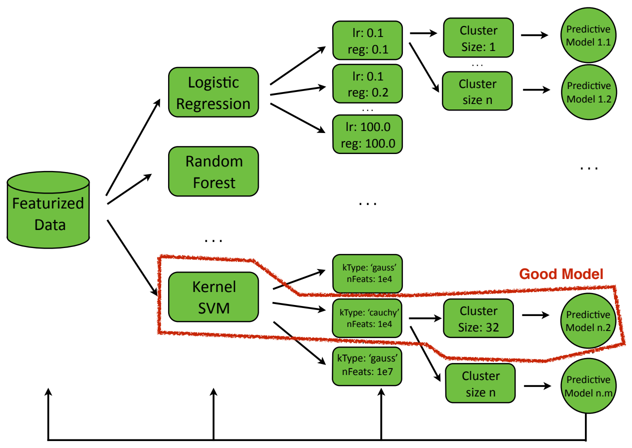


Figure 4.1: Illustration of the model search process.

each of these must be tuned to an appropriate value. Finally, the developer 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. The initial model configuration selected by the developer 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 4.1. 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, a system designed to efficiently and find and train high quality predictive models, automates this process. In the figure, '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.

As I argued in previous chapters, distributed and cloud computing provide a compelling way to accelerate this process, but also present additional challenges. Though parallel storage and processing techniques enable developers 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 developers: 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 developers who are experts in their own field but inexperienced in machine learning and distributed systems.

Existing techniques to automate the search for high-quality predictive models focus on the single node setting; extensions for large-scale problems or distributed settings are very basic [86]. 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, I present TuPAQ (short for **T**raining s**u**pported **P**redictive **A**nalytic **Q**ueries). Central to the system is a planning algorithm that 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 that are unlikely to provide good results.

In this chapter, I make the following contributions:

- I introduce a simple, workload-driven cluster size estimator that determines the appropriate number of machines to use when fitting large-scale ML models.
- I describe the TuPAQ algorithm for large scale model search that combines advanced hyperparameter tuning techniques with physical optimization for efficient execution.
- I describe an implementation of the TuPAQ algorithm in Apache Spark, building on earlier work on the MLbase architecture [88].
- I evaluate several points in the design space with respect to each logical and physical optimization, and demonstrate that proper selection of each can dramatically improve both accuracy and efficiency.
- I 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 chapter, I formally define the model search problem, and provide a high level overview of TuPAQ and its architecture. Next, I present details about TuPAQ's four main optimizations. Then, I present a large-scale evaluation of TuPAQ. I conclude with a discussion of related and future work.

4.2 Model Search and TuPAQ

In this section, I define the model search problem in more detail, and compare two approaches to solving it. The first, which I 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 I describe in further detail in Section 4.3. Finally, I describe the architecture of TuPAQ and how it fits into the broader MLbase architecture.

4.2.1 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. I focus specifically on the supervised learning setting, where each element of the training dataset has a label or score associated with it. In TuPAQ's environment, the developer'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.

4.2.2 Problem Setting

In this chapter, I assume 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 the large-scale speech experiments and $d = 160,000$ in the large scale image experiments presented in Section 4.5. The work is focused on the situation where the data size ($n \times d$) is very large. Section 4.5 reports results from experiments on up to terabytes of training data and TuPAQ is designed to scale even further. Further, this chapter is focused on the classification setting, and I consider only a small number of model families, $f \in F$, each with several hyperparameters, $\lambda \in \Lambda$. Additionally, TuPAQ is focused on model families that are trained via multiple sequential scans of the training data as opposed to model families that require random access to training data to estimate their parameters.

These assumptions map well to reality, as there are only a handful of general-purpose classification methods that are typically deployed in practice. Further, it is reasonable to expect that these techniques will naturally apply to other supervised learning tasks—such as regression and collaborative filtering. For example, a variant of both the batching optimization described here and the early stopping mechanism is implemented in the regression tree algorithm in Apache Spark MLlib. Further, many of the techniques I describe here have been validated in subsequent work [98]. The iterative sequential access pattern encompasses a wide range of learning algorithms [56], especially in the large-scale distributed setting. For instance, efficient distributed implementations of linear regression, tree based models, Naive

Bayes classifiers, and k-means clustering all follow this same access pattern [113, 118]. In particular, I focus on three model families: linear Support Vector Machines (SVM), logistic regression trained via gradient descent, and nonlinear SVMs using random features [126] trained via block coordinate descent. These model families were chosen primarily because of their wide adoption, as well as the ease with which the batching optimization described in Section 4.3 can be applied to them.

The quality of each model is evaluated by computing accuracy on held-out datasets, and search time is measured as the amount of time required to explore a fixed number of models from some model space. This accuracy measure is the measure of model quality used by the system to pick a high quality model. In the large-scale distributed experiments (see Section 4.5) runtimes are reported in parallel.

4.2.3 Connections to Query Optimization

Model search can be considered as a form of query optimization. If one views model search as a task that 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, I turn my attention to scalable model search strategies.

4.2.4 Baseline Model Search

The conventional approach to model search is sequential grid search [65, 120, 91], 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 developers 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—I 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 developer of the application, and is not dynamically made by the search procedure. The intuition is that when data no longer fits in memory on a single node, it makes sense to operate in the data parallel setting. I validate this intuition in Section 4.4.

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 I compare TuPAQ.

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

4.2.5 TuPAQ Model Search

As discussed in Section 4.2.4, 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, I propose the TuPAQ algorithm, described in Algorithm 2, to address these shortcomings via logical and physical optimizations. In the figure, '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 ex-

```
    input  : LabeledData, ModelSpace, Budget
    output: BestModel
 1  bestModel ← ∅;
 2  grid ← gridPoints(ModelSpace, Budget);
 3  while Budget > 0 do
 4      proposal ← nextPoint(grid);
 5      model ← train(proposal, LabeledData,);
 6      if quality(model) > quality(bestModel) then
 7          bestModel ← model;
 8      end
 9      Budget ← Budget − 1;
10  end
11  return bestModel;
```

Algorithm 1: Pseudocode for conventional grid search.

hausted. 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 the TuPAQ model search procedure can now use training history as input. Here, "proposeModel" can be an arbitrary model search algorithm. Second, the TuPAQ algorithm performs *batching* to train multiple models simultaneously (Line 8). Third, the TuPAQ 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 4.3, with a focus on the design space for each of them. In Section 4.4, I evaluate the options in this design space experimentally, and then in Section 4.5 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 4.4.

I now explore the design space for the TuPAQ algorithm.

TuPAQ Architecture

TuPAQ is designed according to the principles of modularization and reuse. In particular, it consists of several components that adhere to an abstract interface. The interaction of these components is illustrated in Figure 4.2.

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* that is responsible for actu-

```
   input  : LabeledData, ModelSpace, Budget, BatchSize
   output: BestModel
1  (bestModel, history, activeProposals) $\leftarrow \emptyset$;
2  searcher.setSearchSpace(ModelSpace);
3  executor.determineExecStrategy(LabeledData, Budget) // Execution Strategy
4  while $Budget > 0$ do
5      freeSlots $\leftarrow$ BatchSize $-$ size(activeProposals);
6      activeProposals $\leftarrow$ activeProposals $\cup$ searcher.proposeModels(freeSlots, history)
           // Hyperparameter Tuning
7      (models, budgetUsed) $\leftarrow$ executor.trainPartial(activeProposals, LabeledData)
           // Batching
8      (finishedModels, activeProposals) $\leftarrow$ banditAllocation(models, history)
           // Bandits
9      history $\leftarrow$ history $\cup$ models;
10     Budget $\leftarrow$ Budget $-$ budgetUsed;
11 end
12 bestModel $\leftarrow$ getBestFromHistory(history);
13 return (bestModel);
```

Algorithm 2: Pseudocode for the planning procedure used by TuPAQ.

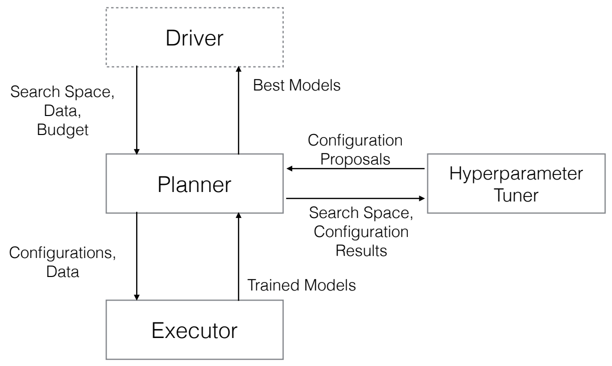


Figure 4.2: TuPAQ is composed of several components, each of which has a standard interface and may have multiple implementations.

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

Table 4.1: A listing of hyperparameter types supported by TuPAQ.

```
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 4.3: An example search space definition.

ally training models given a handle to the developer'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.

Developers define their search space using an extensible API. Table 4.1 lists the data structures used to represent parameters and available parameter types. Developers define parameter spaces by composing potentially nested parameters of various types.

Figure 4.3 shows an example search space defined using the API. Using the small set of parameters from Table 4.1, developers are able to construct rich parameter search spaces. In the figure, the developer supplies parameter ranges to try for Random Forests or Logistic Regression models.

While the system currently expects developers 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), leaving the developer oblivious to the methods and parameters used to fit their models.

Now that I have discussed the architecture of the system, I turn my attention to its design space.

4.3 TuPAQ Design Choices

In this section, I examine the design choices available for the TuPAQ model search procedure. TuPAQ targets 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.

I new describe how system and algorithms presented in Section 4.2 can be optimized to support fast, high quality search. In the remainder of this section, I present the following optimization four optimizations that in concert provide TuPAQ with an order-of-magnitude gain in performance over the baseline approach.

4.3.1 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, data parallel execution 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, I 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.

Estimating Job Latency

Data parallel execution is the natural choice for model search if the compute requirements of the model training become excessive or the data required to train a single model no longer fit in memory. For systems operating in the data parallel setting, one must choose a 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 [6] provides an upper bound on the speedup one 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.

These effects can be captured in the following cost-based model of cluster job execution time:

$$t_{job}(w) = k_0 + k_1 w + \mathbb{I}_{cpu} k_2 \frac{c}{w} + \mathbb{I}_{mem} k_3 \frac{m}{w} + \mathbb{I}_{net} k_4 \frac{b}{w} \tag{4.1}$$

Equation 4.1 describes the TuPAQ estimator, a function of w, the number of worker nodes in the cluster, which encapsulates several important insights. First, the estimator explicitly separates 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 the TuPAQ model easy to use in practice: developers 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. I show an example of how these parameters may be estimated in Section 4.4.1.

In addition, the estimator is based on a roofline [158] model of the tradeoff between job-specific memory and compute requirements. That is, the system asks 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 $\mathbb{I}_{net}$. TuPAQ estimates 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, the TuPAQ 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, TuPAQ estimates 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, experimental results reported in Section 4.4 show that it estimates execution time for data-parallel iterative machine learning jobs reasonably well, making it useful for TuPAQ's model search task. I now describe the use of the TuPAQ model to determine the size of clusters, and will evaluate its use in Section 4.4.

Right Sizing the Cluster

The primary purpose of the TuPAQ 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, one can easily find a closed-form solution by differentiating with respect to w and finding the roots:

$$w^* = \sqrt{\frac{\mathbb{I}_{cpu} k_2 c + \mathbb{I}_{mem} k_3 m + \mathbb{I}_{net} k_4 b}{k_1}}\,. \tag{4.2}$$

The solution in Equation 4.2 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 the TuPAQ model causes there to be a unique minimum in the above solution. To my knowledge, this type of penalty has not been explicitly modeled elsewhere in the cluster computing literature.

I evaluate the effectiveness of this estimator in Section 4.4 but first describe better search algorithms.

4.3.2 Advanced Hyperparameter Tuning

One 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 there exists no closed form expression for it (hence, one cannot 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 I 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 [122], and the Nelder-Mead method [116]. 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, it is unsurprising 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) [20], Sequential Model-based Algorithm Configuration (Auto-WEKA) [148] and Gaussian Process based methods, e.g., Spearmint [138]. These algorithms all share the property that they can search over spaces that 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. Section 4.4 reports the results of experiments designed to evaluate each method on several datasets to determine which method is most suitable for model search.

4.3.3 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 [28]. Indeed, assume the system is 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 the system performs 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.

The setting we consider for TuPAQ differs from this standard setting in two crucial ways. First, several of the search algorithms used in TuPAQ select model configurations to evaluate in an iterative fashion, so the system does not have advanced access to a fixed set of k model configurations. Second, in addition to efficiently allocating resources, the system

```
   input  : currentModels, history
   output: finishedModels, activeProposals
 1 (finishedModels, activeProposals) ← $\emptyset[]$;
 2 bestModel ← getBestFromHistory(history);
 3 for $m$ in currentModels do
 4     if fullyTrained(m) then
 5         finishedModels ← finishedModels ∪ m;
 6     else if $quality(m) * (1 + \epsilon) > quality(bestModel)$ then
 7         activeProposals ← activeProposals ∪ m;
 8     end
 9 end
10 return (finishedModels, activeProposals);
```

Algorithm 3: The bandit allocation strategy used by TuPAQ.

aims to return a reasonable result to a developer as quickly as possible, and hence there is a benefit to finish training promising model configurations once they have been identified.

The bandit selection strategy employed here is a variant of the action elimination algorithm of [54], and to my knowledge this is the first time this algorithm has been applied to hyperparameter tuning. This 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), the system first allocates 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 $(1+\epsilon)$; in these experiments I set $\epsilon = .5$ and thus continue to train all models with quality within 50% of the best quality model observed so far. I chose this value for ϵ because it provided a good tradeoff between maintaining model quality and speeding up search in the 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.

4.3.4 Batching

Batching is a natural system optimization in the context of training machine learning models, with applications for cross validation and ensembling [92, 31], however, it has not previously been applied to model search. For model search, I 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. The system takes advantage of this optimization in line 8 of Algorithm 2.

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

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:

$$\nabla f = \sum_{i=1}^{n} \left[\big(\sigma(w^\top x_i) - y_i\big) x_i \right], \tag{4.3}$$

where σ is the logistic function. The gradient descent algorithm (Algorithm 4) must evaluate this gradient function for all input data points, a task that 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
1 $i \leftarrow 0$;
2 Initialize Model;
3 while $i < MaxIterations$ do
4     read current;
5     Model ← Model - LearningRate * Gradient(Model, X);
6     $i \leftarrow i + 1$;
7 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. The formulation can naturally be extended to multiple models simultaneously if models are represented as a matrix $W \in \mathbb{R}^{d \times k}$, where k is the number of models to train simultaneously, i.e.,

$$\nabla f = \left[X^\top \big(\sigma(XW) - y \big) \right]. \tag{4.4}$$

In effect, by computing the gradient of several models simultaneously, the system is 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 that takes $O(ndk)$ operations and requires a scan of the input data as well as the final multiply by $X^\top$ that 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 [96]—these libraries are tailored to execute exactly dense linear algebra operations as efficiently as possible and are automatically tuned to the architecture the experiments are running on via [157].

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 [110]. 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 [111]. Specifically, on the machines I 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, this resource gap can be reduced.

The Roofline model [158] 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 data is represented 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, the batch size. The exact setting of k that achieves balance (and maximizes throughput) is hardware dependent, but I show in Section 4.5 that on modern machines, $k = 10$ is a reasonable choice.

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.

4.4 Design Space Evaluation

Now that I have laid out the possible optimizations available to TuPAQ, I investigate the potential speedup offered by each in turn. In all experiments I split the 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] I 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.

4.4.1 Execution Strategy Selection

To demonstrate the value of the estimator, I first investigated the consequences of choosing an inappropriate cluster size. I 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

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

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.

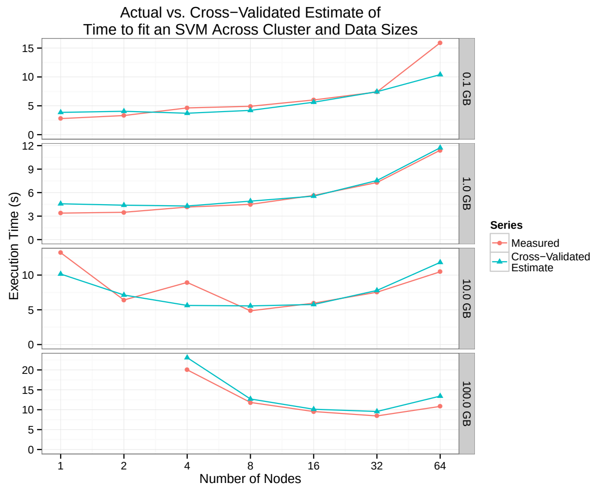


Figure 4.4: Measured vs. Modeled time for an SVM workload.

Next, I evaluated the usefulness of the estimator by instrumenting both a high-quality multi-core machine learning framework [39] and the 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 the 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, the parameters of the TuPAQ model (Equation 4.1) were estimated 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 $R^2 = 80.1\%$). I 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 4.4 demonstrates the tightness of this fit visually, showing the estimated execution times overlaid on the times observed in the experimental data. The figure shows 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 the TuPAQ estimator. One and two machine configurations for the 100.0GB case not shown because the datasets do not fit in cluster memory.

In addition, the coefficients fit by regression matched my knowledge about the hardware the jobs were run on and my 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.

	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 4.2: Time to fit SVM in various systems.

Finally, I used the TuPAQ model to validate the intuition that model parallel execution only makes sense while the data fits in memory on a single node. Table 4.2 reports execution times (in machines $\times$ seconds) of the SVM model training runs described above. The figure compares time 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 size varying from 1 to 32 nodes. 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.

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

4.4.2 Hyperparameter Tuning

I 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 [13]. The 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 [126]. To accommodate the linear scale-up that comes with adding random features, the system down samples the number of data points for each model training by the same proportion.

The 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 $\in (10^{-4}, 10^{2})$.

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

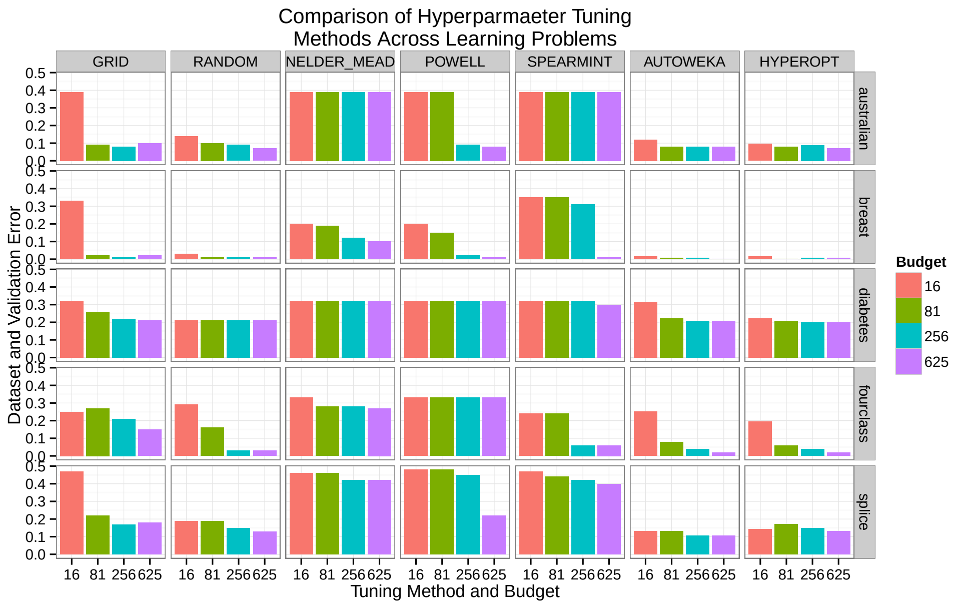


Figure 4.5: A comparison of hyperparameter tuning methods with several datasets and budgets.

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 n is varied 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 4.5. 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, the objective is to find 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 [19], 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 the regime I consider here, each model is expensive to compute, so the resources to try many models are limited. I chose to integrate HyperOpt into the larger experiments because it performed slightly better than Auto-WEKA.

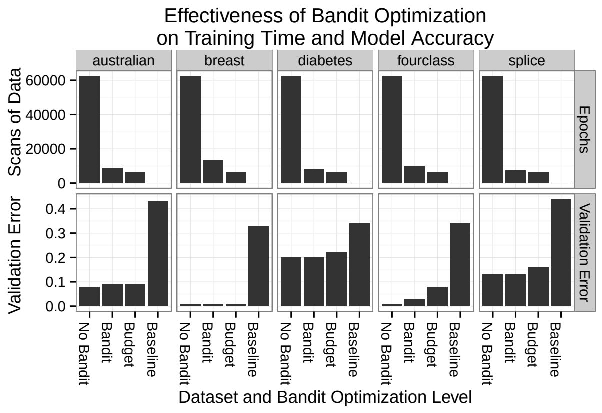


Figure 4.6: Model throughput and accuracy for the bandit strategy vs. other strategies.

4.4.3 Bandit Resource Allocation

I 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 the system could identify and terminate poorly performing models early in the training process without significantly affecting overall search quality.

In Figure 4.6 I 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 that 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 that 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. This heuristic works well for the workloads and datasets we discuss here, and it has already inspired further study [82].

4.4.4 Batching

To evaluate the batching optimization, I 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.

I trained these models on a 16-node cluster of `c3.8xlarge` nodes on Amazon EC2, running Apache Spark 1.1.0. I 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. I implemented both a naive version of this optimization—with while loops evaluating Equation 4.3 over each model in each task—as well as a more sophisticated version of this model that makes BLAS calls to perform the computation described in equation 4.4. For the batching experiments, I ran each algorithm for 10 iterations over the input data.

	D		
Batch Size	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.

	D		
Batch Size	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.

Table 4.3: Effect of batching on a 16 node cluster.

In Table 4.3 I show the total throughput of the system in terms of models trained per hour varying the batch size and the model complexity. In the Table, data sizes ranged from 750MB (D=100) to 75GB (D=10000).

For models trained on the smaller dataset, one can 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

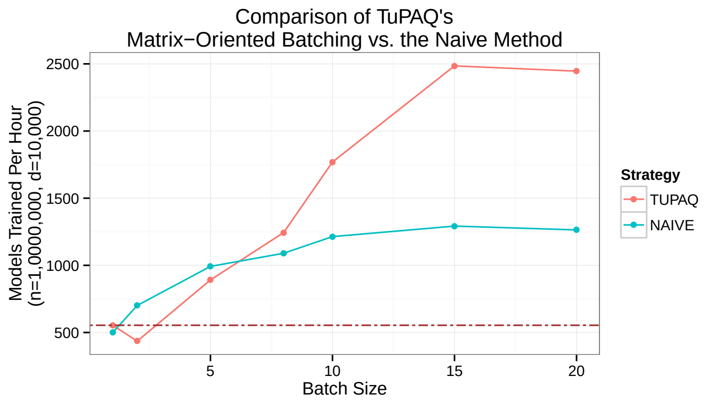


Figure 4.7: Comparison of batching methods vs. an un-batched baselin.

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. I discuss an alternative execution strategy that would better utilize cluster resources for situations where the input dataset is small in Section 4.7.

At the other end of the spectrum in terms of data size and model complexity, one can see the effects of scheduler delay start to lessen, and the system achieves maximum throughput in terms of models per hour at batch size 15. In Figure 4.7 I compare two different strategies of implementing batching—one via the naive method, and the other 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 the batch size to a relatively small constant (O(10)) the system is able to balance this tradeoff.

4.5 Putting It All Together

Having examined each point in the model search design space individually, I evaluate the end-to-end performance of the TuPAQ procedure and show that the techniques evaluated in the previous section yield a 10× 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. I 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.

4.5.1 Platform Configuration

I 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[7], and Apache Spark 1.1.0. Additionally, I made use of Hadoop 1.0.4 configured on local disks as the data store for the large scale experiments. Total runtime would have been similar if I 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, I used 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.

	Tuning Method		
Optimization	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 4.8: Learning time in minutes for a 128-configuration budget across various optimization levels for ImageNet data.

4.5.2 Experimental Setup and Datasets

I used two datasets with two different learning objectives to evaluate the system at scale. The first dataset is a pre-featurized version of the ImageNet 2010 dataset [18], featurized

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 4.9: Search time and best achieved model error for training 128 models on the 16-node ImageNet task.

using a procedure attributed to [48]. This process yields a dataset with 160,000 features and approximately 1,200,000 examples, or 1.4 TB of raw image features. In the 16-node experiments I downsampled 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 I train on the entire dataset. The system explores five hyperparameters here—one parameter for the classifier being train—SVM or logistic regression, as well as learning rate and L2 Regularization parameters for each matching the above experiments. I 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 4.3. 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, the system searches 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. The goal is to reduce validation error as much as possible, but my 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 [60], featurized according to the procedure described in [129]—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 [77]. Following the process of [126], 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. The system explores 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 the 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 [77].

4.5.3 Optimization Effects

In Figure 4.8 one 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 104 minutes 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 10× speedup in the case of random search and a 7× speedup in the slightly slower case of HyperOpt. HyperOpt takes

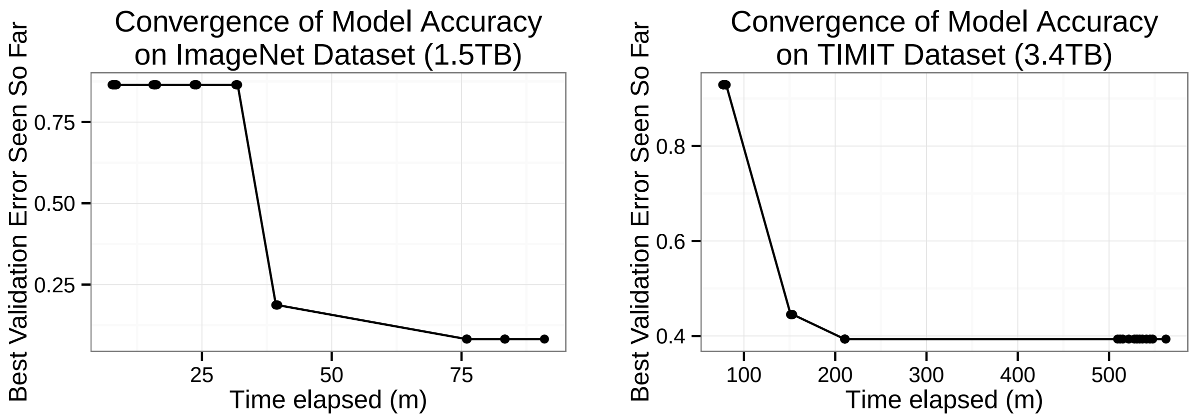


Figure 4.10: Performance of TuPAQ on large-scale problems using a 128-node cluster.

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.

Turning attention to statistical performance illustrated in Figure 4.9, one 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. Both optimized HyperOpt and Random search perform significantly faster than unoptimized Grid search, while HyperOpt yields the best model for this image classification problem.

4.5.4 Large Scale Speech and Vision

Because the system employs data-parallel versions of the 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, I used the same parameter search settings used with the smaller dataset, but I fixed the budget to 32 models to train. The results are illustrated in Figure 4.10(top). Using the fully optimized HyperOpt based search method, the system is 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 attention to an entirely different application area, I 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 4.10(right), 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. In this setting, for this particular dataset, training all 32 models to completion without batching or bandit allocation would have taken 35 hours.

4.6 Related Work

There has been a recent proliferation of systems designed for low-level, ad-hoc distributed predictive analytics, e.g., Apache Spark [160], GraphLab [62], Stratosphere [5], but few provide tooling for searching over a large space of predictive models.

In terms of system-level optimization, both Kumar et. al. [92] and Canny et. al. [31] 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. In this chapter 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. [74] 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, the estimator presented here 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 [148]. As the name suggests, Auto-WEKA aims to automate the use of Weka [155] by applying recent derivative-free optimization algorithms, in particular Sequential Model-based Algorithm Configuration (SMAC) [78], to the hyperparameter tuning problem. In fact, their proposed algorithm is one of the many optimization algorithms used 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 TuPAQ 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. I discuss and evaluate several of these methods in Sections 4.3 and 4.4, and two of these methods are used in TuPAQ.

With respect to the bandit optimization discussed in Section 4.3, both Agarwal et. al. [3] and Jamieson et. al. [82] 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 [155], MLlib [113], Vowpal Wabbit [39], Hyracks [25] and Mahout [10] are notable open-source ML libraries. These systems (all distributed with the exception of Weka), along with proprietary projects such as SystemML [61], 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" [89, 75], the present focus is on learning methods that scale horizontally for efficient use of cluster resources, which can be difficult with Deep Learning methods.

4.7 Future Work and Conclusions

In this chapter, I have described a system for large scale model search that 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 I 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* [17], would become especially important.

Systems extensions. As discussed earlier in Section 5.1, 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 developers 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 [4]. 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 developer'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. I revisit this important problem in Chapter 6.

Before turning to hyperparameter tuning of end-to-end pipelines, I first present KeystoneML, a system built for the construction of scalable, end-to-end ML pipelines and their optimization.

Chapter 5

KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics

Chapter 3 presented an interface for describing large scale ML algorithms. This interface was designed to simplify the succinct construction of new algorithms and present developers with a consistent interface for accessing new algorithms. In this chapter, we turn our attention to interfaces for *pipeline* construction, and discuss the optimization opportunities that exist when end-to-end pipelines are captured by the system. Armed with APIs for algorithm development and pipeline construction, developers are able to construct, optimize, and train end-to-end applications under a particular hyperparameter configuration. The next chapter focuses on efficiently exploring the hyperparameter space for such pipelines.

5.1 Introduction

As I discussed in Chapter 1, machine learning application developers commonly build complex, end-to-end data processing *pipelines* to deliver high quality predictions from raw data.

To assemble such pipelines, developers typically piece together domain specific libraries[1] for feature extraction and general purpose numerical optimization packages [94, 113] for supervised learning. This is often a cumbersome and error-prone process [132]. As previously discussed, these pipelines often need to be completely re-engineered when the training data or features grow by an order of magnitude–often the difference between an application that provides good statistical accuracy and one that does not [68]. As no broader system has purview of the end-to-end application, only narrow optimizations can be applied.

These challenges motivate the need for a system that

- Allows developers to specify end-to-end ML applications in a single system using high level logical operators.
- Scales out dynamically as data volumes and problem complexity change.

[1]e.g. OpenCV for Images (http://opencv.org/), Kaldi for Speech (http://kaldi-asr.org/)

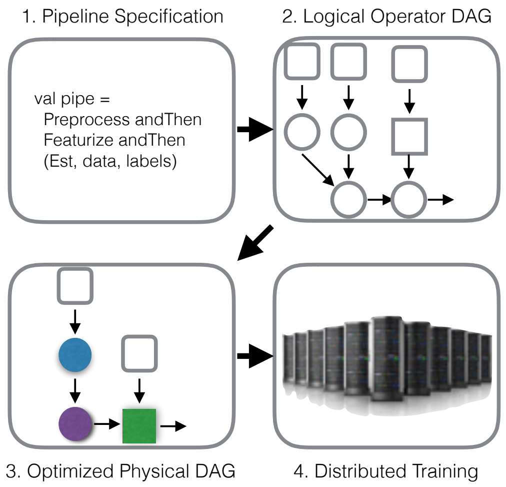


Figure 5.1: KeystoneML takes a high-level ML application specification, optimizes and trains it in a distributed environment. The trained pipeline is used to make predictions on new data.

- Automatically optimizes these applications given a library of ML operators and the developer's compute resources.

While existing efforts in the data management community [73, 61, 113] and in the broader machine learning systems community [94, 120, 105] have built systems to address some of these problems, each of them misses the mark on at least one of the points above.

I present KeystoneML, a framework for ML pipelines designed to satisfy the above requirements. Fundamental to the design of KeystoneML is the observation that model training is only one component of an ML application. While a significant body of recent work has focused on high performance algorithms [162, 127], and scalable implementations [44, 113] for model training, they do not capture the featurization process or the logical intent of the workflow. KeystoneML provides a high-level, type-safe API built around *logical operators* to capture end-to-end applications.

To optimize ML pipelines, database query optimization provides a natural motivation for the core design of such a system [88]. However, compared to relational database query optimization, ML applications present an additional set of concerns. First, ML operators are often *iterative* and may require multiple passes over their inputs, presenting opportunities for data reuse. Second, many ML operators provide only approximate answers to their inputs [127]. Third, numerical data properties such as sparsity and dimensionality are a necessary source of information when selecting optimal execution plans and conventional optimizers do not consider such measures. Finally, the system should be aware of the computation-vs-communication tradeoffs inherent in distributed processing of ML workloads [61, 94] and choose appropriate execution strategies in this regime.

To address these challenges I develop techniques to do both per-operator optimization and end-to-end pipeline optimization for ML pipelines. KeystoneML uses a cost-based optimizer that accounts for both computation and communication costs and the cost model can easily accommodate new operators and hardware configurations. To determine which intermediate states are materialized in memory during iterative execution, I formulate an optimization problem and present a greedy algorithm that works efficiently and accurately in practice.

I measure the importance of cost-based optimization and its associated overheads using real-world workloads from computer vision, speech and natural language processing. I find that end-to-end optimization can improve performance by 7× and that physical operator optimizations combined with end-to-end optimizations can improve performance by up to 15× versus unoptimized execution. I show that poor physical operator selection can result in up to a 260× slowdown. Using an image classification pipeline on over 1M images [131], I show that KeystoneML provides linear performance scalability across various cluster sizes, and statistical performance comparable to recent results [34, 131]. Additionally, KeystoneML can match the performance of a specialized phoneme classification system on a BlueGene supercomputer while using 8× fewer resources.

In summary, in this chapter I make the following contributions:

- I present KeystoneML, a system for describing ML applications using high level logical operators. KeystoneML enables end-to-end optimization of ML applications at both the operator and pipeline level.
- I demonstrate the importance of physical operator selection in the context of input characteristics of three commonly used logical ML operators, and propose a cost model for making this selection.
- I present and evaluate an initial set of whole-pipeline optimizations, including a novel algorithm that automatically identifies a subset of intermediate data to materialize to speed up pipeline execution.
- I evaluate these optimizations in the context of real-world pipelines in a diverse set of domains: phoneme classification, image classification, and textual sentiment analysis, and demonstrate near-linear scalability over 100s of machines with strong statistical performance.
- I compare KeystoneML with several recent systems for large-scale learning and demonstrate superior runtime from these optimization techniques and scale-out strategy.

KeystoneML is open source software[2] and is being used in scientific applications in solar physics [84] and genomics [53]

[2]`http://www.keystone-ml.org/`

```
val textClassifier = Trim andThen
    LowerCase andThen
    Tokenizer andThen
    NGramsFeaturizer(1 to 2) andThen
    TermFrequency(x => 1) andThen
    (CommonSparseFeatures(1e5), data) andThen
    (LinearSolver(), data, labels)
val predictions = textClassifier(testData)
```

Figure 5.2: A text classification pipeline is specified using a small set of logical operators.

5.2 Pipeline Construction and Core APIs

In this section I introduce the KeystoneML API that can be used to express end-to-end ML pipelines. Each pipeline is composed of a number of *operators* that are chained together. For example, Figure 5.2 shows the KeystoneML source code for a complete text classification pipeline. I next describe the building blocks of the KeystoneML API.

5.2.1 Logical ML Operators

Conventional analytics queries are typically composed using a small number of well studied relational database operators. This well-defined environment enables important optimizations. However, ML applications lack such an abstraction and practitioners typically piece together imperative libraries. Recent efforts have proposed using linear algebra operators such as matrix multiplication [61], convex optimization routines [56] or multi-dimensional arrays as logical building blocks [144].

In contrast, with KeystoneML I propose a design where high-level ML operations (such as `PCA`, `LinearSolver`) are used as building blocks. The KeystoneML approach has two major benefits: First, it simplifies building applications. Even complex pipelines can be built using just a handful of operators. Second, this higher level abstraction allows us to perform a wider range of optimizations. The key insight here is that there are usually multiple well studied algorithms for a given ML operator, but that their performance and statistical characteristics vary based on the inputs and system configuration. I next describe the API for operators in KeystoneML.

Pipelines are composed of *operators*. Transformers and Estimators are two abstract types of operators in KeystoneML. An operator is a function that operates on zero or more inputs to produce some output. A *logical operator* satisfies some logical contract. For example, it takes an image and converts it to grayscale. Every logical operator must have at least one *physical operator* associated with it that implements its logic. Logical operators with multiple physical implementations are candidates for *optimization*. They are marked `Optimizable` and have a set of `CostModels` associated with them. Operators that are iterative with respect to their inputs are marked `Iterative`.

A *Transformer* is an operator that can be applied to individual data items (or to a collection of items) and produces a new data item (or a collection of data items)–it is a de-

```
trait Transformer[A, B] extends Pipeline[A, B] {
  def apply(in: Dataset[A]): Dataset[B] = in.map(apply)
  def apply(in: A): B
}
```

```
trait Estimator[A, B] {
  def fit(data: Dataset[A]): Transformer[A, B]
}
```

```
trait Optimizable[T, A, B] {
  val options: List[(CostModel, T[A,B])]
  def optimize(sample: Dataset[A], d: ResourceDesc): T[A,B]
}

class CostProfile(flops: Long, bytes: Long, network: Long)

trait CostModel {
    def cost(sample: Dataset[A], workers: Int): CostProfile
}

trait Iterative {
  def weight: Int
}
```

Figure 5.3: The KeystoneML API consists of two extendable operator types and interfaces for optimization.

terministic unary function without side-effects. Examples of Transformers in KeystoneML include basic data transformations, feature extractors and model application. The deterministic and side-effect free properties affords the ability to reorder and optimize the execution of the functions without changing the result.

An *Estimator* is applied to a distributed collection of data items and produces a Transformer–it is a function generating function. ML algorithms provided by the KeystoneML Standard Library are Estimators, while featurizers are Transformers. For example, `LinearSolver` is an Estimator that takes a data set and labels, finds the linear model that minimizes the square loss between the training data and labels, and produces a Transformer that can apply this model to new data.

5.2.2 Pipeline Construction

Transformers and Estimators are *chained* together into a `Pipeline` using a consistent set of rules. The chaining methods are summarized in Figure 5.4. In addition to linear chaining of nodes using `andThen`, KeystoneML's API allows for pipeline branching. When a developer calls `andThen` a new `Pipeline` object is returned. By calling `andThen` multiple times on the same pipeline, developers can create multiple pipelines that branch out. Developers join the output of multiple pipelines of using `gather`. Redundancy is eliminated via common sub-expression optimization detailed in Section 5.4. I find that these APIs are sufficient for a number of ML applications (Section 5.5), but expect to extend them over time.

```
trait Pipeline[A,B] {
  def andThen[C](next: Pipeline[B, C]): Pipeline[A, C]
  def andThen[C](est: Estimator[B, C], data: Dataset[A]): Pipeline[A, C]
  // Combine the outputs of branches into a sequence
  def gather[A, B](branches: Seq[Pipeline[A, B]]): Pipeline[A, Seq[B]]
}
```

Figure 5.4: Transformers and Estimators are chained using a syntax designed to allow developers to incrementally build pipelines.

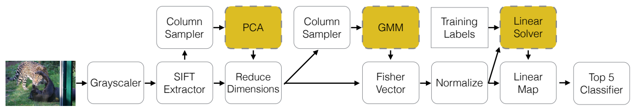


Figure 5.5: A pipeline DAG for image classification. Estimators are shaded.

5.2.3 Pipeline Execution

KeystoneML is designed to run with large, distributed datasets on commodity clusters. The high level API and optimizers can be executed using any distributed data-flow engine. The execution flow of KeystoneML is shown in Figure 5.1. First, developers specify pipelines using the KeystoneML APIs described above. As calls to these APIs are made, KeystoneML incrementally builds an operator DAG for the pipeline. An example operator DAG for image classification is shown in Figure 5.5. Once a pipeline is applied to some data, this DAG is then optimized using a set of optimizations described below– this stage is *optimization time*. Once the application has been optimized, the DAG is traversed depth-first and operators are executed one at a time, with nodes up until pipeline breakers (i.e. Estimators) packed into the same job–this stage is *runtime*. This lazy optimization procedure gives the optimizer full information about the application in question. I now discuss the optimizations made by KeystoneML.

5.3 Operator-Level Optimization

In this section I describe the operator-level optimization procedure used in KeystoneML. Similar to database query optimizers, the goal of the operator-level optimizer is to choose the best physical implementation for every machine learning operator in the pipeline. This is challenging to do because operators in KeystoneML are distributed i.e. they involve computation and communication across the cluster. Operator performance may also depend on statistical properties like sparsity of input data and level of accuracy desired. Finally, as discussed in Section 5.2, KeystoneML consists of a set of high-level operators. The advantage of having high-level operators is that the system can perform more wide-ranging optimizations. But this makes designing an optimizer more challenging because unlike relational operators or linear algebra [61], the set of operators in KeystoneML is not closed. I next discuss how I addressed these challenges.

Approach: The approach taken in KeystoneML is to develop a cost-based optimizer that splits the cost model into two parts: an operator-specific part and a cluster-specific part. The operator-specific part models the computation and communication time given statistics of the input data and number of workers and the cluster specific part is used to weigh their relative importance. More formally, the cost estimate for each physical operator, f can be expressed as:

$$c(f, A_s, R) = R_{exec}c_{exec}(f, A_s, R_w) + R_{coord}c_{coord}(f, A_s, R_w)$$

Where f is the operator in question, A_s contains statistics of a dataset to be used as its input, and R, the *cluster resource descriptor* represents the cluster computing, memory, and networking resources available. The cluster resource descriptor is collected via configuration data and microbenchmarks. Statistics captured include per-node CPU throughput (in GFLOP/s), disk and memory bandwidth (GB/s), and network speed (GB/s), as well as information about the number of nodes available. A_s is determined through a process I discuss in Section 5.4. R_w is the number of cluster nodes available.

The functions, c_{exec}, and c_{coord} are developer-defined operator-specific functions (defined as part of the operator `CostModel`) that describe execution and coordination costs in terms of the longest critical path in the execution graph of the individual operators [158], e.g. the most FLOPS used by a node in the cluster or the amount of data transferred over the most loaded link. Such functions are also used in the analysis of parallel algorithms [14] and are well known for common linear algebra based operators. R_{exec} and R_{coord} are determined by the optimizer from the cluster resource descriptor (R) and capture the relative speed of local and network resources on the cluster.

Splitting the cost model in this fashion allows the the optimizer to easily adapt to new hardware (e.g., GPUs or Infiniband networks) and also for it to work with both existing and future operators. Operator developers only need to implement a `CostModel` and the system accounts for hardware properties. Finally I note that the cost model used here is approximate and that the cost c need not be equal to the actual running time of the operator. Rather, as in conventional query optimizers, the goal of the cost model is to *avoid bad decisions*, which a roughly accurate model will do. At the boundary of two nearly equivalent operators, either should be acceptable in terms of runtime. I next illustrate the cost functions for three central operators in KeystoneML and the performance trade-offs that arise from varying input properties.

Linear Solvers are supervised Estimators that learn a linear map X between an input dataset A in $\mathbb{R}^{n \times d}$ to a labels dataset B in $\mathbb{R}^{n \times k}$ by finding the X that minimizes the value $||AX - B||_F$. In a multi-class classification setting, n is the number of examples or data points, d the number of features and k the number of classes. In the KeystoneML Standard Library there are several implementations of linear solvers, distributed and local, including

- Exact solvers [47] that compute closed form solutions to the least squares loss and return an X to extremely high precision.

Algorithm	Compute	Network	Memory
Local QR	$O(nd(d+k))$	$O(n(d+k))$	$O(d(n+k))$
Dist. QR	$O(\frac{nd(d+k)}{w})$	$O(d(d+k))$	$O(\frac{nd}{w}+d^2)$
L-BFGS	$O(\frac{insk}{w})$	$O(idk)$	$O(\frac{ns}{w}+dk)$
Block Solve	$O(\frac{ind(b+k)}{w})$	$O(id(b+k))$	$O(\frac{nb}{w}+dk)$

Table 5.1: Resource requirements for linear solvers. w is the number of workers in the cluster, i the number of passes over the dataset. For the sparse solvers s is the the average number of non-zero features per example, and b is the block size for the block solver. Compute and Memory requirements are per-node, while network requirements are in terms of the data sent over the most loaded link.

- Block solvers that partition the features into a set of blocks and use second-order Jacobi or Gauss-Seidel [23] updates to converge to the right solution.
- Gradient based methods like SGD [127] or L-BFGS [37] that perform iterative updates using the gradient and converge to a globally optimal solution.

Table 5.1 summarizes the cost model for each method. Constants are omitted for readability but are necessary in practice.

To illustrate these cost tradeoffs empirically, I vary the number of features generated by the featurization stage of two different pipelines and measure the training time and the training loss. I compare the methods on a 16 node cluster.

On an Amazon Reviews dataset (see Table 5.3) with a text classification pipeline, as the number of features are increased from 1k to 16k one can see in Figure 5.6 that L-BFGS performs 5-20× faster than the exact solver and 26-260× faster than the block-wise solver. Additionally the exact solver crashes for greater than 4k features as the memory requirements are too high. The reason for this speedup is that the features generated in text classification problems are sparse and the L-BFGS solver exploits the sparse inputs to calculate gradients cheaply.

The optimal solver choice does not always stay the same as the problem size is increased or as sparsity changes. For the TIMIT dataset, which has dense features, one can see that the exact solver is 3-9× faster than L-BFGS for smaller number of features. However when the number of features goes beyond 8k one can see that the exact solver becomes slower than the block-wise solver, which is also 2-3× faster than L-BFGS.

Principal Component Analysis (PCA) is an Estimator used for tasks ranging from dimensionality reduction to whitening to visualization. PCA takes an input dataset A in $\mathbb{R}^{n\times d}$, and a value k and produces a Transformer which can apply a matrix P in $\mathbb{R}^{d\times k}$, where P consists of the first k eigenvectors of the covariance matrix of A. The P matrix can be found using several techniques including the SVD or via an approximate algorithm, Truncated SVD [69]. In the KeystoneML cost model, SVD has runtime $O(nd^2)$ and offers an exact answer, while TSVD runs in $O(nk^2)$. Both methods may parallelized over a cluster.

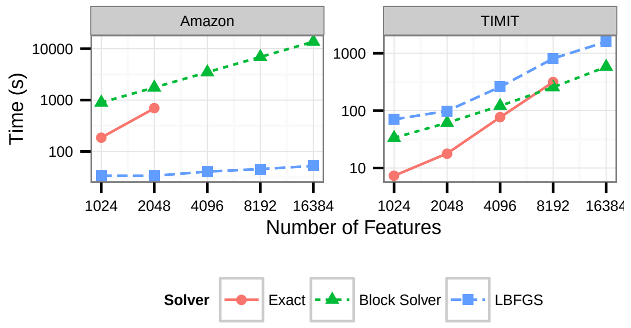


Figure 5.6: A poor choice of solver can mean orders of magnitude difference in runtime. Runtime for exact solve grows quadratically in the number of features and times out with 4096 features for Amazon and 16384 features for TIMIT running on 16 `c3.4xlarge` nodes.

	$d = 256$			$d = 4096$		
	$k = 1$	16	64	$k = 16$	64	1024
$n = 10^4$						
SVD	**0.1**	**0.1**	**0.1**	26	26	**26**
TSVD	0.2	0.3	0.4	**3**	**6**	34
Dist. SVD	1.7	1.7	1.7	106	106	106
Dist. TSVD	4.9	3.8	5.3	6	22	104
$n = 10^6$						
SVD	11	11	11	x	x	x
TSVD	14	30	65	x	x	x
Dist. SVD	**2**	**2**	**2**	260	**260**	**260**
Dist. TSVD	16	59	262	**75**	1,326	8,310

Table 5.2: Comparison of runtimes (in seconds) for approximate and exact PCA operators across different dataset sizes. A dataset has n examples and d features. k is an algorithm input. An x indicates that the operation did not complete.

To better illustrate how the choice of a PCA implementation affects the run time, I constructed a micro-benchmark that varies problem size along n, d, and k, and executed both local and distributed implementations of the approximate and exact algorithm on a 16-node cluster. In Table 5.2, one can see that as data volumes increase in n and d it makes sense to run PCA in a distributed fashion, while for relatively small values of k, it can make sense to use the approximate method.

Convolution is a critical building block of Signal, Speech, and Image Processing pipelines. In image processing, the Transformer takes in an Image of size $n \times n \times d$ and applies a bank of b filters (each of size $k \times k$, where $k < n$) to the Image and returns the b resulting convolved images of size $m \times m$, where $m = n - k + 1$. There are three main ways to implement

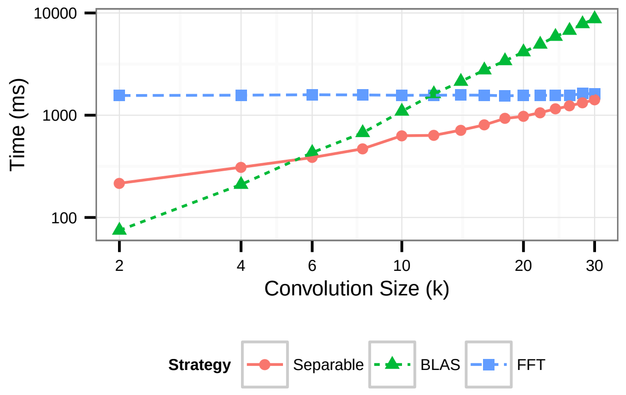


Figure 5.7: Time to perform 50 convolutions on a 256x256 3-channel image. As convolution size increases, the optimal method changes.

convolutions: via a matrix-vector product scheme when convolutions are separable, using BLAS matrix-matrix multiplication [2], or via a Fast Fourier Transform (FFT) [106].

The cost model for the matrix-vector product scheme takes $O(dbk(n-k+1)^2 + bk^3)$ time, but only works when filters are linearly separable. Meanwhile, the matrix-matrix multiplication scheme has a cost of $O(dbk^2(n-k+1)^2)$. Finally, the FFT based scheme takes $O(6dbn^2 \log n + 4dbn^2)$, and does not depend on k.

To illustrate the tradeoffs between these methods, in Figure 5.7, the size of the convolution filter, k, is varied and representative input images and batch sizes are used. For small values of k, one can see that BLAS the is fastest operator. However, as k grows, the algorithm's dependence on k^2 makes this approach inappropriate. If the filters are separable, it is faster to use the matrix-vector algorithm. The FFT algorithm does not depend on k and thus performs the same regardless of k.

Cost Model Evaluation: To evaluate how well the KeystoneML cost model works, I compared the physical operator chosen by the optimizer against the best choice from empirically measured values for linear solvers (Figure 5.6) and PCA (Table 5.2). I found that the optimizer made the right choice 90% of the time for linear solvers and 84% of the time for PCA. In both cases I found that the wrong choices were made when the running time of two operators were close to each other and thus the approximate cost model did not severely impact overall performance. For example, for the linear solver with 4096 dense features, the optimizer chooses the BlockSolver but empirically the Exact solver is about 30% faster.

As seen from the three examples above, the choice of optimal physical execution depends on hardware properties and on properties of the input data. Thus, choices made in support of operator-level optimization depend on upstream processing and cheaply estimating data properties at various points in the pipeline is an important problem. I next discuss how operator chaining semantics can help in achieving this.

5.4 Whole-Pipeline Optimization

5.4.1 Execution Subsampling

Operator optimization in KeystoneML requires the collection of statistics about input data at each pipeline stage. For example, a text featurization operator might map a string into a $10,000$-dimensional sparse feature vector. Without statistics about the input (e.g. vector sparsity) after featurization, a downstream operator will be unable to make its optimization decision. As such, dataset statistics (A_s) are determined by first estimating the size of the initial input dataset (in records), and optimizing the first operator in the pipeline with statistics derived from a sample of the input data. The optimized operator is then executed on the sample, and subsequent operators are optimized. This procedure continues until all nodes have been optimized. Along the way, the system forms a *pipeline profile*, which includes not just the information needed to form A_s at each step, but also information about operator execution time and memory consumption of each operator's execution on the sample. The system uses the pipeline profile to inform the Automatic Materialization optimization described below. I also evaluate the overheads from profiling in Section 5.5.3.

5.4.2 Common Sub-expression Elimination

One of the whole-pipeline rewrites done by KeystoneML is a form of common sub-expression elimination. It is common for training data or the output of featurization stages to be used in several stages of a pipeline. As a concrete example, in a text classification pipeline the system might first tokenize the training data then determine the $100,000$ most common bigrams in a text corpus, featurize the data to a binary vector indicating the presence of each bigram, and then train a classifier on the same training data. Thus, the pipeline needs the bigrams of each document *both* in the most common features calculation as well as when training the classifier. KeystoneML identifies and merges such common sub-expressions to enable computation reuse.

5.4.3 Automatic Materialization

Cache management and automatic selection of materialized views are important optimizations used by database management systems [38] and they have been studied in the context of analytical query systems [164, 70], and feature selection [161]. For ML workloads, materialization of intermediate data is very important for performance because the iterative nature of these workloads means that recomputation costs are multiplied across iterations. By capturing the iterative nature of the pipelines in the DAG, the optimizer is capable of identifying opportunities for reuse, eliminating redundant computation. I next describe a formulation for the materialization problem in iterative pipelines and propose an algorithm to automatically select a good set of intermediate objects to materialize in order to speed up ML pipeline execution.

Given the depth-first execution model and the deterministic and side-effect free nature of KeystoneML operators, a natural strategy is materialization of operator outputs that are visited multiple times during the execution. This optimization works well in the absence of memory constraints.

However, in many applications we have built with KeystoneML, intermediate output can grow to multiple terabytes in size, even for modestly sized inputs. On current hardware, this output is too big to fit in memory, even with hundreds of GB of memory per machine. Commonly used caching policies such as LRU can result in suboptimal run times because the decision to cache a large object (e.g. intermediate features) may evict a smaller object that is needed later in the pipeline and may be expensive to recompute (e.g. image features).

I propose an algorithm to automatically select the items to cache in the presence of memory constraints, given that the system knows how often the objects will be accessed, that it can estimate their size, and that it can estimate the runtime associated with materializing them.

I formulate the problem as follows: Given a memory budget, the objective is to find the set of nodes to include in the cache set that minimizes total execution time.

Let v be the node of interest in a pipeline G, $t(v)$ is the time taken to do the computation that is local to node v per iteration, $C(v)$ is the number of times a node will by called by its direct successors during execution, and w_v is the number of times a node iterates over its inputs. $T(n)$, the total execution time of the pipeline up to and including node v is:

$$T(v) = \frac{w_v(t(v) + \sum_{c\in\chi(v)} T(c))}{C(v)^{\kappa_v}} \tag{5.1}$$

where $\kappa_v \in \{0,1\}$ is a binary indicator variable signifying whether a node is cached or not, and $\chi(v)$ represents the direct predecessors of v in the DAG.

Where $C(v)$ is defined as follows:

$$C(v) = \begin{cases} \sum_{p\in\pi(v)} w_p C(p)^{\kappa_p}, & |\pi(v)| > 0 \\ 1, & \text{otherwise} \end{cases} \tag{5.2}$$

where $\pi(v)$ represents the direct successors of v in the DAG. Because of the DAG structure of the pipeline graph, it is guaranteed that there are no cycles in this graph, thus both $T(v)$ and $C(v)$ are well-defined.

The problem of minimizing pipeline execution time can be stated formally as an optimization problem with linear constraints as follows:

$$\min_{\kappa} T(sink(G))$$

$$s.t. \sum_{v\in V} size(v)\kappa_v \le memSize$$

```
Algorithm GreedyOptimizer
    input : G, t, size, memSize
    output: cache
    cache ← $\emptyset$;
    memLeft ← memSize;
    next ← pickNext (G, cache, size, memLeft, t);
    while $nextNode \neq \emptyset$ do
        cache ← cache ∪ next;
        memLeft ← memLeft - size(next);
        next ← pickNext (G, cache, size, memLeft, t);
    end
    return cache;
Procedure pickNext()
    input : G, cache, size, memLeft, t
    output: next
    minTime ← $\infty$;
    next ← $\emptyset$;
    for $v \in nodes(G)$ do
        runtime ← estRuntime (G, cache ∪ v, t);
        if $runtime < minTime$ & $size(v) < memLeft$ then
            next ← v;
            minTime ← runtime;
        end
    end
    return next;
```

Algorithm 5: The caching algorithm in KeystoneML builds a cache set by finding the node that will maximize time saved subject to memory constraints. `estRuntime` is a procedure that computes $T(v)$ for a given DAG, cache set, and node.

Where $sink(G)$ is the pipeline terminus, $size(v)$ the size of v's output, and $memSize$ the memory constraint.

This problem can also be thought of as problem of finding an optimal cache schedule. It is tempting to reach for classical results [16, 125] in the optimal paging literature to identify an optimal or near-optimal schedule for this problem. However, neither of these results matches the problem setting fully. In particular, Belady's algorithm is only optimal when each item has a fixed cost to bring into cache (as is common in reads from a two-level memory hierarchy), while in the current problem these costs are variable and depend heavily on the computation time to materialize them–in many cases recomputing may be two orders of magnitude faster than reading from disk but an order of magnitude slower than reading from memory, and each operator will have a different computational profile. Second, algorithms for the weighted paging problem don't take into account weights that are dependent on the current state of the cache. e.g. it may be much faster to compute image

Dataset	Train Size (GB)	Num Train (m)	Test Size (GB)	Num Test (m)	Classes	Type	Solve Features	Density	Solve Size (GB)
Amazon	13.97	65	3.88	18	2	text	100000	0.1%	89.1
TIMIT	7.5	2.2	0.39	0.11	147	vector	528000	100%	8857
ImageNet	74	1.2	3.3	0.05	1000	image	262144	100%	2502
VOC	0.428	0.005	0.420	0.005	20	image	40960	100%	1.52
CIFAR-10	0.500	0.5	0.001	0.01	10	image	135168	100%	62.9
Youtube8m	22.07	5.7	6.3	1.6	4800	vector	1024	100%	44.15

Table 5.3: Dataset Characteristics. While raw input sizes may be modest, intermediate state may grow by orders of magnitude before being input to a solver.

features if images are already in cluster memory than if they need to be retrieved from disk.

We will see in the next chapter that it is possible to write this problem down as a mixed-integer linear program. However, the costs of solving such a program may be prohibitive.

Instead, KeystoneML implement the greedy Algorithm 5. Given an unoptimized pipeline DAG, the algorithm chooses to cache the node that will lead to the largest savings in terms of execution time but whose output fits in available memory. This process proceeds iteratively until either no benefit to additional caching is possible or all available memory has been used.

5.5 Evaluation

To evaluate the effectiveness of KeystoneML, I explore its ability to efficiently support large scale ML applications in three domains. I also compare KeystoneML with other systems for large scale ML and show how the system's high-level operators and optimizations can improve performance. Following that I break down the end-to-end benefits of the previously discussed optimizations. Finally, I assess the system's ability to scale and show that KeystoneML scales well by enabling the development of scalable, composable components.

Implementation: I implement KeystoneML on top of Apache Spark, a cluster computing engine that has been shown to have good scalability and performance for many iterative ML algorithms [113]. KeystoneML contains an additional cache-management layer that is aware of the multiple Spark jobs that comprise a pipeline, and implemented ML operators in the KeystoneML Standard Library that are absent from Spark MLlib. While the current implementation of the system is Spark-specific, Spark is merely a distributed execution environment and the system can be ported to other backends.

Experiments are run on Amazon EC2 `r3.4xlarge` instances. Each machine has 8 physical cores, 122 GB of memory, and a 320 GB SSD, and was running Apache Spark 1.3.1, Scala 2.10, and HDFS from the CDH4 distribution of Hadoop. I have also run KeystoneML on Apache Spark 1.5, 1.6 and not encountered any performance regressions. I use OpenBLAS for numerical operations and Vowpal Wabbit [94] v8.0 and SystemML [61] v0.9 in these comparisons. If not otherwise specified, experiments are run on a 16-node cluster.

5.5.1 End-to-End ML Applications

To demonstrate the flexibility and generality of the KeystoneML API, end-to-end machine learning pipelines were implemented in several domains including text classification, image classification and speech recognition. I next describe these pipelines and compare statistical accuracy and performance results obtained using KeystoneML to previously published results. Every effort was taken to recreate these pipelines as they were described by their authors, and it was ensured that pipelines achieved comparable or better statistical results than those reported by each benchmark's respective authors.

The operators used to implement these applications are outlined in Table 5.4, and the datasets used to train them are described in Table 5.3. In each case, the datasets significantly increase in size as part of the featurization process, so at model fitting time the size is substantially larger than the raw data, as shown in the last two columns of the table. The Solve Size is the size of the dataset that is input to a Linear Solver. This may be too large for available cluster memory, as is the case for TIMIT. Accuracy results on each dataset achieved with KeystoneML as well as those achieved with the original authors code or (where code was unavailable) as reported in their respective works, are reported in Table 5.5.

Text Analytics: KeystoneML makes it simple for developers to scale their text pipelines to large datasets. Combined with libraries like CoreNLP [103], KeystoneML allows for scalable implementations of many text classification pipelines such as the one shown in Figure 5.2. A text classification pipeline based on [104] was evaluated on the Amazon Reviews dataset of 65m product reviews [108] with 100k sparse features. It was found that KeystoneML matches the statistical performance of a Vowpal Wabbit [94] pipeline when run on identical resources with the same solver, finishing in $440s$.

Kernel SVM for Speech Recognition: Kernel SVMs can be used in many classification scenarios as they can approximate any function. Often their performance has been shown to be much better than simpler generalized linear models [76]. Kernel evaluations can be efficiently approximated using random feature transformations [126, 137] and pipelines are a natural way to specify such transformations. Statistical operators like FFTs and cosine transformations and APIs to merge features help us succinctly describe the pipeline in KeystoneML. A kernel SVM solver was evaluated on the TIMIT dataset with 528k features. Using KeystoneML this pipeline runs in 138 minutes on 64 machines. By contrast, a 256 node IBM Blue Gene machine with 16 cores per machine takes around 120 minutes [137]. In this case, while KeystoneML may be 11% slower, it is using only $\frac{1}{8}$ the number of cores to solve this computationally demanding problem.

Image Classification: Image classification systems are useful in many settings. As images carry local information (i.e. information specific to where in the image a feature appears), locality sensitive techniques, e.g. convolutions or spatially-pooled fisher vectors [131], can be used to generate training features. KeystoneML makes it easy to modularize the pipeline to use efficient implementations of image processing operators like SIFT [102] and Fisher Vectors [131, 34]. Many of the same operators considered here are necessary components of "deep-learning" pipelines [90] that typically train neural networks via stochastic

Task	Type	Operators Used
Amazon Reviews Classification	Text	LowerCase, Tokenize NGrams, TermFrequency LogisticRegression
TIMIT Kernel SVM	Speech	RandomFeatures, Pipeline.gather LinearSolver
ImageNet Classification	Image	GrayScale, SIFT, LCS, PCA, GMM FisherVector, LinearSolver
VOC Classification	Image	GrayScale, SIFT, PCA, GMM FisherVector, LinearSolver
CIFAR-10 Classification	Image	Windower, PatchExtractor ZCAWhitener, Convolver, LinearSolver SymmetricRectifier, Pooler

Table 5.4: Operators used in constructing pipelines for datasets in Table 5.3.

gradient descent and back-propagation.

Using the VOC dataset, the pipeline described in [34] was implemented. This pipeline executes end-to-end on 32 nodes using KeystoneML in just 7 minutes. Using the authors original source code the same workload takes 1 hour and 27 minutes to execute on a single 16-core machine with 256 GB of RAM–KeystoneML achieves a 12.4× speedup with 16× the cores. A Fisher Vector based pipeline was evaluated on ImageNet with 256k features. The KeystoneML pipeline runs in 4.5 hours on 100 machines. The original pipeline takes four days [130] to run using a highly specialized codebase on a 16-core machine, a 21× speedup on 50× the cores.

In summary, using KeystoneML achieves one to two orders of magnitude improvement in end-to-end throughput versus a single node, and equivalent or better performance over cluster systems running similar workloads. These improvements mean much quicker ML application development, which leads to higher developer productivity. Next I compare KeystoneML to other large scale learning systems.

5.5.2 KeystoneML vs. Other Systems

I compare runtimes for the KeystoneML solver with both a specialized system, **Vowpal Wabbit** [94], built to estimate linear models, and **SystemML** [61], a general purpose ML system, which optimizes the implementation of linear algebra operators used in specific algorithms (e.g., Conjugate Gradient Method), but does not choose among logically equivalent algorithms. I compare solver performance across different feature sizes for two binary classification problems: Amazon and a binary version of TIMIT. The systems were

[3] I report accuracy on 64k features for ImageNet, while time is reported on 256k features due to lack of consistent reporting by the original authors. The workloads are otherwise similar.

Dataset	KeystoneML		Reported	
	Accuracy	Time (m)	Accuracy	Time (m)
Amazon [104]	91.6%	3.3	-	-
TIMIT [77]	66.06%	138	66.33%	120
ImageNet [131][3]	67.43%	270	66.58%	5760
VOC 2007 [34]	57.2%	7	59.2%	87
CIFAR-10 [147]	84.0%	28.7	84.0%	50.0

Table 5.5: Time to Accuracy with KeystoneML obtained on ML pipelines described in the relevant publication. Accuracy for VOC is mean average precision. Accuracy for ImageNet is Top-5 error.

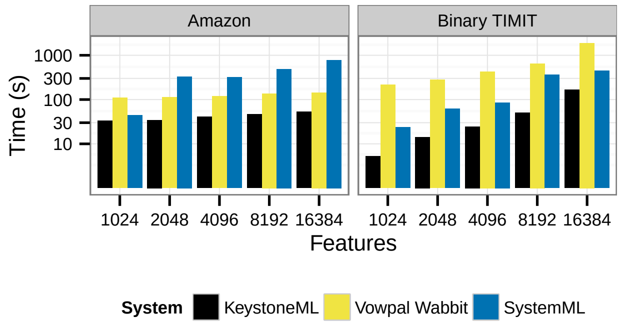


Figure 5.8: KeystoneML's optimizing linear solver outperforms both a specialized and optimizing ML system for two problems across feature sizes. Times are on log scale.

run with identical inputs and objective functions, and I report end-to-end solve time. For this comparison, I solve binary problems because SystemML does not include a multiclass linear solver.

The results are shown in Figure 5.8. The optimized solver in KeystoneML outperforms both Vowpal Wabbit and SystemML because it selects an *appropriate algorithm* to solve the logical problem, as opposed to relying on a one-size fits all operator. At 1024 features for the Binary TIMIT problem, KeystoneML chooses to run an exact solve, while from 2048 to 32768 features it chooses a Dense L-BFGS implementation. At 65536 features (not pictured), KeystoneML finishes in 17 minutes, while SystemML takes 1 hour and 40 minutes to converge to *worse* training loss over 10 iterations, a speedup of 5.5×.

The reasons for these performance differences are twofold: first, since KeystoneML raises the level of abstraction to the logical level, the system can automatically select, for example, a sparse solver for sparse data or an exact algorithm when the number of features is low, or a block solver when the features are high. In the middle, particularly for KeystoneML vs. SystemML on the Binary TIMIT dataset, the algorithms are similar in terms of complexity and access patterns. In this case KeystoneML is faster because feature

Machines	1	2	4	8	16	32
TensorFlow (strong)	184	90	57	67	122	292
TensorFlow (weak)	184	135	135	114	xxx	xxx
KeystoneML	235	125	69	43	32	29

Table 5.6: Time, in minutes, to 84% accuracy on the CIFAR-10 dataset with KeystoneML and TensorFlow configured for both strong and weak scaling. In large weak scaling regimes TensorFlow failed to converge to a good model.

extraction is *pipelined* with the solver, while SystemML requires a conversion process for data to be fed into a format suitable for the solver. If *only* the solve step of the pipeline is considered, KeystoneML is roughly 1.5× faster than SystemML for this problem.

TensorFlow is a newly open-sourced ML system developed by Google [105]. Developed concurrently to KeystoneML, TensorFlow also represents pipelines as graph of dataflow operators. However, the design goals of the two systems are fundamentally different. KeystoneML is designed to support horizontally scalable workloads to offer good scale out performance for conventional machine learning applications consisting of featurization and model estimation, while TensorFlow is designed to support neural network models trained via mini-batch SGD with back-propagation. I compare against TensorFlow v0.8 and adapt a multi-GPU example [147] to a distributed setting in a procedure similar to [36].

To illustrate the differences, I compare the systems' performance on CIFAR-10 top-1 classification performance. While the learning tasks are identical (i.e., make good predictions on a test dataset, given a training dataset), the workloads are not identical. Specifically, TensorFlow implements a model similar to the one presented in [90], while in KeystoneML I implement a version of the model similar to [40]. TensorFlow was run with default parameters and I experimented with strong scaling (fixed 128 image batch size) and weak scaling (batch size of $128 \times Machines$).

For this workload, TensorFlow achieves its best performance on 4-node cluster with 32 total CPU cores, running in 57 minutes. Meanwhile, KeystoneML surpasses its performance at 8 nodes and continues to improve in total runtime out to 32 nodes, achieving a minimum runtime of 29 minutes, or a 1.97× speedup. These results are summarized in Table 5.6. I ran TensorFlow on CPUs for the sake of comparability. Prior benchmarks [147] have shown that the speed of a single multi-core CPU is comparable to a single GPU; thus the same pipeline finishes in 50 minutes on a 4 GPU machine.

TensorFlow's lack of scalability on this task is fundamental to the chosen model and the algorithm being used to fit it. Minimizing a non-convex loss function via minibatch Stochastic Gradient Descent (SGD) requires coordination of the model parameters after a small number of examples are seen. In this case, the coordination requirements surpass the savings from parallelism at a small number of nodes. While TensorFlow has better scalability on some model architectures [146], it is not scalable for other architectures. By contrast, by using a communication-avoiding solver KeystoneML's performance on this task can be scaled out significantly further.

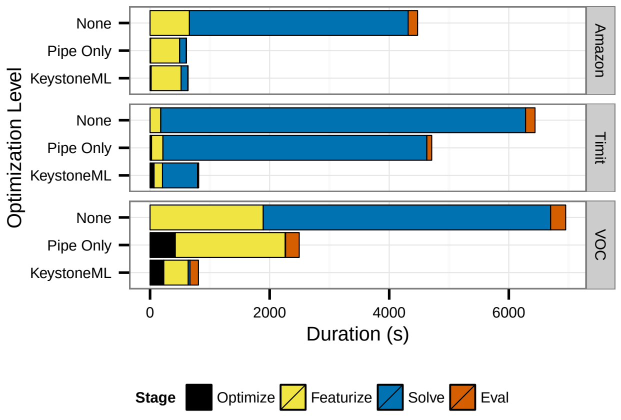


Figure 5.9: Impact of optimization levels on three applications, broken down by stage.

Finally, a recent benchmark dataset from YouTube [1] describes learning pipelines involving featurization with a neural network [146] followed by a logistic regression model or SVM. The authors claim that "models train to convergence in less than a day on a single machine using the publicly-available TensorFlow framework." A best-effort replication of this pipeline was performed using KeystoneML. We are unable to replicate the author's claimed accuracy–the KeystoneML pipeline achieves 21% mAP while they report 28% mAP. KeystoneML trains a linear classifier on this dataset in 3 minutes, and a converged logistic regression model with worse accuracy in 90 minutes (31 batch gradient evaluations) on a 32-node cluster. The ability to choose an appropriate solver and readily scale out are the key enablers of KeystoneML's performance.

I now evaluate the effectiveness of KeystoneML's optimizations at improving the throughput of these example workloads.

5.5.3 Optimization Levels

The end-to-end results reported earlier in this section are achieved by taking advantage of the complete set of optimizations available in KeystoneML. To understand how important the per-operator and whole-pipeline optimizations described in Sections 5.3 and 5.4 are I compare three different levels of optimization: a default unoptimized configuration (`None`), a configuration where only whole-pipeline optimizations are used (`Pipe Only`) and a configuration with operator-level and whole-pipeline optimizations (`KeystoneML`).

Results comparing these levels, with a breakdown of stage-level timings on the VOC, Amazon and TIMIT pipelines are shown in Figure 5.9. For the Amazon pipeline the whole-pipeline optimizations improve performance by 7×, but the operator optimizations do not help further, because the Amazon pipeline uses CoreNLP featurizers which do not have statistical optimizations associated with them, and the default L-BFGS solver turns out to be optimal. The performance gains come from caching intermediate features just before

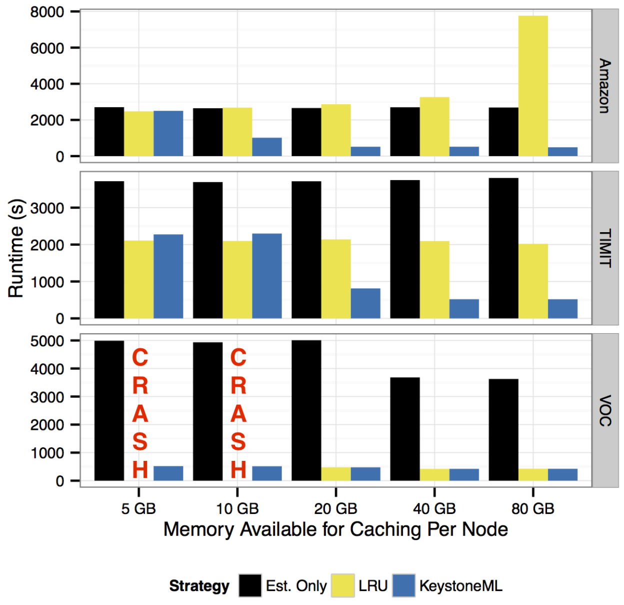


Figure 5.10: The KeystoneML caching strategy outperforms a rule-based and LRU caching strategy at many levels of memory constraints and responds well to memory pressure.

the L-BFGS solve. For the TIMIT pipeline, run with 16k features, one can see that the end-to-end optimizations only give a 1.3× speedup but that selecting the appropriate solver results in a 8× speedup over the baseline. Finally in the VOC pipeline the whole pipeline optimization gives around 3× speedup. Operator-level optimization chooses good PCA, GMM and solver operators resulting in a 12× improvement over the baseline, or 15× if I amortize the optimization costs across many runs of a similar pipeline. Optimization overheads are insignificant except for the VOC pipeline. This dataset has relatively few examples, so the sampling strategy takes more time relative to the other datasets.

5.5.4 Automatic Materialization Strategies

As discussed in Section 5.4, one key optimization enabled by KeystoneML's ability to capture the complete application DAG to dynamically determine where to materialize reused intermediate objects, particularly in the presence of memory constraints. In Figure 5.10 I demonstrate the effectiveness of the greedy caching algorithm proposed in Section 5.4. Since the algorithm needs local profiles of each node's performance, I measured each node's running time on two samples of 512 and 1024 examples. The system extrapolates the node's memory usage and runtime to full scale using linear regression. I found that memory estimates from this process are highly accurate and runtime estimates were within 15% of actual runtimes. If estimates are inaccurate, the system falls back to an LRU replacement policy for the cache set determined by this procedure. While this measurement process is imperfect, it is adequate at identifying relative running times and thus is sufficient for the purpose of resource management.

I compare this strategy with two alternatives–the first is a simple rule-based approach which only caches the results of Estimators. This is a sensible rule to follow, as the result of an Estimator (a Transformer or model) is computationally expensive to acquire and typically holds a small memory footprint. However, this is not sufficient for most practical pipelines because if a pipeline contains more than one Estimator, often the input to the first Estimator will be used downstream, thus presenting an opportunity for reuse. The second approach is a Least Recently Used (LRU) policy: in a regime where memory is unconstrained, LRU matches the ideal strategy and further, LRU is the default memory management strategy used by Apache Spark. However, LRU does not take into account that datasets from other jobs (even ones in the same pipeline) are vying for presence in cluster memory.

From Figure 5.10 one can notice several important trends. First, the KeystoneML strategy is nearly always better than either of the other strategies. In the unconstrained case, the algorithm is going to remember all reused items as late in their journey through the pipeline as possible. In the constrained case, it will do as least as well as remembering the (small) estimators which are by definition reused later in the pipeline. Additionally, the strategy degrades effectively, mixing between the best performance of the limited-memory rule-based strategy and the LRU based “cache everything” strategy which works well in unconstrained settings. Curiously, as memory available to caching per-node was increased, the LRU strategy performed *worse* for the Amazon pipeline. Upon further investigation, this is because Spark has an implicit admission control policy which only allows objects under some proportion of the cache size to be admitted to the cache at runtime. As the cache size gets bigger in the LRU case, massive objects which are not then reused are admitted to the cache and evict smaller objects which *are* reused and thus need to be recomputed.

To give a concrete example of the optimizer in action, consider the VOC pipeline shown in Figure 5.5 in Section 5.2. When memory is not constrained (80 GB per node), the outputs from the `SIFT`, `ReduceDimensions`, `Normalize` and `TrainingLabels` are cached. When memory is restricted (5 GB per node) only the output from `Normalize` and `TrainingLabels` are cached.

These results show that both per-operator and whole-pipeline optimizations are important for end-to-end performance improvements. I next study the scalability of the system on three workloads

5.5.5 Scalability

As discussed in previous sections, KeystoneML’s API design encourages the construction of scalable operators. However, some estimators like linear solvers need coordination [47] among workers to compute correct results. In Figure 5.11 I demonstrate the scaling properties from 8 to 128 nodes of the text, image, and Kernel SVM pipelines on the Amazon, ImageNet (with 16k features) and TIMIT datasets (with 65k features) respectively. The ImageNet pipeline

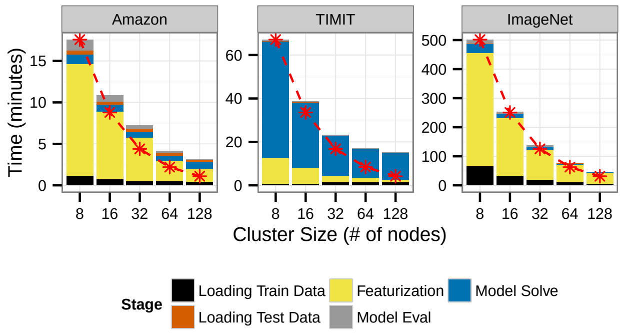


Figure 5.11: Time breakdown of workloads by stage. The red line indicates ideal strong scaling performance over 8 nodes.

exhibits near-perfect horizontal scalability up to 128 nodes, while the Amazon and TIMIT pipeline scale well up to 64 nodes.

To understand why the Amazon and TIMIT pipeline do not scale linearly to 128 nodes, I further analyze the breakdown of time take by each stage. One can see that each pipeline is dominated by a different part of its computation. The TIMIT pipeline is dominated by its solve stage, while featurization dominates the Amazon and ImageNet pipelines. Scaling linear solvers is known to require coordination [47], which leads directly to sub-linear scalability of the whole pipeline. Similarly, in the Amazon pipeline, one of the featurization steps uses an aggregation tree which does not scale linearly.

5.6 Related Work

ML Frameworks: ML researchers have traditionally used MATLAB or R packages to develop ML routines. The importance of feature engineering has led to tools like scikit-learn [120] and KNIME [21] adding support for featurization for small datasets. Further, existing libraries for large scale ML [29] like Vowpal Wabbit [94], GraphLab [101], MLlib [113], RIOT [163], DimmWitted [162] focus on efficient implementations of learning algorithms like regression, classification and linear algebra routines. In KeystoneML, I focus on pipelines that include featurization and show how to optimize performance with end-to-end information. Work in Parameter Servers [99] has studied how to share model updates. KeystoneML presents a high-level API for linear solvers and can leverage parameter servers in our architecture.

Closely related to KeystoneML is SystemML [61], which also uses an optimization based approach to determine the physical execution strategy of ML algorithms. However, SystemML places less emphasis on support for UDFs and featurization, while instead focusing on linear algebra operators that have well specified semantics. To handle featurization I develop an extensible API in KeystoneML that allows for cost profiling of arbitrary nodes and uses these cost estimates to make node-level and whole-pipeline optimizations.

Other work [161, 2] has looked at optimizing caching strategies and operator selection in the regime of feature selection and feature generation workloads. KeystoneML considers similar problems in the context of distributed ML operators and end-to-end learning pipelines. Developed concurrently to KeystoneML is TensorFlow [105]. While designed to support different learning workloads the optimizations that are a part of KeystoneML can also be applied to systems like TensorFlow.

Projects such as Bismarck [56], MADLib [73], and GLADE [123] have proposed techniques to integrate ML algorithms inside database engines. In KeystoneML, I develop a high level API and show how the system can achieve similar benefits of modularity and end-to-end optimization while also being scalable. These systems do not present cross-operator optimizations and do not consider tradeoffs at the operator level that are considered in KeystoneML. Finally, Spark ML [112] represents an early design of a similar high-level API for machine learning. We present a type safe API and optimization framework for such a system. The version presented here differs in its use of type-safe operations, support for complex data flows, internal DAG representation and optimizations discussed in Sections 5.3 and 5.4.

Query Optimization, Modular Design, Caching: There are several similarities between the optimizations made by KeystoneML and traditional relational query optimizers. Even the earliest relational query optimizers [133] used multiple physical implementations of equivalent logical operators, and like many relational optimizers, the KeystoneML optimizer is cost-based. However, KeystoneML supports a much richer set of data types than a traditional relational query system, but its operators lack some relational algebra semantics, such as commutativity, limiting the system's ability to perform certain optimizations. Further, KeystoneML switches among operators that provide exact answers vs approximate ones to save time due to the workload setting. Data characteristics such as sparsity are not traditionally considered by optimizers.

As previously stated, the caching strategy employed by KeystoneML can be viewed as a form of view selection for materialized view maintenance over queries with expensive user-defined functions [38, 72], I focus on materialization for intra-query optimization, as opposed to inter-query optimization [70, 35, 164, 52, 121]. While much of the related work focuses on the challenging problem of view maintenance in the presence of updates, KeystoneML exploits the iterative nature and immutable properties of this state.

5.7 Future Work and Conclusion

KeystoneML represents a significant first step towards easy-to-use, robust, and efficient end-to-end ML at massive scale. The existing KeystoneML operator APIs are synchronous and the pipelines presented here are acyclic. Future study could examine how algorithms like asynchronous SGD [99] or back-propagation can be integrated with the robustness and scalability that KeystoneML provides.

In this chapter, I have presented the design of KeystoneML, a system that enables the development end-to-end ML pipelines. By capturing the end-to-end application, KeystoneML can automatically optimize execution at both the operator and whole-pipeline levels, enabling solutions that automatically adapt to changes in data, hardware, and other environmental characteristics.

In the next chapter, I show how combining whole-pipeline optimizations such as those presented here with accelerated hyperparameter tuning methods such as those presented in Chapter 4 can lead to decreased ML application development time.

Chapter 6

Piperplanned: Resource Aware Pipeline Hyperparameter Tuning

In Chapter 4 I discussed optimizations designed to accelerate hyperparameter tuning of individual learning algorithms. In Chapter 5 I focused on APIs that enable the construction of end-to-end learning pipelines and the optimizations that are enabled by capturing these workloads end-to-end in a single system. A natural follow-on question is whether these techniques can be combined to accelerate end-to-end large scale ML application development. In this chapter, I explore this question in depth.

6.1 Introduction

As described in the preceding chapters, ML pipelines combine multiple stages of data-preprocessing, feature extraction, and supervised and unsupervised machine learning to achieve high statistical accuracy on problems in various of domains [131, 57]. Recent machine learning tool kits have been designed to enable such complex workflows [112, 120], including the work presented in Chapter 5. However, each stage of a pipeline may have one or more configuration parameters, or *hyperparameters*, that must be set by the application developer. The task of identifying a hyperparameter configuration that achieves high statistical performance, a problem called *hyperparameter optimization*, has thus emerged as a crucial component in the development of ML applications in order to boost predictive accuracy.

Hyperparameter optimization also introduces a significant computational burden. Executing a pipeline with a fixed hyperparameter configuration can take hours to weeks even when executing on hundreds of machines in a cluster, and classical methods such as grid search or random search typically evaluate tens to thousands of configurations. Several recent methods have been developed to improve upon these classical methods, and these modern techniques roughly fall into two categories. 'Configuration-selection' techniques select hyperparameter configurations in an adaptive manner, e.g., [138, 57], while 'configuration-evaluation' methods allocate more resources to promising hyperparameter configurations and quickly eliminate poor ones. Inspired by the work I presented in Chapter 4, researchers from Berkeley and UCLA proposed a configuration-evaluation method called HYPERBAND, which is a simple and principled multi-armed bandit-based approach that exhibits state-of-the-art empirical accuracy on several benchmarks [98]. In spite of their differences, all three classes of approaches (classical, configuration-selection, configuration-evaluation) evaluate each hyperparameter configuration *independently*. Unfortunately, this independence assumption fundamentally limits the usefulness of existing methods on modern pipelines, especially since the number of hyperparameters usually grows with number of pipeline stages (causing the size of the corresponding search space to grow exponentially).

However, hyperparameters at various pipeline stages are often highly interdependent, and in fact, this underlying pipeline structure can be naturally leveraged by removing the independence assumption associated with existing methods. To illustrate this idea, consider the three basic pipelines in Figure 6.1(a), which consist of three operators (A, B, C), each of which has a single hyperparameter (denoted by the subscript). The three pipelines are composed of the same three operators, but differ in their hyperparameter configurations. Notably, all three pipelines have an identical hyperparameter configuration for operator A, and the first two pipelines also have the same configuration for operator B. By taking this structure into account, one can succinctly represent the joint computation graph for all three pipelines, as illustrated in Figure 6.1(b). Now, assuming that each operator represents a single unit of computation, the potential for savings is immediately obvious. The first set of graphs consists of 9 operators, while the second graph consists of only 6, for a hypothetical savings of 33%. Alternatively, if operator A were the computational bottleneck and a system jointly evaluated 20 pipelines rather than three, one could expect to see nearly a 20× speedup. Although this is just a simple example, these ideas can naturally be extended by modeling the dependencies among various pipelines via a directed acyclic graph (DAG) that encodes all putative pipeline configurations. By considering more complex pipelines and larger batch sizes (i.e., evaluating more pipelines simultaneously), a developer can potentially experience significant computational speedups via efficient traversal of the corresponding DAGs.

I formalize these ideas and develop a novel *pipeline-aware* approach for hyperparameter optimization, integrated with KEYSTONEML, called PIPERPLANNED. This approach is based on three critical observations.

1. Hyperparameter optimization involves evaluating pipelines that are very similar but not identical, leading to redundant work when evaluating hyperparameter configurations independently.

2. By analyzing pipeline structure and carefully planning computation jointly across *multiple* hyperparameter configurations, the system can eliminate this redundant work and substantially decrease end-to-end runtime.

3. This approach is complementary to existing hyperparameter optimization methods. The set of hyperparameter configurations that are considered can either be selected

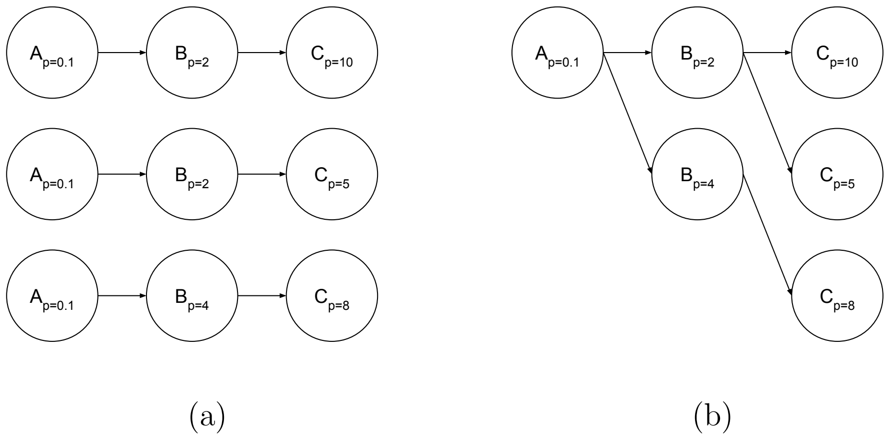


Figure 6.1: Three pipelines with a overlapping hyperparameter configurations. (a): Three independent computation pipelines. (b): Multiple computation pipelines can be merged into a single tree to eliminate redundancy.

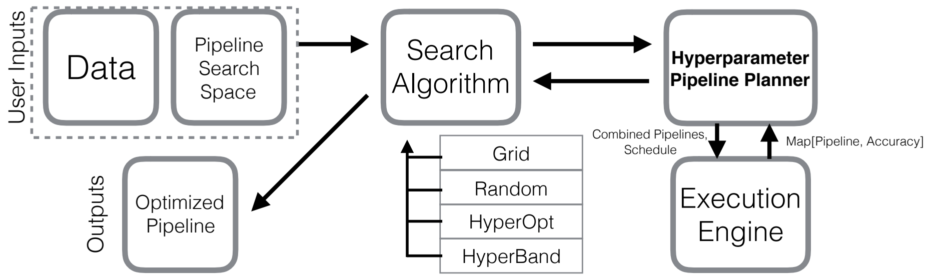


Figure 6.2: The PiperPlanned architecture.

randomly for classical and configuration-evaluation methods, or from some non-uniform posterior distribution for configuration-selection methods.

The architecture of this approach is shown in Figure 6.2. Using a syntax defined in Section 6.3, developers describe a *pipeline search space*, which delineates the space of allowable pipeline configurations, the available hyperparameters, and their associated ranges. These hyperparameters can be real valued, integral, or discrete, and can be nested and chained to form complex spaces. This space is used as input to a *hyperparameter search algorithm* that can *enumerate* or *sample* hyperparameter *configurations* from the hyperparameter space. In conventional systems, these hyperparameter samples would be executed sequentially. In this architecture, however, multiple configurations are fed to the *hyperparameter pipeline planner* that consolidates multiple pipelines into a single execution plan and automatically optimizes its execution. In this design, the hyperparameter pipeline planner is independent of the search algorithm.

Caching intermediate pipeline results is a natural approach to encourage reuse. However, in large-scale settings, these intermediate results will often be large, and the system will face memory-constraints when storing them for later reuse.

Furthermore, conventional caching strategies fail to account for the dependency structure among data objects encoded in the data flow graph, leading to sub-optimal performance on real-world workloads even on single-shot pipelines, as was illustrated in the previous chapter.

To efficiently execute many hyperparameter configurations simultaneously, the executing system needs a resource-aware cache control schedule for a given dataflow graph representing a hyperparameter tuning workload that minimizes total execution time for the graph subject to memory constraints. In this chapter, I formulate the problem of finding such a schedule as a constrained mixed integer linear program. The solution to this program is the *optimal* schedule for the given workload. While actually computing the optimal schedule may be infeasible for large graphs under this formulation, it serves as an absolute lower bound on the performance of *any* cache schedule and thus I use it to study the effectiveness of several conventional algorithms on sample workloads.

In the remainder of this chapter, I first describe the issues with existing hyperparameter optimization efforts in further depth in Section 6.2. I describe the salient features of PiperPlanned in Section 6.3 before establishing formalism to describe the maximum savings available due to reuse in the unconstrained memory setting in Section 6.4. I then formally describe the challenging problem of finding an optimal cache control schedule for a data flow graph in Section 6.4.3 and propose a solution to this problem. Finally, I evaluate the practicality of finding and implementing the optimal policy on a number of synthetic and real workloads and compare it with several existing caching algorithms in Section 6.5, including the one presented in Chapter 5.

6.2 Hyperparameter Optimization

In this section, I describe the hyperparameter optimization problem and illustrate why knowledge of application structure is crucial to accelerating hyperparameter optimization.

6.2.1 Hyperparameter Search

As discussed in Chapter 2 and in Chapter 4, hyperparameter optimization, can be formally stated as an optimization problem as follows:

$$h^\star = \underset{h \in H}{\text{minimize}} \quad l(m(h, D_{\text{train}}), D_{\text{val}}) \tag{6.1}$$

Here, h represents a hyperparameter configuration from some larger hyperparameter configuration space H. D_{train} represents training data, and D_{val} represents validation data. m here is a function that returns a model for a given training configuration and training data, and l is a function that computes a loss metric, such as prediction error of a model, on test data. Simply put, the goal of hyperparameter optimization is to find the hyperparameter setting that minimizes validation loss.

In general, the hyperparameter loss function, l is not convex or even differentiable, and thus optimizing it is difficult.

Conventional hyperparameter optimization methods attempt to solve the problem of finding an optimal hyperparameter configuration via brute force, enumerating some number of configurations from X and picking the best one. On the other hand, configuration selection and configuration evaluation techniques attempt to learn the curvature of the space of l for a hyperparameter space X by sampling observations from x and exploring regions more likely to minimize the loss function.

These methods are powerful and general purpose in part because they make no assumptions about the underlying model fitting process, m, and treat m as a function that can be evaluated independently across runs. However, in the context of a system like KeystoneML, it is not necessary to treat model fitting runs independently.

6.2.2 Pipelines

In practice, machine learning applications are composed of multi-stage computational pipelines, where each stage has its own hyperparameters. Downstream stages are dependent on upstream stages. As I discussed in the introduction to this chapter, and in Chapter 2, the training of a model can be expressed as functional operations on data. Returning to the example from Chapter 2, one can represent a hyperparameter tuning pipeline as the following:

$$(x^\star, \lambda_1^\star, \lambda_2^\star) = \underset{x,\lambda_1,\lambda_2}{\text{minimize}} \quad \|s(f(D), \lambda_1)x - b\|_2^2 + \lambda_2\|x\|_1 \tag{6.2}$$

In this modified example, the hypothetical pipeline is represented with two preprocessing steps: $s(.)$, which takes a dataset and a single hyperparameter, and $f(.)$, which takes only a dataset and is parameter free. Solving for a model for *any* instance of a hyperparameter setting, (λ_1, λ_2) relies on evaluating $f(D)$, which is parameter-free. Further, any configurations with the same value for λ_1 will rely on evaluating $s(f(D), \lambda_1)$. If the result of $s(.)$ or $f(.)$ is expensive to compute, storing their result could lead to substantial savings during hyperparameter search.

Further, adding pipeline parameters increases the complexity of the traditional hyperparameter optimization setup *dramatically* due to the curse of dimensionality. Even under the assumption that each hyperparameter is binary, if there are h hyperparameters, then there are 2^h possible hyperparameter settings for a given pipeline. In reality, the situation is much worse. Hyperparameters may be continuous, discrete, integral, or even *compound* parameters of any of the above.

6.2.3 Pipeline Tuning

The key insight behind PiperPlanned is that by analyzing pipeline structure (or the functional structure of all of the pipeline configurations to be trained by a system at runtime) the traditional black-box optimization problem can be opened up for reuse.

```
case class TransformerP[PT,I,O](name: String, x: PT => Transformer[I,O], p:
    Parameter[PT]) extends PipelineParameter
case class  EstimatorP[PT,I,E,O](name: String, x: PT => Estimator[E,O], p:
    Parameter[PT], data: RDD[I]) extends PipelineParameter
case class LabelEstimatorP[PT,I,E,O,L](name: String, x: PT => LabelEstimator[E,O
    ,L], p: Parameter[PT], data: RDD[I], labels: RDD[L]) extends
    PipelineParameter

trait PipelineParameter extends PPChainable {
  def toParameterPipeline: ParameterPipeline = {
    new ParameterPipeline(Seq(this))
  }
}

trait PPChainable {
  def andThen(p: PipelineParameter): ParameterPipeline = {
    new ParameterPipeline(this.toParameterPipeline.nodes :+ p)
  }
}
```

Figure 6.3: Pipeline hyperparameter API.

I now describe the implementation of the PiperPlanned hyperparameter tuning subsystem for KeystoneML. I then derive a an expression for the maximum reuse achievable within such a system and show how to best leverage reuse in a limited memory setting.

6.3 Implementation

I now describe the implementation of PiperPlanned and its integration with KeystoneML.

6.3.1 Search Space Description

Inspired by the syntax presented in Chapters 5 and 4 and in [79], I have implemented a set of APIs designed to allow developers to construct pipelines with a hyperparameter search space.

This API allows developers to write down pipelines in a concise and familiar syntax, chaining together parameterized nodes with `andThen` syntax, similar to the API used in KeystoneML. The API is comprised of two parts: the parameter space for the individual pipeline elements, which is captured in Table 4.1, and an API used to capture pipeline structure. The API used to describe the search space is defined in Figure 6.3.

The three new objects introduced by this API are `TransformerP`, `EstimatorP`, and `LabelEstimatorP`, each takes a function that transforms a parameter setting into a concrete instance of an operator of the appropriate type, and an appropriately typed parameter from Table 4.1. Using the `andThen` syntax from KeystoneML, developers are able to chain arbitrary search spaces together.

```
 1 val pipeSpace = TransformerP("Trim", {x: Unit => Trim}, EmptyParameter())
     andThen
 2   TransformerP("LowerCase", {x: Unit => LowerCase()}, EmptyParameter()) andThen
 3   TransformerP("Tokenizer", {x: Unit => Tokenizer()}, EmptyParameter()) andThen
 4   TransformerP("NGramsFeaturizer",
 5     {x:Int => NGramsFeaturizer(1 to x)},
 6     IntParameter("maxGrams", conf.nGramsMin, conf.nGramsMax)) andThen
 7   TransformerP("TermFrequency", {x:Unit => tf}, EmptyParameter()) andThen
 8   EstimatorP("CommonSparseFeatures",
 9     {x:Int => CommonSparseFeatures[Seq[String]](math.pow(10,x).toInt)},
10     IntParameter("commonSparseFeatures", conf.commonFeatsMin, conf.
         commonFeatsMax),
11     trainData.data) andThen
12   LabelEstimatorP("NaiveBayesEstimator",
13     {x:Double => NaiveBayesEstimator[SparseVector[Double]](numClasses,x)},
14     ContinuousParameter("lambda", conf.lambdaMin, conf.lambdaMax, scale=Scale.
         Log),
15     trainData.data,
16     trainData.labels) andThen
17   TransformerP("MaxClassifier", {x:Unit => MaxClassifier}, EmptyParameter())
```

Figure 6.4: An example pipeline search space definition.

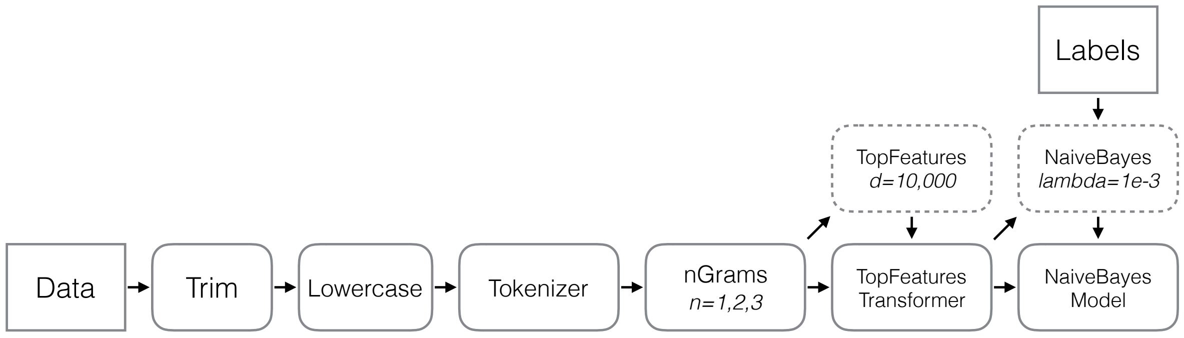
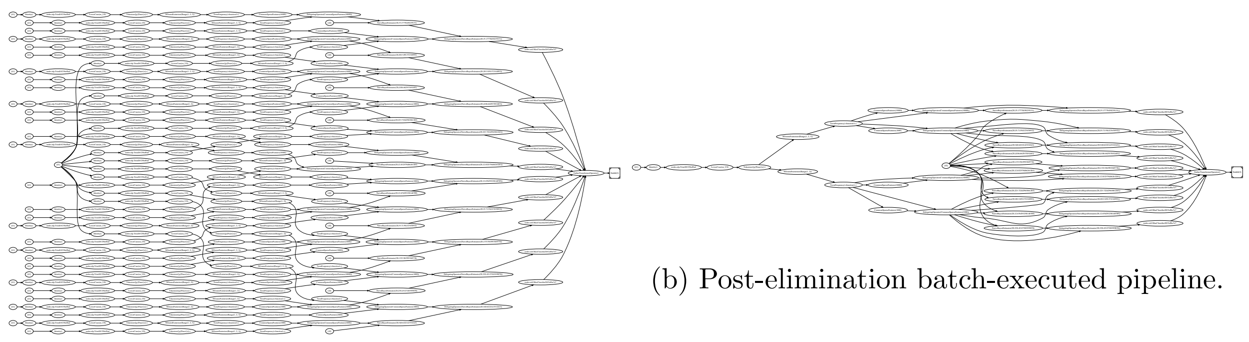


Figure 6.5: A sample hyperparameter configuration.

An example search space is shown in Figure 6.4. This pipeline is similar to the pipeline I illustrated in Figure 5.2, but is annotated with a pipeline search space. While the current syntax has substantial boilerplate code, the parameters are identified in bold in the Figure: In line 6, an `IntParameter` is associated with featurization process, in line 10 an `IntParameter` controls how many sparse features should be kept, and in line 14 a `ContinuousParameter` on a log scale determines the regularization used in the `NaiveBayesEstimator`. Of course, alternative pipeline search spaces could easily be constructed. Likewise, additional hyperparameters could be added to the existing pipeline.

Once a search space is defined, example pipelines may be *sampled* from it, and a sample pipeline is returned to be executed. The output of the `sample` is a pipeline with a single hyperparameter configuration. An example sample pipeline generated from the search space in Figure 6.4 is shown in Figure 6.5. The pipeline has not yet been *fit*, just constructed (i.e.



(a) Pre-elimination batch-executed pipeline.

(b) Post-elimination batch-executed pipeline.

Figure 6.6: Ten sample hyperparameter configurations before and after common subexpression elimination.

its parameters have not been estimated). In this chapter, the system takes advantage of this lazy execution model to sample several pipelines before fitting them in tandem.

6.3.2 Operator Graphs

The output of the `sample` procedure above is a pipeline graph. This graph represents all of the computation that needs to be done to produce a *model* capable of making predictions on unseen data.

I then introduce the concept of a *batch executor* that takes in a number of pipelines with a handle to training data and labels and a metric by which these pipelines should be evaluated.

The batch executor returns the best (estimated) pipeline to the caller among this set. It can take an arbitrary number of inputs, but here we consider 10s to 100s.

Internally, the pipeline executor takes the operator graphs as input, merges them, eliminates common prefixes (common subexpressions), all using standard KeystoneML optimizations. Before-and after snapshots of this process for 10 example pipeline configurations are shown in Figure 6.6. This figure zooms out on Figure 6.5 and shows the common structure among 10 hyperparameter configurations of the same pipeline. While the individual items in the pipeline are not identifiable, one can see that a great degree of redundancy is eliminated in this example.

6.3.3 Execution Planner

One of the most significant pieces of the system is the execution planner that looks at the (merged) execution plan and decides how to execute the plan operators and what intermediate states to materialize. The key challenge in building the execution planner has been determining the cache management policy to use to take advantage of maximum reuse when executing hyperparameter pipelines in limited-memory settings. In the next Section, I formally define this problem and provide one solution.

6.4 Optimal Reuse in Pipeline Execution

I now define and set bounds on the maximum speedups achievable due to reuse in the execution graphs that are used as input to the batch executor in the case where memory is unconstrained. I next introduce memory constraints to the execution model and introduce the concept of a *cache control policy* as a mechanism for minimizing execution time in a setting where intermediate state is too large to be entirely stored in fast memory and some must be recomputed. Finally, I frame the problem of finding an optimal cache control policy for a given dataflow graph under the execution model we described in Chapter 5 as a mixed-integer linear program (ILP).

6.4.1 Maximum Achievable Reuse

In the absence of memory constraints, execution time for a single pipeline p is simply the time required to evaluate each of its local operators, or

$$T(p) = \sum_{v \in V_p} t(v) \tag{6.3}$$

Similarly, the time taken to compute the outputs of a set of pipelines P, independently is:

$$TP(P) = \sum_{p \in P} T(p) = \sum_{p \in P} \sum_{v \in V_p} t(v) \tag{6.4}$$

For the purposes of illustration, if one holds $t(v)$ constant and assumes that each pipeline has the same length, $|V|$, it is simple to see that $TP(p) = |P||V|t(v)$, the runtime for each pipeline, is proportional to the number of operators in the pipeline.

Using the common subexpression elimination rules I discussed in Chapter 5, it is possible to merge together multiple pipelines and produce savings if the pipelines share any prefixes in common.

The system merges two expressions if:

- The root of the expressions are "equal" (that is, they are the same operation with the same hyperparameters).
- Their dependencies are all equal.

That is, two operators are equivalent if they are the same operation on the same inputs The runtime for a set of merged pipelines is $T(\cup_{p \in P}) = \sum_{v \in V_{p_{\text{merged}}}} t(v)$, where $\cup$ is the merge operator. The ratio, $\frac{TP(P)}{T(\cup_{p \in P})}$ is the speedup achieved from merging two pipelines.

Again, for the purposes of illustration, let's hold $|V|$ equal across all of the pipelines, and hold $t(v)$ fixed. If all the pipelines in the set are disjoint (that is, there is no opportunity for merging, then $TP(P) = T(\cup_{p \in P})$, and the speedup is $1\times$. On the other hand, if all the pipelines are *maximally similar* that is, they differ only in the *last* node, then the total execution time $T(\cup_{p \in P}) = |V| + |P| - 1$, yielding a maximum speedup ratio in this setting of $\frac{|V||P|}{|V|+|P|-1}$. At the limit, when all pipelines are long, speedup tends to $|P|$, the number of pipelines in the set. If one relaxes the assumption that $t(v)$ is fixed, and assumes that the *mergeable* pipelines are expensive relative to the *last* stages of the pipeline, then the speedup is $O(|P||V| - |P| - |V|)$, which is potentially substantial.

6.4.2 Memory Constraints

The previous section assumed that it is always possible to store the output of an expression in some fast memory and refer to it later without paying the cost $T(p)$ to recompute it. Unfortunately, fast memory is finite and in big data settings, the intermediate expression results can be very large. Additionally, in many applications we have studied, the cost of reading from disk is substantially more than the cost of recomputing results in memory.

In Chapter 5 I discussed an algorithm for picking a minimal set of intermediate states to materialize in order to speed up training time for a *single* pipeline. Importantly, because of the limited total number of execution stages in such a pipeline, I did *not* consider the time-varying nature of cache contents. Specifically, the algorithm picks a static, conservative set of objects to materialize before executing the pipeline.

In the hyperparameter tuning setting, this is an unacceptable restriction of the problem, because as soon as a pipeline has been trained, none of its (unique) intermediate state needs to be kept in cache, regardless of how much time it saved during the training process.

Given a *merged* pipeline, it is possible to express its execution plan as a depth-first traversal of the operator graph from desired output to input. This *linearization* is the sequence of operations that will produce the dataflow graph's desired output.

At each stage of pipeline execution, a new result is produced. Once memory is full, the system has a decision to make–does it store the current result in cache for later, and if so, what does it evict to make room? The set of such decisions taken over the lifetime of the program is a *cache management schedule*, and a schedule that minimizes total execution time is called the *optimal schedule.* An algorithm that determines an optimal cache management schedule is said to be an *optimal cache management policy.* In the current setting, this problem is seemingly made easier by the fact that the system has knowledge of the entire execution plan of the graph, and can plan ahead. Before presenting a solution, I review existing optimal algorithms. While in general it is true that perfect information about future program execution may be a necessary input to optimal policy algorithms, I show that this does not necessarily mean that computing such a policy is trivial. Classically, finding an optimal cache management policy is known as the *paging problem* [15].

Optimal Algorithms

Belady's algorithm [16] describes an optimal cache management policy for two-level cache hierarchies. The policy, simply stated is: given an input program specified as a sequence of reads, $\boldsymbol{\rho} = (\rho_1, \rho_2, \rho_3, ..., \rho_T)$, evict the item that will be next needed *furthest* in the future. If there exist any items in the cache *never* needed in the future, evict one uniformly at random. It can be proven that this simple strategy will result in the minimal number of cache misses, and thus lowest execution time.

Classically, Belady's algorithm makes two key assumptions to achieve optimality: that all pages are the same size, and that all reads have the same *cost*. This algorithm is still optimal in the case that objects have different sizes. If each $\rho_i \in \boldsymbol{\rho}$ has an integer size $s(\rho_i)$ associated with it, it is possible to simply re-write ρ_i as a length $s(\rho_i)$ sequence of smaller reads all of size 1. The weighted cost of the online algorithm is still optimal. The second assumption is more difficult to deal with, and requires a different algorithm.

LRU is an *online* algorithm that is a popular and well studied and does not need complete future knowledge of program execution to operate. It can be shown that LRU is *k-competitive* with Belady's algorithm in this setting. An algorithm is k-competitive if its worst case performance is no more than a factor of k worse than the optimal algorithm, where k denotes the size of the cache.

A particularly interesting variant of the paging problem is the *weighted paging problem*, where each $\rho_i \in \boldsymbol{\rho}$ carries with it a different *cost*, $t(\rho_i)$ of reading. Each item may be the same size (or, as in the previous subsection, cut into equal-sized chunks), but each item may have a different cost of retrieval. The objective of the *weighted paging problem* is to find a policy that minimizes the total cost associated with program execution. It can be shown that Belady's algorithm is not optimal in this setting, and that LRU is not k-competitive because it does not factor in the costs of requesting each page in. It has been shown that the optimal offline policy for the weighted paging problem can be obtained by solving an Integer Linear Program that is relaxed into a standard Linear Program [15]. However, a simple online policy where items are evicted with probability $\frac{1}{t(\rho_i)}$ can be shown to be k-competitive in this setting.

In our setting, there are two additional constraint. The first is is evident from Equation 5.1 and Equation 5.2: the *weight*, or time to calculate an item is dependent on the state of the cache at time t. The second constraint arises in our setting due to the memory model of Apache Spark, the target execution engine. Specifically, in Spark, not every intermediate output needs to fit in cache memory, because cache is a separate memory region from working memory. As such, a traditional assumption of paging problems, that all items in the read set will enter cache and evict something else does not hold. Effectively, this means there is a choice on both the *admission* and *eviction* policy. This means that the optimal policy has more flexibility, but also adds additional complexity. This is a strictly more challenging problem than the weighted paging problem, and I call this the *cache-dependent weighted paging problem*.

6.4.3 Optimal Dataflow Reuse with Constrained Memory

Given this background, I now formally define the cache-dependent weighted paging problem in the context of dataflow execution and define an optimal offline algorithm for solving it as a mixed-integer linear program. I evaluate the viability of solving this problem in the next section and use results from solving such programs on real and synthetic graphs to motivate a choice of cache management policy for PiperPlanned.

Let G denote a directed acyclic dataflow graph (DAG) consisting of vertices V and edges, E. To *execute* the graph, the system walks from the *sinks* of the graph (any $v_x \in V$ such that $v_x \neq v_i \forall (v_i, v_j) \in E$), to the *sources* of the graph (any $v_x \in V$ such that $v_x \neq v_j \forall (v_i, v_j) \in E$). For a given source and destination pair, there may be multiple paths through the dag between them. Each path is considered in the execution plan, and the current path is called the *active path.*

Consider the graph in figure 6.1.

The execution plan becomes: $A_{p=0.1}B_{p=2}C_{p=10}A_{p=0.1}B_{p=2}C_{p=5}A_{p=0.1}B_{p=4}C_{p=8}$

The *execution plan* can be thought of as all of the outputs the program would would have to compute in the absence of a cache–the only item in working memory being an operator's results and its direct inputs.

A program ($\boldsymbol{i}$) is a vector length T where i_t is the index (node id) of an operation to be executed at time t. The DAG and the execution plan jointly define a matrix A (defined below) that tells us, given the cache state, whether or not the active operation, i_t, should be counted in the execution of a program. At each time, the cost of computing a value is given by the value of the cache and the state of its successors. If there exists no un-cached node between a node i_t and the "active" sink on the current execution path, then time devoted to i_t is 0, otherwise its c_t. $c_t = C(i_t)$ where C is a cost vector mapping node id (i_t) to a non-negative real number.

In the worst case (cache size of 0), total runtime is upper-bounded by the total time to execute each operator, every time it occurs in the execution plan. For a program $\boldsymbol{i}$, this is $\sum_{i_t \in \boldsymbol{i}} C(i_t)$.

The state of a cache set at time t is:

$$X_{j,t} = \sum_{s<t} \Delta_{j,s} \forall j \in V$$

where j is the index of a node in the input DAG.

I define Δ as a sequence of valid actions on the cache, where Δ is defined as follows:

$$\Delta_{j,t} = \begin{cases} 1 & \text{if} \quad j \quad \text{is } \textit{added} \text{ to the cache at time t} \\ -1 & \text{if} \quad j \quad \text{is } \textit{removed} \text{ from the cache at time t} \\ 0 & \text{otherwise} \end{cases}$$

The set of *valid* actions on the cache is severely limited. $\Delta_{j,t}$ may only be positive if $j = i_t$, and may only be negative if $X_{j,t-1} > 0$. That is, the system can only add the active item, and can only evict items that are currently in the cache.

Let $A \in \{0,1\}^{V \times T}$ be a matrix that says whether or not a downstream operator influences the state of the cache. That is,

$$A_{j,t} = \begin{cases} 1 & \text{if} \quad j \quad \text{is on the active path between} \quad i_t \quad \text{and the sink} \\ 0 & \text{otherwise} \end{cases}$$

Specifically, $A = 1$ for the active node (e.g. if the active node is already cached and doesn't need to be recomputed), and all of its successors along the active path in the tree.

Finally, the goal is to find the cache policy (deltas) that minimizes the following expression:

$$\begin{aligned}
\underset{X}{\text{minimize}} \quad & \sum_{t \in T} c_t \max(0, 1 - X_t^\top A_t) \\
\text{subject to} \quad & \Delta_{j,0} = 0 \quad \forall j \in V, \\
& \Delta_{j,t} \leq 0 \quad \forall j \neq i_t, \\
& \Delta_{i_t,t} \geq 0, \\
& -1 \leq \Delta_{j,t} \leq 1 \quad \forall j, t, \\
& \Delta_{j,t} \in \mathbb{Z} \quad \forall j, t, \\
& X_{j,t} = \sum_{k=1}^{t} \Delta_{j,k} \quad \forall j, \\
& 0 \leq X_{j,t} \leq 1 \quad \forall j, t, \\
& 0 \leq \sum_{i=1}^{n} X_{i,t} m_i \leq M, \quad \forall t
\end{aligned}$$

That is, the goal is to minimize total runtime of the DAG where the runtime is dependent on the structure of the graph (A_t), the state of the cache at each point in time (X_t). The system counts the node in the runtime if it has to run ($\max(0, 1 - X_t^\top A_t) > 0$). The first four constraints on Δ enforce that the cache must be initially empty, that only the active node may be added to cache, and that only non-active nodes may be removed from the cache, and that Δ is bounded above and below by 1 and -1. The fifth constraint, that Δ must be integral makes this program a mixed-integer linear program as opposed to a constrained linear program. The equality constraint simply says that X is the linear sum of the *changes* to the cache at each time t, and the final constraint says that at any time t, the values of the cache, as represented by the sum of each of its objects individual sizes, must be below a positive real number, M.

The function being minimized here is a convex function of X. This can be seen as follows: 0 is convex, $1 - X_t^\top A_t$ is convex, and the *max* of any two convex functions is convex, as is the same function scaled by non-negative number. Finally, the sum of convex functions is also convex. Further, each of the constraints on the program are linear or equality constraints, with the exception of one integral constraint. As such, this program is a mixed integer linear program (ILP).

Despite the fact that solving ILPs is NP hard, several commercial solvers exist to assist in solving such problems. I implemented the above as an ILP for graphs generated by the PiperPlanned system and solve them using an off-the-shelf solver for such problems.

In the next section, I first study the practical scalability of solving these programs on current hardware on synthetic datasets, and use these to motivate a choice of cache replacement policy for PiperPlanned.

6.5 Evaluation

In this section, I evaluate the effectiveness of PiperPlanned. First, I discuss the feasibility of running the optimal solver on real world pipelines with synthetic examples. Then, I explore the expected runtime of several potential cache policies on simulated datasets and compare their performance to the lower bound given by the solution to the ILP before finally evaluating speedups achievable due to reuse on several real-world hyperparameter tuning workloads.

6.5.1 Experimental Setup

In order to study the effectiveness of different cache policies at achieving maximal reuse in the hyperparameter tuning, I have constructed a simulation framework that estimates time to execute a fully merged hyperparameter pipeline DAG under a given replacement policy at a particular memory level. The pipelines enter the simulator as DAGs with all of the necessary metadata required to estimate their execution time associated with them- specifically the local execution time for each node, memory footprint of each node's output, and number of iterations associated with each node.

I constructed these experiments on a branch of the KeystoneML master source code as of June 2016 with Scala 2.10 and Apache Spark 1.6.1. The simulation framework is written in Python 2.7 and I used Anaconda 4.0.0 with cvxpy 0.4.5 and Gurobi 6.5.2.

6.5.2 Limitations on Solving the ILP

As previously discussed, it is well known that solving integer linear programs is NP-hard. However, there are a number of commercially available ILP solvers that can be used to solve practical problems. I use one such solver, Gurobi [67], called via the cvxpy package to solve the cache-dependent weighted paging problem.

In this set of experiments, the goal is to see how far the simulator can be driven in terms of total problem complexity. I generated synthetic workloads designed to look like common hyperparameter workloads in terms of size and rough complexity. The workload is a k-ary operator graph of depth d that is designed to represent a workload of grid search with k grid points for pipelines of length d with one hyperparameter at each node. While unimportant for this setting, I assigned each operator a fixed output size of 10 memory units and a total

k	d	p	n	T	Size of X	Runtime (s)
2	2	4	7	12	84	0.5
2	3	8	15	32	480	1.3
2	4	16	31	80	2480	3.8
2	5	32	63	192	12096	12.9
3	2	9	13	27	351	3.5
3	3	27	40	108	4320	70.9
3	4	81	121	405	49005	x
3	5	243	364	1458	530712	x
4	2	16	21	48	1008	27.3
4	3	64	85	256	21760	x
4	4	256	341	1280	436480	x
4	5	1024	1365	6144	8386560	x

Table 6.1: Time to solve ILP for various pipeline sizes.

cache size of 100 units. The root operator is assigned a computational cost of 100 while all other operator have a cost of 1.

The size of each tree, n is given by the following formula:

$$n = \frac{k^{d+1} - 1}{k - 1}$$

The total number of pipelines $p = k^d$, and execution steps is given by $T = k^d(d+1)$. Therefore, variable X in the ILP solve is size $n \times T$.

Empirical runtimes of the optimal algorithm in this setup for each tree size are given in Table 6.1. Observations with an x timed out after 5 minutes.

While by no means exhaustive, this table provides guidance for when solving the offline cache policy may be worthwhile. The results show that in under 5 minutes one can effectively compute an optimal cache management policy for up to 10s of pipelines. While this may not be enough for an exhaustive grid search, it may be plenty for algorithms like Hyperband or Bayesian search, which may only need to operate on 10s of pipelines at a time.

6.5.3 Evaluating Cache Strategies

Given the results in the previous subsection, it is clear that beyond a certain limit, solving an ILP to optimize reuse for this problem may not be practical in the hyperparameter tuning setting. In this subsection I compare the optimal strategy to several online strategies across four different workloads and maximum memory sizes.

The cache strategies I evaluate are:

- **OPT:** The optimal strategy presented in this chapter.
- **LRU:** Least Recently Used, a k-optimal strategy for the paging problem
- **KEYSTONE:** The strategy presented in Chapter 5.

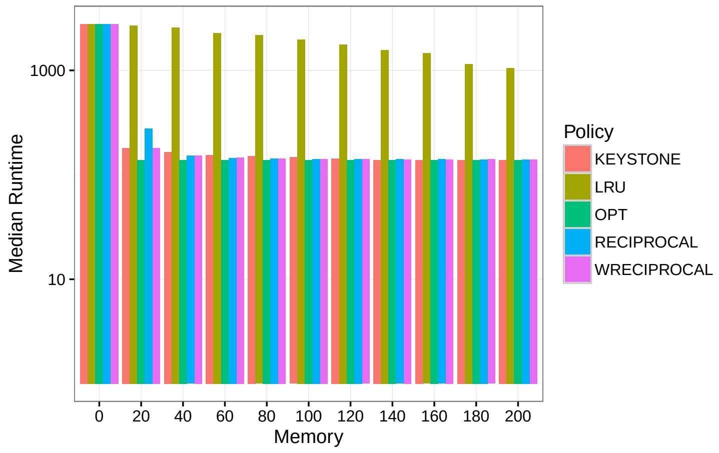


Figure 6.7: A comparison of cache management policies for 27 hyperparameter pipelines with each operator having equal memory size and execution time.

- **RECIPROCAL:** A strategy where items are evicted with probability inversely proportional to their cost to acquire. This is a k-optimal strategy for the weighted paging problem.
- **WRECIPROCAL:** A weighted variant of the reciprocal strategy where objects are evicted with probability inversely proportional to their cost to acquire and directly proportional to their size.

When considering the randomized strategies, I ran each strategy 100 times to account for variance.

The first workload I examine is the tree above where $k = 3$ and $d = 3$. Memory size of each result is still fixed at 10, and runtimes are all 1 except for the first node. Figure 6.7 shows the results of this experiment. One can see that LRU is a terrible strategy for this configuration precisely because the root node in the tree uses 100 units of computation and LRU doesn't account for this. The other strategies converge to within 10% of the execution time of the optimal strategy relatively quickly, with the KEYSTONE strategy converging on the same runtime as the optimal strategy at 140 units of memory. This plot also demonstrates the power of reuse. Even with a small amount of memory available for cache, the workload can be made 20$\times$ faster.

Next, I allowed the memory sizes to be variable, and each node is randomly given a size of either 10 or 50 units uniformly at random and a corresponding cost of 1 or 100. Figure 6.8 shows the competitiveness of each strategy vs. the optimal strategy. Competitiveness is defined as the ratio of the runtime of each strategy to the runtime of the optimal strategy. In this setting, LRU is again the least effective policy. The KEYSTONE policy is slightly less effective than the randomized policies.

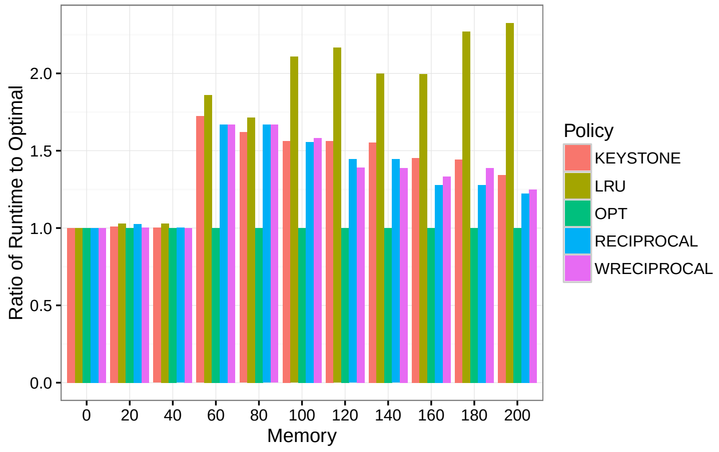


Figure 6.8: A comparison of relative runtime of cache management policies for 27 hyperparameter pipelines with each operator having equal execution time but variable memory size.

	Cache Size (GB)							
Policy	5	10	15	20	100	250	500	1500
OPT	2.85	2.85	2.86	2.86	2.86	2.94	3.00	3.00
KEYSTONE	2.85	2.85	2.85	2.85	2.86	2.94	2.98	3.00
LRU	2.08	2.09	2.09	2.09	2.86	2.94	2.92	3.00
RECIPROCAL	2.70	2.85	2.09	2.85	2.86	2.86	2.92	3.00
WRECIPROCAL	2.55	2.85	2.31	2.85	2.86	2.94	2.94	3.00

Table 6.2: A comparison of speedups achieved by several cache management strategies on the VOC workload.

Finally, I validate that the optimal strategy works on an iterative pipeline. In this case I took the VOC pipeline and uses two-pass iterative solver on its inputs. This is a *single* pipeline with limited iteration, so the savings due to reuse are modest. Nevertheless, with modest memory available for caching, one can see a savings due to reuse up to a factor of 3, and I find that aside from LRU, the strategies are all within 1% of optimal.

6.5.4 Results on Real Hyperparameter Configurations

In the evaluation, we take a number of real-world pipelines as described in Chapter 5 and construct realistic hyperparameter search spaces for each pipeline. The pipelines in question are Newsgroups [139] and TIMIT-Huang [77] datasets. In Figure 6.9 we summarize the hyperparameter spaces searched over for each of these pipelines. For these experiments, we sample random hyperparameter configurations randomly from a grid defined by each of these spaces.

We examine potential savings due to reuse for 10 and 100 configurations trained simultaneously. Given the results from the previous section, it is not feasible to find the optimal

Name	Type	Values
nGrams	Integer	(2,4)
Features	Integer	(10^3, 10^5)
λ	Continuous (log)	(0, 10^4)

(a) Search space for Newsgroups workload.

Name	Type	Values
γ	Continuous (log)	($5.5 \times 10^{-}4$, 5.5×10^4)
Distribution	Discrete	{Cauchy, Gaussian}
λ	Continuous (log)	(0, 10^5)

(b) Search space for TIMIT workload.

Figure 6.9: Hyperparameter search spaces.

	Cache Size (GB)				
Policy	0	0.05	0.1	0.5	5
KEYSTONE	1.00	9.85	16.08	18.23	18.32
LRU	1.00	4.01	6.54	18.23	19.22
RECIPROCAL	1.00	10.70	14.34	18.23	18.81
WRECIPROCAL	1.00	11.71	15.38	18.21	19.11

(a) Achievable speedups from running 10 Newsgroups pipelines simultaneously.

	Cache Size (GB)				
Policy	0	0.05	0.1	0.5	5
KEYSTONE	1.00	16.67	40.58	70.62	71.40
LRU	1.00	2.01	3.07	11.39	18.64
RECIPROCAL	1.00	18.79	23.14	70.25	71.40
WRECIPROCAL	1.00	22.34	33.78	70.59	71.86

(b) Achievable speedups from running 100 Newsgroups pipelines simultaneously.

Figure 6.10: Effect of reuse on 10 and 100 Newsgroups configurations.

policy for 100 pipelines at many memory levels in a reasonable amount of time, but as we saw, online methods tend to give reasonable performance for these sorts of workloads. However, based on calculations described in Section 6.4 of this chapter, we *are* able to bound the maximum possible speedup due to reuse *without* memory constraints. This gives us an upper bound for speedup, and a good cache management policy will reach this bound more quickly than a bad one. In each of these experiments, we plot this bound with a red line.

On the Newsgroups pipeline, seen in Figure 6.10, aside from LRU, all strategies perform well, offering up to a 72× speedup over executing without any reuse. This is because in this pipeline the *learning* part of the work is relatively light compared with the featurization part of the work. By caching, we save on several shared featurization steps: tokenization, NGrams generation, and common sparse feature estimation. By comparison, model estimation time is relatively small and so caching helps substantially. We calculate the maximum speedups for these pipelines in the presence of no memory constraints to be 19.31× and 72.19×, respectively.

On the TIMIT workload, we see lower total achievable speedups. In this workload, the *last* stage of pipeline training takes approximately the same amount of time as the featurization process. Because of this characteristic, the maximum speedup due to reuse in both cases is 9.74× and 9.75×, respectively. Eventually, the time is dominated by the slowest step (the solve), and driving the feature training time down to 0 does not help speedup any further.

In both cases, we can see that choice of cache management policy can make a big differ-

	Cache Size (GB)		
Policy	100	500	1000
KEYSTONE	1.02	1.12	1.40
LRU	1.02	9.68	9.70
RECIPROCAL	1.02	9.67	9.68
WRECIPROCAL	1.02	9.68	9.71

(a) Achievable speedups from running 10 TIMIT pipelines simultaneously.

	Cache Size (GB)		
Policy	100	500	1000
KEYSTONE	1.02	1.03	1.05
LRU	1.00	9.66	1.76
RECIPROCAL	1.02	9.66	9.67
WRECIPROCAL	1.02	9.68	9.71

(b) Achievable speedups from running 100 TIMIT pipelines simultaneously.

Figure 6.11: Effect of reuse on 10 and 100 TIMIT configurations.

ence. In particular, we see that choosing an LRU policy can lead to unpredictable effects as memory size scales due to the fact that LRU does not take into account the cost of computing the result. Meanwhile, while the KEYSTONE algorithm performs relatively well in the limited memory setting, its conservative nature means that in the larger memory settings with more iterations it does not perform as well as the randomized eviction algorithms. Thus, in the hyperparameter setting, we anticipate using the WRECIPROCAL as the default cache management algorithm in PiperPlanned.

6.6 Related Work

As discussed in Chapter 4, hyperparameter tuning has received substantial attention in the Bayesian optimization community [138, 19, 20, 78, 143]. However, these algorithms generally view the functions mapping a hyperparameter configuration to a model quality score as a black box, and make no optimizations based on the underlying computational structure relating these models. More recent works have begun to open the black box and exploit the iterative nature of these algorithms, which is an important first step [82, 98] towards solving this challenging problem. Other works have studied the incorporation of feature preprocessing into hyperparameter tuning but have ignored the opportunities for computational reuse that present themselves in these settings [57].

The study of optimal policies for cache management is a classical problem in computer science and has been extremely well studied in the theory as well as systems communities. In particular, the paging problem, which assumes the presence of a two-level memory hierarchy and seeks to find an optimal replacement policy has had a known optimal offline solution since the 1960s [16], and it is well known that online strategies are k-optimal. In the case when pages have constant but variable weights associated with them, it is known that the offline solution can be computed by solving a linear program, and that a simple randomized algorithm is k-optimal [15]. In the advanced analytics setting, the *all or nothing* property of a cache replacement algorithm has been shown to be important for these workloads, particularly because data reading tends to be a key bottleneck [8]. Online replacement policies that incorporate cost of computation and size of intermediate state have also been proposed in this setting [66]. However, none of these works have incorporated the dataflow dependency graph into their replacement policies.

6.7 Future Work and Conclusions

In this chapter, I have described the important problem of pipeline-aware hyperparameter optimization. I have demonstrated that by opening up the black box of learning pipelines, new opportunities for acceleration are presented. I studied this problem in depth and mapped it to a dataflow optimization problem. I then presented an a mixed-integer linear programming formulation to identify the optimal cache management policy for hyperparameter tuning. I then evaluated the practical viability of this strategy and compared it with existing cost-aware cache eviction policies in a simulation framework. Finally, I used this simulation framework to study the effectiveness of these policies on realistic hyperparameter search workloads.

Chapter 7

Future Work and Conclusions

In this thesis, I have described my work toward the construction of systems for practical large scale ML. This work has covered programming interfaces for distributed ML algorithm and application construction, the optimization opportunities that arise in such systems, and the integration of automatic hyperparameter tuning techniques into these systems. I have combined techniques from database query optimization, linear algebra, compilers, and distributed and high performance computing to deliver systems that are capable of training high quality learning applications significantly faster and with higher scalability than existing solutions. In this chapter I summarize the main contributions of this work and briefly discuss future challenges to be addressed in follow-on work.

7.1 Contributions

In Chapter 3 I proposed a programming model for defining scalable machine learning algorithms that also provides ML application developers with a sensible interface to these algorithms. I evaluated these interfaces from the perspective of programmer productivity via comparisons of code complexity versus a high level programming environment for single-node systems and found that by leveraging these interfaces it is possible to achieve an order of magnitude reduction in code complexity over existing distributed ML systems, while delivering end-to-end performance that is as good or better than existing solutions. The MLI system I described served as a research prototype for Apache Spark MLlib and many of the core ideas have found their way into that system.

In Chapter 4 I then focused on the problem of model search via hyperparameter tuning in the large scale setting. I evaluated several optimizations, including model-based cluster sizing, derivative-free optimization techniques, early stopping heuristics, and joint data and model batching to accelerate hyperparameter search for massive datasets on commodity clusters. I evaluated these techniques both in the context of isolated microbenchmarks as well as in the context of a model search system, TuPAQ, on two multi-terabyte training data sets.

In Chapter 5 I then evaluated the design and implementation of a system for end-to-end learning pipelines KeystoneML that allows ML application developers to specify large-scale end-to-end learning pipelines by composing a set of domain-specific and general-purpose ML operators. I discussed several potential avenues of optimization that arise in such a system and grouped them loosely into operator-level and whole-pipeline optimizations. I then used the system to reproduce several recent academic learning pipelines from a number of diverse domains, delivering high-scale throughput for problems that is up to an order of magnitude faster than existing specialized solutions and uses up to 8× fewer resources, before comparing the performance of the system to other state-of-the-art large-scale learning systems. Finally, I performed ablation studies on the various optimizations in the system and studied its scalability in detail. KeystoneML is open source software with an active community of scientific and industrial ML application developers.

Finally, in Chapter 6 I described my work on hyperparameter tuning of end-to-end pipelines in the context of PiperPlanned. This work is predicated on the simple observation that a single instance of a learning pipeline can be viewed as a computation graph and that the computations done across multiple learning pipelines can often be redundant. By combining computation graphs via standard techniques from the compiler literature is possible to achieve large savings. I provided bounds on the total savings due to reuse in the context of hyperparameter tuning, before turning my attention to finding an execution of the computation graph that would maximize savings subject to finite memory constraints. I described a mixed-integer linear programming solution to this challenging problem and used it as a tool to evaluate the effectiveness of several standard and non-standard caching strategies.

7.2 Limitations

This work has demonstrated the practical viability of systems support for large scale machine learning in modern datacenter architectures using a dataflow execution engine. Many other techniques for speeding up large scale learning exist, including scale-up solutions that use high-end servers with hundreds of cores, terabytes of memory, and specialized hardware such as GPUs and ASICs to achieve throughput. Many of these solutions are complimentary to this work but I do not evaluate the effectiveness of combining such solutions.

Further, this work does not support every flavor of machine learning. The programming model is inherently geared toward models that work over batches of static data in an iterative fashion. While this encompasses both approximate and exact classical discriminative learning methods trained via algorithms like Gauss-Seidel and flavors of Gradient and Coordinate descent, it can also be extended to support Bayesian models that are traditionally trained via Expectation-Minimization and similar algorithms, as well as discriminative models such as decision trees that do not make use of gradient-based techniques to train.

Finally, in its current form our systems do not support describing or training Deep Learning pipelines. The reasons for this are not fundamental to the programming model or implementation of the system: introducing back-propagation through our operators would be a natural extension. However, tremendous care should be taken when considering training deep learning models in a distributed setting. The size of deep learning models and communication requirements required to update them can overwhelm even the fastest available commodity networking hardware.

7.3 Future Challenges

One possible criticism of the approach presented here is that in general the applications expressed in KeystoneML and MLI make the assumption that training data is both voluminous and fixed. Many of the solutions presented here are designed with these assumptions in mind. However, initial work on extending the programming model to support incremental updates to complete pipelines indicates that the system can be cleanly integrated with existing streaming interfaces such as Spark Streaming. Early results indicate that only minor changes to the underlying APIs need to be made to offer such updates and that much of the existing optimization infrastructure can be reused in this setting. Further, a ripe area for investigation is how to take these solutions and execute them efficiently on individual machines and custom hardware to get the best of both worlds: speed when the data is small and dynamic scalability as the problem grows large.

One particularly interesting area for future work is in the realm of Deep Learning. These general techniques have been shown to achieve high statistical performance on a number of important problems in recent years. My collaborators and I have ongoing work exploring the fundamental limits of scalability of the conventional algorithms used to estimate deep learning models. This work uses bottom-up cost estimation similar to what I described in Chapter 5 to determine the limits of scalability for an input model on developer-supplied compute and networking hardware under a number of communication schemes. Communication avoiding algorithms for deep learning could prove to be a tremendously impactful research area.

The work presented in Chapter 6 on pipeline-oriented hyperparameter tuning opens the door for new research in cache optimization in the setting of dataflow graph execution. One particularly interesting research challenge is the proposal of an efficient online algorithm that is k-competitive with the optimal algorithm I propose. Another area of research is finding a convex relaxation of the proposed integer linear program that does not violate memory constraints. This would potentially enable the efficient calculation of an optimal cache policy in an offline fashion. Finally, another area of research along this line of work would be in dynamic reordering of pipeline execution to make better use of available resources. The current analysis applies to a given *fixed* execution schedule, but reordering the execution schedule would not affect program semantics and could further increase throughput. Addressed naively this is an enormous combinatorial problem, but it is a reasonable conjecture that simple algorithms exist to provide at least good approximations to optimal instruction reordering.

7.4 Final Remarks

Large Scale Machine Learning Applications have become an increasingly important use case that designers of data processing systems need to support. Once developed, these applications may completely replace existing solutions. But, the pace at which new applications can be developed is gated by developer expertise, ability to store and preprocess large quantities of data, and the rate at which ideal configurations for these applications can be discovered.

In this work, I have examined how to substantially reduce the end-to-end time to solution for a variety of large scale ML application development workloads in the context of datacenter and cloud infrastructure. While this work is necessarily an academic prototype, many of the solutions I have proposed have been adopted into real-world systems and commercial product offerings.

This work has shown that data driven decision making and semi-automatic ML application development at the scale of today's massive datasets is practical on current commodity datacenter and cloud-based computing infrastructure. Better algorithms, infrastructure, and more high quality data will lead to applications that currently seem far beyond the realm of possibility.

Bibliography

[1] S. Abu-El-Haija et al. “YouTube-8M: A Large-Scale Video Classification Benchmark”. In: *arXiv preprint arXiv: 1609.08675* (2016).

[2] Firas Abuzaid et al. “Caffe con Troll: Shallow Ideas to Speed Up Deep Learning.” In: *CoRR abs/1504.04343* (2015).

[3] Alekh Agarwal, John Duchi, and Peter Bartlett. “Oracle inequalities for computationally adaptive model selection”. In: *arXiv.org* (Aug. 2012).

[4] Sameer Agarwal et al. “Knowing when You’Re Wrong: Building Fast and Reliable Approximate Query Processing Systems”. In: *SIGMOD* (2014).

[5] Alexander Alexandrov et al. “The Stratosphere Platform for Big Data Analytics”. In: *VLDB* (2014).

[6] Gene 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.

[7] *Anaconda python distribution*. URL: `http://docs.continuum.io/anaconda/`.

[8] Ganesh Ananthanarayanan et al. “PACMan: coordinated memory caching for parallel jobs”. In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association. 2012, pp. 20–20.

[9] Edward Anderson et al. *LAPACK Users’ guide*. Vol. 9. SIAM, 1999.

[10] *Apache Mahout*. URL: `http://mahout.apache.org/`.

[11] Michael Armbrust et al. “Above the clouds: A berkeley view of cloud computing”. In: (2009).

[12] “Artificial Intelligence: Google’s AlphaGo Beats Go Master Lee Se-dol”. In: *BBC News Online* (2016).

[13] K. Bache and M. Lichman. *UCI Machine Learning Repository*. 2013. URL: `http://archive.ics.uci.edu/ml`.

[14] Grey Malone Ballard. “Avoiding Communication in Dense Linear Algebra”. PhD thesis. University of California, Berkeley, 2013.

[15] Nikhil Bansal, Niv Buchbinder, and Joseph Seffi Naor. "A primal-dual randomized algorithm for weighted paging". In: *Journal of the ACM (JACM)* (2012).

[16] Laszlo A. Belady. "A study of replacement algorithms for a virtual-storage computer". In: *IBM Systems journal* 5.2 (1966), pp. 78–101.

[17] Yoav Benjamini and Yosef Hochberg. "Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing". In: *JRSS B* (1995).

[18] Alex Berg, Jia Deng, and Fei-Fei Li. *ImageNet Large Scale Visual Recognition Challenge 2010 (ILSVRC2010)*. 2010.

[19] James Bergstra and Yoshua Bengio. "Random search for hyper-parameter optimization". In: *JMLR* (2012).

[20] James Bergstra et al. "Algorithms for Hyper-Parameter Optimization". In: *NIPS* (2011).

[21] Michael R Berthold et al. "KNIME: The Konstanz information miner". In: *Data analysis, machine learning and applications*. Springer, 2008, pp. 319–326.

[22] Dimitri P Bertsekas. *Nonlinear programming*. Athena scientific Belmont, 1999.

[23] Dimitri P Bertsekas and John N Tsitsiklis. *Parallel and distributed computation: numerical methods*. Prentice-Hall, Inc., 1989.

[24] Haran Boral et al. "Prototyping Bubba, a highly parallel database system". In: *IEEE Transactions on Knowledge and Data Engineering* 2.1 (1990), pp. 4–24.

[25] Vinayak R. Borkar et al. "Hyracks: A Flexible and Extensible Foundation for Data-Intensive Computing". In: *ICDE*. 2011.

[26] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[27] Yingyi Bu et al. "Scaling Datalog for Machine Learning on Big Data". In: *CoRR* abs/1203.0160 (2012).

[28] Sébastien Bubeck and Nicolò Cesa-Bianchi. "Regret Analysis of Stochastic and Nonstochastic Multi-armed Bandit Problems". In: *Foundations and Trends in Machine Learning* (2012).

[29] Zhuhua Cai et al. "A comparison of platforms for implementing and running very large scale machine learning algorithms". In: *SIGMOD 2014*. 2014, pp. 1371–1382.

[30] John Canny and Huasha Zhao. "Bidmach: Large-scale learning with zero memory allocation". In: *BigLearning, NIPS Workshop*. 2013.

[31] John Canny and Huashua Zhao. "Big data analytics with small footprint: squaring the cloud". In: *KDD*. 2013.

[32] Bryan Catanzaro et al. "SEJITS: Getting productivity and performance with selective embedded JIT specialization". In: *Programming Models for Emerging Architectures* 1.1 (2009), pp. 1–9.

[33] Chih-Chung Chang et al. “LIBSVM: A library for support vector machines”. In: *ACM TIST* 2 (3 2011).

[34] K. Chatfield et al. “The devil is in the details: an evaluation of recent feature encoding methods”. In: *British Machine Vision Conference*. 2011.

[35] Surajit Chaudhuri and Vivek R Narasayya. “AutoAdmin ’What-if’ Index Analysis Utility.” In: *SIGMOD* (1998).

[36] Jianmin Chen et al. “Revisiting Distributed Synchronous SGD”. In: *arXiv preprint arxiv:1604.00981* (2016).

[37] Weizhu Chen, Zhenghao Wang, and Jingren Zhou. “Large-scale L-BFGS using MapReduce”. In: *NIPS*. 2014, pp. 1332–1340.

[38] Rada Chirkova and Jun Yang. “Materialized Views”. In: *Foundations and Trends in Databases* (2012).

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

[40] Adam Coates and Andrew Y Ng. “Learning Feature Representations with K-Means”. In: *Neural Networks: Tricks of the Trade*. 2012.

[41] Adam Coates, Andrew Y Ng, and Honglak Lee. “An Analysis of Single-Layer Networks in Unsupervised Feature Learning.” In: *AISTATS* (2011), pp. 215–223.

[42] Andrew R Conn, Katya Scheinberg, and Luis N Vicente. *Introduction to Derivative-free Optimization*. SIAM, 2009.

[43] Daniel Crankshaw et al. “The Missing Piece in Complex Analytics: Low Latency, Scalable Model Management and Serving with Velox”. In: *CIDR 2015*.

[44] Andrew Crotty, Alex Galakatos, and Tim Kraska. “Tupleware: Distributed Machine Learning on Small Clusters”. In: *IEEE Data Eng. Bull* 37.3 (2014).

[45] Ryan R. Curtin et al. “MLPACK: A Scalable C++ Machine Learning Library”. In: *NIPS Big Learning Workshop*. 2011.

[46] Abhinandan S Das et al. “Google news personalization: scalable online collaborative filtering”. In: *WWW*. ACM. 2007.

[47] James Demmel et al. “Communication-optimal parallel and sequential QR and LU factorizations”. In: *SIAM Journal on Scientific Computing* 34.1 (2012), A206–A239.

[48] Jia Deng et al. “Hedging your bets: Optimizing accuracy-specificity trade-offs in large scale visual recognition”. In: *CVPR* (2012).

[49] Amol Deshpande and Samuel Madden. “MauveDB: supporting model-based user views in database systems”. In: *SIGMOD* (2006).

[50] David J. DeWitt et al. “The Gamma database machine project”. In: *IEEE Transactions on Knowledge and Data Engineering* 2.1 (1990), pp. 44–62.

[51] Ted Dunning. *Statistical identification of language.* 1994.

[52] Iman Elghandour and Ashraf Aboulnaga. “ReStore: reusing results of MapReduce jobs”. In: *PVLDB.* 2012.

[53] *ENCODE-DREAM in vivo Transcription Factor Binding Site Prediction Challenge.* 2016. URL: `https://www.synapse.org/%5C#!Synapse:syn6131484`.

[54] Eyal Even-Dar, Shie Mannor, and Yishay Mansour. “Action Elimination and Stopping Conditions for the Multi-Armed Bandit and Reinforcement Learning Problems”. In: *JMLR* (2006).

[55] Rong-En Fan et al. “LIBLINEAR: A Library for Large Linear Classification”. In: *JMLR* 9 (2008), pp. 1871–1874.

[56] Xixuan Feng et al. “Towards a unified architecture for in-RDBMS analytics”. In: *SIGMOD.* 2012.

[57] M. Feurer et al. “Efficient and Robust Automated Machine Learning.” In: *NIPS.* 2015.

[58] David A Forsyth and Jean Ponce. *Computer vision: a modern approach.* Prentice Hall Professional Technical Reference, 2002.

[59] Mark Gales and Steve Young. “The application of hidden Markov models in speech recognition”. In: *Foundations and trends in signal processing* 1.3 (2008), pp. 195–304.

[60] John Garofolo et al. “TIMIT Acoustic-Phonetic Continuous Speech Corpus”. In: (1993).

[61] Amol Ghoting et al. “SystemML: Declarative machine learning on MapReduce”. In: *ICDE.* IEEE. 2011, pp. 231–242.

[62] Joseph E Gonzalez et al. “PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs”. In: *OSDI.* 2012.

[63] *Google Prediction API.* URL: `http://developers.google.com/prediction/`.

[64] M. Grant and S. Boyd. “Graph implementations for nonsmooth convex programs”. In: *Recent Advances in Learning and Control.* Ed. by V. Blondel, S. Boyd, and H. Kimura. Lecture Notes in Control and Information Sciences. Springer-Verlag Limited, 2008, pp. 95–110.

[65] *GraphLab Create Documentation: model parameter search.* URL: `https://dato.com/products/create/docs/graphlab.toolkits.model_parameter_search.html`.

[66] Pradeep Kumar Gunda et al. “Nectar: Automatic Management of Data and Computation in Datacenters.” In: *OSDI.* Vol. 10. 2010, pp. 1–8.

[67] Inc. Gurobi Optimization. *Gurobi Optimizer Reference Manual.* 2015. URL: `http://www.gurobi.com`.

[68] Alon Halevy, Peter Norvig, and Fernando Pereira. "The unreasonable effectiveness of data". In: *Intelligent Systems, IEEE* 24.2 (2009), pp. 8–12.

[69] N Halko, P G Martinsson, and J A Tropp. "Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions". In: *SIAM Review* (2011).

[70] Venky Harinarayan, Anand Rajaraman, and Jeffrey D Ullman. "Implementing Data Cubes Efficiently." In: *SIGMOD* (1996), pp. 205–216.

[71] Trevor Hastie et al. *The Elements of Statistical Learning.* Springer, 2001.

[72] Joseph M Hellerstein and Jeffrey F Naughton. "Query execution techniques for caching expensive methods". In: *SIGMOD* (1997).

[73] Joseph M Hellerstein et al. "The MADlib analytics library: or MAD skills, the SQL". In: *PVLDB* 5.12 (2012), pp. 1700–1711.

[74] Herodotos Herodotou and Shivnath Babu. "Profiling, what-if analysis, and cost-based optimization of MapReduce programs". In: *VLDB* (2011).

[75] Geoffrey Hinton et al. "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups". In: *Signal Processing Magazine, IEEE* (2012).

[76] Chih-Wei Hsu, Chih-Chung Chang, Chih-Jen Lin, et al. *A practical guide to support vector classification.* 2003. URL: `https://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf`.

[77] Po-Sen Huang et al. "Kernel methods match deep neural networks on TIMIT". In: *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP).* IEEE. 2014, pp. 205–209.

[78] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. "Sequential model-based optimization for general algorithm configuration". In: *International Conference on Learning and Intelligent Optimization.* Springer. 2011, pp. 507–523.

[79] *Hyperopt: Distributed Asynchronous Hyperparameter Optimization in Python.* URL: `http://hyperopt.github.io/hyperopt/`.

[80] Forrest N Iandola et al. "FireCaffe: near-linear acceleration of deep neural network training on compute clusters". In: *arXiv preprint arXiv:1511.00175* (2015).

[81] Gareth James et al. *An introduction to statistical learning.* Vol. 6. Springer, 2013.

[82] Kevin Jamieson and Ameet Talwalkar. "Non-stochastic Best Arm Identification and Hyperparameter Optimization". In: *CoRR* (2015).

[83] Yangqing Jia et al. "Caffe: Convolutional Architecture for Fast Feature Embedding". In: *arXiv preprint arXiv:1408.5093* (2014).

[84] Eric Jonas et al. "Flare Prediction Using Photospheric and Coronal Image Data". In: *AGU Fall Meeting* (2016).

[85] Eddie Kohler et al. “The Click modular router”. In: *ACM Transactions on Computer Systems (TOCS)* (2000).

[86] Brent Komer et al. “Hyperopt-sklearn: Automatic hyperparameter configuration for scikit-learn”. In: *ICML workshop on AutoML*. 2014.

[87] Yehuda Koren et al. “Matrix factorization techniques for recommender systems”. In: *Computer* 42.8 (2009), pp. 30–37.

[88] Tim Kraska et al. “MLbase: A Distributed Machine-learning System”. In: *CIDR* (2013).

[89] Alex Krizhevsky et al. “Imagenet classification with deep convolutional neural networks”. In: *NIPS* (2012).

[90] Alex Krizhevsky and G Hinton. “Convolutional Deep Belief Networks on CIFAR-10”. In: *Unpublished manuscript* (2010).

[91] Max Kuhn et al. *caret: Classification and Regression Training*. R package version 6.0-41. 2015. URL: `http://CRAN.R-project.org/package=caret`.

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

[93] Arun Kumar, Feng Niu, and Christopher Ré. “Hazy: making it easier to build and maintain big-data analytics”. In: *Communications of the ACM* 56.3 (2013), pp. 40–49.

[94] John Langford, Lihong Li, and Alex Strehl. *Vowpal wabbit online learning project*. 2007.

[95] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *CGO*. 2004.

[96] C. L. Lawson et al. “Basic Linear Algebra Subprograms for Fortran Usage”. In: *ACM Trans. Math. Softw.* (1979).

[97] Yann LeCun et al. “Backpropagation applied to handwritten zip code recognition”. In: *Neural computation* 1.4 (1989), pp. 541–551.

[98] Lisha Li et al. “Efficient Hyperparameter Optimization and Infinitely Many Armed Bandits”. In: *CoRR* (2016).

[99] Mu Li et al. “Scaling Distributed Machine Learning with the Parameter Server.” In: *OSDI* (2014).

[100] Yucheng Low et al. “GraphLab: A New Framework For Parallel Machine Learning”. In: *UAI*. 2010.

[101] Yucheng Low et al. “Distributed GraphLab: a framework for machine learning and data mining in the cloud”. In: *PVLDB* 5.8 (2012), pp. 716–727.

[102] David G Lowe. “Object recognition from local scale-invariant features”. In: *ICCV*. Vol. 2. IEEE. 1999, pp. 1150–1157.

[103] Christopher D. Manning et al. “The Stanford CoreNLP Natural Language Processing Toolkit”. In: *ACL*. 2014.

[104] Christopher Manning and Dan Klein. “Optimization, maxent models, and conditional estimation without magic”. In: *HLT-NAACL, Tutorial Vol 5*. 2003.

[105] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: `http://tensorflow.org/`.

[106] Michael Mathieu, Mikael Henaff, and Yann LeCun. “Fast Training of Convolutional Networks through FFTs”. In: *ICLR* (2014).

[107] *MATLAB Distributed Computing Server*. URL: `http://www.mathworks.com/products/distriben/`.

[108] Julian McAuley, Rahul Pandey, and Jure Leskovec. “Inferring Networks of Substitutable and Complementary Products”. In: *KDD*. 2015, pp. 785–794.

[109] Jon McCallum. *Historical Cost of Memory Storage*. 2016. URL: `http://www.jcmit.com/diskprice.htm`.

[110] John D. McCalpin. “Memory Bandwidth and Machine Balance in Current High Performance Computers”. In: *TCCA Newsletter* (1995).

[111] John D. McCalpin. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Tech. rep. University of Virginia, 1991-2007.

[112] Xiangrui Meng et al. *ML Pipelines: A New High-Level API for MLlib*. 2015. URL: `https://databricks.com/blog/2015/01/07/ml-pipelines-a-new-high-level-api-for-mllib.html`.

[113] Xiangrui Meng et al. “Mllib: Machine learning in apache spark”. In: *JMLR* 17.34 (2016), pp. 1–7.

[114] Philipp Moritz et al. “SparkNet: Training Deep Networks in Spark”. In: *arXiv preprint arXiv:1511.06051* (2015).

[115] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[116] John A Nelder and Roger Mead. “A Simplex Method for Function Minimization”. In: *The computer journal* (1965).

[117] Feng Niu et al. “DeepDive: Web-scale Knowledge-base Construction using Statistical Learning and Inference.” In: *VLDS* 12 (2012), pp. 25–28.

[118] Biswanath Panda et al. “Planet: Massively Parallel Learning of Tree Ensembles with MapReduce”. In: *VLDB* (2009).

[119] Karl Pearson. “LIII. On lines and planes of closest fit to systems of points in space”. In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2.11 (1901), pp. 559–572.

[120] F. Pedregosa, G. Varoquaux, A. Gramfort, et al. “Scikit-learn: Machine Learning in Python”. In: *JMLR* 12 (2011), pp. 2825–2830.

[121] L.L. Perez and C.M. Jermaine. “History-aware query optimization with materialized intermediate views”. In: *ICDE*. Mar. 2014, pp. 520–531.

[122] Michael JD Powell. “An Efficient Method for Finding the Minimum of a Function of Several Variables Without Calculating Derivatives”. In: *The computer journal* (1964).

[123] C Qin and F Rusu. “Scalable I/O-bound Parallel Incremental Gradient Descent for Big Data Analytics in GLADE”. In: *DanaC*. 2013.

[124] *R for Hadoop*. URL: `http://www.revolutionanalytics.com/products/r-for-hadoop.php`.

[125] Prabhakar Raghavan and Marc Snir. “Memory versus randomization in on-line algorithms”. In: *International Colloquium on Automata, Languages, and Programming*. Springer. 1989, pp. 687–703.

[126] Ali Rahimi and Benjamin Recht. “Random Features for Large-Scale Kernel Machines”. In: *NIPS*. 2007.

[127] Benjamin Recht et al. “Hogwild!: A lock-free approach to parallelizing stochastic gradient descent”. In: *NIPS*. 2011, pp. 693–701.

[128] Stephan R. Richter et al. “Playing for Data: Ground Truth from Computer Games”. In: *European Conference on Computer Vision (ECCV)*. Ed. by Bastian Leibe et al. Vol. 9906. LNCS. Springer International Publishing, 2016, pp. 102–118.

[129] Tara N Sainath et al. “Exemplar-based sparse representation features: From TIMIT to LVCSR”. In: *IEEE Transactions on Audio, Speech, and Language Processing* (2011).

[130] Jorge Sanchez, Florent Perronnin, and Thomas Mensink. *Improved Fisher Vector for Large Scale Image Classification*. URL: `http://image-net.org/challenges/LSVRC/2010/ILSVRC2010_XRCE.pdf`.

[131] Jorge Sánchez et al. “Image classification with the fisher vector: Theory and practice”. In: *International journal of computer vision* 105.3 (2013), pp. 222–245.

[132] D. Sculley et al. “Machine Learning: The High Interest Credit Card of Technical Debt”. In: *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*. 2014.

[133] P Griffiths Selinger et al. “Access path selection in a relational database management system”. In: *SIGMOD* (Aug. 1979), pp. 141–152.

[134] *SHOGUN*. URL: `http://shogun-toolbox.org/page/home/`.

[135] Konstantin Shvachko et al. “The hadoop distributed file system”. In: *MSST*. IEEE. 2010.

[136] Tom Simonite. “Intel Puts the Brakes on Moores Law”. In: *MIT Technology Review* (Mar. 2016).

[137] Vikas Sindhwani and Haim Avron. "High-performance Kernel Machines with Implicit Distributed Optimization and Randomization". In: *CoRR* abs/1409.0940 (2014).

[138] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. "Practical bayesian optimization of machine learning algorithms". In: *Advances in neural information processing systems*. 2012, pp. 2951–2959.

[139] *Software/Classifier/20 Newsgroups*. URL: `http://nlp.stanford.edu/wiki/Software/Classifier/20_Newsgroups`.

[140] Evan R. Sparks et al. "Automating Model Search for Large Scale Machine Learning". In: *SoCC '15*. 2015.

[141] Evan R. Sparks et al. "KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics". In: *ArXiv e-prints* (2016). arXiv: `1610.09451 [cs.LG]`.

[142] Evan R. Sparks, Ameet Talwalkar, et al. "MLI: An API for Distributed Machine Learning". In: *ICDM*. 2013.

[143] Niranjan Srinivas et al. "Gaussian process optimization in the bandit setting: No regret and experimental design". In: *arXiv preprint arXiv:0912.3995* (2009).

[144] Michael Stonebraker et al. "Requirements for Science Data Bases and SciDB." In: *CIDR*. Vol. 7. 2009, pp. 173–184.

[145] Arvind K. Sujeeth et al. "OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning". In: *ICML*. 2011.

[146] Christian Szegedy, Vincent Vanhoucke, et al. "Rethinking the Inception Architecture for Computer Vision". In: *arXiv preprint arXiv:1512.00567* (2015).

[147] *TensorFlow CIFAR-10 Performance as reported in TensorFlow Source Code*. URL: `https://git.io/v2b4J`.

[148] Chris Thornton et al. "Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms". In: *KDD*. 2013.

[149] Engin Tola, Vincent Lepetit, and Pascal Fua. "Daisy: An efficient dense descriptor applied to wide-baseline stereo". In: *IEEE TPAMI* 32.5 (2010), pp. 815–830.

[150] Stephen Tu et al. "Large Scale Kernel Learning using Block Coordinate Descent". In: *CoRR* (2016). URL: `http://arxiv.org/abs/1602.05310`.

[151] Shivaram Venkataraman et al. "Presto: distributed machine learning and graph processing with sparse matrices". In: *EuroSys*. 2013.

[152] Shivaram Venkataraman et al. "Ernest: efficient performance prediction for large-scale advanced analytics". In: *NSDI*. 2016.

[153] *Vowpal Wabbit*. URL: `http://hunch.net/~vw/`.

[154] Daisy Zhe Wang et al. "BayesStore: managing large, uncertain data repositories with probabilistic graphical models". In: *VLDB* (2008).

[155] *WEKA*. URL: `http://www.cs.waikato.ac.nz/ml/weka/`.

[156] Peter D Welch. “The use of fast Fourier transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms”. In: *IEEE Transactions on audio and electroacoustics* 15.2 (1967), pp. 70–73.

[157] R Clint Whaley and Jack J Dongarra. “Automatically Tuned Linear Algebra Software”. In: *ACM/IEEE conference on Supercomputing*. 1998.

[158] Samuel Williams, Andrew Waterman, and David Patterson. “Roofline: An Insightful Visual Performance Model for Multicore Architectures”. In: *CACM* (2009).

[159] Yuan Yu et al. “DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language.” In: *OSDI* (2008).

[160] Matei Zaharia et al. “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing”. In: *NSDI* (2012).

[161] Ce Zhang, Arun Kumar, and Christopher Ré. “Materialization optimizations for feature selection workloads”. In: *SIGMOD*. 2014.

[162] Ce Zhang and Christopher Ré. “DimmWitted: A study of main-memory statistical analytics”. In: *PVLDB* 7.12 (2014), pp. 1283–1294.

[163] Yi Zhang, Herodotos Herodotou, and Jun Yang. “RIOT: I/O-Efficient Numerical Computing without SQL”. In: *CIDR*. 2009.

[164] Daniel C Zilio et al. “Recommending Materialized Views and Indexes with IBM DB2 Design Advisor”. In: *ICAC 2004*. May 2004.