Title
The Design and Implementation of Low-Latency Prediction Serving Systems

Author
Crankshaw, Daniel

Publication Date
2019

Peer reviewed|Thesis/dissertation

The Design and Implementation of Low-Latency Prediction Serving Systems

by

Daniel Crankshaw

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 Joseph Gonzalez, Chair
Professor Ion Stoica
Professor Fernando Perez

Fall 2019

The Design and Implementation of Low-Latency Prediction Serving Systems

Abstract

The Design and Implementation of Low-Latency Prediction Serving Systems

by

Daniel Crankshaw

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Joseph Gonzalez, Chair

Machine learning is being deployed in a growing number of applications which demand real-time, accurate, and cost-efficient predictions under heavy query load. These applications employ a variety of machine learning frameworks and models, often composing several models within the same application. However, most machine learning frameworks and systems are optimized for model training and not deployment.

In this thesis, I discuss three prediction serving systems designed to meet the needs of modern interactive machine learning applications. The key idea in this work is to utilize a decoupled, layered design that interposes systems on top of training frameworks to build low-latency, scalable serving systems. Velox introduced this decoupled architecture to enable fast online learning and model personalization in response to feedback. Clipper generalized this system architecture to be framework-agnostic and introduced a set of optimizations to reduce and bound prediction latency and improve prediction throughput, accuracy, and robustness without modifying the underlying machine learning frameworks. And InferLine provisions and manages the individual stages of prediction pipelines to minimize cost while meeting end-to-end tail latency constraints.

To my parents

Contents

List of Figures

Acknowledgments

This dissertation would not be possible without the help of so many.

To start with, I want to thank all of my teachers that got me to the starting line by encouraging me to explore and ask questions. In particular, I'd like to thank Randal Burns and Alex Szalay, who gave me my first tastes of research and were extremely generous with their time and advice.

I came to UC Berkeley for graduate school in large part due to the collaborative research environment, and I was not disappointed. I'd like to thank all of my early research mentors, including Mike Franklin, my first adviser, who always pushed me to understand why a research problem mattered, as well as Peter Bailis and Shivaram Venkataram, who offered me invaluable advice on doing research and navigating the realities of grad school. I'd like to thank Ion Stoica for all his feedback and guidance, as well as being generous enough to serve as my dissertation committee chair. And thank you to Fernando Perez for serving on my dissertation committee and Jonathan Ragan-Kelley for serving on my qualifying exam committee. Alexey Tumanov was a pleasure to collaborate with and learn from, and taught me to never stop digging into a problem. I was fortunate to work with many other collaborators over the years, including Haoyuan Li, Zhao Zhang, Ali Ghodsi, Michael Jordan, Xin Wang, Giulio Zhou, and Eyal Sela. I'd like to thank Dan Haas, Evan Sparks, Vikram Sreekanti, Johann Schleier-Smith, Kevin Jamieson, Rolando Garcia, and Neeraka Yadwalkar for many fruitful conversations. I'd also like to thank Joe Hellerstein for his periodic advice and guidance throughout my time at Berkeley.

I had the privilege of working with many incredibly talented and hardworking undergraduate and masters students. I'd particularly like to thank Corey Zumar, Simon Mo, Rehan Durrani, and Hari Balakrishnan. This work would not exist without them.

Joey Gonzalez made this thesis, and my whole PhD, possible. From my first weeks in grad school working with Joey when he was a post-doc in the AMPLab to having him proofread my dissertation as my PhD adviser, Joey has been there. Thank you for all you've taught me, and the research, mental, and emotional support you've offered me every step of the way. You're an incredibly kind and supportive mentor, and I'm so grateful that I had the opportunity to be advised by you.

I'd like to thank all of my friends for supporting me and being understanding when I would disappear for weeks at a time. Tyler, Ethan, Dan, Emily, Nick, Danny, Max, Johnny, Mac, Cecilia, thank you for keeping me sane.

Hannah, thank you so much for your love and patience as I finish this thesis.

Finally, I'd like to thank my parents for all their love and support over the years. This thesis is dedicated to you.

Chapter 1

Introduction

Machine learning is an enabling technology that transforms data into decisions by extracting patterns that generalize to new data. Much of machine learning can be reduced to learning a model – a function that maps an input (e.g. a photo) to a prediction (e.g. objects in the photo). Once trained, these models can be used to make predictions on new inputs (e.g., new photos) and as part of more complex decisions (e.g., whether to unlock a door). While there are thousands of papers published each year on how to design and train models, there is surprisingly less research on how to manage and deploy such models once they are trained. It is this latter, often overlooked, topic that is the subject of this thesis.

1.1 The Machine Learning Lifecycle

As the field of machine-learning advances, models are increasingly being deployed as part of large-scale, interactive services. While these services span application domains, technologies, and use cases, the process by which they are developed, deployed, and maintained has the same high level structure. To motivate the rest of this work, we begin by briefly examining how machine learning applications are developed and used. As a running example, consider the development of an ecommerce fraud-detection application that decides whether transactions are fraudulent using machine learning.

Model Development First, a data scientist must develop and train a model. This is illustrated in the first two stages depicted in Figure 1.1.

For example, when developing the fraud detection model, a data scientist will likely start with some exploratory data analysis to experiment with different ways to featurize the transaction. They might look at user profile data (is this a new user?), user purchase history (is this item similar to other items the user has purchased?), as well as features about the transaction (is this a foreign transaction?, are they using a suspicious payment method?, etc). They will experiment with these features to find the combination that provides the best signal and therefore most accurate model. Along with this exploratory data analysis and feature engineering, they will experiment with different model

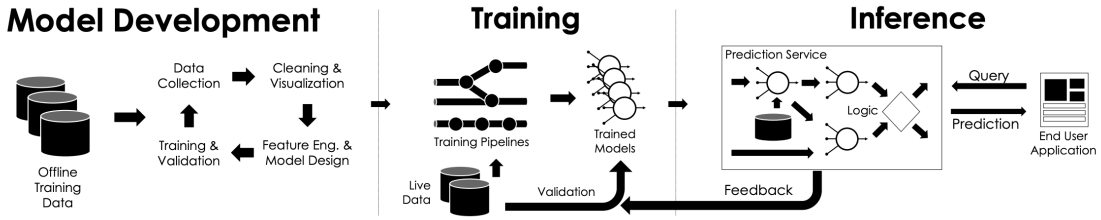


Figure 1.1: A simplified view of the machine learning model lifecycle [60]. This illustrates the three stages of a machine learning model. The first two stages encompass model development, where a data scientist first develops the model then trains it for production deployment on large dataset. Once the model is trained, it can be deployed for inference and integrated into a target application. This latter stage typically involves deploying the model to an ML serving platform for scalable and efficient inference and an application developer integrating with this ML platform to use the model to drive an application.

architectures such as random forests, logistic regression, gradient boosting, or even deep learning architectures, as well as do hyper-parameter tuning to develop the best model possible. At this point, they will perform large-scale training on the full training dataset in preparation for model deployment.

Typically, data scientists use a machine learning training framework [145, 117, 105, 159, 128, 109] for model development (see Section 1.3). These frameworks provide APIs and abstractions that simplify the model development process, and are optimized to accelerate large-scale batch training.

Once the model has been developed and trained, it will be deployed to an application to render decisions. For example, the fraud detection model will be integrated with the rest of the checkout and payments functionality of the ecommerce website and evaluated on every transaction to stop fraudulent ones from being completed.

After a first model is developed and deployed, a data scientist or team of data scientists will continue to iterate, deploying new models, experimenting with different model architectures and featurizations, and even re-using or composing models for different applications.

Integration into Applications Once a model is deployed, an application developer must integrate the model into the rest of application, using the model's predictions to drive the application, as illustrated in the Inference stage in Figure 1.1. For example, the fraud detection model must be integrated with the checkout and payment sections of the website. The ecommerce application must construct the feature vector by fetching the relevant elements from the user profile along with any other necessary information about the transaction. This data must be transformed into a numerical feature vector and provided to the model in the required format. Finally, once the model has finished evaluating the transaction, the resulting prediction must be provided back to the application. For user-facing applications such as ecommerce, the application must be able to get predictions at low-latencies (10s-100s of milliseconds) to avoid slowing down the application.

The frontend application often has the ability to observe the results of the prediction and collect feedback on the quality of the model's predictions, as illustrated by the left-pointing Feedback arrow

in Figure 1.1. This feedback can be used to improve the model quality by collecting fresh training data. It can also be used to detect when a model's accuracy has degraded and needs to be improved. For example, in the case of fraud detection, adversarial users will observe what transactions the model flags as fraudulent and modify their behavior to attempt to evade the model. When this happens, the data scientist will need to look for new features that reflect this new behavior.

Deployment and Maintenance Finally, a devops or platform engineer must transform this machine-learning application into a highly-available service that can meet the demands of the serving workload. This is the prediction service depicted in the Inference stage in Figure 1.1. They must be able to scale the service as demand changes or a user-base grows, ensure the service is reliable and meets latency and throughput Service Level Objectives (SLOs), maintain it over time, and efficiently manage physical resources to keep the service affordable. For example, if it costs more money to run the fraud detection model than the company saves by preventing fraudulent transactions, the model will not be worth deploying.

Developing a machine learning application requires the joint effort of three groups of developers with very different skill sets and responsibilities. Data scientists are experts in the models they develop, but are often not trained software engineers, and have little experience building high performance and scalable systems. Frontend engineers often have little or no experience with machine learning, but deeply understand their applications and can reason about how to use the decisions produced by models, even if they do not understand the underlying model that produces those decisions. And platform engineers have the skills to build and maintain user-facing, production services but typically focus on maintaining the service and identifying bottlenecks or problems in the application, while relying on the application developers to fix them. Furthermore, many machine learning applications, especially at smaller organizations, do not have dedicated platform engineers and instead, data scientists and frontend application developers must ensure that the application is production ready and stable themselves.

1.2 Requirements For Prediction Serving Systems

Next, I examine some critical properties of user-facing, interactive machine learning applications. These properties serve as the system design principles for building general-purpose prediction serving systems.

Low and Predictable Latencies The response time window for human interactivity lies somewhere between 100 milliseconds and 1 second, depending on the application [113]. According to Google's guidelines for web developers [81], page load time is critical for retaining users and increasing the length of a user's session. Even a 100ms decrease in page load time can have significant revenue implications [24]. As machine learning is increasingly embedded in every day user applications, it is critical that predictions can be delivered with low-latency to ensure that these services remain interactive.

Furthermore, the decisions provided by a model are typically only one part of the application processing that needs to happen before returning a response to the user. Even before a prediction can be rendered, other services such as authentication might need to run. For personalized or context-dependent predictions, user data such as recent browsing history may need to be fetched to serve as one of the inputs to the model. And once a prediction has been made, it often serves as an input for further processing. For example, once a recommendation model has decided which content should be shown, the content itself must be fetched from a data store and returned to the user. Finally, the latency of a user communicating with an application server over a wide area network (WAN) connection can be high and variable. In order to provide a latency buffer to allow for this uncontrollable latency overhead, applications will have internal SLOs lower than the human interactivity window. In this work, we target latency SLOs of 10s to 100s of milliseconds, supporting SLOs as low as a few milliseconds when serving lightweight models. Latency SLOs lower than that are better served by embedding a model directly in an application, rather than calling out to a separate service and are more typical of realtime control systems.. And latency SLOs higher than a few hundred milliseconds indicate that the response is not blocking a latency-critical application and can be served by streaming systems.

Finally, modern web and mobile applications are often built with a scaleout microservice architecture. Microservice architectures enable individual services to be both developed and provisioned independently. For example, a modern web application may have an authentication service, multiple data storage services (e.g. one for large static content with a large cache, another for user-specific data keyed by user), a machine learning service hosting models making many different decisions, a logging service that records all the user's interactions, etc. When a request comes into the main application service, the application calls out to each of these services (which may call out to other services in turn), waiting for all of the responses before assembling the final response to the user. The result is a microservice architecture with a high degree of fanout. This means that the latency of the final response to the user depends on the latency of *all of these services*, as the application must wait until all of them have returned before it can respond. Because of this, it is insufficient to simply minimize the average latency of a service. Instead, it is critical to both minimize and bound the tail latency in order to build response applications out of these microservice building blocks. These requirements are formalized in service level objectives (SLOs) that specify an upper bound on the tail latency distribution (e.g. 95% or 99% of all requests must have a latency below the SLO bound).

Requirement: Prediction serving systems must be able to meet tail latency SLOs on the order of 10s to 100s of milliseconds.

High Throughput and Scalability Machine learning is increasingly integrated into everyday applications (see Section 1.3). For example, as of 2016, Google Translate translated 140 billion words per day [92], a number that has only increased since then. As machine learning becomes an integral part of these applications, it must be able to serve the enormous volume of requests that many modern web applications experience. And it must provide the same scalability properties as the other components of a modern web serving tier, including the ability to be easily scaled up in

response to unexpected traffic increases or periodic changes in user behavior (e.g. the diurnal web browsing behavior of North America). In addition, much of the value of machine learning is to make applications more intelligent and therefore more valuable. A good model (e.g. for recommending content) can increase the popularity of an application and therefore the demand for the application, which increases the workload that the model must serve, creating a positive feedback loop.

Requirement: Prediction serving systems must provide low-latency predictions, efficiently use hardware resources to minimize cost, and scale to sustain high throughput and dynamic workloads.

Accurate and adaptable Many of the most common machine learning applications are user-facing applications with millions or even billions of users (e.g., Facebook and Google). With such a large number of diverse users, the best decision will vary on a per-user basis. These applications all use some form of model personalization to customize predictions. And even for a single user, the best-decision to make is often context dependent. For example, depending on a user's current activity, the right items for Amazon to recommend for them to buy will vary. Similarly, depending on the user's mood, the right songs for Spotify to include in a personalized radio station will change. As a result, the best machine learning models use context, whether that is a user's identity (personalization), recent usage history (contextualization), or some other form of context clue to improve the accuracy of a model.

Even seemingly context-oblivious applications such as object detection in images can benefit from having additional context. For example, it is common to take a computer vision or natural language model and first train it on a large, general-purpose training dataset, then fine-tune [156] it to improve its accuracy on a more specific application (e.g. detecting cars in images from a specific camera).

Finally, the act of deploying a model into an application can change user behavior as they adapt their behavior to the idiosyncrasies of a specific model. This creates feedback loops which can be exploited [12] to actually improve model accuracy, but also has the effect of causing models to become stale over time [56, 72]. Furthermore, data science and model development is itself an iterative activity [162] with new models being deployed periodically (anywhere from every few minutes to nightly to weekly to quarterly).

Requirement: Prediction serving systems should enable new models to be deployed automically without a human in the loop, as well as supporting personalized and contextual predictions.

Affordable While making a single prediction is much cheaper than training a model, the scale at which machine learning applications are deployed means that the aggregate computational cost of an inference workload can be very high. Especially with the growing adoption of deep learning models (Section 1.3), inference is often the dominant computational cost in applications that use machine learning. Furthermore, there are a growing number of specialized hardware accelerators for inference. These hardware accelerators are quite expensive when compared to a CPU, but can often provide predictions at much lower latencies and higher throughput when compared to CPUs, justifying their cost. However, in order to use these accelerators efficiently, predictions must be processed in batches to utilize the parallel vector processing units of these accelerators. Specialized

accelerators are most useful when serving computaionally expensive deep learning models, but even classical machine learning models running on CPUs can have a large computational footprint leading to high costs.

Finally, properly tuning a machine learning application to minimize cost is complex and often outside the expertise of both application developers building the application and data scientists developing the model. Doing this correctly requires an understanding of the underlying hardware available, knowledge of the application requirements (latency SLOs and throughput requirements), and an understanding of the specific performance characteristics of the trained model. Any time any of these factors changes (e.g., a model with a new architecture is deployed), the application must be reconfigured.

Requirement: Prediction serving systems should automatically maximize resource efficiency to minimize inference cost while still meeting other application constraints.

1.3 Trends in Modern Machine Learning

Over the last several years, there has been an explosion in the use of machine learning in standard applications. Prior to that, the most common uses of machine learning were in sophisticated but highly proprietary ad-targeting systems and search engines, or were recommender systems targeting applications like Netflix [111]. The first version of Google Translate was released in 2006 and the initial version of Apple's Siri virtual assistant was first released as a third party app in 2010 and integrated into Apple's iOS operating system in 2011.

Fifteen years later in 2019, nearly every widely used user-facing technology uses machine learning as an integral part of the application. These include ad-targeting, video monitoring, content recommendation ranging from Amazon's shopping suggestions to Netflix's personalized home page to Spotify's personalized playlists, intelligent virtual assistants such as Amazon's Alexa, Apple's Siri, Google's Home, Microsoft's Cortana, smart appliances such as home video monitoring systems or intelligent thermostats such as Nest.

Machine learning is also widely adopted in industrial and enterprise settings, including ways to make sales, marketing, and customer support be more efficient [125] and improving manufacturing efficiency [27]. Finally, machine learning, especially deep neural network computer vision models and reinforcement learning are integral to making autonomous vehicles a reality. Machine learning is even finding uses in highly skilled professional fields such as medicine [131] where it is helping with diagnostics and personalized treatment plans, and helping lawyers do legal research [123].

This explosion in adoption of machine learning has many factors. The rise of elastic cloud computing and the excitement around "Big Data" in the 2000s and early 2010s led to a renewed interest in machine learning research, as previously intractable problems could now be attacked with orders of magnitude more data and compute power. Additional data and compute resources, along with the work of a very clever grad student to utilize the parallel processing power of GPUs to train deep neural networks [85], led to the rise of deep learning. Deep neural networks have achieved huge increases in accuracy in many tasks, especially for sensory perception models such as speech, natural language, and computer vision. Finally, much of modern life is conducted in digital

spaces and is mediated by technology. Billions of people using the Internet creates an enormous demand for better technology, leading to widespread investment across both academic researchers and industry in using machine learning to improve the functionality, usability, and efficiency of digital life.

Understanding how machine learning is developed and where it is going is critical for designing prediction serving systems. In the remainder of this section, I discuss three trends that have accompanied the recent widespread adoption of machine learning and that have informed the design of the systems I discuss in this work.

Proliferation of ML Frameworks

As part of the growing adoption of machine learning, there have been many machine learning frameworks developed to help train models. Several of these frameworks have come out of the research community, initally developed to facilitate a researcher or research group's own work. Many of these earlier machine learning frameworks were targeted at supporting specific types of models or applications, namely those the developers were most interested in studying [61, 79, 100, 74, 17, 38]. State-of-the-art machine learning results were often achieved in these frameworks, which were nearly always open-sourced but were often closer to research prototypes than industrial strength systems.

As machine learning made its way from research labs to industry, many of these open-source research systems became quite popular for developing models in industry. This lead to getting many more open-source contributors who improved both the stability and functionality of the software. Industrial users of these frameworks started to sponsor the projects, allocating some of their own employees to work on the open-source system, or even adopting them as their own, hiring the creators and maintainers of the software but having them continue to work on these projects full time on behalf of the company [109, 74, 159].

In addition, several large technology companies at the cutting edge of industrial machine learning, including Google (TensorFlow [145]), Facebook (PyTorch [117]), and Uber (Horovod [130]) started to develop their own frameworks internally and then open-sourced them. This approach allows the researchers at these companies, who often come from academic and open-source backgrounds, to contribute back to the community. It also creates network effects around the use of these frameworks that directly benefit the companies. For example, having a strong ecosystem around TensorFlow increases the value of Google's cloud offering which has native support for TensorFlow models, including the ability to run on Google's proprietary TPU hardware accelerator [76]. And for all these companies, the more researchers and students that use that company's machine learning framework, the more likely it is that state-of-the-art ML results will be developed in that framework and the larger the pool of potential future employees already trained to use the companies internal ML infrastructure.

The result of all this activity is a rich ecosystem of machine learning training frameworks specialized for different use cases and available in different programming languages. Data scientists have many tools available to choose from, depending on their use case. They might use Scikit-Learn for early prototyping or on simple problems, which has a massive library of classical machine

learning models as well as lots of support for feature engineering, hyper-parameter tuning, and model selection. On the other hand, if they want to train a new deep learning model or research new DNN architectures, they might turn to PyTorch or TensorFlow which are specialized for training neural networks and offer support for distributed training on specialized hardware such as GPUs. Or if they come from a statistics or bio-informatics background, they might prefer to use R which is popular in those communities.

However, all of these frameworks have a few design goals in common: namely they are optimized for the model development and training portions of the machine learning lifecycle. Their users are those in the ML research and data science communities, so functionality and interfaces for these frameworks are intended to simplify and facilitate model development. As a result, the performance optimizations in these frameworks are focused on maximizing throughput and reducing model training time. They exploit data-parallel optimizations and batched computation to maximize throughput, such as using efficient parallel implementations of the linear algebra routines [21, 87] that forms the basis of most machine learning optimization algorithms.

The Rise of Deep Learning

A second trend is the rise of deep learning, starting with Alex Krizhevsky's resounding win [85] in the ImageNet competition [124] in 2012. That paper had two critical results. It found that the depth of the model was critical for attaining high accuracy, and that they could use GPUs to accelerate model training. Since then, thousands of papers on deep learning have been written, many of which employ GPUs or other specialized hardware to train the models. Deep learning models have achieved huge advances in state of the art accuracy on common tasks ranging from image classification using convolutional neural networks (the subject of Krizhevsky's original AlexNet paper), to more sophisticated computer vision tasks such as object detection and tracking, to natural language processing and machine translation, to content recommendation with sparse neural networks, and many other applications.

At this point, it is important to note that deep learning, and even machine learning more broadly, is not a magical technology that is automatically the right fit for a problem. Deep learning requires massive datasets to train an accurate model and expertise to tune the many hyperparameters (everything from model architecture to learning rate to stopping time) critical for training an accurate model. There are many applications of deep learning where a well-understood and relatively simple model will be adequate to the decision problem, as well as being simpler to train and requiring less data.

But there are many tasks where deep neural networks provide much higher accuracy than any other known approaches. Deep learning is behind many of most common widely used applications. Facebook uses it to create your Newsfeed and detect objectional content. Google uses it in applications ranging from search to photos to Google Translate. And hundreds of other companies and applications are finding ways to leverage deep learning.

While deep learning comes with the promise of increased accuracy, it comes at a cost as well. Deep learning models are often extremely computationally expensive, requiring tens of billions of floating point operations to render a single prediction, with computational cost often increasing

super-linearly with model accuracy [152]. Furthermore, the inputs to deep learning models are often large. In domains like computer vision where the inputs are images or video, or in speech processing where the inputs are audio files, the cost of simply moving these large inputs around a cluster for distributed training can add substantial overhead.

Specialized Hardware

Due in large part to the rise of deep learning, there has been a significant investment in developing compute hardware for machine learning workloads. These accelerators often have twin goals of faster computation and lower power consumption.

Starting in 2012 with Alexnet [85], GPUs have been widely used for both training and serving machine learning models. NVIDIA has capitalized on this renewed interest to build architecture level support for machine learning into their GPUs such as mixed precision matrix multiply and accumulate [114], as well as developing software libraries such as cuDNN [44] and cuBLAS [43] to help machine learning developers effectively leverage the specialized GPU hardware. Furthermore, GPUs are widely available in all of the major cloud computing platforms and can be purchased off the shelf and easily installed, making them the easiest hardware accelerators for machine learning researchers and developers to gain access to. However, while GPUs offer substantial performance gains over CPUs for both machine learning training and inference, they are still quite expensive and have high power consumption. As a result, companies doing machine learning at scale have begun to invest in even more specialized hardware to reduce the hardware cost and power consumption of their massive machine learning workloads.

The first major specialized accelerator specifically for machine learning was Google's Tensor Processing Unit (TPU) [76]. Google began work on the TPU in 2013 and had deployed it to run production workloads by 2015. They designed the TPU to be able to perform model inference with lower latency and cost as well as be rolled out quickly in Google's existing datacenters. Even the first version of the TPU was able to achieve order of magnitude better performance/Watt over contemporary CPUs and GPUs. Google has created two new versions of the TPU since then, and they now support both training and inference. TPUs run many of Google's internal machine learning workloads and can be used by customers of Google Cloud Platform to run their own machine learning workloads.

Since the TPU was announced, several other companies have started work on their own specialized processors.

Facebook has created an Open Compute Project accelerator module form factor to enable multiple hardware vendors to develop their own specialized accelerators that will all be compatible [51]. It has also begun development of its own specialized inference accelerator, although has released few details about the design [50].

In November of 2018, Amazon announced that it was developing the Inferentia inference accelerator to provide high performance inference at low cost as part of Amazon Web Services investment in supporting machine learning workloads [71]. They are now available in the EC2 Inf1 instance type.

Microsoft's Project Brainwave is an effort to use FPGAs to efficiently perform machine learning inference at low batch sizes [36]. While GPUs offer high performance inference, they require increasingly large batch sizes to maximize resource utilization. These larger batch sizes increase latency, which presents problems for real-time systems. The Brainwave architecture achieves over an order of magnitude improvement in latency and throughput compared to GPUs at a batch size of one. Brainwave models are used in Bing search and are available externally to Microsoft Azure cloud customers.

1.4 Decoupling the Serving System

In this thesis, I argue that modern machine learning applications demand a new class of systems infrastructure: the *prediction serving system*. This thesis introduces three such systems designed to provide low-latency and accurate predictions at scale.

The key design principle for this new class of infrastructure is to adopt a decoupled design that layers systems on top of training frameworks to achieve serving requirements. First, by decoupling the specific model implementation from the serving system, we can lift the serving functionality above the models to build general-purpose systems that can serve arbitrary models. General-purpose systems allow data scientists to deploy models from any of the myriad machine learning frameworks in use today. Second, encapsulating the model implementation – including all software dependencies – behind a uniform interface by leveraging containerization [48] both simplifies the model deployment process for data scientists as well as enabling the system to make automated decisions about how to run each model to minimize cost and latency. This in turn opens the door for automated resource provisioning and self-scaling systems. Further, it ensures that ML applications deployed on prediction serving infrastructure are already compatible with new developments in specialized hardware without needing to be rewritten. And finally, by abstracting away ML framework heterogeneity we enable cross-framework model composition and can efficiently serving heterogeneous prediction pipelines. We can even perform online model exploration and adaption across these heterogenous pipelines.

1.5 Summary and Contributions

In the rest of this thesis, I discuss three prediction serving systems. I first discuss Velox (Chapter 2), which explores the decoupled serving architecture in the context of serving models trained with Apache Spark. Velox focuses on easing the model deployment process and introduces methods for dynamically personalizing and updating models online in response to feedback. I then discuss Clipper (Chapter 3), which extends Velox's layered architecture beyond Apache Spark models by introducing a simple, cross-language prediction API to deploy models from any machine learning framework. Clipper also introduces a set of system optimizations to reduce and bound prediction latency and improve prediction throughput, accuracy, and robustness without modifying the underlying machine learning frameworks. Finally, I introduce InferLine (Chapter 4), which

provisions and manages ML prediction pipelines to minimize cost subject to end-to-end tail latency constraints through the use of a low-frequency combinatorial planner combined with a high-frequency auto-scaling tuner. The low-frequency planner leverages stage-wise profiling, discrete event simulation, and constrained combinatorial search to automatically select hardware type, replication, and batching parameters for each stage in the pipeline. The high-frequency tuner uses network calculus to auto-scale each stage to meet tail latency goals in response to changes in the query arrival process. I conclude with a discussion of some of the lessons learned from building these systems and some future directions for the research.

In summary, the contributions of this thesis are:

- The introduction of a decoupled and layered architecture for general-purpose prediction serving systems. The decoupled architecture is first explored in Velox's split model decomposition Chapter 2, generalized to arbitrary models and frameworks in Chapter 3, and extended to prediction pipelines in Chapter 4.
- Interfaces and abstractions for online learning, personalization, and model selection (Chapter 2 and Chapter 3).
- System optimizations to reduce and bound prediction tail latency for single model serving systems (Chapter 3).
- An accurate end-to-end latency estimation procedure for prediction pipelines spanning multiple hardware and model configurations (Chapter 4).
- A workload-aware low-frequency pipeline planner that minimizes hardware costs subject to end-to-end latency constraints (Chapter 4).
- A high-frequency auto-scaling tuner that monitors the query arrival process and scales prediction pipelines to maintain high SLO attainment at low cost under dynamic workloads (Chapter 4).

Chapter 2

The Missing Piece in Complex Analytics: Low Latency, Scalable Model Management and Serving with Velox

2.1 Introduction

The rise of large-scale commodity cluster compute frameworks has enabled the increased use of complex analytics tasks at unprecedented scale. A large subset of these complex tasks, which we call *model training* tasks, facilitate the production of statistical models that can be used to make predictions about the world, as in applications such as personalized recommendations, targeted advertising, and intelligent services. By providing scalable platforms for high-volume data analysis, systems such as Hadoop [13] and Spark [159] have created a valuable ecosystem of distributed model training processes that were previously confined to an analyst's R console or otherwise relegated to proprietary data-parallel warehousing engines. The database and broader systems community has expended considerable energy designing, implementing, and optimizing these frameworks, both in academia and industry.

This otherwise productive focus on model training has overlooked a critical component of real-world analytics pipelines: namely, how are trained models actually deployed, served, and managed? Consider the implementation of a collaborative filtering model to recommend songs for an online music service. We could use a data-parallel modeling interface, such as MLbase [83] on Spark [159], to build a model of user preferences (say, in the form of a matrix representing predictions for user-item pairs) based on historical data—a batch-oriented task for which Spark is well suited and optimized. However, if we wish to actually *use* this model to deliver predictions on demand (e.g., as part of a web service) on interactive timescales, a data-parallel compute framework such as Spark is the wrong solution. Batch-oriented designs sacrifice latency for throughput, while the mid-query fault tolerance guarantees provided by modern cluster compute frameworks are overkill and too costly for fine-grained jobs. Instead, the overwhelming trend in industry is to simply dump computed models into general-purpose data stores that have no knowledge of the model

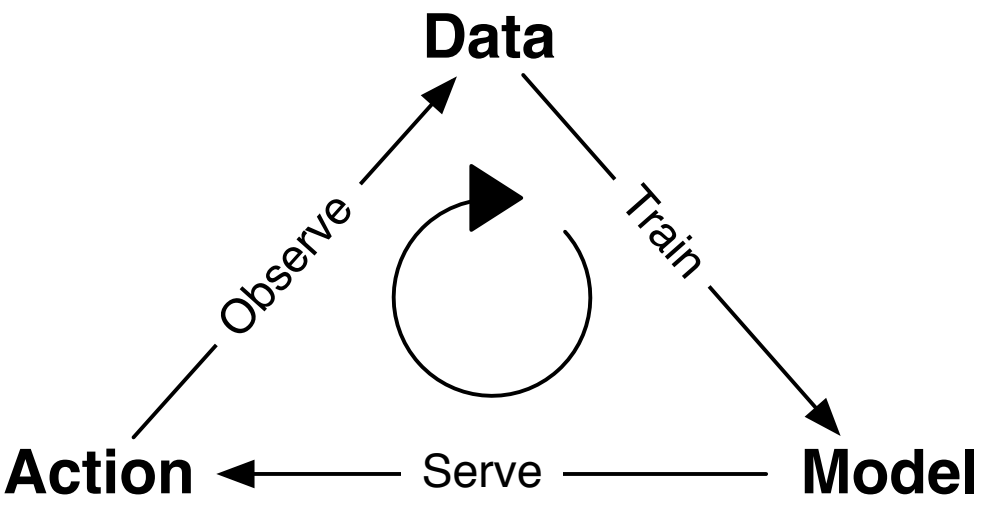


Figure 2.1: Velox Machine Learning Lifecycle Velox manages the ML model lifecycle, from model training on raw data (the focus of many existing "Big Data" analytics training systems) to the actual actions and predictions that the models inform. Actions produce additional observations that, in turn, lead to further model training. Velox facilitates this cycle by providing functionality to support the missing components in existing analytics stacks.

semantics. The role of interpreting and serving models is relegated to another set of application-level services, and the management of the machine learning life cycle (Figure 2.1) is performed by yet another separate control procedure tasked with model refresh and maintenance.

As a data management community, it is time to address this missing piece in complex analytics pipelines: machine learning model management and serving at scale. Towards this end, we present Velox, a model management platform within the Berkeley Data Analytics Stack (BDAS). In a sense, BDAS is prototypical of the real-world data pipelines above: prior to Velox, BDAS contained a data storage manager [94], a dataflow execution engine [159], a stream processor, a sampling engine, and various advanced analytics packages [139]. However, BDAS lacked any means of actually serving this data to end-users, and the many industrial users of the stack (e.g., Yahoo!, Baidu, Alibaba, Quantifind) rolled their own solutions to model serving and management. Velox fills this gap.

Specifically, Velox provides end-user applications with a low-latency, intuitive interface to models at scale, transforming the raw statistical models computed in Spark into full-blown, end-to-end data products. Given a description of the statistical model expressed as a Spark UDF, Velox performs two key tasks. First, Velox exposes the model as a service through a generic model serving API providing low latency predictions for a range of important query types. Second, Velox keeps the models up-to-date by implementing a range of both offline and online incremental maintenance strategies that leverage both advances in large-scale cluster compute frameworks as well as online and bandit-based learning algorithms.

In the remainder of this paper we present the design of the Velox system and present observations from our initial, pre-alpha implementation. In Section 2.2, we describe the key challenges in the

design of data products and outline how Velox addresses each. In Section 2.3, we provide an overview of the Velox system, including the serving, maintenance, and modeling components. We discuss our current design and implementation of each in Sections 2.4 through 2.6. Finally, we survey related work in Section 2.7 and conclude with a discussion of ongoing and future work in Section 2.8.

2.2 Background and Motivation

Many of today's large scale complex analytics tasks are performed in service of *data products*: applications powered by a combination of machine learning and large amounts of input data. These data products are used in a diverse array of settings ranging from targeting ads and blocking fraudulent transactions to personalized search, real-time automated language translation, and digital assistants.

An example application: To illustrate the implications of data product design on data management infrastructure, and, as a running example of a data product, consider the task of building a service to suggest songs to users. This music suggestion service is an example of a *recommender system*, a popular class of data products which target content to individuals based on implicit (e.g., clicks) or explicit (e.g., ratings) feedback.

To begin making recommendations, we start with a *training dataset* that contains an existing set of user ratings for songs (e.g., songs that interns have manually labeled) and fit a statistical model to the training set. After this initial training, we iteratively improve the quality of the model as we collect more ratings.

There are numerous techniques for modeling this data; in this example, we consider widely used matrix factorization models [82]. At a high level, these models embed users and songs into a high-dimensional space of latent factors and use a distance metric between each as a proxy for similarity. More formally, these models *learn* a d-dimensional latent factor $w_u \in \mathbb{R}^d$ for each user u (collected into a matrix $W \in \mathbb{R}^{|\text{users}| \times d}$) and $x_i \in \mathbb{R}^d$ for each song i (and corresponding $X \in \mathbb{R}^{|\text{songs}| \times d}$). These latent factors encode information about unmeasured properties of the user (e.g., *DeadHead*) and song (e.g., *PartyAnthem*) and are learned by solving the following optimization problem:

$$\arg\min_{W,X} \lambda \left(||W||_2^2 + ||X||_2^2 \right) + \sum_{(u,i) \in \text{Obs}} (r_{ui} - w_u^T x_i)^2$$

Given the W and X matrices, we can calculate a user u's expected rating for a song i by appropriately projecting W and X to yield u's weights w_u and i's weights x_i, and taking their dot product:

$$\text{rating}(u,i) = w_u^T x_i$$

Therefore, an implementation of our data product has two natural phases. The first calculates the optimal W and X matrices containing factors for each user and item. The second uses W and X to make predictions for specific user-item pairs.

Similar to many companies providing actual data products, we could implement our song recommendation service by combining cluster compute frameworks with traditional data management

and serving systems. For the training phase, we might compute W and X periodically (e.g., daily) using a large-scale cluster compute framework like Spark or GraphLab [61] based on a snapshot of the ratings logs stored in a distributed filesystem like HDFS [133]. In this architecture, models are trained on stale data and not updated in response to new user feedback until the next day.

For the serving phase, there are several options. The simplest strategy would precompute all predictions for every user and song combination and load these predictions into a lower-latency data store. While this approach hides the modeling complexity from the serving tier, it has the disadvantage of materializing potentially billions of predictions when only a small fraction will likely be required. Alternatively, a more sophisticated approach might load the latent factors in to a data management system and compute the predictions online in the application tier. Given the current lack of a general-purpose, scalable prediction service, this is likely to be an ad-hoc task that will be repeated for each data product.

Challenges and Opportunities

While the above straw-man proposal is a reasonable representation of how users today implement data products, there are several opportunities for improving the model management experience. Here, we highlight the challenges in managing a data product that are not addressed by either traditional offline analytics systems or online data management systems.

Low Latency: Because many data products are consumed by user-facing applications it is essential that they respond within the window of interactivity to prediction queries and quickly learn from new information. For example, a user listening to an online radio station expects their feedback to influence the next songs played by the music service. These demands for real-time responsiveness both in their ability to make predictions and learn from feedback are not well addressed by traditional cluster compute frameworks designed for scalable but batch-oriented machine learning. Additionally, while data management systems provide low latency query serving, they are not capable of retraining the underlying models.

Velox's approach: Velox provides low latency query serving by intelligently caching computation, scaling out model prediction and online training, and introducing new strategies for fast incremental model updates.

Large scale: With access to large amounts of data, the machine learning community has developed increasingly sophisticated techniques capable of modeling data at the granularity of individual entities. In our example recommender service, we learn factor representations for individual users and songs. Furthermore, these latent factor representations are interdependent: changes in the song factors affects the user factors. The size and interdependency of these models poses unique challenges to our ability to serve and maintain these models in a low latency environment.

Velox's approach: Velox addresses the challenge of scalable learning by leveraging existing cluster compute frameworks to initialize the model training process and infer global properties offline and then applies incremental maintenance strategies to efficiently update the model as more data is observed. To serve models at scale, Velox leverages distributed in memory storage and computation.

Model lifecycle management: Managing multiple models and identifying when models are underperforming or need to be retrained is a key challenge in the design of effective data products. For example, an advertising service may run a series of ad campaigns, each with separate models over the same set of users. Alternatively, existing models may no longer adequately reflect the present state of the world. For example, a recommendation system that favors the songs in the Top 40 playlists may become increasingly less effective as new songs are introduced and the contents of the Top 40 evolves. Being able to identify when models need to be retrained, coordinating offline training, and updating the online models is essential to provide accurate predictions. Existing solutions typically bind together separate monitoring and management services with scripts to trigger retraining, often in an ad-hoc manner.

Velox's approach: Velox maintains statistics about model performance and version histories, enabling easier diagnostics of model quality regression and simple rollbacks to earlier model versions. In addition, Velox uses these statistics to automatically trigger offline retraining of models that are under performing and migrates those changes to the online serving environment.

Adaptive feedback: Because data products influence the actions of other services and ultimately the data that is collected, their decisions can affect their ability to learn in the future. For example, a recommendation system that only recommends sports articles may not collect enough information to learn about a user's preferences for articles on politics. While there are a range of techniques in the machine learning literature [153, 95] to address this feedback loop, these techniques must be able to modify the predictions served and observe the results, and thus run in the serving layer.

Velox's approach: Velox adopts bandit techniques [95] for controlling feedback that enable the system to optimize not only prediction accuracy but its ability to effectively learn the user models.

2.3 System Architecture

In response to the challenges presented in Section 2.2, we developed Velox, a system for serving, incrementally maintaining, and managing machine learning models at scale within the existing BDAS ecosystem. Velox allows BDAS users to build, maintain, and operate full, end-to-end data products. In this section, we outline the Velox architecture.

Velox consists of two primary architectural components. First, the **Velox model manager** orchestrates the computation and maintenance of a set of pre-declared machine learning models, incorporating feedback and new data, evaluating model performance, and retraining models as necessary. The manager performs fine-grained model updates but, to facilitate large scale model re-training, uses Spark for efficient batch computation.

Second, the **Velox model predictor** implements a low-latency, up-to-date prediction interface for end-users and other data product consumers. There are a range of pre-materialization strategies for model predictions, which we outline in detail in Section 2.5.

To persist models and training data, Velox uses a configurable storage backend. By default, Velox is configured to use Tachyon [94], a fault-tolerant, memory-optimized distributed storage system in BDAS. In our current architecture, both the model manager and predictor are deployed as

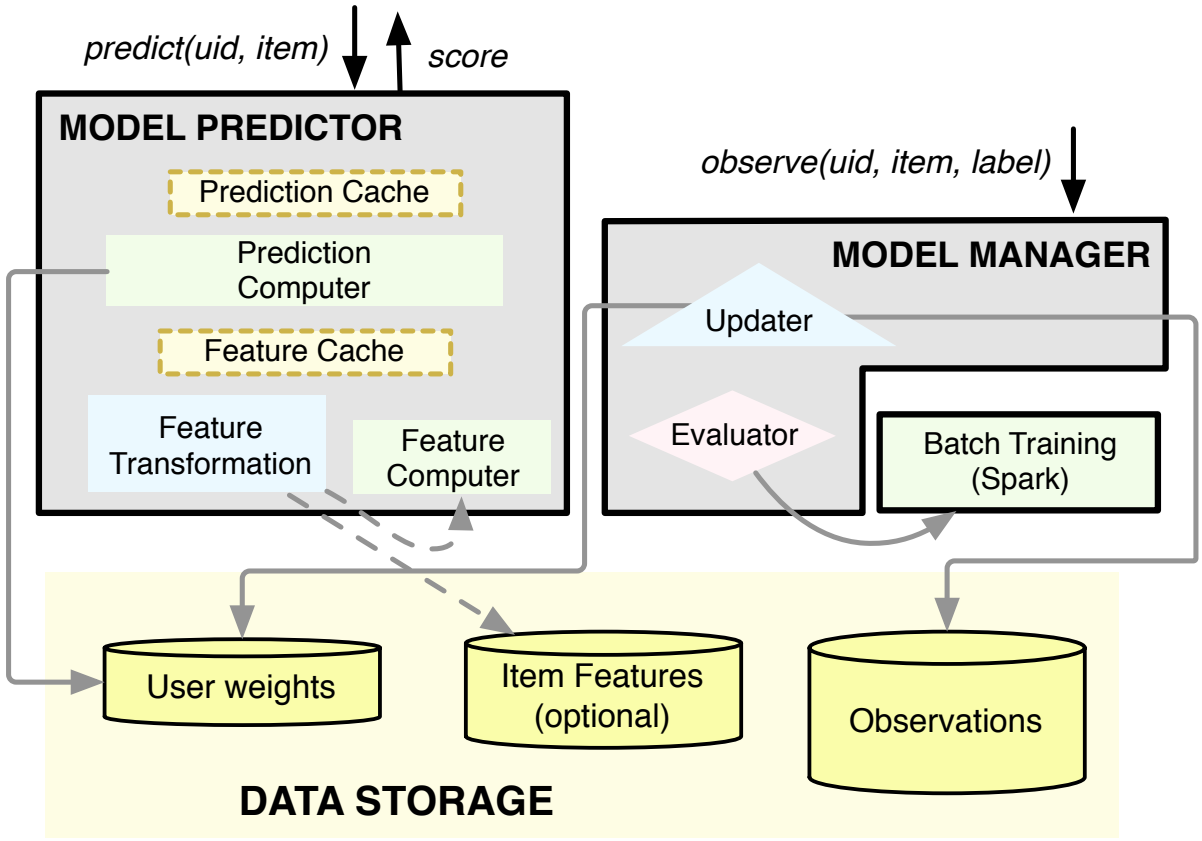


Figure 2.2: Velox System Architecture Two core components, the Model Predictor and Model Manager, orchestrate low latency and large-scale serving of data products computed (originally, in batch) by Spark and stored in a lightweight storage layer (e.g., Tachyon).

a pair of co-located processes that are resident with each Tachyon worker process. When coupled with an intelligent routing policy, this design maximizes data locality.

Current modeling functionality: The current Velox implementation provides native support for a simple yet highly expressive family of personalized linear models that generalizes the matrix factorization model presented in Section 2.2. (We discuss extensions in Section 2.8.) Each model consists of a d-dimensional weight vector $w_u \in \mathbb{R}^d$ for each user u, and a feature function f which maps an input object (e.g., a song) into a d-dimensional feature space (e.g., its latent factor representation). A prediction is computed by evaluating:

$$\text{prediction}(u,x) = w_u^T f(x,\theta) \tag{2.1}$$

The feature parameters θ in conjunction with the feature function f enable this simple model to incorporate a wide range of models including support vector machines (SVMs) [25], deep neural networks [16], and the latent factor models used to build our song recommendation service.

Recommendation service behavior: In our music recommendation data product, the Velox prediction service (Section 2.5) computes predictions for a given user and item (or set of items) by reading the user model w_u and feature parameters θ from Tachyon and evaluating Eq. (2.1). The Velox model maintenance service (Section 2.4) updates the user-models w_u as new observations enter the system and evaluates the resulting model quality. When the model quality is determined to have degraded below some threshold, the maintenance service triggers Spark, the offline training component, to retrain the feature parameters θ. Spark consumes newly observed data from the storage layer, recomputes the user models and feature parameters, and writes the results back to Tachyon.

In the subsequent sections, we provide greater detail about the design of each component, the interfaces they expose, and some of the design decisions we have made in our early prototype. While our focus is on exposing these generalized linear models as data products to end-users, we describe the process of implementing additional models in Section 2.6.

2.4 Model Management

The model management component of Velox is responsible for orchestrating the model life-cycle including: collecting observation and model performance feedback, incrementally updating the user specific weights, continuous evaluation of model performance, and the offline retraining of feature parameters.

Feedback and Data Collection

As users interact with applications backed by Velox, the front-end applications can gather more data (both explicitly and implicitly) about a user's behavior. Velox exposes an *observation* interface to consume this new interaction data and update the user's model accordingly. To insert an observation about a user-item pair into Velox, a front-end application calls `observe` (Listing 2.1), providing the user's ID, the item data (for feature extraction), and the correct label y for that item. In addition to being used to trigger online updates, the observation is written to Tachyon for use by Spark when retraining the model offline.

Offline + Online Learning

Learning in the Velox modeling framework consists of estimating the user specific weights w_u as well as the parameters θ of the feature function. To simultaneously enable low latency learning and personalization while also supporting sophisticated models we divide the learning process into online and offline phases. The offline phase adjusts the feature parameters θ and can run infrequently because the feature parameters capture aggregate properties of the data and therefore evolve slowly. The online phase exploits the independence of the user weights and the linear structure of Eq. (2.1) to permit lightweight conflict free per user updates.

The infrequent offline retraining phase leverages the bulk computation capabilities of large-scale cluster compute frameworks to recompute the feature parameters θ. The training procedure for computing the new feature parameters is defined as an opaque Spark UDF and depends on the current user weights and all the available training data. The result of offline training are new feature parameters as well as potentially updated user weights.

Because the offline phase modifies both the feature parameters and user weights it invalidates both prediction and feature caches. To alleviate some of the performance degradation resulting from invalidating both caches, the batch analytics system also computes all predictions and feature transformations that were cached at the time the batch computation was triggered. These are used to repopulate the caches when switching to the newly trained model. We intend to investigate further improvements in this process, as it is possible that the set of hot items may change as the retrained models redistribute the distribution of popularity among items.

The online learning phase runs continuously and adapts the user specific models w_u to observations as they arrive. While the precise form of the user model updates depends on the choice of error function (a configuration option) we restrict our attention to the widely used squared error (with L_2 regularization) in our initial prototype. As a consequence the updated value of w_u can be obtained analytically using the normal equations:

$$w_u \leftarrow (F(X,\theta)^T F(X,\theta) + \lambda I_n)^{-1} F(X,\theta)^T Y \tag{2.2}$$

where $F(X,\theta) \in \mathbb{R}^{n_u \times d}$ is the matrix formed by evaluating the feature function on all n_u examples of training data obtained for that particular user and λ is a fixed regularization constant. While this step has cubic time complexity in the feature dimension d and linear time complexity in the number of examples n it can be maintained in time quadratic in d using the Sherman-Morrison formula for rank-one updates. Nonetheless using a naive implementation we are able to achieve (Figure 2.3) acceptable latency for a range of feature dimensions d on a real-world collaborative filtering task.

By updating the user weights online and the feature parameters offline, we are provide an *approximation* to continuously retraining the entire model. Moreover, while the feature parameters evolve slowly, they still change. By not continuously updating their value, we potentially introduce inaccuracy into the model. To assess the impact of the hybrid online + offline incremental strategy adopted by Velox, we evaluated the accuracy of Velox on the MovieLens10M dataset[1]. By initializing the latent features with 10 ratings from each user and then using an additional 7 ratings, we were able to achieve 1.6% improvement in prediction accuracy[2] by applying the online strategy. This is comparable to the 2.3% increase in accuracy achieved using full offline retraining.

We first used offline training to initialize the feature parameters θ on half of the data and then evaluated the prediction error of the proposed strategy on the remaining data. By using the Velox's incremental online updates to train on 70% of the remaining data, we were able to achieve a held out prediction error that is only slightly worse than complete retraining.

[1] `http://grouplens.org/datasets/movielens`

[2] Differences in accuracy on the MovieLens dataset are typically measured in small percentages.

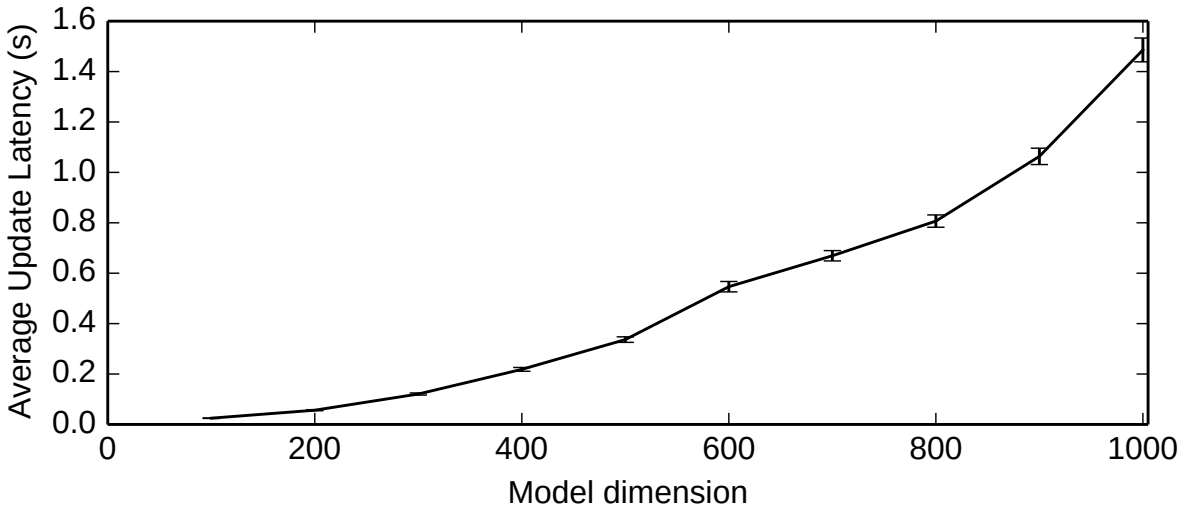


Figure 2.3: Update latency vs model complexity Average time to perform an online update to a user model as a function of the number of factors in the model. The results are averaged over 5000 updates of randomly selected users and items from the MovieLens 10M rating data set. Error bars represent 95% confidence intervals.

Model Evaluation

Monitoring model performance is an essential part of any predictive service. Velox relies on model performance metrics to both aid administrators in managing deployed models and to identify when offline retraining of feature parameters is required. To assess model performance, Velox applies several strategies. First, Velox maintains running per-user aggregates of errors associated with each model. Second, Velox runs an additional cross-validation step during incremental user weight updates to assess generalization performance. Finally, when the `topK` prediction API is used, Velox employs bandit algorithms to collect a pool of validation data that is not influenced by the model. When the error rate on any of these metrics exceeds a pre-configured threshold, the model is retrained offline.

2.5 Online Prediction Service

The Velox prediction service exposes model predictions to other services and applications through a simple interface (Listing 2.1) The `predict` function serves point predictions for the provided user and item, returning the item and its predicted score. The `topK` function evaluates the best item from the provided set for the given `uid`. Support for the latter function is necessary for Velox to implement the bandit methods described later in this section and can be used to support pre-filtering items according to application level policies.

```
def predict(s: ModelSchema, uid: UUID, x: Data)
  : (Data, Double)
def topK(s: ModelSchema, uid: UUID, x: List[Data])
  : List[(Data, Double)]
def observe(uid: UUID, x: Data, y: Double)
```

Listing 2.1: The Prediction and Observation API These methods form the front-end API for a prediction and model management service (i.e., Velox).

Caching: The dominant expense when serving predictions in Velox is evaluating the feature function f. In particular, when f represents a materialized feature function (e.g., matrix factorization models), the distributed lookup of the latent factor in θ is the dominant cost. Alternatively, when f represents a computational feature function (e.g., a deep neural network) the computation becomes the dominant cost. These costs reflect two opportunities for optimization: caching the results of feature function evaluations and efficiently partitioning and replicating the materialized feature tables to limit remote data accesses. Velox performs both caching strategies in the Velox predictor process, corresponding to the *Feature Cache* in Figure 2.2. In addition, we can cache the final prediction for a given (user,item) pair, often useful for repeated calls to `topK` with overlapping itemsets, corresponding to the *Prediction Cache* in Figure 2.2.

To demonstrate the performance improvement that the prediction cache provides, we evaluated the prediction latency of computing `topK` for itemsets of varying size. We compare the prediction latency for the optimal case, when all predictions are cached (i.e., 100% cache hit rate) with the prediction latencies for models of several different sizes. As Figure 2.4 demonstrates, the relationship between itemset size and prediction latency grows linearly, which is to be expected. And as the model size grows (a simple proxy for the expense of computing a prediction, which is a product of both the prediction expense and the feature transformation expense), the benefits of caching grow.

To distribute the load across a Velox cluster and reduce network data transfer, we exploit the fact that every prediction is associated with a specific user and partition W, the user weight vectors table, by `uid`. We then deploy a routing protocol for incoming user requests to ensure that they are served by the node containing that user's model. This partitioning serves a dual purpose. It ensures that lookups into W can always be satisfied locally, and it provides a natural load-balancing scheme for distributing both serving load and the computational cost of online updates. This also has the beneficial side-effect that all writes — online updates to user weight vectors — are local.

When the feature function f is materialized as a pre-computed lookup table, the table is also partitioned across the cluster. Therefore, evaluating f may involve a data transfer from a remote machine containing the required item-features pair. However, while the number of items in the system may be large, item popularity often follows a Zipfian distribution [103]. Consequently, many items are not frequently accessed, and a small subset of items are accessed very often. This suggests that caching the hot items on each machine using a simple cache eviction strategy like LRU, will tend to have a high hit rate. Fortunately, because the materialized features for each item are only updated during the offline batch retraining, cached items are invalidated infrequently.

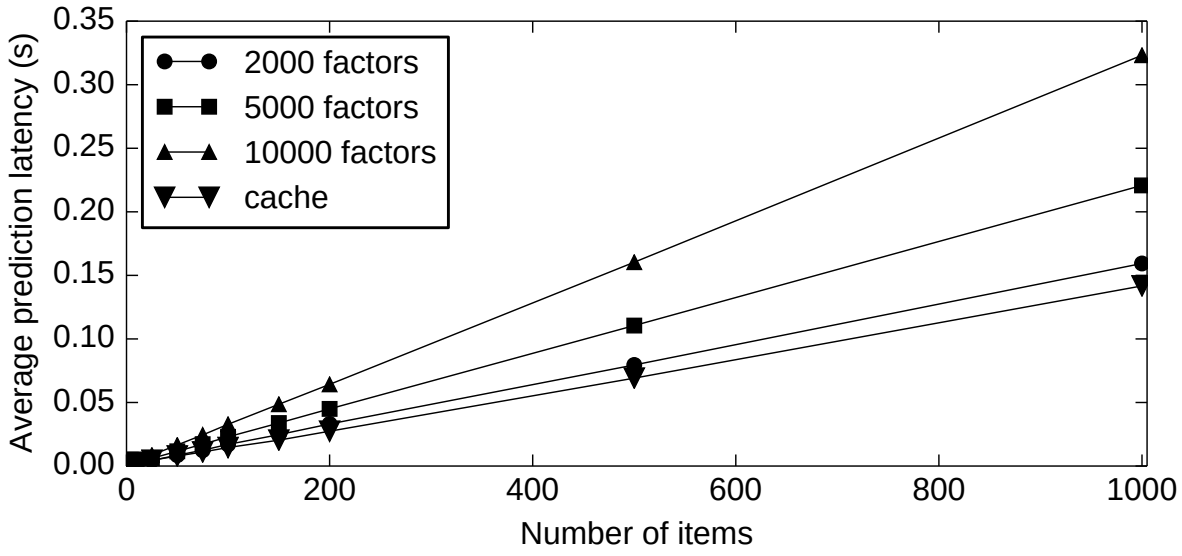


Figure 2.4: Prediction latency vs model complexity Single-node `topK` prediction latency for both cached and non-cached predictions for the Movie Lens 10M rating dataset, varying size of input set and dimension (d, or, factor). Results are averaged over 10,000 trials.

When the feature transformation f is computational, caching the results of computing the basis functions can provide similar benefits. For popular items, caching feature function evaluation reduces prediction latency by eliminating the time to compute the potentially expensive feature function, and reduces computational load on the serving machine, freeing resources for serving queries.

Bootstrapping: One of the key challenges in predictive services is how to model new users. In Velox, we currently adopt a simple heuristic in which new users are assigned a recent estimate of the average of the existing user weight vectors:

$$\bar{w}^T f(x,\theta) = \frac{1}{|\text{users}|} \sum_{u \in \text{users}} w_u^T f(x,\theta)$$

This also corresponds predicting the average score for all users.

Bandits and Multiple Models: Model serving influences decisions that may, in turn, be used to train future models. This can lead to feedback loops. For example, a music recommendation service that only plays the current Top40 songs will never receive feedback from users indicating that others songs are preferable. To escape these feedback loops we rely on a form of the *contextual bandits algorithm* [95], a family of techniques developed to avoid these feedback loops. These techniques assign each item an *uncertainty* score in addition to its predicted store. The algorithm improves models greedily by reducing uncertainty about predictions. To reduce the total uncertainty in the model, the algorithm recommends the item with the best *potential* prediction score (i.e., the item

```
class VeloxModel:
  val name: String  // <- user provided
  val state: Vector // <- feature parameters
  val version: Int  // <- system provided
  def VeloxModel(state: Opt[Vector])
  // Feature Transformation Function
  def features(x: Data, materialized: Boolean)
      : Vector
  // Learning
  def retrain(f: (Data) => Vector,
              w: Table[String, Vector],
              newData: Seq[Data])
      : ((Data) => Vector, Table[String,Vector])
  // Quality Evaluation
  def loss(y: Label, yPredict: Label,
           x: Data, uid: UUID): Double
```

Listing 2.2: The `VeloxModel` Interface Developers can add new models and feature transformation functions to Velox by implementing this interface. Implementations specify how to featurize items, perform offline training, and how to evaluate model quality.

with max sum of score and uncertainty) as opposed to recommending the item with the absolute best prediction score. When Velox observes the correct score for that recommendation and an online update is triggered, that update will reduce the uncertainty in the user weight vector more so than an observation about an item with less uncertainty. The bandits algorithms exploit the `topK` interface to select the item that has the highest potential predicted rating. For example, if Velox is unsure to what extent a user is a *DeadHead* it will occasionally select songs such as "New Potato Caboose" to evaluate this hypothesis even if those songs do not have the highest prediction score.

2.6 Adding New Models

It is possible to express a wide range of models and machine learning techniques within Velox by defining new feature functions. To add a new model to Velox, a data scientist implements and uploads a new `VeloxModel` instance (Listing 2.2).

Shared state: Each `VeloxModel` may be instantiated with a vector, used to provide any global, immutable state (i.e., θ from Section 2.2) needed during the featurization process. For example, this state may be the parameters for a set of SVMs learned offline and used as the feature transformation function.

Feature transformations: The `VeloxModel` function `features` implements the feature transformation function. The `features` function may implement a computation on some input data, as is the case when the feature transformation is the computation of a set of basis functions. Alternatively,

the `features` function may implement a lookup of the latent features in a table, similar to the table W used to store the user models. The implementor indicates which of these two strategies is used by explicitly specifying whether the features are materialized or are computed. Continuing with the ensemble of SVMs example, `features` would evaluate a set of SVMs with different parameters (stored in the member `state`) passed in on instance construction. We are investigating automatic ways of analyzing data dependencies through techniques like UDF byte-code inspection.

Quality evaluation and model retraining: The user provides two functions, `retrain` and `loss`, that allow Velox to automatically detect when models are stale and retrain them. The `loss` is evaluated every time new data is observed (i.e., every time a user model is updated) and if the loss starts to increase faster than a threshold value, the model is detected as stale. Once a model has been detected as stale, Velox retrains the model offline using the cluster compute framework. `retrain` informs the cluster compute framework how to train this `VeloxModel`, as well as where to find and how to interpret the observation data needed for training. When offline training completes, Velox automatically instantiates a new `VeloxModel` and new W — incrementing the version — and transparently upgrades incoming prediction requests to be served with the newly trained user-models and `VeloxModel`.

2.7 Related Work

Velox draws upon a range of related work from the intersection of database systems and complex analytics. We can broadly characterize this work as belonging to three major areas:

Predictive database systems: The past several years have seen several calls towards tight coupling of databases and predictive analytics. The Longview system [6] integrates predictive models as first-class entities in PostgreSQL and introduces a declarative language for model querying. Similarly, Bismarck [52] allows users to express complex analytics via common user-defined aggregates. A large body of work studies probabilistic databases, which provide first-class support for complex statistical models but, in turn, focus on modeling uncertainty in data [141, 151]. Commercially, the PMML markup language and implementations like Oryx[3] provide support for a subset of the data product concerns addressed by Velox. Our focus in Velox is to provide predictive analytics as required for modern data products in a large scale distributed setting. In doing so, we focus on user-specific personalization, online model training, and the challenges of feedback loops in modern predictive services.

In contrast with prior work on model management in database systems, which was largely concerned with managing deterministic metadata (such as schema) [18], Velox focuses on the use and management of statistical models from the domain of machine learning.

View materialization: The problem of maintaining models at scale can be viewed as an instance of complex materialized view maintenance. In particular, MauveDB exploits this connection in a single-node context, providing a range of materialization strategies for a set of *model-based*

[3] `https://github.com/cloudera/oryx`

views including regression and Kalman filtering [47] but does not address latent feature models or personalized modeling. Similarly, Columbus [160] demonstrates the power of caching and model reuse in in-situ feature learning. We see considerable opportunity in further exploiting the literature on materialized view maintenance [34] in the model serving setting.

Distributed machine learning: There are a bevy of systems suitable for the performing batch-oriented complex analytics tasks [13], and a considerable amount of work implementing specific tasks. For example, Li et al. [93] explored a strategy for implementing a variant of SGD within the Spark cluster compute framework that could be used by Velox to improve offline training performance. Our focus is on leveraging these existing algorithms to provide better *online* predictive tasks. However, we aggressively exploit these systems' batch processing ability and large install bases in our solution.

2.8 Conclusions and Future Work

In this paper, we introduced Velox, a system for performing machine learning model serving and model maintenance at scale. Velox leverages knowledge of prediction semantics to efficiently cache and replicate models across a cluster. Velox updates models to react to changing user patterns, automatically monitoring model quality and delegating offline retraining to existing cluster compute frameworks. In doing so, Velox fills a void in current production analytics pipelines, simplifying front-end applications by allowing them to consume predictions from automatically maintained complex statistical models.

We have completed an initial Velox prototype that exposes a RESTful client interface and integrates with existing BDAS components, relying on Spark and Tachyon for offline training and distributed data storage. By running tests against the MovieLens10M dataset we demonstrated that our early prototype performs well on basic serving and model update tasks. In addition we have evaluated our online incremental update strategy and demonstrated that it closely recovers the prediction accuracy of offline batch retraining.

Chapter 3

Clipper: A Low-Latency Online Prediction Serving System

3.1 Introduction

The past few years have seen an explosion of applications driven by machine learning, including recommendation systems [67, 150], voice assistants [136, 40, 62], and ad-targeting [63, 3]. These applications depend on two stages of machine learning: *training* and *inference*. Training is the process of building a model from data (e.g., movie ratings). Inference is the process of using the model to make a prediction given an input (e.g., predict a user's rating for a movie). While training is often computationally expensive, requiring multiple passes over potentially large datasets, inference is often assumed to be inexpensive. Conversely, while it is acceptable for training to take hours to days to complete, inference must run in real-time, often on orders of magnitude more queries than during training, and is typically part of user-facing applications.

For example, consider an online news organization that wants to deploy a content recommendation service to personalize the presentation of content. Ideally, the service should be able to recommend articles at interactive latencies (<100ms) [158], scale to large and growing user populations, sustain the throughput demands of flash crowds driven by breaking news, and provide accurate predictions as the news cycle and reader interests evolve.

The challenges of developing these services differ between the training and inference stages. On the training side, developers must choose from a bewildering array of machine learning frameworks with diverse APIs, models, algorithms, and hardware requirements. Furthermore, they may often need to migrate between models and frameworks as new, more accurate techniques are developed. Once trained, models must be *deployed* to a prediction serving system to provide low-latency predictions at scale.

Unlike model development, which is supported by sophisticated infrastructure, theory, and systems, model deployment and prediction-serving have received relatively little attention. Developers must cobble together the necessary pieces from various systems components, and must integrate and support inference across multiple, evolving frameworks, all while coping with ever-

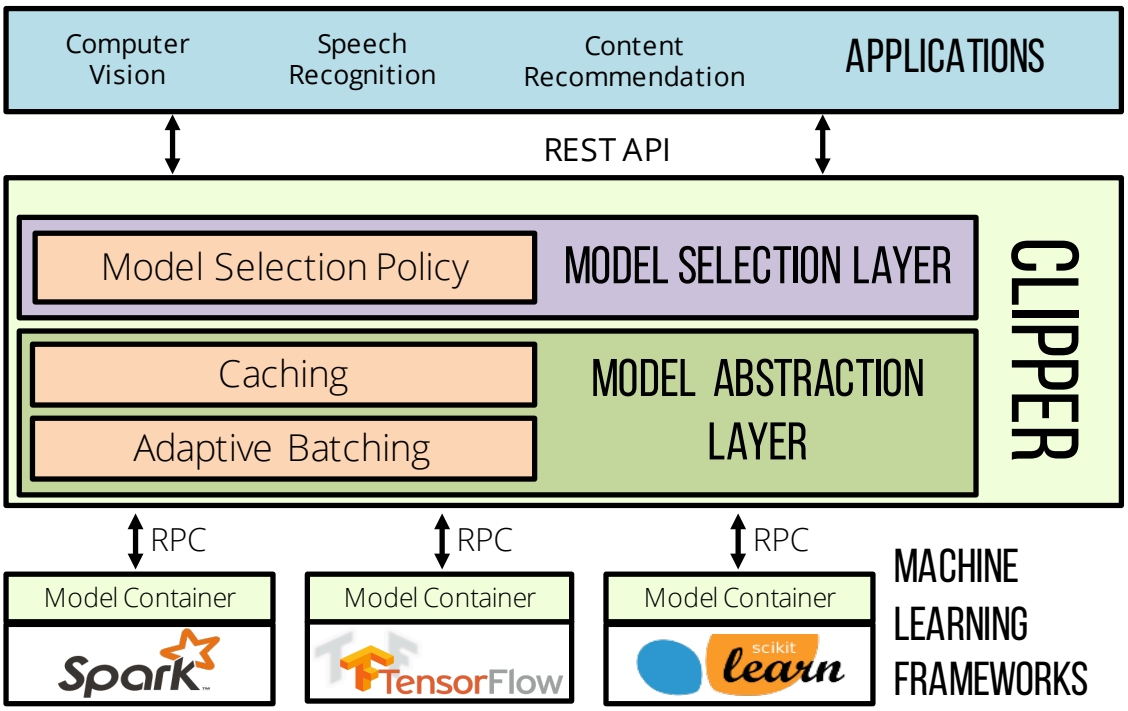


Figure 3.1: The Clipper Architecture.

increasing demands for scalability and responsiveness. As a result, the deployment, optimization, and maintenance of machine learning services is difficult and error-prone.

To address these challenges, we propose Clipper, a *layered architecture* system (Figure 3.1) that reduces the complexity of implementing a prediction serving stack and achieves three crucial properties of a prediction serving system: *low latencies*, *high throughputs*, and *improved accuracy*. Clipper is divided into two layers: (1) the model abstraction layer, and (2) the model selection layer. The first layer exposes a common API that abstracts away the heterogeneity of existing ML frameworks and models. Consequently, models can be modified or swapped transparently to the application. The model selection layer sits above the model abstraction layer and dynamically selects and combines predictions across competing models to provide more accurate and robust predictions.

To achieve low latency, high throughput predictions, Clipper implements a range of optimizations. In the model abstraction layer, Clipper caches predictions on a per-model basis and implements adaptive batching to maximize throughput given a query latency target. In the model selection layer, Clipper implements techniques to improve prediction accuracy and latency. To improve accuracy, Clipper exploits bandit and ensemble methods to robustly select and combine predictions from multiple models and estimate prediction uncertainty. In addition, Clipper is able to adapt the model selection independently for each user or session. To improve latency, the model selection layer adopts a straggler mitigation technique to render predictions without waiting for

slow models. Because of this layered design, neither the end-user applications nor the underlying machine learning frameworks need to be modified to take advantage of these optimizations.

We implemented Clipper in Rust and added support for several of the most widely used machine learning frameworks: Apache Spark MLLib [104], Scikit-Learn [128], Caffe [74], TensorFlow [1], and HTK [157]. While these frameworks span multiple application domains, programming languages, and system requirements, each was added using fewer than 25 lines of code.

We evaluate Clipper using four common machine learning benchmark datasets and demonstrate that Clipper is able to render low and bounded latency predictions (<20ms), scale to many deployed models even across machines, quickly select and adapt the best combination of models, and dynamically trade-off accuracy and latency under heavy query load. We compare Clipper to the Google TensorFlow Serving system [147], an industrial grade prediction serving system tightly integrated with the TensorFlow training framework. We demonstrate that Clipper's modular design and broad functionality impose minimal performance cost, achieving comparable prediction throughput and latency to TensorFlow Serving while supporting substantially more functionality. In summary, our key contributions are:

- A layered architecture that abstracts away the complexity associated with serving predictions in existing machine learning frameworks (Section 3.3).
- A set of novel techniques to reduce and bound latency while maximizing throughput that generalize across machine learning frameworks (Section 3.4).
- A model selection layer that enables online model selection and composition to provide robust and accurate predictions for interactive applications (Section 3.5).

3.2 Applications and Challenges

The machine learning life-cycle (Figure 3.2) can be divided into two distinct phases: *training* and *inference*. Training is the process of estimating a model from data. Training is often computationally expensive requiring multiple passes over large datasets and can take hours or even days to complete [29, 106, 68]. Much of the innovation in systems for machine learning has focused on model training with the development of systems like Apache Spark [159], the Parameter Server [96], PowerGraph [61], and Adam [33].

A wide range of machine learning frameworks have been developed to address the challenges of training. Many specialize in particular models such as TensorFlow [1] for deep learning or Vowpal Wabbit [86] for large linear models. Others are specialized for specific application domains such as Caffe [74] for computer vision or HTK [157] for speech recognition. Typically, these frameworks leverage advances in parallel and distributed systems to scale the training process.

Inference is the process of evaluating a model to render predictions. In contrast to training, inference does not involve complex iterative algorithms and is therefore generally assumed to be easy. As a consequence, there is little research studying the process of inference and most machine learning frameworks provide only basic support for offline batch inference – often with the singular goal of evaluating the model training algorithm. However, scalable, accurate, and reliable inference

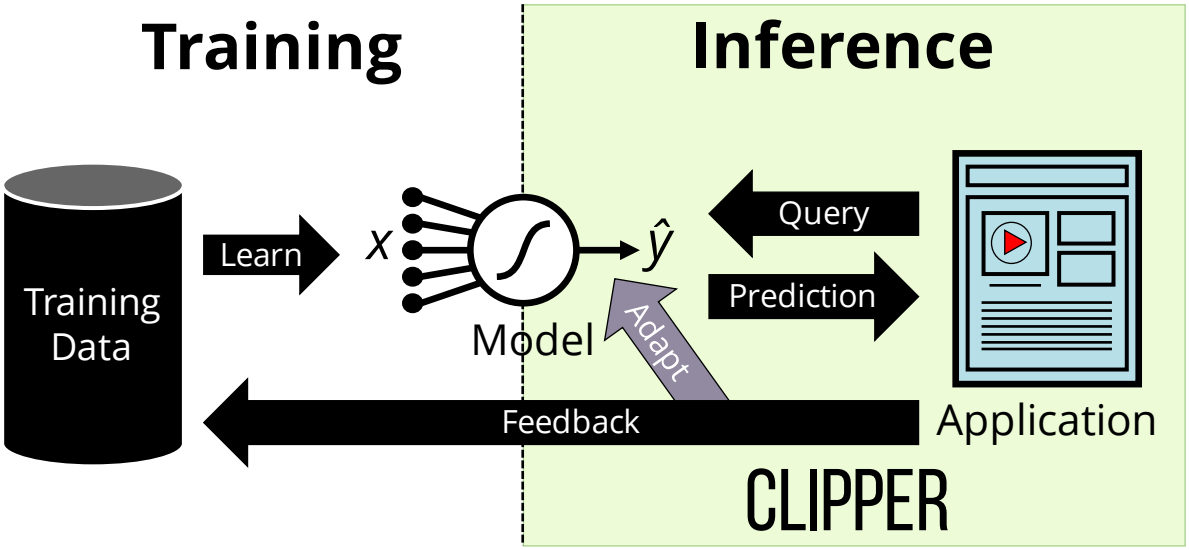


Figure 3.2: Machine Learning Lifecycle.

presents fundamental system challenges that will likely dominate the challenges of training as machine learning adoption increases. In this paper we focus on the less studied but increasingly important challenges of *inference*.

Application Workloads

To illustrate the challenges of inference and provide a benchmark on which to evaluate Clipper, we describe two canonical real-world applications of machine learning: *object recognition* and *speech recognition*.

Object Recognition

Advances in deep learning have spurred rapid progress in computer vision, especially in object recognition problems – the task of identifying and labeling the objects in a picture. Object recognition models form an important building block in many computer vision applications ranging from image search to self-driving cars.

As users interact with these applications, they provide feedback about the accuracy of the predictions, either by explicitly labeling images (e.g., tagging a user in an image) or implicitly by indicating whether the provided prediction was correct (e.g., clicking on a suggested image in a search). Incorporating this feedback quickly can be essential to eliminating failing models and providing a more personalized experience for users.

Benchmark Applications: We use the well studied MNIST [89], CIFAR-10 [84], and ImageNet [124] datasets to evaluate increasingly difficult object recognition tasks with correspondingly

larger inputs. For each dataset, the prediction task requires identifying the correct label for an image based on its pixel values. MNIST is a common baseline dataset used to demonstrate the potential of a new algorithm or technique, and both deep learning and more classical machine learning models perform well on MNIST. On the other hand, for CIFAR-10 and Imagenet, deep learning significantly outperforms other methods. By using three different datasets, we evaluate Clipper's performance when serving models that have a wide variety of computational requirements and accuracies.

Automatic Speech Recognition

Another successful application of machine learning is automatic speech recognition. A speech recognition model is a function from a spoken audio signal to the corresponding sequence of words. Speech recognition models can be relatively large [28] and are often composed of many complex sub-models trained using specialized speech recognition frameworks (e.g., HTK [157]). Speech recognition models are also often personalized to individual users to accommodate variations in dialect and accent.

In most applications, inference is done online as the user speaks. Providing real-time predictions is essential to user experience [4] and enables new applications like real-time translation [137]. However, inference in speech models can be costly [28] requiring the evaluation of large tensor products in convolutional neural networks.

As users interact with speech services, they provide implicit signal about the quality of the speech predictions which can be used to identify the dialect. Incorporating this feedback quickly improves user experience by allowing us to choose models specialized for a user's dialect.

Benchmark Application: To evaluate the benefit of personalization and online model-selection on a dataset with real user data, we built a speech recognition service with the widely used TIMIT speech corpus [59] and the HTK [157] machine learning framework. This dataset consists of voice recordings for 630 speakers in eight dialects of English. We randomly drew users from the test corpus and simulated their interaction with our speech recognition service using their pre-recorded speech data.

Challenges

Motivated by the above applications, we outline the key challenges of prediction serving and describe how Clipper addresses these challenges.

Complexity of Deploying Machine Learning

There is a large and growing number of machine learning frameworks [38, 74, 1, 32, 17]. Each framework has strengths and weaknesses and many are optimized for specific models or application domains (e.g., computer vision). Thus, there is no dominant framework and often multiple frameworks may be used for a single application (e.g., speech recognition and computer vision in automatic captioning). Furthermore, machine learning is an iterative process and the best framework may change as an application evolves over time (e.g., as a training dataset grows to require

distributed model training). Although common model exchange formats have been proposed [119, 118], they have never achieved widespread adoption because of the rapid and fundamental changes in state-of-the-art techniques and additional source of errors from parallel implementations for training and serving. Finally, machine learning frameworks are often developed by and for machine learning experts and are therefore heavily optimized towards model development rather than deployment. As a consequence of these design decisions, application developers are forced to accept reduced accuracy by forgoing the use of a model well-suited to the task or to incur the substantially increased complexity of integrating and supporting multiple machine learning frameworks.

Solution: Clipper introduces a model abstraction layer and common prediction interface that isolates applications from variability in machine learning frameworks (Section 3.4) and simplifies the process of deploying a new model or framework to a running application.

Prediction Latency and Throughput

The *prediction latency* is the time it takes to render a prediction given a query. Because prediction serving is often on the critical path, predictions must both be fast and have bounded tail latencies to meet service level objectives [158]. While simple linear models are fast, more sophisticated and often more accurate models such as support vector machines, random forests, and deep neural networks are much more computationally intensive and can have substantial latencies (50-100ms) [32] (see Figure 3.11 for details). In many cases accuracy can be improved by combining models but at the expense of stragglers and increased tail latencies. Finally, most machine learning frameworks are optimized for offline batch processing and not single-input prediction latency. Moreover, the low and bounded latency demands of interactive applications are often at odds with the design goals of machine learning frameworks.

The computational cost of sophisticated models can substantially impact prediction throughput. For example, a relatively fast neural network which is able to render 100 predictions per second is still orders of magnitude slower than a modern web-server. While batching prediction requests can substantially improve throughput by exploiting optimized BLAS libraries, SIMD instructions, and GPU acceleration it can also adversely affect prediction latency. Finally, under heavy query load it is often preferable to marginally degrade accuracy rather than substantially increase latency or lose availability [58, 3].

Solution: Clipper automatically and adaptively batches prediction requests to maximize the use of batch-oriented system optimizations in machine learning frameworks while ensuring that prediction latency objectives are still met (Section 3.4). In addition, Clipper employs straggler mitigation techniques to reduce and bound tail latency, enabling model developers to experiment with complex models without affecting serving latency (Section 3.5).

Model Selection

Model development is an iterative process producing many models reflecting different feature representations, modeling assumptions, and machine learning frameworks. Typically developers must decide which of these models to deploy based on offline evaluation using stale datasets or

Dataset	Type	Size	Features	Labels
MNIST [89]	Image	70K	28x28	10
CIFAR [84]	Image	60k	32x32x3	10
ImageNet [124]	Image	1.26M	299x299x3	1000
Speech [59]	Sound	6300	5 sec.	39

Table 3.1: Datasets. The collection of real-world benchmark datasets used in the experiments.

engage in costly online A/B testing. When predictions can influence future queries (e.g., content recommendation), offline evaluation techniques can be heavily biased by previous modeling results. Alternatively, A/B testing techniques[2] have been shown to be statistically inefficient — requiring data to grow *exponentially* in the number of candidate models. The resulting choice of model is typically static and therefore susceptible to changes in model performance due to factors such as feature corruption or concept drift[129]. In some cases the best model may differ depending on the context (e.g., user or region) in which the query originated. Finally, predictions from more than one model can often be combined in ensembles to boost prediction accuracy and provide more robust predictions with confidence bounds.

Solution: Clipper leverages adaptive online model selection and ensembling techniques to incorporate feedback and automatically select and combine predictions from models that can span multiple machine learning frameworks.

Experimental Setup

Because we include microbenchmarks of many of Clipper's features as we introduce them, we present the experimental setup now. For each of the three object recognition benchmarks, the prediction task is predicting the correct label given the raw pixels of an unlabeled image as input. We used a variety of models on each of the object recognition benchmarks. For the speech recognition benchmark, the prediction task is predicting the phonetic transcription of the raw audio signal. For this benchmark, we used the HTK Speech Recognition Toolkit [157] to learn Hidden Markov Models whose outputs are sequences of phonemes representing the transcription of the sound. Details about each dataset are presented in Table 3.1.

Unless otherwise noted, all experiments were conducted on a single server. All machines used in the experiments contain 2 Intel Haswell-EP CPUs and 256 GB of RAM running Ubuntu 14.04 on Linux 4.2.0. TensorFlow models were executed on a Nvidia Tesla K20c GPUs with 5 GB of GPU memory and 2496 cores. In the scaling experiment presented in Figure 3.6, the servers in the cluster were connected with both a 10Gbps and 1Gbps network. For each network, all the servers were located on the same switch. Both network configurations were investigated.

3.3 System Architecture

Clipper is divided into *model selection* and *model abstraction* layers (see Figure 3.1). The model abstraction layer is responsible for providing a common prediction interface, ensuring resource isolation, and optimizing the query workload for batch oriented machine learning frameworks. The model selection layer is responsible for dispatching queries to one or more models and combining their predictions based on feedback to improve accuracy, estimate uncertainty, and provide robust predictions.

Before presenting the detailed Clipper system design we first describe the path of a prediction request through the system. Applications issue prediction requests to Clipper through application facing REST or RPC APIs. Prediction requests are first processed by the model selection layer. Based on properties of the prediction request and recent feedback, the model selection layer dispatches the prediction request to one or more of the models through the model abstraction layer.

The model abstraction layer first checks the prediction cache for the query before assigning the query to an adaptive batching queue associated with the desired model. The adaptive batching queue constructs batches of queries that are tuned for the machine learning framework and model. A cross language RPC is used to send the batch of queries to a model container hosting the model in its native machine learning framework. To simplify deployment, we host each model container in a separate Docker container. After evaluating the model on the batch of queries, the predictions are sent back to the model abstraction layer which populates the prediction cache and returns the results to the model selection layer. The model selection layer then combines one or more of the predictions to render a final prediction and confidence estimate. The prediction and confidence estimate are then returned to the end-user application.

Any feedback the application collects about the quality of the predictions is sent back to the model selection layer through the same application-facing REST/RPC interface. The model selection layer joins this feedback with the corresponding predictions to improve how it selects and combines future predictions.

We now present the model abstraction layer and the model selection layer in greater detail.

3.4 Model Abstraction Layer

The **Model Abstraction Layer** (Figure 3.1) provides a common interface across machine learning frameworks. It is composed of a prediction cache, an adaptive query-batching component, and a set of model containers connected to Clipper via a lightweight RPC system. This modular architecture enables caching and batching mechanisms to be shared across frameworks while also scaling to many concurrent models and simplifying the addition of new frameworks.

Overview

At the top of the model abstraction layer is the prediction cache (Section 3.4). The prediction caches provides a partial pre-materialization mechanism for frequent queries and accelerates the adaptive

model selection techniques described in Section 3.5 by enabling efficient joins between recent predictions and feedback.

The batching component (Section 3.4) sits below the prediction cache and aggregates point queries into mini-batches that are dynamically resized for each model container to maximize throughput. Once a mini-batch is constructed for a given model it is dispatched via the RPC system to the container for evaluation.

Models deployed in Clipper are each encapsulated within their own lightweight container (Section 3.4), communicating with Clipper through an RPC mechanism that provides a uniform interface to Clipper and simplifies the deployment of new models. The lightweight RPC system minimizes the overhead of the container-based architecture and simplifies cross-language integration.

In the following sections we describe each of these components in greater detail and discuss some of the key algorithmic innovations associated with each.

Caching

For many applications (e.g., content recommendation), predictions concerning popular items are requested frequently. By maintaining a prediction cache, Clipper can serve these frequent queries without evaluating the model. This substantially reduces latency and system load by eliminating the additional cost of model evaluation.

In addition, caching in Clipper serves an important role in model selection (Section 3.5). To select models intelligently Clipper needs to join the original predictions with any feedback it receives. Since feedback is likely to return soon after predictions are rendered [102], even infrequent or unique queries can benefit from caching.

For example, even with a small ensemble of four models (a random forest, logistic regression model, and linear SVM trained in Scikit-Learn and a linear SVM trained in Spark), prediction caching increased feedback processing throughput in Clipper by 1.6x from roughly 6K to 11K observations per second.

The prediction cache acts as a function cache for the generic prediction function:

```
Predict(m: ModelId, x: X) -> y: Y
```

that takes a model id *m* along with the query *x* and computes the corresponding model prediction *y*. The cache exposes a simple non-blocking *request* and *fetch* API. When a prediction is needed, the *request* function is invoked which notifies the cache to compute the prediction if it is not already present and returns a boolean indicating whether the entry is in the cache. The *fetch* function checks the cache and returns the query result if present.

Clipper employs an LRU eviction policy for the prediction cache, using the standard CLOCK [39] cache eviction algorithm. With an adequately sized cache, frequent queries will not be evicted and the cache serves as a partial pre-materialization mechanism for hot items. However, because adaptive model selection occurs *above the cache* in Clipper, changes in predictions due to model selection do not invalidate cache entries.

Batching

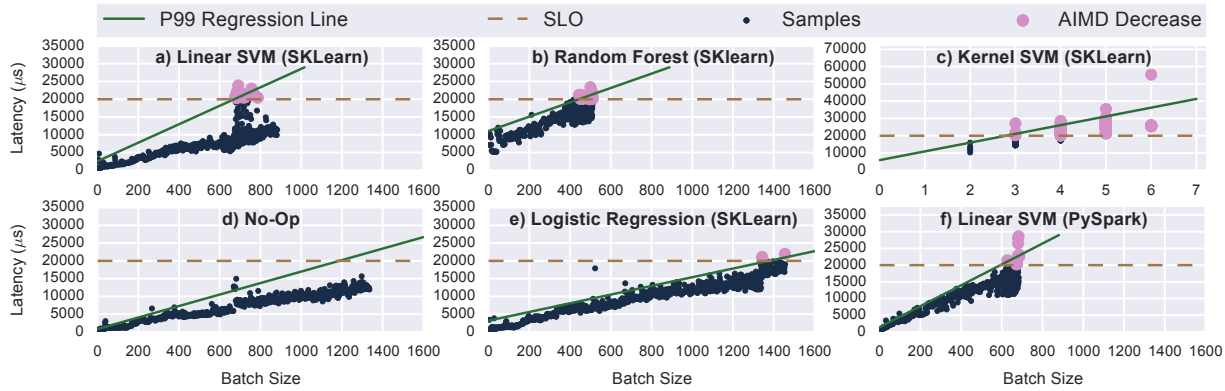


Figure 3.3: Model Container Latency Profiles. We measured the batching latency profile of several models trained on the MNIST benchmark dataset. The models were trained using Scikit-Learn (SKLearn) or Spark and were chosen to represent several of the most widely used types of models. The No-Op Container measures the system overhead of the model containers and RPC system.

The Clipper batching component transforms the concurrent stream of prediction queries received by Clipper into batches that more closely match the workload assumptions made by machine learning frameworks while simultaneously amortizing RPC and system overheads. Batching improves throughput and utilization of often costly physical resources such as GPUs, but it does so at the expense of increased latency by requiring all queries in the batch to complete before returning a single prediction.

We exploit an explicitly stated latency service level objective (SLO) to *increase latency* in exchange for substantially improved throughput. By allowing users to specify a latency objective, Clipper is able to tune batched query evaluation to maximize throughput while still meeting the latency requirements of interactive applications. For example, requesting predictions in sufficiently large batches can improve throughput by up to 26x (the Scikit-Learn SVM in Figure 3.4) while meeting a 20ms latency SLO.

Batching increases throughput via two mechanisms. First, batching amortizes the cost of RPC calls and internal framework overheads such as copying inputs to GPU memory. Second, batching enables machine learning frameworks to exploit existing data-parallel optimizations by performing batch inference on many inputs simultaneously (e.g., by using the GPU or BLAS acceleration).

As the model selection layer dispatches queries for model evaluation, they are placed on queues associated with model containers. Each model container has its own adaptive batching queue tuned to the latency profile of that container and a corresponding thread to process predictions. Predictions are processed in batches by removing as many queries as possible from a queue up to the maximum batch size *for that model container* and sending the queries as a single batch prediction RPC to the container for evaluation. Clipper imposes a *maximum* batch size to ensure that latency objectives are met and avoid excessively delaying the first queries in the batch.

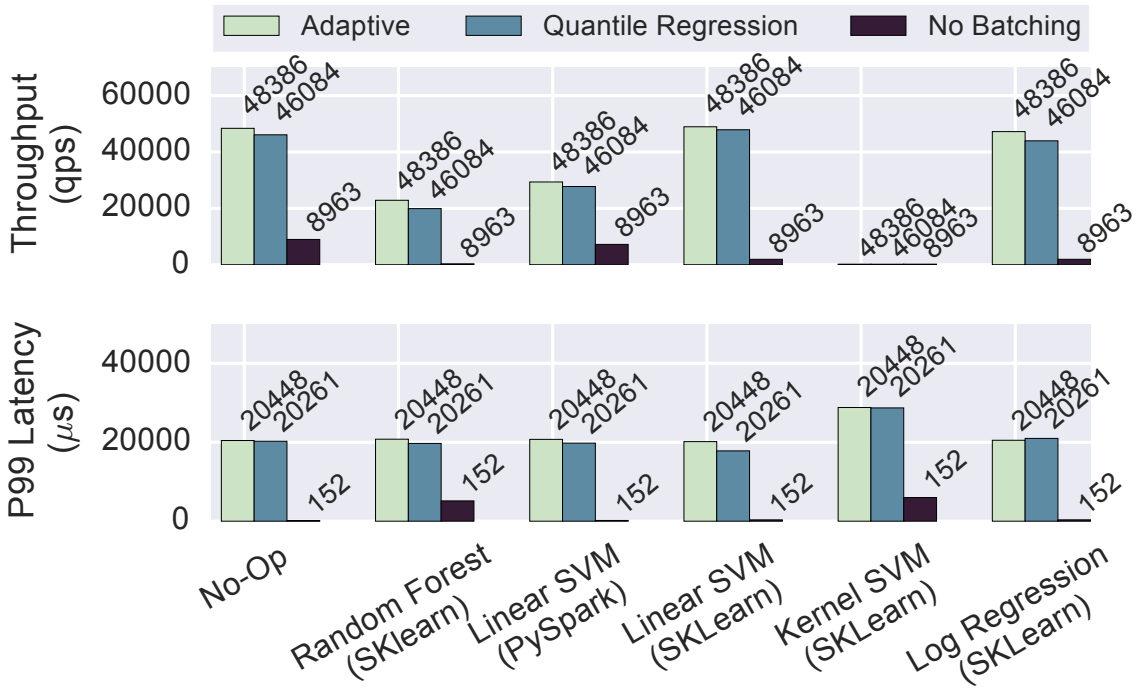


Figure 3.4: Comparison of Dynamic Batching Strategies.

Frameworks that leverage GPU acceleration such as TensorFlow often enforce static batch sizes to maintain a consistent data layout across evaluations of the model. These frameworks typically encode the batch size directly into the model definition in order to fully exploit GPU parallelism. When rendering fewer predictions than the batch size, the input must be padded to reach the defined size, reducing model throughput without any improvement in prediction latency. Careful tuning of the batch size should be done to maximize inference performance, but this tuning must be done offline and is fixed by the time a model is deployed.

However, most machine learning frameworks can efficiently process variable-sized batches at serving time. Yet differences between the framework implementation and choice of model and inference algorithm can lead to orders of magnitude variation in model throughput and latency. As a result, the latency profile – the expected time to evaluate a batch of a given size – varies substantially between model containers. For example, in Figure 3.3 we see that the maximum batch size that can be executed within a 20ms latency SLO differs by 241x between the linear SVM which does a very simple vector-vector multiply to perform inference and the kernel SVM which must perform a sequence of expensive nearest-neighbor calculations to evaluate the kernel. As a consequence, the linear SVM can achieve throughput of nearly 30,000 qps while the kernel SVM is limited to 200 qps under this SLO. Instead of requiring application developers to manually tune the batch size for each new model, Clipper employs a simple adaptive batching scheme to dynamically find and adapt the maximum batch size.

Dynamic Batch Size

We define the optimal batch size as the batch size that maximizes throughput subject to the constraint that the batch evaluation latency is under the target SLO. To automatically find the optimal maximum batch size for each model container we employ an additive-increase-multiplicative-decrease (AIMD) scheme. Under this scheme, we additively increase the batch size by a fixed amount until the latency to process a batch exceeds the latency objective. At this point, we perform a small multiplicative backoff, reducing the batch size by 10%. Because the optimal batch size does not fluctuate substantially, we use a much smaller backoff constant than other Additive-Increase, Multiplicative-Decrease schemes [35].

Early performance measurements (Figure 3.3) suggested a stable linear relationship between batch size and latency across several of the modeling frameworks. As a result, we also explored the use of quantile regression to estimate the 99th-percentile (P99) latency as a function of batch size and set the maximum batch size accordingly. We compared the two approaches on a range of commonly used Spark and Scikit-Learn models in Figure 3.4. Both strategies provide significant performance improvements over the baseline strategy of no batching, achieving up to a 26x throughput increase in the case of the Scikit-Learn linear SVM, demonstrating the performance gains that batching provides. While the two batching strategies perform nearly identically, the AIMD scheme is significantly simpler and easier to tune. Furthermore, the ongoing adaptivity of the AIMD strategy makes it robust to changes in throughput capacity of a model (e.g., during a garbage collection pause in Spark). As a result, Clipper employs the AIMD scheme as the default.

Delayed Batching

Under moderate or bursty loads, the batching queue may contain less queries than the maximum batch size when the next batch is ready to be dispatched. For some models, briefly delaying the dispatch to allow more queries to arrive can significantly improve throughput under bursty loads. Similar to the motivation for Nagle's algorithm [110], the gain in efficiency is a result of the ratio of the fixed cost for sending a batch to the variable cost of increasing the size of a batch.

In Figure 3.5, we compare the gain in efficiency (measured as increased throughput) from delayed batching for two models. Delayed batching provides no increase in throughput for the Spark SVM because Spark is already relatively efficient at processing small batch sizes and can keep up with the moderate serving workload using batches much smaller than the optimal batch size. In contrast, the Scikit-Learn SVM has a high fixed cost for processing a batch but employs BLAS libraries to do efficient parallel inference on many inputs at once. As a consequence, a 2ms batch delay provides a 3.3x improvement in throughput and allows the Scikit-Learn model container to keep up with the throughput demand while remaining well below the 10-20ms latency objectives needed for interactive applications.

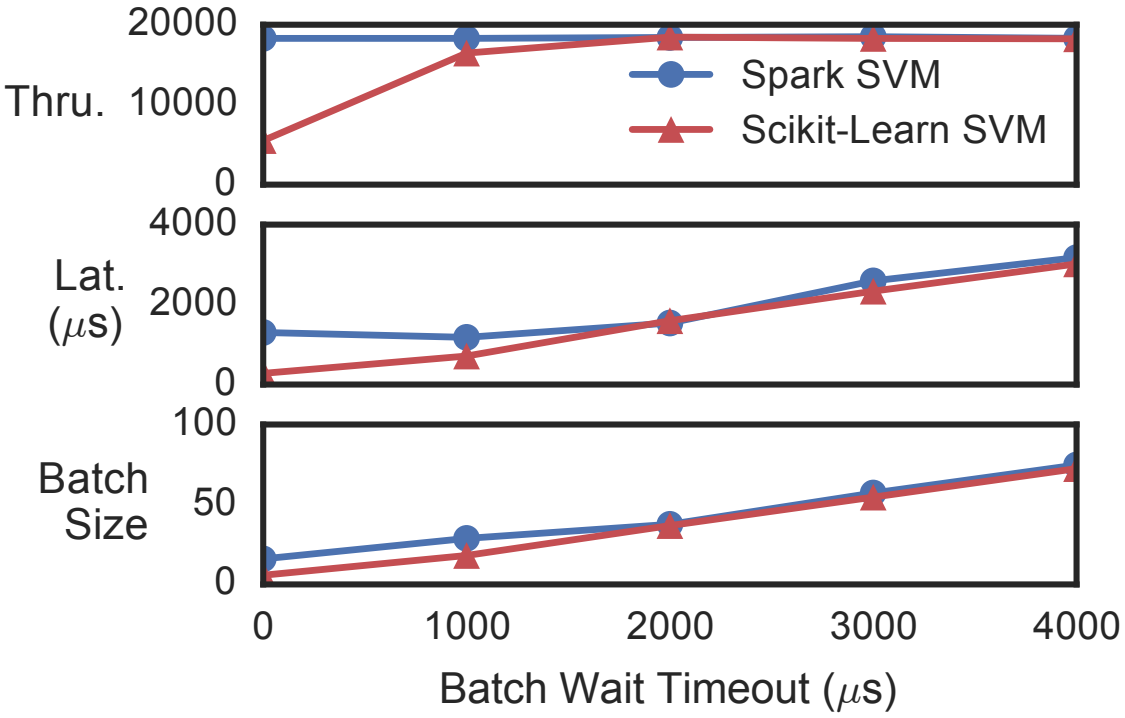


Figure 3.5: Throughput Increase from Delayed Batching.

```
interface Predictor<X,Y> {
  List<List<Y>> pred_batch(List<X> inputs);
}
```

Listing 3.1: Common Batch Prediction Interface for Model Containers. The batch prediction function is called via the RPC interface to compute the predictions for a batch of inputs. The return type is a nested list because each input may produce multiple outputs.

Model Containers

Model containers encapsulate the diversity of machine learning frameworks and model implementations within a uniform "narrow waist" remote prediction API. To add a new type of model to Clipper, model builders only need to implement the standard batch prediction interface in Listing 3.1. Clipper includes language specific container bindings for C++, Java, and Python. The model container implementations for most of the models in this paper only required a few lines of code.

To achieve process isolation, each model is managed in a separate Docker container. By placing models in separate containers, we ensure that variability in performance and stability of relatively immature state-of-the-art machine learning frameworks does not interfere with the overall availability of Clipper. Any state associated with a model, such as the model parameters, is provided

to the container during initialization and the container itself is stateless after initialization. As a result, resource intensive machine learning frameworks can be replicated across multiple machines or given access to specialized hardware (e.g., GPUs) when needed to meet serving demand.

Container Replica Scaling

Clipper supports replicating model containers, both locally and across a cluster, to improve prediction throughput and leverage additional hardware accelerators. Because different replicas can have different performance characteristics, particularly when spread across a cluster, Clipper performs adaptive batching independently for each replica.

In Figure 3.6 we demonstrate the linear throughput scaling that Clipper can achieve by replicating model containers across a cluster. With a four-node GPU cluster connected through a 10Gbps Ethernet switch, Clipper gets a 3.95x throughput increase from 19,500 qps when using a single model container running on a local GPU to 77,000 qps when using four replicas each running on a different machine. Because the model containers in this experiment are computationally intensive and run on the GPU, GPU throughput is the bottleneck and Clipper's RPC system can easily saturate the GPUs. However, when the cluster is connected through a 1Gbps switch, the aggregate throughput of the GPUs is higher than 1Gbps and so the network becomes saturated when replicating to a second remote machine. As machine-learning applications begin to consume increasingly bigger inputs, scaling from handcrafted features to large images, audio signals, or even video, the network will continue to be a bottleneck to scaling out prediction serving applications. This suggests the need for research into efficient networking strategies for remote predictions on large inputs.

3.5 Model Selection Layer

The **Model Selection Layer** uses feedback to dynamically select one or more of the deployed models and combine their outputs to provide more accurate and robust predictions. By allowing many candidate models to be deployed simultaneously and relying on feedback to adaptively determine the best model or combination of models, the model selection layer simplifies the deployment process for new models. By continuously learning from feedback throughout the lifetime of an application, the model selection layer automatically compensates for failing models without human intervention. By combining predictions from multiple models, the model selection layer boosts application accuracy and estimates prediction confidence.

There are a wide range of techniques for model selection and composition that span a tradeoff space of computational overhead and application accuracy. However, most of these techniques can be expressed with a simple *select*, *combine*, and *observe* API. We capture this API in the model selection policy interface (Listing 3.2) which governs the behavior of the model selection layer and allows users to introduce new model selection techniques themselves.

The model selection policy (Listing 3.2) defines four essential functions as well as a few basic types. In addition to the query and prediction types X and Y, the state type S encodes the learned

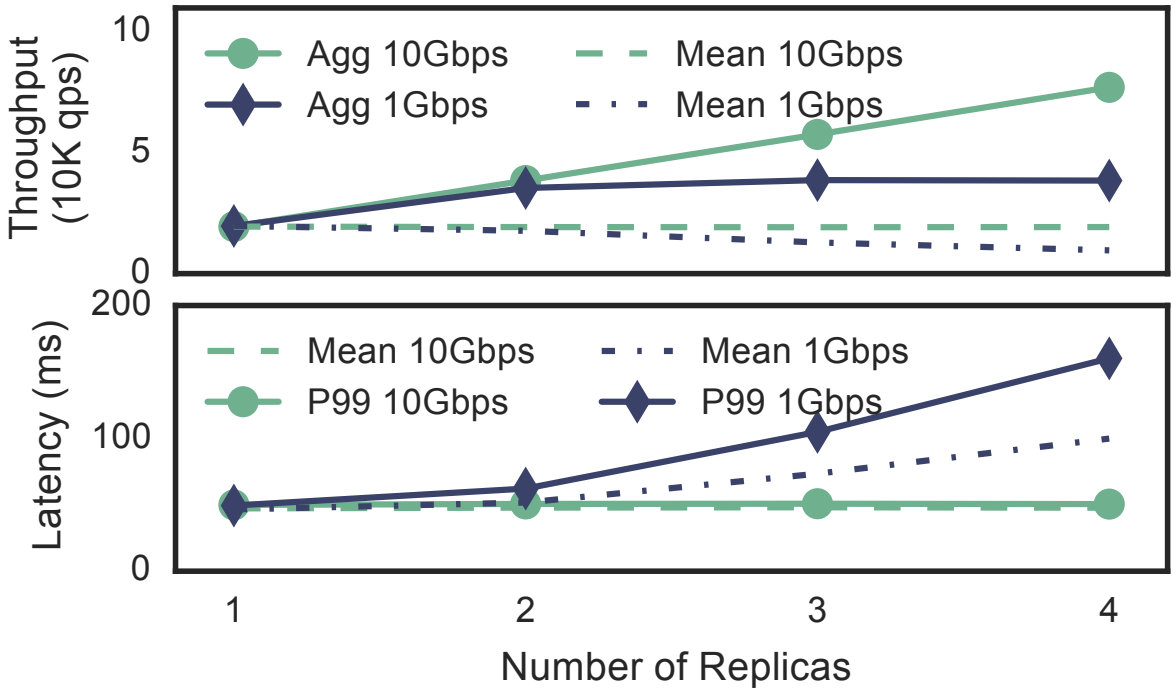


Figure 3.6: Scaling the Model Abstraction Layer Across a GPU Cluster. The solid lines refer to aggregate throughput of all the model replicas and the dashed lines refer to the mean per-replica throughput.

```
interface SelectionPolicy<S, X, Y> {
  S init();
  List<ModelId> select(S s, X x);
  pair<Y, double> combine(S s, X x,
    Map<ModelId, Y> pred);
  S observe(S s, X x, Y feedback,
    Map<ModelId, Y> pred);
}
```

Listing 3.2: Model Selection Policy Interface.

state of the selection algorithm. The *init* function returns an initial instance of the selection policy state. We isolate the selection policy state and require an initialization function to enable Clipper to efficiently instantiate many instances of the selection policy for fine-grained contextualized model selection (Section 3.5). The *select* and *combine* functions are responsible for choosing which models to query and how to combine the results. In addition, the *combine* function can compute other information about the predictions. For example, in Section 3.5 we leverage the *combine* function to provide a prediction confidence score. Finally, the *observe* function is used to update the state S based on feedback from front-end applications.

In the current implementation of Clipper we provide two generic model selection policies based on robust bandit algorithms developed by Auer et al. [12]. These algorithms span a trade-off between computation overhead and accuracy. The single model selection policy (Section 3.5) leverages the Exp3 algorithm to optimally *select* the best model based on noisy feedback with minimal computational overhead. The ensemble model selection policy (Section 3.5) is based on the Exp4 algorithm which adaptively *combines* the predictions to improve prediction accuracy and estimate confidence at the expense of increased computational cost from evaluating all models for each query. By implementing model selection policies that provide different cost-accuracy tradeoffs, as well as an API for users to implement their own policies, Clipper provides a mechanism to easily navigate the tradeoffs between accuracy and computational cost on a per-application basis. Furthermore, users can modify this choice over time as application workloads evolve and resources become more or less constrained.

Single Model Selection Policy

We can cast the model-selection process as a multi-armed bandit problem [108]. The multi-armed bandit[1] problem refers the task of optimally choosing between k possible actions (e.g., models) each with a stochastic reward (e.g., feedback). Because only the reward for the *selected* action can be observed, solutions to the multi-armed bandit problem must address the trade-off between *exploring* possible actions and *exploiting* the estimated best action.

There are numerous algorithms for the multi-armed bandits problem with a wide range of trade-offs. In this work we first explore the use of the simple randomized Exp3 [12] algorithm which makes few assumptions about the problem setting and has strong optimality guarantees. The Exp3 algorithm associates a weight $s_i = 1$ for each of the k deployed models and then randomly selects model i with probability $p_i = s_i / \sum_{j=1}^{k} s_j$. For each prediction $\hat{y}$, Clipper observes a loss $L(y, \hat{y}) \in [0, 1]$ with respect to the true value y (e.g., the fraction of words that were transcribed correctly during speech recognition). The Exp3 algorithm then updates the weight, $s_i \leftarrow s_i \exp(-\eta L(y, \hat{y})/p_i)$, corresponding to the selected model i. The constant η determines how quickly Clipper responds to recent feedback.

The Exp3 algorithm provides several benefits over manual experimentation and A/B testing, two common ways of performing model-selection in practice. Exp3 is both simple and robust, scaling well to model selection over a large number of models. It is a lightweight algorithm that requires only a single model evaluation for each prediction and thus performs well under heavy loads with negligible computational overhead. And Exp3 has strong theoretical guarantees that ensure it will quickly converge to an optimal solution.

Ensemble Model Selection Policies

It is a well-known result in machine learning [20, 108, 31, 70] that prediction accuracy can be improved by combining predictions from multiple models. For example, bootstrap aggregation [22]

[1] The term bandits refers to pull-lever slot machines found in casinos.

Framework	Model	Size (Layers)
Caffe	VGG[135]	13 Conv. and 3 FC
Caffe	GoogLeNet[143]	96 Conv. and 5 FC
Caffe	ResNet[68]	151 Conv. and 1 FC
Caffe	CaffeNet[49]	5 Conv. and 3 FC
TensorFlow	Inception[144]	6 Conv, 1 FC, & 3 Incept.

Table 3.2: Deep Learning Models. The set of deep learning models used to evaluate the ImageNet ensemble selection policy.

(a.k.a., bagging) is used widely to reduce variance and thereby improve generalization performance. More recently, ensembles were used to win the Netflix challenge [134], and a carefully crafted ensemble of deep neural networks was used to achieve state-of-the-art accuracy on the speech recognition corpus Google uses to power their acoustic models [70]. The ensemble model selection policies adaptively combine the predictions from *all* available models to improve accuracy, rather than select individual models.

In Clipper we use linear ensemble methods which compute a weighted average of the base model predictions. In Figure 3.7, we show the prediction error rate of linear ensembles on two benchmarks. In both cases linear ensembles are able to marginally reduce the overall error rate. In the ImageNet benchmark, the ensemble formulation achieves a 5.2% relative reduction in the error rate simply by combining off-the-shelf models (Table 3.2). While this may seem small, on the difficult computer vision tasks for which these models are used, a lot of time and energy is spent trying to achieve even small reductions in error, and marginal improvements are considered significant [124].

There are many methods for estimating the ensemble weights including linear regression, boosting [108], and bandit formulations. We adopt the bandits approach and use the Exp4 algorithm [12] to learn the weights. Unlike Exp3, Exp4 constructs a weighted *combination* of all base model predictions and updates weights based on the individual model prediction error. Exp4 confers many of the same theoretical guarantees as Exp3. But while the accuracy when using Exp3 is bounded by the accuracy of the single best model, Exp4 can further improve prediction accuracy as the number of models increases. The extent to which accuracy increases depends on the relative accuracies of the set of base models, as well as the independence of their predictions. This increased accuracy comes at the cost of increased computational resources consumed by each prediction in order to evaluate all the base models.

The accuracy of a deployed model can silently degrade over time. Clipper's online selection policies can automatically detect these failures using feedback and compensate by switching to another model (Exp3) or down-weighting the failing model (Exp4). To evaluate how quickly and effectively the model selection policies react in the presence of changes in model accuracy, we simulated a severe model degradation while receiving real-time feedback. Using the CIFAR dataset we trained five different Caffe models with varying levels of accuracy to perform object recognition. During a simulated run of 20K sequential queries with immediate feedback, we degraded the accuracy of the best-performing model after 5K queries and then allowed the model to recover after 10K queries.

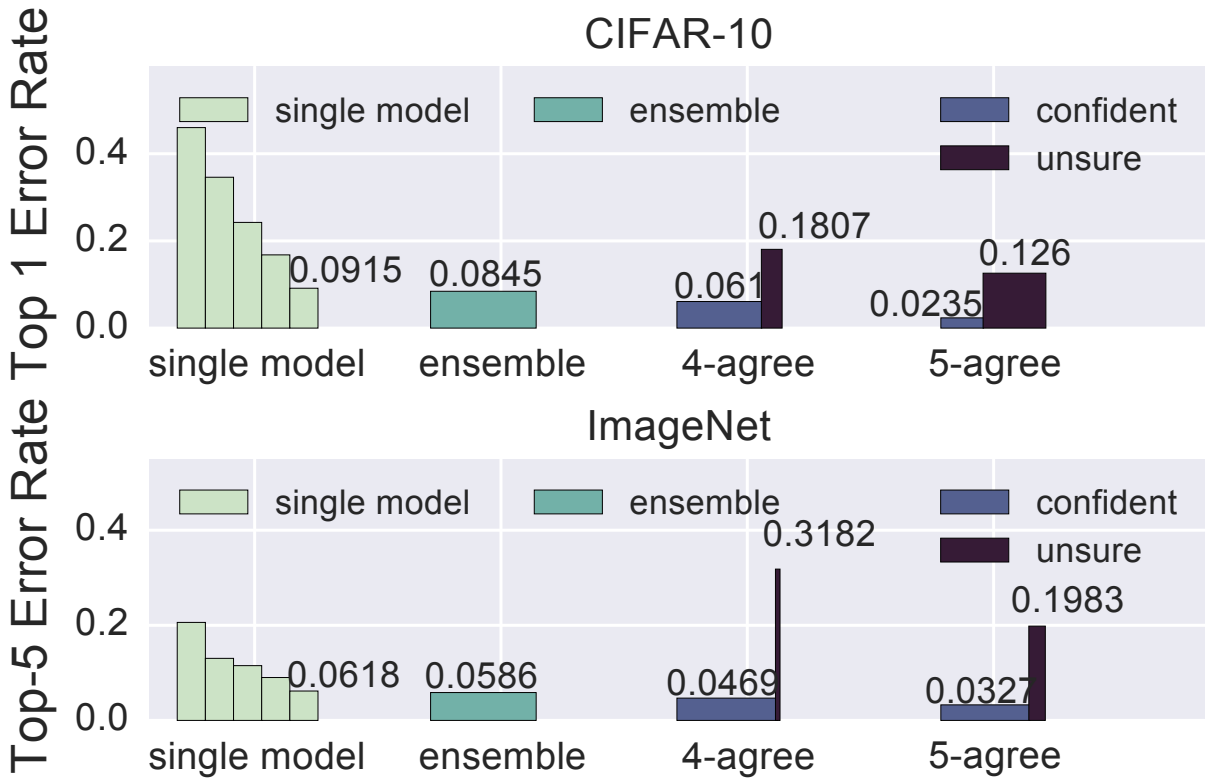


Figure 3.7: Ensemble Prediction Accuracy. The linear ensembles are composed of five computer vision models (Table 3.2) applied to the CIFAR and ImageNet benchmarks. The 4-agree and 5-agree groups correspond to ensemble predictions in which the queries have been separated by the ensemble prediction confidence (four or five models agree) and the width of each bar defines the proportion of examples in that category.

In Figure 3.8 we plot the cumulative average error rate for each of the five base models as well as the single (Exp3) and ensemble (Exp4) model selection policies. In the first 5K queries both model selection policies quickly converge to an error rate near the best performing model (model 5). When we degrade the predictions from model 5 its cumulative error rate spikes. The model selection policies are able to quickly mitigate the consequences of the increase in errors by learning to divert queries to the other models. When model 5 recovers after 10K queries the model selection policies also begin to improve by gradually sending queries back to model 5.

Robust Predictions

The advantages of online model selection go beyond detecting and mitigating model failures to leveraging new opportunities to improve application accuracy and performance. For many real-time decision-making applications, knowing the confidence of the prediction can significantly improve the end-user experience of the application.

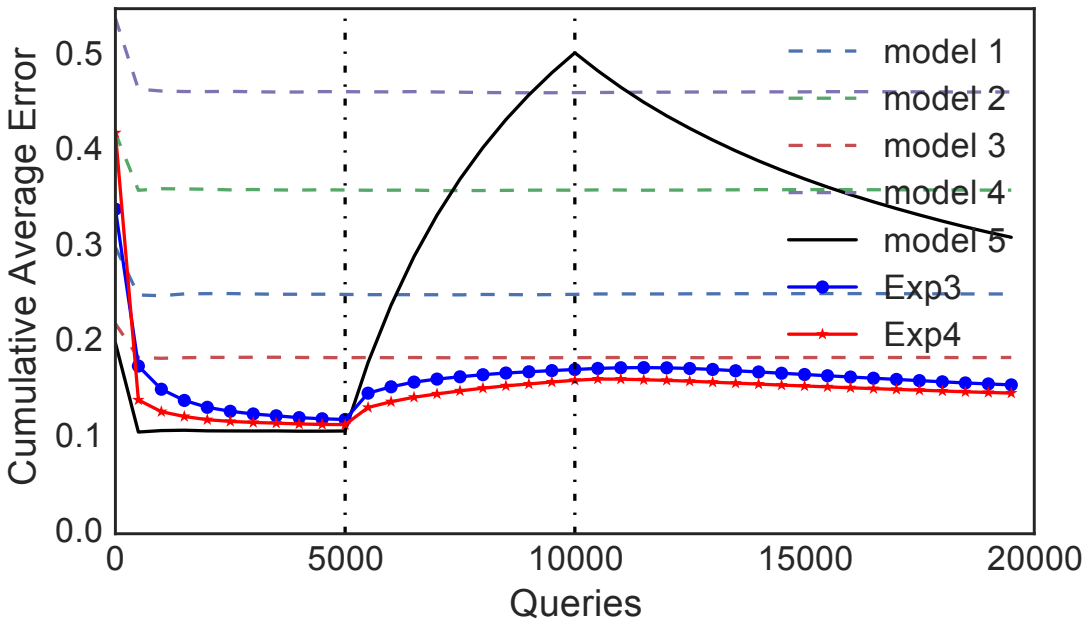
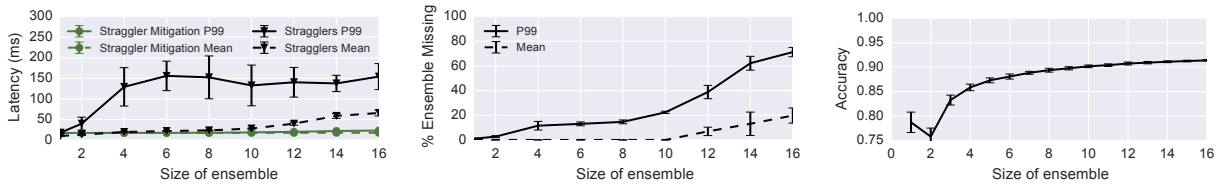


Figure 3.8: Behavior of Exp3 and Exp4 Under Model Failure. After 5K queries the performance of the lowest-error model is severely degraded, and after 10k queries performance recovers. Exp3 and Exp4 quickly compensate for the failure and achieve lower error than any static model selection.

For example, in many settings, applications have a sensible default action they can take when a prediction is unavailable. This is critical for building highly available applications that can survive partial system failures or when building applications where a mistake can be costly. Rather than blindly using all predictions regardless of the confidence in the result, applications can choose to only accept predictions above a confidence threshold by using the robust model selection policy. When the confidence in a prediction for a query falls below the confidence threshold, the application can instead use the sensible default decision for the query and avoid a costly mistake.

By evaluating predictions from multiple competing models concurrently we can obtain an estimator of the confidence in our predictions. In settings where models have high variance or are trained on random samples from the training data (e.g., bagging), agreement in model predictions is an indicator of prediction confidence. When evaluating the *combine* function in the ensemble selection policy we compute a measure of confidence by calculating the number of models that agree with the final prediction. End user applications can use this confidence score to decide whether to rely on the prediction. If we only consider predictions where multiple models agree, we can substantially reduce the error rate (see Figure 3.7) while declining to predict a small fraction of queries.



(a) Latency **(b)** Missing Predictions **(c)** Accuracy

Figure 3.9: Increase in stragglers from bigger ensembles. The **(a)** latency, **(b)** percentage of missing predictions, and **(c)** prediction accuracy when using the ensemble model selection policy on SK-Learn Random Forest models applied to MNIST. As the size of an ensemble grows, the prediction accuracy increases but the latency cost of blocking until all predictions are available grows substantially. Instead, Clipper enforces bounded latency predictions and transforms the latency cost of waiting for stragglers into a reduction in accuracy from using a smaller ensemble.

Straggler Mitigation

While the ensemble model selection policy can improve prediction accuracy and help quantify uncertainty, it introduces additional system costs. As we increase the size of the ensemble the computational cost of rendering a prediction increases. Fortunately, we can compensate for the increased prediction cost by scaling-out the model abstraction layer. Unfortunately, as we add model containers we increase the chance of stragglers adversely affecting tail latencies.

To evaluate the cost of stragglers, we deployed ensembles of increasing size and measured the resulting prediction latency (Figure 3.9a) under moderate query load. Even with small ensembles we observe the effect of stragglers on the P99 tail latency, which rise sharply to well beyond the 20ms latency objective. As the size of the ensemble increases and the system becomes more heavily loaded, stragglers begin to affect the mean latency.

To address stragglers, Clipper introduces a simple best-effort straggler-mitigation strategy motivated by the design choice that rendering a *late* prediction is worse than rendering an *inaccurate* prediction. For each query the model selection layer maintains a latency deadline determined by the latency SLO. At the latency deadline the *combine* function of the model selection policy is invoked with the *subset* of the predictions that are available. The model selection policy must render a final prediction using only the available base model predictions and communicate the potential loss in accuracy in its confidence score. Currently, we substitute missing predictions with their average value and define the confidence as the fraction of models that agree on the prediction.

The best-effort straggler-mitigation strategy prevents model container tail latencies from propagating to front-end applications by maintaining the latency objective as additional models are deployed. However, the straggler mitigation strategy reduces the size of the ensemble. In Figure 3.9b we plot the reduction in ensemble size and find that while tail latencies increase significantly with even small ensembles, most of the predictions arrive by the latency deadline. In Figure 3.9c we plot the effect of ensemble size on accuracy and observe that this ensemble can tolerate the loss of small numbers of component models with only a slight reduction in accuracy.

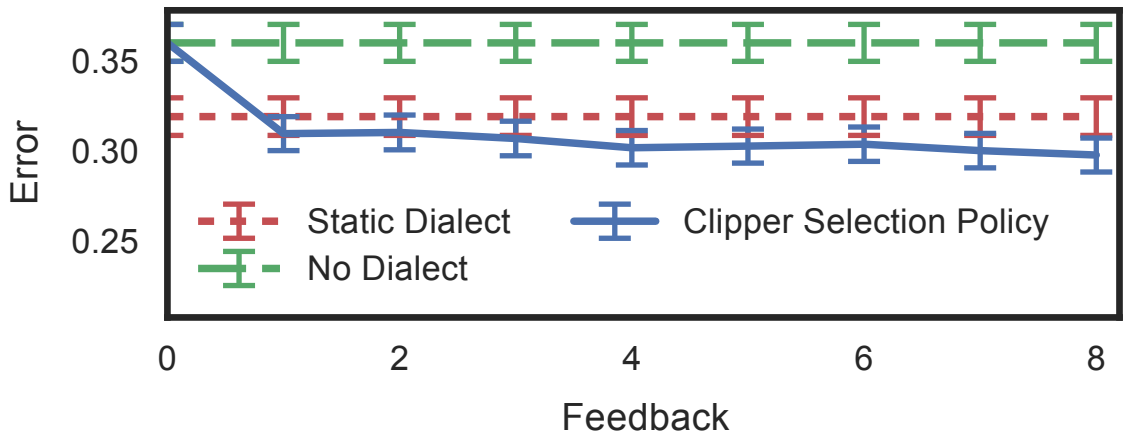


Figure 3.10: Personalized Model Selection. Accuracy of the ensemble selection policy on the speech recognition benchmark.

Contextualization

In many prediction tasks the accuracy of a particular model may depend heavily on context. For example, in speech recognition a model trained for one dialect may perform well for some users and poorly for others. However, selecting the right model or composition of models can be difficult and is best accomplished online in the model selection layer through feedback. To support context specific model selection, the model selection layer can be configured to instantiate a unique model selection state for each user, context, or session. The context specific session state is managed in an external database system. In our current implementation we use Redis.

To demonstrate the potential gains from personalized model selection we hosted a collection of TIMIT [59] voice recognition models each trained for a different dialect. We then evaluated (Figure 3.10) the prediction error rates using a single model trained across all dialects, the users' reported dialect model, and the Clipper ensemble selection policy. We first observe that the dialect-specific models out-perform the dialect-oblivious model, demonstrating the value of context to improve prediction accuracy. We also observe that the ensemble selection policy is able to quickly identify a combination of models that out-performs even the users' designated dialect model by using feedback from the serving workload.

3.6 System Comparison

In addition to the microbenchmarks presented in Section 3.4 and Section 3.5, we compared Clipper's performance to TensorFlow Serving and evaluate latency and throughput on three object recognition benchmarks.

TensorFlow Serving [147] is a recently released prediction serving system created by Google to

accompany their TensorFlow machine learning training framework. Similar to Clipper, TensorFlow Serving is designed for serving machine learning models in production environments and provides a high-performance prediction API to simplify deploying new algorithms and experimenting with new models without modifying frontend applications. TensorFlow Serving supports general TensorFlow models with GPU acceleration through direct integration with the TensorFlow machine learning framework and tightly couples the model and serving components in the same process.

TensorFlow Serving also employs batching to accelerate prediction serving. Batch sizes in TensorFlow Serving are static and rely on a purely timeout based mechanism to avoid starvation. TensorFlow Serving does not explicitly incorporate prediction latency objectives which must be achieved by manually tuning the batch size. Furthermore, TensorFlow Serving was designed to serve one model at a time and therefore does not directly support feedback, dynamic model selection, or composition.

To better understand the performance overheads introduced by Clipper's layered architecture and decoupled model containers, we compared the serving performance of Clipper and TensorFlow Serving on three TensorFlow object recognition deep networks of varying computational cost: a 4-layer convolutional neural network trained on the MNIST dataset [107], the 8-layer AlexNet [85] architecture trained on CIFAR-10 [84], and Google's 22-layer Inception-v3 network [144] trained on ImageNet. We implemented two Clipper model containers for each TensorFlow model, one that calls TensorFlow from the more standard and widely used Python API and one that calls TensorFlow from the more efficient C++ API. All models were run on a GPU using hand-tuned batch sizes (MNIST: 512, CIFAR: 128, ImageNet: 16) to maximize the throughput of TensorFlow Serving. The serving workload measured the maximum sustained throughput and corresponding prediction latency for each system.

Despite Clipper's modular design, we are able to achieve comparable throughput to TensorFlow Serving across all three models (Figure 3.11). The Python model containers suffer a 15-18% performance hit compared to the throughput of TensorFlow Serving, but the C++ model containers achieve nearly identical performance. This suggests that the high-level Python API for TensorFlow imposes a significant performance cost in the context of low-latency prediction-serving but that Clipper does not impose any additional performance degradation.

For these serving workloads, the throughput bottleneck is inference on the GPU. Both systems utilize additional queuing in order to saturate the GPU and therefore maximize throughput. For the Clipper model containers, we decomposed the prediction latency into component functions to demonstrate the overhead of the modular system design. The *predict* bar is the time spent performing inference within TensorFlow framework code. The *queue* bar is time spent queued within the model container waiting for the GPU to become available. The top bar includes the remaining system overhead, including query serialization and deserialization as well as copying into and out of the network stack. As Figure 3.11 illustrates, the RPC overheads are minimal on these workloads and the next prediction batch is queued as soon as the current batch is dispatched to the GPU for inference. TensorFlow Serving utilizes a similar queueing method to saturate the GPU, but because of the tight integration between TensorFlow Serving and the TensorFlow inference code, they are able to push the queueing into the TensorFlow framework code itself running in the same process.

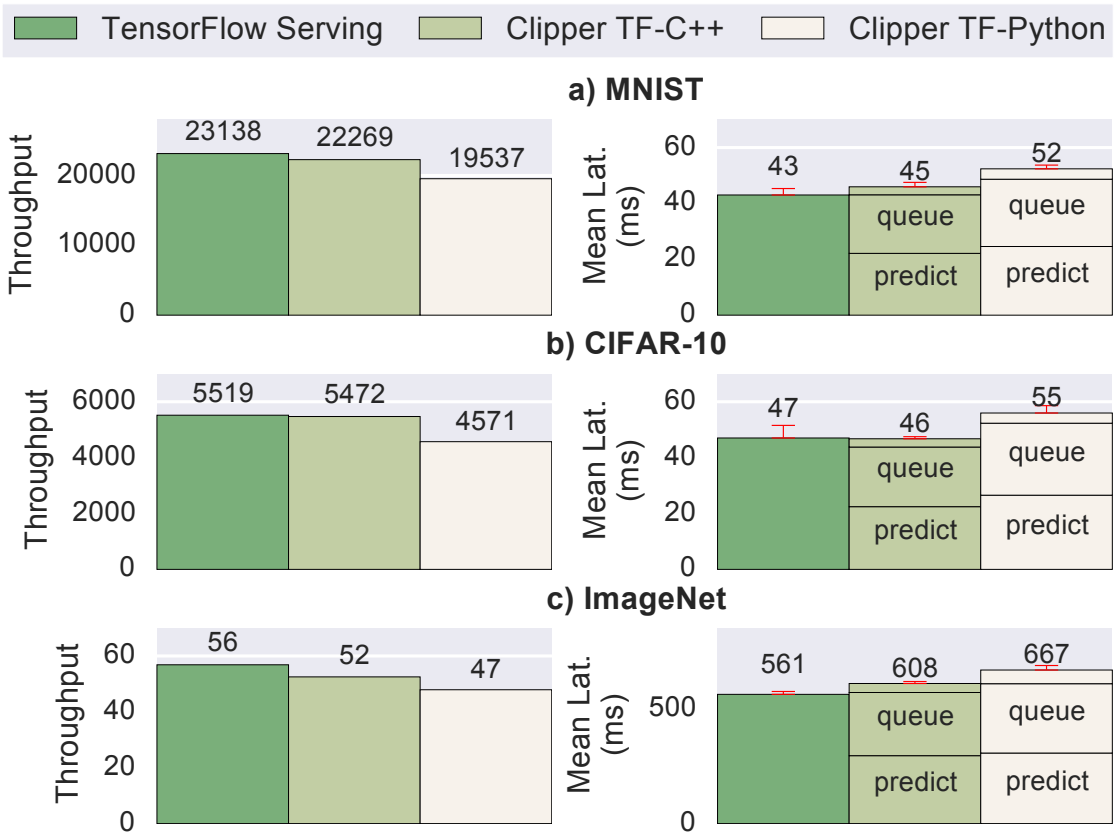


Figure 3.11: TensorFlow Serving Comparison. Comparison of peak throughput and latency (p99 latencies shown in error bars) on three TensorFlow models of varying inference cost. TF-C++ uses TensorFlow's C++ API and TF-Python the Python API.

By achieving comparable performance across this range of models, we have demonstrated that through careful design and implementation of the system, the modular architecture and substantially broader set of features in Clipper do not come at a cost of reduced performance on core prediction-serving tasks.

3.7 Limitations

While Clipper attempts to address many challenges in the context of prediction serving there are a few key limitations when compared to other designs like TensorFlow Serving. Most of these limitations follow directly from the design of the Clipper architecture which assumes models are below Clipper in the software stack, and thus are treated as black-box components.

Clipper does not optimize the execution of the models within their respective machine learning

frameworks. Slow models will remain slow when served from Clipper. In contrast, TensorFlow Serving is tightly integrated with model evaluation, and hence is able to leverage GPU acceleration and compilation techniques to speedup inference on models created with TensorFlow.

Similarly, Clipper does not manage the training or re-training of the base models within their respective frameworks. As a consequence, if all models are out-of-date or inaccurate Clipper will be unable to improve accuracy beyond what can be accomplished through ensembles.

3.8 Related Work

The closest projects to Clipper are LASER [3], Velox [42], and TensorFlow Serving [147]. The LASER system was developed at LinkedIn to support linear models for ad-targeting applications. Velox is a UC Berkeley research project to study personalized prediction serving with Apache Spark. TensorFlow Serving is the open-source prediction serving system developed by Google for TensorFlow models. In our experiments we only compare against TensorFlow Serving, because LASER is not publicly available, and the current prototype of Velox has very limited functionality.

All three systems propose mechanisms to address latency and throughput. Both LASER and Velox utilize caching at various levels in their systems. In addition, LASER also uses a straggler mitigation strategy to address slow feature evaluation. Neither LASER or Velox discuss batching. Conversely, TensorFlow Serving does not employ caching and instead leverages batching and hardware acceleration to improve throughput.

LASER and Velox both exploit a form of model decomposition to incorporate feedback and context similar to the linear ensembles in Clipper. However, LASER does not incorporate feedback in real-time, Velox does not support bandits and neither system supports cross framework learning. Moreover, the techniques used for online learning and contextualization in both of these systems are captured in the more general Clipper selection policy. In contrast, TensorFlow Serving has no mechanism to achieve personalization or adapt to real-time feedback.

Finally, LASER, Velox, and TensorFlow Serving are all vertically integrated; they focused on serving predictions from a single model or framework. In contrast, Clipper supports a wide range of machine learning models and frameworks and simultaneously addresses latency, throughput, and accuracy in a single serving system.

Application Specific Prediction Serving: There has been considerable prior work in application and model specific prediction-serving. Much of this work has focused on content recommendation, including video-recommendation [45], ad-targeting [102, 63], and product-recommendations [91]. Outside of content recommendation, there has been recent success in speech recognition [90, 136] and internet-scale resource allocation [58]. While many of these applications require real-time predictions, the solutions described are highly application-specific and tightly coupled to the model and workload characteristics. As a consequence, much of this work solves the same systems challenges in different application areas. In contrast, Clipper is a general-purpose system capable of serving many of these applications.

Parameter Server: There has been considerable work in the learning systems community on parameter-servers [46, 96, 5, 155]. While parameter-servers do focus on reduced latency and

caching, they do so in the context of *model training*. In particular they are a specialized type of key-value store used to coordinate updates to model parameters in a distributed training system. They are not typically used to serve predictions.

General Serving Systems: The high-performance serving architecture of Clipper draws from prior work on highly-concurrent serving systems[116, 154, 127, 112]. The division of functionality into vertical stages introduced by [154] is similar to the division of Clipper's architecture into independent layers. Notably, while the dominant cost in data-serving systems tends to be IO, in prediction serving it is computation. This changes both physical resource allocation and batching and latency-hiding strategies.

3.9 Conclusion

In this work we identified three key challenges of prediction serving: latency, throughput, and accuracy, and proposed a new layered architecture that addresses these challenges by interposing between end-user applications and existing machine learning frameworks.

As an instantiation of this architecture, we introduced the Clipper prediction serving system. Clipper isolates end-user applications from the variability and diversity in machine learning frameworks by providing a common prediction interface. As a consequence, new machine learning frameworks and models can be introduced without modifying end-user applications.

We addressed the challenges of prediction serving latency and throughput within the Clipper Model Abstraction layer. The model abstraction layer lifts caching and adaptive batching strategies above the machine learning frameworks to achieve up to a 26x improvement in throughput while maintaining strict bounds on tail latency and providing mechanisms to scale serving across a cluster. We addressed the challenges of accuracy in the Clipper Model Selection Layer. The model selection layer enables many models to be deployed concurrently and then dynamically selects and combines predictions from each model to render more robust, accurate, and contextualized predictions while mitigating the cost of stragglers.

We evaluated Clipper using four standard machine-learning benchmark datasets spanning computer vision and speech recognition applications. We demonstrated Clipper's capacity to bound latency, scale heavy workloads across nodes, and provide accurate, robust, and contextual predictions. We compared Clipper to Google's TensorFlow Serving system and achieved parity on throughput and latency performance, demonstrating that the modular container-based architecture and substantial additional functionality in Clipper can be achieved with minimal performance penalty.

Chapter 4

InferLine: ML Prediction Pipeline Provisioning and Management for Tight Latency Objectives

4.1 Introduction

Applications today increasingly rely on ML inference over multiple models linked together in a dataflow DAG. Examples include a digital assistant service (e.g., Amazon Alexa), which combines audio pre-processing with downstream models for speech recognition, topic identification, question interpretation and response and text-to-speech to answer a user's question. The natural evolution of these applications leads to a growth in the complexity of the prediction pipelines. At the same time, their latency-sensitive nature dictates tight tail latency constraints (e.g., 200-300ms). As the pipelines grow and the models used become increasingly sophisticated, they present a unique set of systems challenges for provisioning and managing these pipelines.

Each stage of the pipeline must be assigned the appropriate hardware accelerator (e.g., CPU, GPU, TPU) — a task complicated by increasing hardware heterogeneity. Each model must be configured with the appropriate query batch size — necessary for optimal utilization of the hardware. And each pipeline stage can be replicated to meet the application throughput requirements. Per-stage decisions with respect to the hardware type and batch size affect the latency contributed by each stage towards the end-to-end pipeline latency bound by the application-specified Service Level Objective (SLO). This creates a combinatorial search space with three control dimensions per model (hardware type, batch size, number of replicas) and global constraints on aggregate latency.

A number of prediction serving systems exist today, including Clipper [41], TensorFlow Serving [147], and NVIDIA TensorRT Inference Server [146] that optimize for single model serving. This pushes the complexity of coordinating cross-model interaction and, particularly, the questions of per-model configuration to meet application-level requirements, to the application developer. To the best of our knowledge, no system exists today that automates the process of pipeline provisioning and configuration, subject to specified tail latency SLO in a cost-aware manner. Thus, the goal of

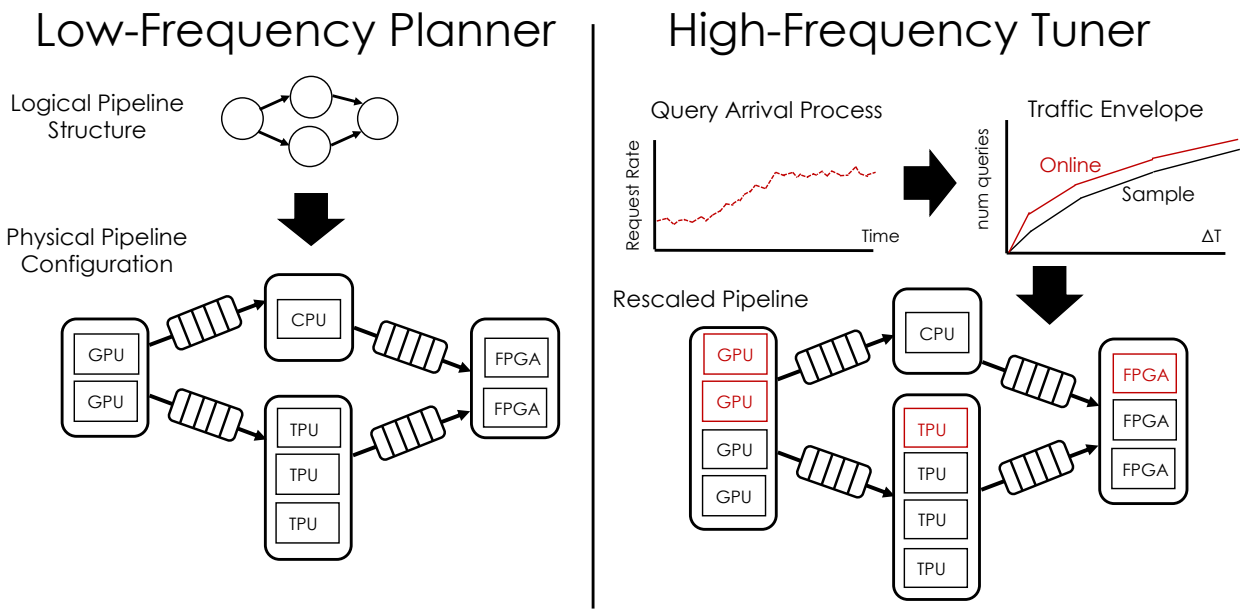


Figure 4.1: InferLine System Overview

this paper is to address the problem of configuring and managing multi-stage prediction pipelines subject to end-to-end tail latency constraints cost efficiently.

We propose InferLine — a system for provisioning and management of ML inference pipelines. It composes with existing prediction serving frameworks, such as Clipper and TensorFlow Serving. It is necessary for such a system to contain two principal components: a low-frequency *planner* and a high-frequency *tuner*. The low-frequency planner is responsible for navigating the combinatorial search space to produce per-model pipeline configuration relatively infrequently to minimize cost. It is intended to run periodically to correct for workload drift or fundamental changes in the steady-state, long-term query arrival process. It is also necessary for integrating new models added to the repository and to integrate new hardware accelerators. The high frequency component is intended to operate at time scales three orders of magnitude faster. It monitors instantaneous query arrival traffic and tunes the running pipeline to accomodate unexpected query spikes cost efficiently to maintain latency SLOs under bursty and stochastic workloads.

To enable efficient exploration of the combinatorial configuration space, InferLine profiles each stage in the pipeline individually and uses these profiles and a discrete event simulator to accurately estimate end-to-end pipeline latency given the hardware configuration and batchsize parameters. The low-frequency *planner* uses a constrained greedy search algorithm to find the cost-minimizing pipeline configuration that meets the end-to-end tail latency constraint determined using the discrete event simulator on a sample planning trace.

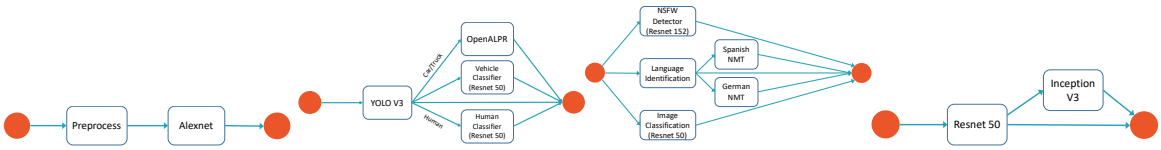
The InferLine high-frequency *tuner* leverages traffic envelopes built using network calculus tools to capture the arrival process dynamics across multiple time scales and determine when and how to react to changes in the arrival process. As a consequence, the tuner is able to maintain the

latency SLO in the presence of transient spikes and sustained variation in the query arrival process.

In summary, the primary contribution of this paper is a system for provisioning and managing machine learning inference pipelines for latency-sensitive applications cost efficiently. It consists of two principal components that operate at time scales orders of magnitude apart to configure the system for near-optimal performance. The planner builds on a high-fidelity model-based networked queueing simulator, while the tuner uses network calculus techniques to rapidly adjust pipeline configuration, absorbing unexpected query traffic variation cost efficiently.

We apply InferLine to provision and manage resources for multiple state-of-the-art prediction serving systems. We show that InferLine significantly outperforms alternative pipeline configuration baselines by a factor of up to 7.6X on cost, while exceeding 99% latency SLO attainment—the highest level of attainment achieved in relevant prediction serving literature.

4.2 Background and Motivation



(a) Image Processing **(b)** Video Monitoring **(c)** Social Media **(d)** TF Cascade

Figure 4.2: Example Pipelines. We evaluate InferLine on four prediction pipelines that span a wide range of models, control flow, and input characteristics.

Prediction pipelines combine multiple machine learning models and data transformations to support complex prediction tasks [138]. For instance, state-of-the-art visual question answering services [8, 99] combine language models with vision models to answer the question.

A prediction pipeline can be represented as a directed acyclic graph (DAG), where each vertex corresponds to a model (e.g., mapping images to objects in the image) or a basic data transformation (e.g., extracting key frames from a video) and edges represent dataflow between these vertices.

In this paper we study several (Figure 4.2) representative prediction pipeline motifs. The Image Processing pipeline consists of basic image pre-processing (e.g., cropping and resizing) followed by image classification using a deep neural network. The Video Monitoring pipeline was inspired by [161] and uses an object detection model to identify vehicles and people and then performs subsequent analysis including vehicle and person identification and license plate extraction on any relevant images. The Social Media pipeline translates and categorizes posts based on both text and linked images by combining computer vision models with multiple stages of language models to identify the source language and translate the post if necessary. Finally, the TensorFlow (TF) Cascade pipeline combines fast and slow TensorFlow models, invoking the slow model only when necessary.

In the Social Media, Video Monitoring, and TF Cascade pipelines, a subset of models are invoked based on the output of earlier models in the pipeline. This conditional evaluation pattern appears in bandit algorithms [95, 12] used for model personalization as well as more general cascaded prediction pipelines [101, 66, 9, 142].

We show that for such pipelines InferLine is able to maintain latency constraints with P99 service level objectives (99% of query latencies must be below the constraint) at low cost, even under bursty and unpredictable workloads.

Challenges

Prediction pipelines present new challenges for the design and provisioning of prediction serving systems. First, we first discuss how the proliferation of specialized hardware accelerators and the need to meet end-to-end latency constraints leads to a combinatorially large configuration space. Second, we discuss some of the complexities of meeting tight latency SLOs under bursty stochastic query loads. Third, we contrast this work with ideas from the data stream processing literature, which shares some *structural* similarities, but is targeted at fundamentally different applications and performance goals.

Combinatorial Configuration Space Many machine learning models can be computationally intensive with substantial opportunities for parallelism. In some cases, this parallelism can result in orders of magnitude improvements in throughput and latency. For example, in our experiments we found that TensorFlow can render predictions for the relatively large ResNet152 neural network at 0.6 queries per second (QPS) on a CPU and at 50.6 QPS on an NVIDIA Tesla K80 GPU, an 84x difference in throughput (Figure 4.3). However, not all models benefit equally from hardware accelerators. For example, several widely used classical models (e.g., decision trees [23]) can be difficult to parallelize on GPUs, and common data transformations (e.g. text feature extraction) often cannot be efficiently computed on GPUs.

In many cases, to fully utilize the available parallel hardware, queries must be processed in batches (e.g., ResNet152 required a batch size of 32 to maximize throughput on the K80). However, processing queries in a batch can also increase latency, as we see in Figure 4.3. Because most hardware accelerators operate at vector level parallelism, the first query in a batch is not returned until the last query is completed. As a consequence, it is often necessary to set a *maximum* batch size to bound query latency. However, the choice of the maximum batch size depends on the hardware and model and affects the end-to-end latency of the pipeline.

Finally, in heavy query load settings it is often necessary to replicate individual operators in the pipeline to provide the throughput demanded by the workload. As we scale up pipelines through replication, each operator scales differently, an effect that can be amplified by the use of conditional control flow within a pipeline causing some components to be queried more frequently than others. Low cost configurations require fine-grained scaling of each operator.

Allocating parallel hardware resources to a single model presents a complex model dependent trade-off space between cost, throughput, and latency. This trade-off space grows exponentially with each model in a prediction pipeline. Decisions made about the choice of hardware, batching

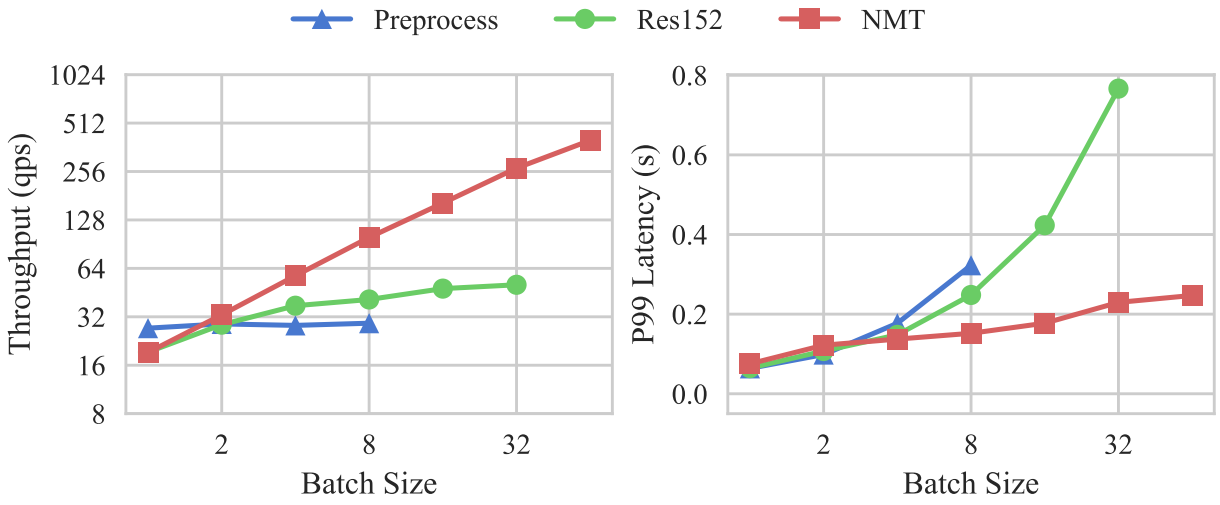


Figure 4.3: Example Model Profiles on K80 GPU. The preprocess model has no internal parallelism and cannot utilize a GPU. Thus, it sees no benefit from batching. Res152 (image classification) & TF-NMT(text translation model) benefit from batching on a GPU but at the cost of increased latency.

parameters, and replication factor at one stage of the pipeline affect the set of feasible choices at the other stages due to the need to meet *end-to-end* latency constraints. For example, trading latency for increased throughput on one model by increasing the batch size reduces the latency budget of other models in the pipeline and, as a consequence, constrains feasible hardware configurations as well.

Queueing Delays As stages of a pipeline may operate at different speeds, due to resource and model heterogeneity, it is necessary to have a queue per stage. Queueing also allows to absorb query inter-arrival process irregularities and can be a significant end-to-end latency component. Queueing delay must be explicitly considered during pipeline configuration, as it directly depends on the relationship between the inter-arrival process and system configuration.

Stochastic and Unpredictable Workloads Prediction serving systems must respond to bursty, stochastic query streams. At a high-level these stochastic processes can be characterized by their average arrival rate λ and their coefficient of variation, a dimensionless measure of variability defined by $CV = \frac{\sigma^2}{\mu^2}$, where $\mu = \frac{1}{\lambda}$ and σ are the mean and standard-deviation of the query inter-arrival time. Processes with higher CV have higher variability and often require additional over-provisioning to meet latency objectives. Clearly, over-provisioning the whole pipeline on specialized hardware can be prohibitively expensive. Therefore, it is critical to be able to identify and provision the bottlenecks in a pipeline to accommodate the bursty arrival process. Finally, as the workload changes, we need mechanisms to monitor, quickly detect, and *tune* individual stages in the pipeline.

Comparison to Stream Processing Systems Many of the challenges around configuring and scaling pipelines have been studied in the context of generic data stream processing systems [53, 126, 148, 140]. However, these systems focus their effort on supporting more traditional data processing workloads, which include stateful aggregation operators and support for a variety of windowing operations. As a result, the concept of per-query latency is often ill-defined in data pipelines, and instead these systems tend to focus on maximizing throughput while avoiding backpressure, with latency as a second order performance goal (Section 4.8).

4.3 System Design and Architecture

In this section, we provide a high-level overview of the main system components in InferLine (Figure 4.1). The system requires a *planner* that operates infrequenty and re-configures the whole pipeline w.r.t. all of our control parameters and a *tuner* that makes adjustments to the pipeline configurations in response to dynamically observed query traffic patterns.

InferLine runs on top of any prediction serving system that meets a few simple requirements. The underlying serving system must be able to 1) deploy multiple replicas of a model and scale the number of replicas at runtime, 2) allow for batched inference with the ability to configure a maximum batch size, and 3) use a centralized batched queueing system to distribute batches among model replicas. The first two properties are necessary for InferLine to configure the serving engine, and a centralized queueing system provides deterministic queueing behavior that can be accurately simulated by the **Estimator**. In our experimental evaluation, we run InferLine with both Clipper [41] and TensorFlow Serving [147]. Both systems needed only minor modifications to meet these requirements.

Using InferLine: To deploy a new prediction pipeline managed by InferLine, developers provide a driver program, sample query trace used for planning, and a latency service level objective. The driver function interleaves application-specific code with asynchronous calls to models hosted in the underlying serving system to execute the pipeline.

The **Planner** runs as a standalone Python process that runs periodically independent of the prediction serving framework. The **Tuner** runs as a standalone process implemented in C++. It observes the incoming arrival trace streamed to it by the centralized queueing system and triggers model addition/removal executed by serving-framework-specific APIs.

Low-Frequency Planning: The first time planning is performed, InferLine uses the **Profiler** to create performance profiles of all the individual models referenced by the driver program. A performance profile captures model throughput as a function of hardware type and maximum batch size. An entry in the model profile is measured empirically by evaluating the model in isolation in the given configuration using the queries in the sample trace. The model profiles are saved and reused in subsequent runs of the planner.

The **Planner** finds a cost-efficient initial pipeline configuration subject to the end-to-end latency SLO and the specified arrival process. It uses a globally-aware, cost-minimizing optimization

algorithm to set the three control parameters for each model in the pipeline. In each iteration of the optimization algorithm, the Planner uses the model profiles to select a cost-minimizing step while relying on the **Estimator** to check for latency constraint violations. After the initial configuration is generated and the pipeline is deployed to serve live traffic, the Planner is re-run periodically (hours to days) on the most recent arrival history to find a cost-optimal configuration for the current workload. This also allows integrating new models and hardware.

High-Frequency Tuning: The **Tuner** monitors the dynamic behavior of the arrival process to adjust per-model replication factors and maintain high SLO attainment at low cost. The Tuner continuously monitors the current traffic envelope [88] to detect deviations from the planning trace traffic envelope at different timescales simultaneously. By analyzing the timescale at which the deviation occurred, the Tuner is able to take appropriate mitigating action within seconds to ensure that SLOs are met without unnecessarily increasing cost. It ensures that latency SLOs are maintained during unexpected changes to the arrival workload in between runs of the Planner.

4.4 Low-Frequency Planning

During planning, the **Profiler**, **Estimator** and **Planner** are used to estimate model performance characteristics and optimally provision and configure the system for a given sample workload and latency SLO. In this section, we expand on each of these three components.

Profiler

The **Profiler** creates performance profiles for each of the models in the pipeline as a function of batch size and hardware. Profiling begins with InferLine executing the sample set of queries on the pipeline. This generates input data for profiling each of the component models individually. We also track the frequency of queries visiting each model, called the *scale factor*, s. The scale factor represents the conditional probability that a model will be queried given a query entering the pipeline, independent of the behavior of any other models. It is used by the **Estimator** to simulate the effects of conditional control flow on latency (Section 4.4) and the **Tuner** to make scaling decisions (Section 4.5).

The Profiler captures model throughput as a function of hardware type and batch size to create per-model performance profiles. An individual model configuration corresponds to a specific value for each of these parameters as well as the model's replication factor. Because the models scale horizontally, profiling a single replica is sufficient. Profiling only needs to be performed once for each hardware and batch size pair and is re-used in subsequent runs of the Planner.

Estimator

The **Estimator** is responsible for rapidly estimating the end-to-end latency of a given pipeline configuration for the sample query trace. It takes as input a pipeline configuration, the individual

Algorithm 1: Find an initial, feasible configuration

```
1  Function Initialize(pipeline, slo):
2      foreach model in pipeline do
3          model.batchsize = 1;
4          model.replicas = 1;
5          model.hw = BestHardware(model);
6      if ServiceTime(pipeline) ≤ slo then
7          return False;
8      else
9          while not Feasible(pipeline, slo) do
10             model = FindMinThru(pipeline);
11             model.replicas += 1;
12         return pipeline;
```

model profiles, and a sample trace of the query workload, and returns accurate estimates of the latency for *each query* in the trace. The Estimator is implemented as a continuous-time, discrete-event simulator [15], simulating the entire pipeline, including queueing delays. The simulator maintains a global logical clock that is advanced from one discrete event to the next with each event triggering future events that are processed in temporal order. Because the simulation only models discrete events, we are able to faithfully simulate hours worth of real-world traces in hundreds of milliseconds.

The Estimator simulates the deterministic behavior of queries flowing through a centralized batched queueing system. It combines this with the model profile information which informs the simulator how long a model running on a specific hardware configuration will take to process a batch of a given size.

Planning Algorithm

At a high-level, the planning algorithm is an iterative constrained optimization procedure that greedily minimizes cost while ensuring that the latency constraint is satisfied. The algorithm can be divided into two phases. In the first (Alg. 1), it finds a feasible initial configuration that meets the latency SLO while ignoring cost. In the second (Alg. 2), it greedily modifies the configuration to reduce the cost while using the Estimator to identify and reject configurations that violate the latency SLO. The algorithm converges when it can no longer make any cost reducing modifications to the configuration without violating the SLO.

Initialization (Alg. 1): First, an initial latency-minimizing configuration is constructed by setting the batch size to 1 using the lowest latency hardware available for each model (lines 2-5). If the service time under this configuration (the sum of the processing latencies of all the models on

Algorithm 2: Find the min-cost configuration

```
1  Function MinimizeCost(pipeline, slo):
2      pipeline = Initialize(pipeline, slo);
3      if pipeline == False then
4          return False;
5      actions = [IncreaseBatch, RemoveReplica, DowngradeHW ];
6      repeat
7          best = NULL;
8          foreach model in pipeline do
9              foreach action in actions do
10                 new = action(model, pipeline);
11                 if Feasible(new) then
12                     if new.cost < best.cost then
13                         best = new;
14         if best is not NULL then
15             pipeline = best;
16     until best == NULL;
17     return pipeline;
```

the longest path through the pipeline DAG) is greater than the SLO then the latency constraint is infeasible given the available hardware and the Planner terminates (lines 6-7). Otherwise, the Planner then iteratively determines the *throughput bottleneck* and increases that model's replication factor until it is no longer the bottleneck (lines 9-11).

Cost-Minimization (Alg. 2): In each iteration of the cost-minimizing process, the Planner considers three candidate modifications for each model: increase the batch size, decrease the replication factor, or downgrade the hardware (line 5), searching for the modification that maximally decreases cost while still meeting the latency SLO. It evaluates each modification on each model in the pipeline (lines 8-10), discarding candidates that violate the latency SLO according to the Estimator (line 11).

The **batch size** only affects throughput and does not affect cost. It will therefore only be the cost-minimizing modification if the other two would create infeasible configurations. Increasing the batch size does increase latency. The batch size is increased by factors of two as the throughput improvements from larger batch sizes have diminishing returns (observe Figure 4.3). In contrast, decreasing the **replication factor** directly reduces cost. Removing replicas is feasible when a previous iteration of the algorithm has increased the batch size for a model, increasing the per-replica throughput.

Downgrading hardware is more involved than the other two actions, as the batch size and

replication factor for the model must be re-evaluated to account for the differing batching behavior of the new hardware. It is often necessary to reduce the batch size and increase replication factor to find a feasible pipeline configuration. However, the reduction in hardware price sometimes compensates for the increased replication factor. For example, in Figure 4.9, the steep decrease in cost when moving from an SLO of 0.1 to 0.15 can be largely attributed to downgrading the hardware of a language identification model from a GPU to a CPU.

To evaluate a hardware downgrade, we first freeze the configurations of the other models in the pipeline and perform the initialization stage for that model using the next cheapest hardware. The planner then performs a localized version of the cost-minimizing algorithm to find the batch size and replication factor for the model on the newly downgraded resource allocation needed to reduce the cost of the previous configuration. If there is no cost reducing feasible configuration the hardware downgrade action is rejected.

At the point of termination, the planning algorithm provides the following guarantees: (1) If there is a configuration that meets the latency SLO, then the algorithm will return a valid configuration. (2) There is no single action that can be taken to reduce cost without violating the SLO.

4.5 High-Frequency Tuning

InferLine's Planner finds an efficient, low-cost configuration that is guaranteed to meet the provided latency objective. However, this guarantee only holds for the sample planning workload provided to the planner. Real workloads evolve over time, changing in both arrival rate (change in λ) as well as becoming more or less bursty (change in CV). When the serving workload deviates from the sample, the original configuration will either suffer from queue buildups leading to SLO misses or be over-provisioned and incur unnecessary costs. The **Tuner** both *detects* these changes as they occur and takes the appropriate *scaling action* to maintain both the latency constraint and cost-efficiency objective.

In order to maintain P99 latency SLOs, the Tuner must be able to detect changes in the arrival workload dynamics across multiple timescales simultaneously. The Planner guarantees that the pipeline is adequately provisioned for the sample trace. The Tuner's detection mechanism detects when the current request workload exceeds the sample workload. To do this, we draw on the idea of traffic envelopes from network calculus [88] to characterize the workloads.

A traffic envelope for a workload is constructed by sliding a window of size ΔT_i over the workload's inter-arrival process and capturing the maximum number of queries seen anywhere within this window. Thus, each $x = \Delta T_i$ is mapped to $y = q_i$ (number of queries) for all x over the duration of a trace. This powerful characterization captures how much the workload can burst in any given interval of time. In practice, we discretize the x-axis by setting the smallest ΔT_i to T_s, the service time of the system, and then double the window size up to 60 seconds. For each such interval, the maximum arrival rate r_i for this interval can be computed as $r_i = \frac{q_i}{\Delta T_i}$. By measuring r_i across all ΔT_i *simultaneously* we capture a fine-grain characterization of the arrival workload that enables simultaneous detection of changes in both short term (burstiness) and long term (average arrival rate) traffic behavior.

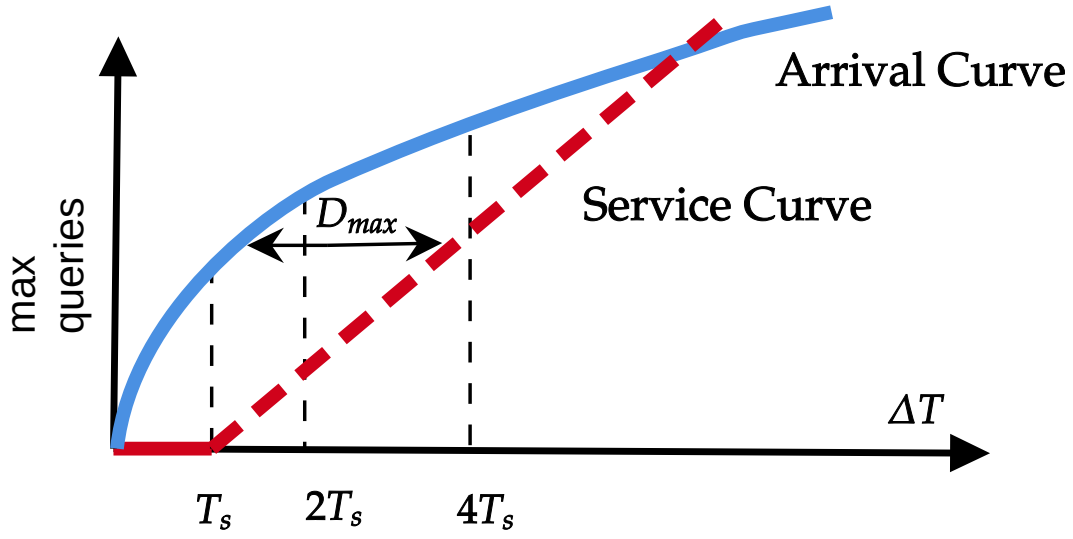


Figure 4.4: Arrival and Service Curves. The arrival curve captures the maximum number of queries to be expected in any interval of time x seconds wide. The service curve plots the expected number of queries processed in an interval of time x seconds wide.

Initialization During planning, the Planner constructs the traffic envelope for the sample arrival trace. The Planner also computes the max-provisioning ratio for each model $\rho_m = \frac{\lambda}{\mu_m}$, the ratio of the arrival rate λ to the maximum throughput of the model μ in its current configuration. While the max-provisioning ratio is not a fundamental property of the pipeline, it provides a useful heuristic to measure how much "slack" the Planner has determined is needed for this model to be able to absorb bursts and still meet the SLO. The Planner then provides the Tuner with the traffic envelope for the sample trace, the max-provisioning ratio ρ_m and single replica throughput μ_m for each model in the pipeline.

In the low-latency applications that InferLine targets, failing to scale up the pipeline in the case of an increased workload results in missed latency objectives and degraded quality of service, while failing to scale down the pipeline in the case of decreased workload only results in slightly higher costs. We therefore handle the two situations separately.

Scaling Up The Tuner continuously computes the traffic envelope for the current arrival workload. This yields a set of arrival rates for the current workload that can be directly compared to those of the sample workload. If any of the current rates exceed their corresponding sample rates, the pipeline is underprovisioned and the Tuner checks whether it add replicas for any models in the pipeline.

At this point, not only has the Tuner detected that rescaling may be necessary, it also knows what arrival rate it needs to reprovision the pipeline for: the current workload rate r_{max} that triggered rescaling. If the overall λ of the workload has not changed but it has become burstier, this will be a rate computed with a smaller ΔT_i, and if the burstiness of the workload is stationary but the λ has

increased, this will be a rate with a larger ΔT_i. In the case that multiple rates have exceeded their sample trace counterpart, we take the max rate.

To determine how to reprovision the pipeline, the Tuner computes the number of replicas needed for each model to process r_{max} as $k_m = \left\lceil \frac{r_{max} s_m}{\mu_m \rho_m} \right\rceil$. s_m is the scale factor for model m, which prevents over-provisioning for a model that only receives a portion of the queries due to conditional logic. ρ_m is the max-provisioning ratio, which ensures enough slack remains in the model to handle bursts. The Tuner then adds the additional replicas needed for any models that are detected to be underprovisioned.

Scaling Down InferLine takes a conservative approach to scaling down the pipeline to prevent unnecessary configuration oscillation which can cause SLO misses. Drawing on the work in [57], the Tuner waits for a period of time after any configuration changes to allow the system to stabilize before considering any down scaling actions. InferLine uses a delay of 15 seconds (3x the 5 second activation time of spinning up new replicas in the underlying prediction serving frameworks), but the precise value is unimportant as long as it provides enough time for the pipeline to stabilize after a scaling action. Once this delay has elapsed, the Tuner computes the max request rate λ_{new} that has been observed over the last 30 seconds, using 5 second windows.

The Tuner computes the number of replicas needed for each model to process λ_{new} similarly to the procedure for scaling up, setting $k_m = \left\lceil \frac{\lambda_{new} s_m}{\mu_m \rho_p} \right\rceil$. In contrast to scaling up, when scaling down we use the minimum max provisioning factor in the pipeline $\rho_p = \min(\rho_m \forall m \in \text{models})$. Because the max provisioning factor is a heuristic that has some dependence on the sample trace, using the min across the pipeline provides a more conservative downscaling algorithm and ensures the Tuner is not overly aggressive in removing replicas. If the workload has dropped substantially, the next time the Planner runs it will find a new lower-cost configuration that is optimal for the new workload.

4.6 Experimental Setup

To evaluate InferLine we constructed four prediction pipelines (Figure 4.2) representing common application domains and using models trained in a variety of machine learning frameworks [117, 145, 120, 115]. We configure each pipeline with varying input arrival processes and latency budgets. We evaluate the latency SLO attainment and pipeline cost under a range of both synthetic and real world workload traces.

Coarse-Grained Baseline Comparison Current prediction serving systems do not provide functionality for provisioning and managing prediction pipelines with end-to-end latency constraints. Instead, the individual pipeline components are each deployed as a separate microservice to a prediction serving system such as [147, 7, 41, 146] and a pipeline is manually constructed by individual calls to each service.

Any performance tuning for end-to-end latency or cost treats the entire pipeline as a single black-box service and tunes it as a whole. We therefore use this same approach as our baseline

for comparison. Throughout the experimental evaluation we refer to this as the *Coarse-Grained* baseline. We deploy pipelines configured with both InferLine and the coarse-grained baseline to the same underlying prediction-serving franework. All experiments used Clipper [41] as the prediction-serving framework except for those in Figure 4.13 which compare InferLine running on Clipper and TensorFlow Serving [147]. Both prediction-serving frameworks were modified to add a centralized batched queueing system.

We use the techniques proposed in [57] to do both low-frequency planning and high-frequency tuning for the coarse-grained pipelines as a baseline for comparison. In this baseline, we profile the entire pipeline as a single black box to identify the single maximum batch size capable of meeting the SLO, in contrast to InferLine's per-model profiling. The pipeline is then replicated as a single unit to achieve the required throughput as measured on the same sample arrival trace used by the **Planner**. We evaluate two strategies for determining required throughput. *CG-Mean* uses the mean request rate computed over the arrival trace while *CG-Peak* determines the peak request rate in the trace computed using a sliding window of size equal to the SLO. The coarse-grained tuning mechanism scales the number of pipeline replicas using the scaling algorithm introduced in [57].

Physical Execution Environment We ran all experiments in a distributed cluster on Amazon EC2. The pipeline driver client was deployed on an `m4.16xlarge` instance which has 64 vCPUs, 256 GiB of memory, and 25Gbps networking across two NUMA zones. We used large client instance types to ensure that network bandwidth from the client is not a bottleneck. Models were deployed to a cluster of up to 16 `p2.8xlarge` GPU instances. This instance type has 8 NVIDIA K80 GPUs, 32 vCPUs, 488.0 GiB of memory and 10Gbps networking all within a single NUMA zone. All instances ran Ubuntu 16.04 with Linux Kernel version 4.4.0.

CPU costs were computed by dividing the total hourly cost of an instance by the number of CPUs. GPU costs were computed by taking the difference between a GPU instance and its equivalent non-GPU instance (all other hardware matches), then dividing by the number of GPUs. This cost model provides consistent prices across instance sizes.

Workload Setup We generated synthetic traces by sampling inter-arrival times from a gamma distribution with differing mean μ to vary the request rate, and coefficient of variation CV to vary the workload burstiness. When reporting performance on a specific workload as characterized by $\lambda = \frac{1}{\mu}$ and CV, a trace for that workload was generated once and reused across all comparison points to provide a more direct comparison of performance. We generated separate traces with the same performance characteristics for profiling and evaluation to avoid overfitting to the sample trace.

To generate synthetic time-varying workloads, we evolve the workload generating function between different Gamma distributions over a specified period of time, the transition time. This allows us to generate workloads that vary in mean throughput, CV, or both, and thus evaluate the performance of the **Tuner** under a wide range of conditions.

In Figure 4.6 we evaluate InferLine on traces derived from real workloads studied in the AutoScale system [57]. These workloads only report the average request rate each minute for an

hour, rather than providing the full trace of query inter-arrival times. To derive traces from these workloads, we followed the approach used by [57] to re-scale the max throughput to 300 QPS, the maximum throughput supported by the coarse-grained baseline pipelines on a 16 node (128 GPU) cluster. We then iterated through each of the mean request rates in the workload and sample from a Gamma distribution with CV 1.0 for 30 seconds. We use the first 25% of the trace as the sample for the Planner, and the remaining 75% as the live serving workload (see Figure 4.6).

4.7 Experimental Evaluation

In this section we evaluate InferLine's performance. First, we evaluate end-to-end performance of InferLine relative to current state of the art methods for configuring and provisioning prediction pipelines with end-to-end latency constraints (Section 4.7). We show that InferLine outperforms the baselines on latency SLO attainment and cost for synthetic and real-world derived workloads with both stable and unpredictable workload dynamics. Second, we demonstrate that InferLine is robust to unplanned dynamics of the arrival process (Section 4.7): changes in the arrival rate as well as unexpected inter-arrival bursts, as the **Tuner** rapidly re-scales the pipeline in response to these changes. Third, we perform an ablation study to show that the system benefits from both the low-frequency planning and high-frequency tuning. We conclude by showing that InferLine composes with multiple underlying prediction-serving frameworks (Section 4.7).

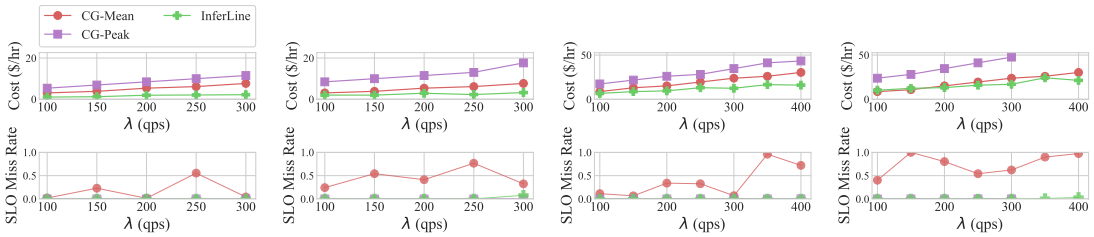
End-to-end Evaluation

We first establish that InferLine's planning and tuning components outperform state-of-the-art pipeline-level configuration alternatives in an end-to-end evaluation (Section 4.7). InferLine is able to achieve the same throughput at significantly lower cost, while maintaining zero or near-zero latency SLO miss rate.

Low-Frequency Planning In the absence of a workload-aware planner (Section 4.4), the options are limited to either (a) provisioning for the peak (CG Peak), or (b) provisioning for the mean (CG Mean) request rate. We compare InferLine to these two end-points of the configuration continuum across 2 pipelines (Figure 4.5). InferLine meets latency SLOs at the lowest cost. CG Peak meets SLOs, but at much higher cost, particularly for burstier workloads. And CG Mean is not provisioned to handle arrival bursts which results in high SLO miss rates.

The **Planner** consistently finds lower cost configurations than both coarse-grained provisioning strategies and is able to achieve up to a *7.6x reduction in cost* by minimizing pipeline imbalance. Finally, we observe that the Planner consistently finds configurations that meet the SLO for workloads with the same characteristics as the sample trace used for planning. Next, we evaluate the Tuner's ability to meet SLOs during *unexpected* changes in workload.

High-Frequency Tuning InferLine is able to (1) maintain a negligible SLO miss rate, and (2) and reduce cost by up to 4.2x when compared to the state-of-the-art approach [57] when handling



(a) Image Processing CV 1.0 **(b)** Image Processing CV 4.0 **(c)** Video Monitoring CV 1.0 **(d)** Video Monitoring CV 4.0

Figure 4.5: Comparison of InferLine's Planner to coarse-grained baselines (150ms SLO) InferLine outperforms both baselines, consistently providing both the lowest cost configuration and highest SLO attainment (lowest miss rate). CG-Peak was not evaluated on $\lambda > 300$ because the configurations exceeded cluster capacity.

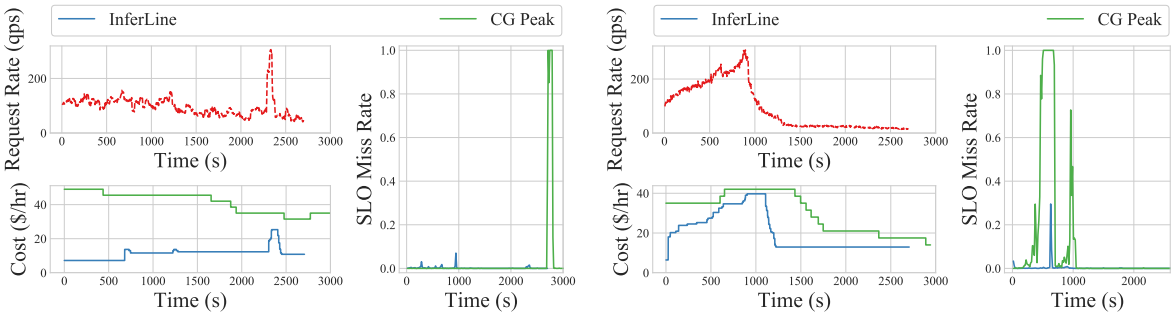
unexpected changes in the arrival rate and burstiness. In Figure 4.6 we evaluate the *Social Media* pipeline on 2 traces derived from real workloads studied in [57]. The **Planner** finds a *5x cheaper* initial configuration than coarse-grained provisioning (Figure 4.6a). Both systems achieve near-zero SLO miss rates throughout most of the workload, and when the big spike occurs we observe that InferLine's **Tuner** quickly reacts by scaling up the pipeline as described in Section 4.5. As soon as the spike dissipates, InferLine scales the pipeline down to maintain a cost-efficient configuration. In contrast, the coarse-grained tuning mechanism operates much slower and, therefore, is ill-suited for reacting to rapid changes in the request rate of the arrival process.

In Figure 4.6b, InferLine scales up the pipeline smoothly and recovers rapidly from an instantaneous spike, unlike the CG baseline. As the workload drops quickly after 1000 seconds, InferLine rapidly responds by shutting down replicas to reduce cluster cost. In the end, InferLine and the coarse-grained pipelines converge to similar costs due to the low terminal request rate which hides the effects of pipeline imbalance, but InferLine has a *34.5x lower SLO miss rate* than the baseline.

We further evaluate the differences between the InferLine and coarse-grained tuning algorithms on a set of synthetic workloads with increasing arrival rates in Figure 4.7. We observe that the traffic envelope monitoring described in Section 4.5 enables InferLine to detect the increase in arrival rate earlier and therefore scale up the pipeline sooner to maintain a low SLO miss rate. In contrast, the coarse-grained baseline only reacts to the increase in request rate at the point when the pipeline is overloaded and therefore reacts when the pipeline is already in an infeasible configuration. The effect of this delayed reaction is compounded by the longer provisioning time needed to replicate an entire pipeline, resulting in the coarse-grained baselines being unable to recover before the experiment ends. They will eventually recover as we see in Figure 4.6 but only after suffering a period of 100% SLO miss rate.

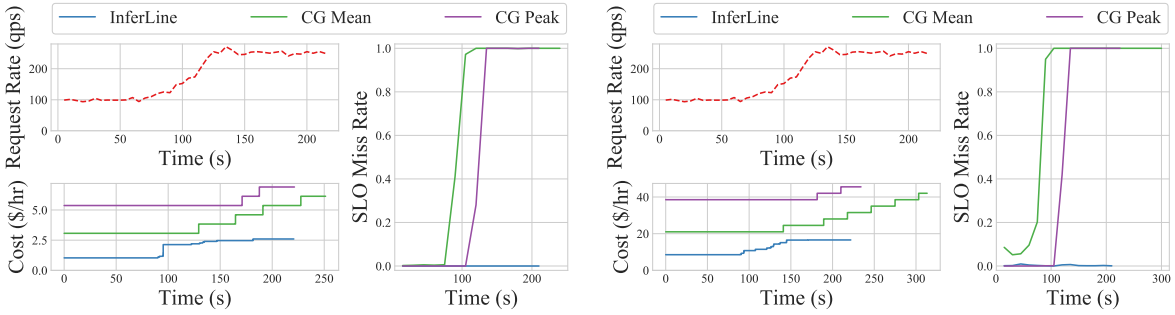
Sensitivity Analysis

We evaluate the sensitivity and robustness of the **Planner** and the **Tuner**. We analyze the accuracy of the **Estimator** in estimating tail latencies from the sample trace and the **Planner**'s response to



(a) AutoScale Big Spike **(b)** AutoScale Slowly Varying

Figure 4.6: Performance comparison of the high-frequency tuning algorithms on traces derived from real workloads. These are the same workloads evaluated in [57] which forms the basis for the coarse-grained baseline. Both workloads were evaluated on the Social Media pipeline with a 150ms SLO. In Figure 4.6a, InferLine maintains a 99.8% SLO attainment overall at a total cost of $8.50, while the coarse-grained baseline has a 93.7% SLO attainment at a cost of $36.30. In Figure 4.6b, InferLine has a 99.3% SLO attainment at a cost of $15.27, while the coarse-grained baseline has a 75.8% SLO attainment at a cost of $24.63, a 34.5x lower SLO miss rate.



(a) Image Processing **(b)** Social Media

Figure 4.7: Performance comparison of the high-frequency tuning algorithms on synthetic traces with increasing arrival rates. We observe that InferLine outperforms both coarse-grained baselines on cost while maintaining a near-zero SLO miss rate for the entire duration of the trace.

varying arrival rates, latency SLOs, and burstiness factors. We also analyze the **Tuner**'s sensitivity to changes in the arrival process and ability to re-scale individual pipeline stages to maintain latency SLOs during these unexpected changes to the workload.

Planner Sensitivity We first evaluate how closely the latency distribution produced by the **Estimator** reflects the latency distribution of the running system in Figure 4.8. We observe that the estimated and measured P99 latencies are close across all four experiments. Further, we see that

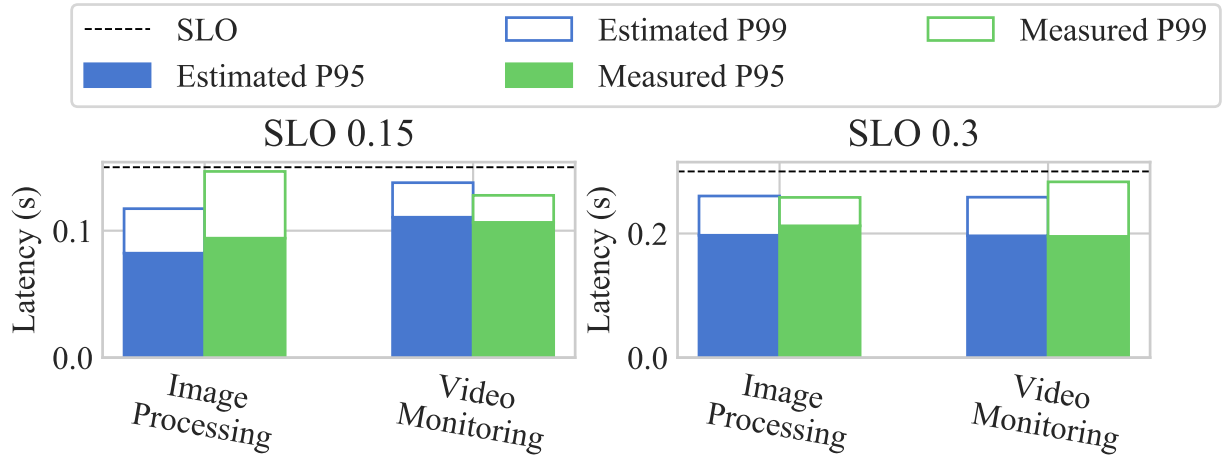


Figure 4.8: Comparison of estimated and measured tail latencies. We compare the latency distributions produced by the Estimator on a workload with λ of 150 qps and CV of 4, observing that in all cases the estimated and measured latencies are both close to each other and below the latency SLO.

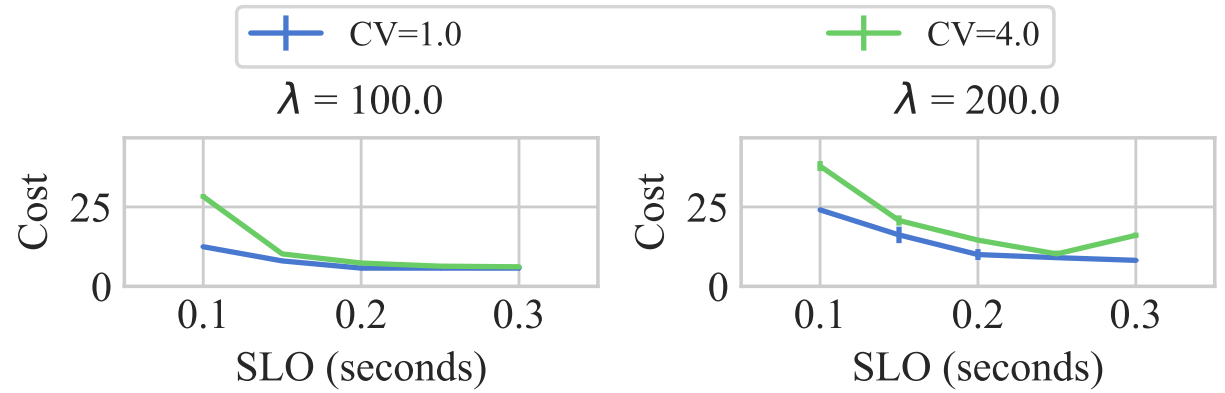


Figure 4.9: Planner sensitivity: Variation in configuration cost across different arrival processes and latency SLOs for the Social Media pipeline. We observe that 1) cost decreases as SLO increases, 2) burstier workloads require higher cost configurations, and 3) cost increases as λ increases.

the Estimator has the critical property of ensuring that the P99 latency of feasible configurations is below the latency objective. The near-zero SLO miss rates in Figure 4.5 are a further demonstration of the Estimator's ability to detect infeasible configurations.

Next, we evaluate the **Planner**'s performance under varying load, burstiness, and end-to-end latency SLOs. We observe three important trends in Figure 4.9. First, increasing burstiness (from CV=1 to CV=4) requires more costly configurations as the Planner provisions more capacity to ensure that transient bursts do not cause the queues to diverge more than the SLO allows. We also see the cost gap narrowing between CV=1 and CV=4 as the SLO increases. As the SLO increases,

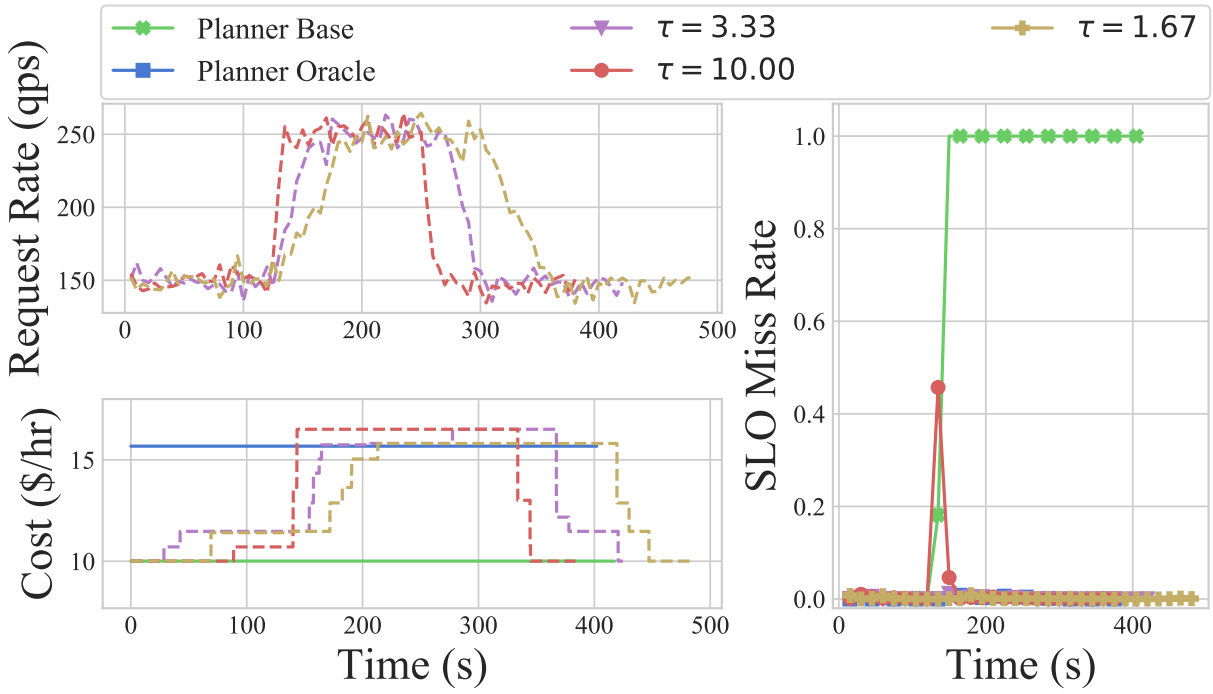


Figure 4.10: Sensitivity to arrival rate changes (Social Media pipeline). We observe that the Tuner quickly detects and scales up the pipeline in response to increases in λ. Further, the Tuner finds cost-efficient configurations that either match or are close to those found by the Planner given full oracle knowledge of the trace.

additional slack in the deadline can absorb more variability in the arrival process and therefore fewer pipeline replicas are needed to process transient bursts within the SLO. Second, the cost decreases as a function of the latency SLO. While this downward cost trend generally holds, the optimizer occasionally finds sub-optimal configurations, as it makes locally optimal decisions to change a resource assignment. Third, the cost increases as a function of expected arrival rate, as more queries require more model replicas.

Tuner Sensitivity A common type of unpredictable behavior is a change in the arrival rate. We compare the behavior of InferLine with and without its Tuner enabled as the arrival rate changes from the planned-for 150 QPS to 250 QPS. We vary the rate of arrival throughput change τ. InferLine is able to maintain the SLO miss rate close to zero while matching or beating two alternatives: (a) a pipeline with only the Planner enabled but given full oracle knowledge of the arrival trace, and (b) a pipeline with only the Planner enabled and provided only the sample planning trace. Neither of these baselines responds to changes in workload during live serving. As we see in Figure 4.10, InferLine continues to meet the SLO, and increases the cost of the pipeline only for the duration of the unexpected burst. The oracle Planner with full knowledge of the workload is able to find the cheapest configuration at the peak because it is equipped with the ability to configure batch

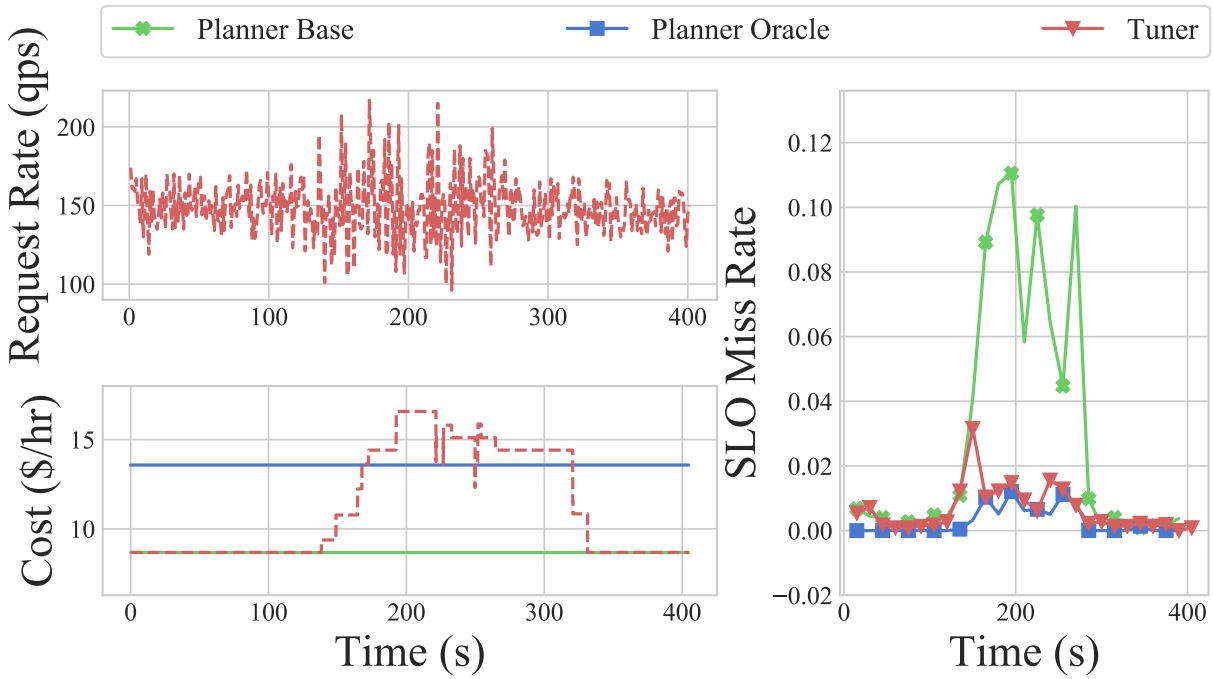


Figure 4.11: Sensitivity to arrival burstiness changes (Social Media Pipeline). We observe that the network-calculus based detection mechanism of the Tuner detects changes in workload burstiness and takes the appropriate scaling action to maintain a near-zero SLO miss rate.

size and hardware type along with replication factor. But it pays this cost for the entire duration of the workload. The Planner without oracular knowledge starts missing latency SLOs as soon as the ingest rate increases as it is unable to respond to unexpected changes in the workload without the Tuner.

A less obvious but potentially debilitating change in the arrival process is an increase in its burstiness, even while maintaining the same mean arrival rate λ. This type of arrival process change is also harder to detect, as the common practice is to look at moments of the arrival rate distribution, such as the mean or 99th percentile. In Figure 4.11 we show that **Tuner** is able to *detect* deviation from expected arrival burstiness and react to meet the latency SLOs by employing the traffic-envelope detection mechanism described in Section 4.5.

Attribution of Benefit

InferLine benefits from (a) low-frequency planning and (b) high-freqency tuning. Thus, we evaluate the following comparison points: baseline coarse grain planning (Baseline Plan), InferLine's planning (InferLine Plan), InferLine planning with baseline tuning (InferLine Plan + Baseline Tune), and InferLine planning with InferLine tuning (InferLine Plan + InferLine Tune), building up from pipeline-level configuration to the full feature set InferLine provides. InferLine's Planner

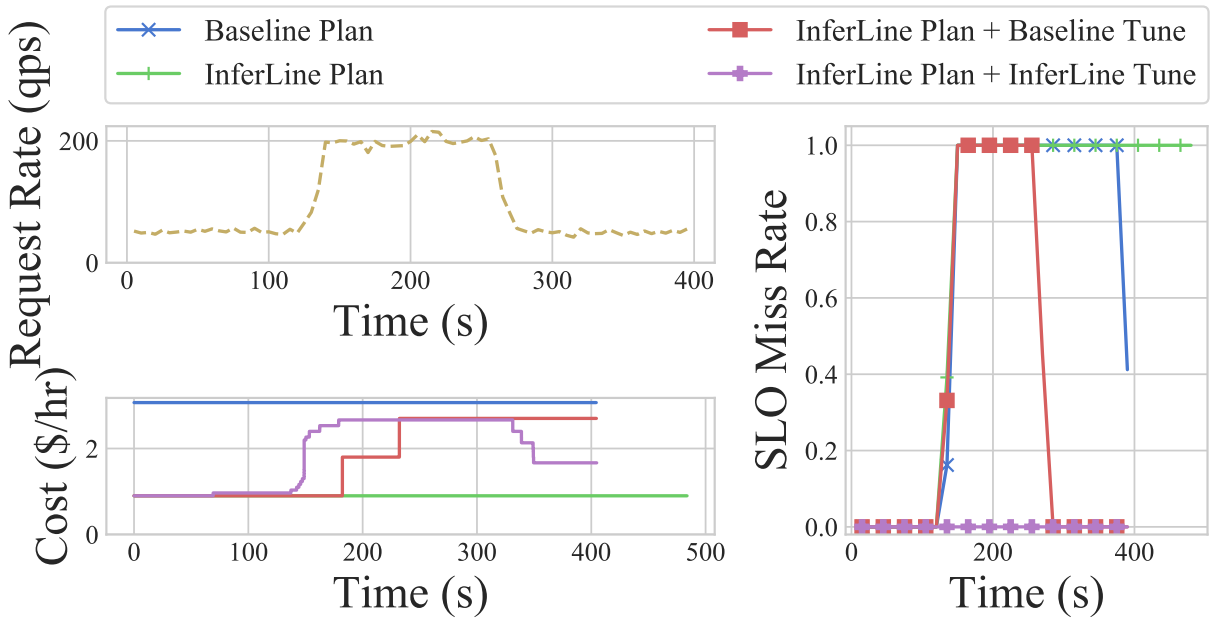


Figure 4.12: Attribution of benefit between the InferLine low-frequency Planner and high-frequency Tuner on the Image Processing pipeline. We observe that the Planner finds a more than 3x cheaper configuration than the baseline. We also observe that InferLine's Tuner is the only alternative that maintains the latency SLO throughout the workload.

reduces the cost of the initial pipeline configuration by more than 3x (Figure 4.12), but starts missing latency SLOs when the request rate increases. Adding the baseline tuning mechanism (InferLine Plan + Baseline Tune) adapts the configuration, but too late to completely avoid SLO misses, although it recovers faster than planning-only alternatives. The InferLine Tuner has the highest SLO attainment and is the only alternative that maintains the SLO across the entirety of the workload. This emphasizes the need for both the Planner for initial cost-efficient pipeline configuration, and the Tuner to promptly and cost-efficiently adapt to unexpected workload changes.

Multiple Prediction-Serving Frameworks

The contributions of this work generalize to different underlying serving frameworks. Here, we evaluate the InferLine **Planner** running on top of both Clipper and TensorFlow Serving (TFS). In this experiment, we achieve the same low latency SLO miss rate for both prediction-serving frameworks. This indicates the generality of the planning algorithms used to configure individual models in InferLine. In Figure 4.13 we show both the SLO attainment rates and the cost of pipeline provisioning when running InferLine on the two serving frameworks. The cost for running on TFS is slightly higher due to some additional RPC serialization overheads not present in Clipper.

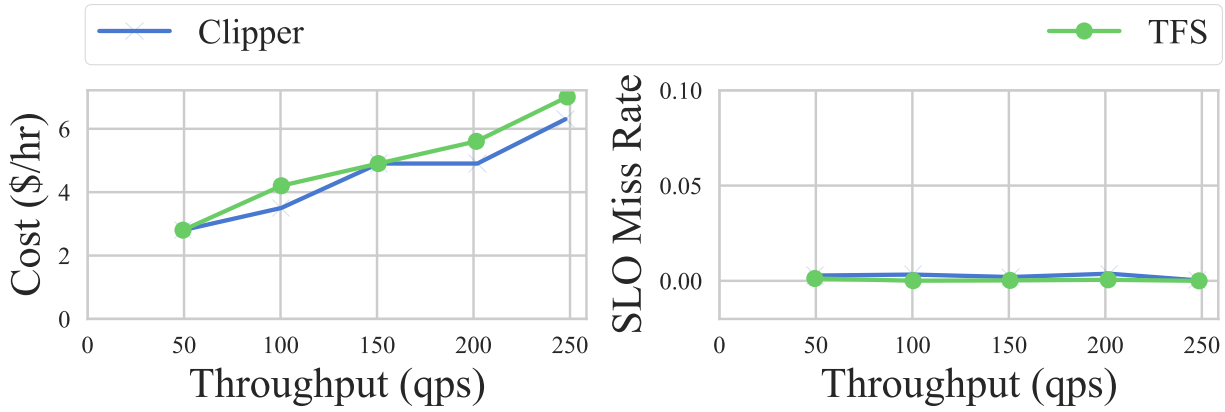


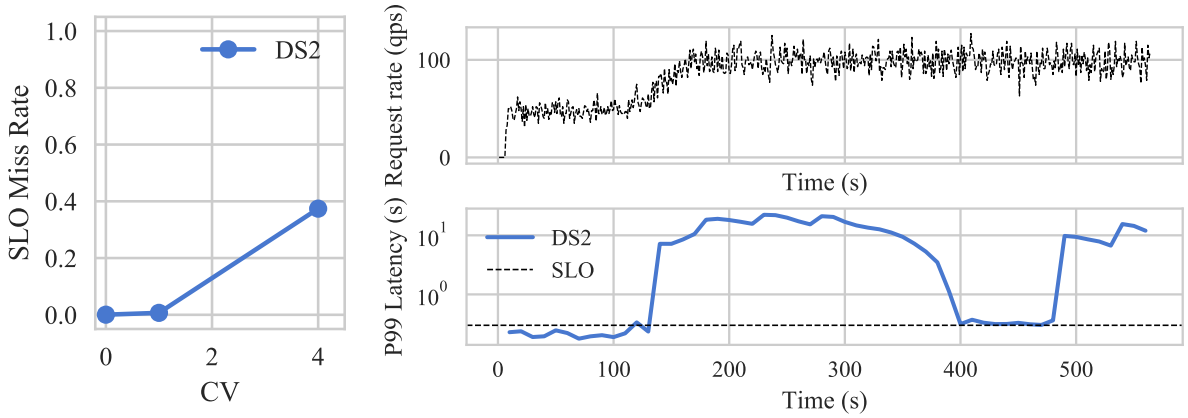
Figure 4.13: Comparison of the InferLine Planner provisioning the TF Cascade pipeline in the Clipper and TensorFlow Serving (TFS) prediction-serving frameworks. The SLO is 0.15 and the CV is 1.0.

4.8 Related Work

A number of recent efforts study the design of generic prediction serving systems [41, 14, 147, 146]. TensorFlow Serving [147] is a commercial grade prediction serving system primarily designed to support prediction pipelines implemented using TensorFlow [145], but does not provide any automatic provisioning or support latency constraints. Clipper adopts a containerized design allowing each model to be individually managed, configured, and deployed in separate containers, but does not support prediction pipelines or reasoning about latency deadlines across models. TensorRT Inference Server [146] from NVIDIA adopts a similar design to TensorFlow Serving and is optimized for NVIDIA GPUs.

Several systems have explored offline pipeline configuration for data pipelines [73, 19]. However, these target generic data streaming pipelines. They use black box optimization techniques that require running the pipeline end-to-end to measure the performance of each candidate configuration. InferLine instead leverages performance profiles of each stage and a simulation-based performance estimator to explore the configuration space without needing to run the pipeline.

Dynamic pipeline scaling is a critical feature in data streaming systems to avoid backpressure and overprovisioning. Systems such as [78, 54] are throughput-oriented with the goal of maintaining a well-provisioned system under changes in the request rate. The DS2 autoscaler in [78] estimates true processing rates for each operator in the pipeline by instrumenting the underlying streaming system. They use these processing rates in conjunction with the pipeline topology structure to estimate the optimal degree of parallelism for all operators at once. In contrast, [54] identifies a single bottleneck stage at a time, taking several steps to converge from an under-provisioned to a well-provisioned system. Both systems provision for the average ingest rate and ignore any burstiness in the workload which can transiently overload the system. In contrast, InferLine maintains a traffic envelope of the request workload and uses this to ensure that the pipeline is well-provisioned for the peak workload across several timescales simultaneously, including any



(a) Increasing CV

(b) Increasing Arrival Rate

Figure 4.14: Performance of DS2 on Bursty and Stochastic Workloads. We observe that DS2 is unable to maintain the latency SLO under (a) bursty workloads, and (b) workloads with increasing request rate.

burstiness (see Section 4.5).

In Figure 4.14 we evaluate the performance of DS2 [78] on its ability to meet latency SLOs under a variety of workloads. We deployed the Image Processing pipeline (Figure 4.2a) in DS2 running on Apache Flink [53] without any batching. As we can see in Figure 4.14a, provisioning for the average request rate is sufficient to meet latency objectives under uniform workloads. But as CV increases to 4.0, the latency SLO miss rate increases as bursts in the request rate transiently overload the system, causing queueing delays until the system recovers. In addition, DS2 occasionally misinterprets transient bursts as changes in the overall request rate and scales up the pipeline, requiring Apache Flink to halt processing and save state before migrating to the new configuration.

We observe this same degradation under non-stationary workloads in Figure 4.14b where we measure P99 latency over time for a workload that starts out with a CV of 1.0 and a request rate of 50 qps, then increases the request rate to 100 qps over 60 seconds. It takes nearly 300 seconds after the request rate increase for the system to re-stabilize and the queues to fully drain from the repeated pipeline re-configurations. In contrast, as we see in Figure 4.10 and Figure 4.11, InferLine is capable of maintaining SLOs under a variety of changes to the workload dynamics.

A few streaming autoscaling systems consider latency-oriented performance goals [55, 97]. The closest work to InferLine, [97] from Lohrmann et al. as part of their work on Nephele [98], treats each stage in a pipeline as a single-server queueing system and uses queueing theory to estimate the total queue waiting time of a job under different degrees of parallelism. They leverage this queueing model to greedily increase the parallelism of the stage with the highest queue waiting time until they can meet the latency SLO. However, their queueing model only considers average latency, and provides no guarantees about the behavior of tail latencies. InferLine's Tuner automatically provisions for worst-case latencies.

VideoStorm [161] explores the design of a streaming video processing system that adopts a distributed design with pipeline operators provisioned across compute nodes and explores the combinatorial search space of hardware and model configurations. VideoStorm jointly optimizes for quality and lag and does not provide latency guarantees.

Nexus [132] is a recent system configures DNN inference pipelines for video-streaming applications. Similar to InferLine, it uses model profiles to understand model batching behavior and provisions pipelines for end-to-end latency objectives. However, they do not configure which hardware to run models on, instead assuming a homogenous cluster of GPUs. Furthermore, they rely on admission control to reject queries during transient spikes while InferLine's Tuner quickly re-scales the pipeline to maintain SLOs without rejecting queries.

A large body of prior work leverages profiling for scheduling, including recent work on workflow-aware scheduling [122, 77]. In contrast, InferLine exploits the compute-intensive and side-effect free nature of ML models to estimate end-to-end pipeline performance based on individual model profiles.

Autoscale [57] comprehensively surveys work aimed at automatically scaling the number of servers reactively, subject to changing load in the context of web services. Autoscale works well for single model replication without batching as it assumes bit-at-a-time instead of batch-at-a-time query processing. However, we find that the InferLine high-frequency Tuner outperforms the coarse-grain baselines using the Autoscale scaling mechanism on both latency SLO attainment and cost (Section 4.7).

4.9 Limitations and Generality

One limitation of the Planner is its assumption that the available hardware has a total ordering of latency across all batch sizes. As specialized accelerators for ML continue to proliferate, there may be settings where one accelerator is slower than another at smaller batch sizes but faster at larger batch sizes. This would require modifications to the hardware downgrade portion of the planning algorithm to account for this batch-size dependent ordering.

A second limitation is the assumption that the inference latency of ML models is independent of their input. There are emerging classes of machine learning tasks where state-of-the-art models have inference latency that varies based on the input. For example, object detection models [121, 120] will take longer to make predictions on images with many objects in them. One way of modifying InferLine to account for this is to measure this latency distribution during profiling based on the variability in the sample queries and use the tail of the distribution (e.g., 99% or k standard deviations above the mean) as the processing time in the estimator, which will lead to feasible but more costly configurations.

Finally, while we only study machine learning prediction pipelines in this work, there may be other classes of applications that have similarly predictable performance and can therefore be profiled. We leave the extension of InferLine to applications beyond machine learning as future work.

4.10 Conclusion

In this paper we studied the problem of provisioning and managing prediction pipelines to meet end-to-end tail latency requirements at low cost and across heterogeneous parallel hardware. We introduced InferLine–a system which efficiently provisions prediction pipelines subject to end-to-end latency constraints. InferLine combines a low-frequency Planner that finds cost-optimal configurations with a high-frequency Tuner that rapidly re-scales pipelines to meet latency SLOs in response to changes in the query workload. The low-frequency Planner combines profiling, discrete event simulation, and constrained combinatorial optimization to find the cost minimizing configuration that meets the end-to-end tail latency requirements without ever instatiating the system (Section 4.4). The high-frequence Tuner uses network-calculus to quickly auto-scale each stage of the pipeline to accommodate changes in the query workload (Section 4.5). In combination, these components achieve the combined effect of *cost-efficient* heterogeneous prediction pipeline provisioning that can be deployed to a variety of prediction-serving frameworks to serve applications with a range of tight end-to-end latency objectives. As a result, we achieve up to 7.6x improvement in cost and 34.5x improvement in SLO attainment for the same throughput and latency objectives over state-of-the-art provisioning alternatives.

Chapter 5

Discussion and Future Directions

This work explored the use of a decoupled architecture to build general-purpose systems for serving modern machine learning applications.

In Velox (Chapter 2), we introduced the split model decomposition, which formed the foundation for the work on model composition and adaptability throughput this thesis. In Velox, we decomposed models into components that changed – and therefore needed to be learned – at different rates and granularities. We lifted the parts of the models that needed to be retrained quickly into the online serving system by introducing interfaces for online learning and personalization using lightweight linear models. These lightweight models are then composed with more powerful but slower changing feature models that must be retrained offline in batch processing systems, but are necessary for achieving the prediction power needed for modern machine learning applications.

However, Velox had a few limitations. It was tightly integrated with the Berkeley Data Analytics Stack. This tight integration had proved valuable for all the data processing components of the stack, but ultimately was too limiting for a prediction serving system. Velox could only serve Apache Spark models, yet data scientists were using a wide variety of machine learning frameworks (Section 1.3) to train their models, not just Spark. Furthermore, the online learning in Velox was limited to simple linear regression models and could not support other techniques such as A/B testing or contextual bandit methods. And finally, while Velox was architected for high performance serving, it had no explicit mechanisms to control for tail latency or meet specific application SLOs.

In Clipper (Chapter 3), we addressed several of Velox's limitations through a decoupled system architecture inspired by the Velox split model. Clipper's first design goal was the extension of the Velox split model – with the partial online learning and personalization that it enabled – to arbitrary models trained in any machine learning framework. We accomplished this by the introducing a framework- and programming language-agnostic prediction interface to abstract away the heterogeneity inherent in the model development process. This narrow waist system architecture decoupled the model implementation from the serving system functionality. As a result, Clipper extended the Velox split model design in two ways. First, we introduced a more generic interface for online learning and personalization that could support both the simple linear regression models of Velox, as well as bandit algorithms and any other stateful online learning above the prediction interface. Second, Clipper was able to serve not only Spark models but arbitrary models

trained in any framework. Furthermore, Clipper introduced general-purpose techniques to bound prediction tail latency to meet application-specific SLOs and maximize compute resource utilization while treating the underlying models as black boxes. These techniques then helped ease the model deployment process for data scientists. The system would automatically take care of optimizing inference performance, rather than the data scientists needing to do it themselves manually before deployment.

After getting positive industrial feedback from the initial research prototype of Clipper, we decided to develop the prototype into a production-ready open-source serving system [37]. This involved significant implementation effort but was well-received, resulting in several companies using Clipper in production. We also learned a lot about the practical challenges involved in deploying machine learning to production from interactions with both current and potential users of Clipper. In particular, there were two pieces of feedback we heard repeatedly. The first was that users often had prediction pipelines they wanted to serve. These pipelines used several different models whose predictions they needed to combine in some way to make a final decision, such as feeding the output of one model as an input to another. Clipper could serve each of the models involved in the decision but would treat them as independent applications rather than a single decision pipeline with end-to-end latency constraints. The second piece of feedback was that users had no idea how to control the cost of serving their machine learning applications while still ensuring they were properly provisioned to meet workload demands and latency SLOs. Often, they weren't even sure what type of compute hardware to run on, and would either be deploying models on GPUs but getting very low GPU utilization, or would think that GPUs were only useful for training and be unable to meet latency SLOs. While Clipper supported deploying models to heterogeneous hardware and scaling up applications, it relied on users for making the decisions about which hardware to use and how to scale.

We developed InferLine (Chapter 4) to help users navigate the complex cost-latency-throughput tradeoff space for provisioning and serving machine learning inference pipelines. InferLine is a generic system for configuring and auto-scaling heterogenous machine learning applications subject to end-to-end latency constraints. One of the key challenges in InferLine was how to navigate the large configuration space, one that grows exponentially in both the size of the pipeline and the kinds of available hardware (see Section 1.3). Once again, we leveraged a decoupled and layered architecture to address this challenge. First, InferLine relied on the Clipper prediction interface to abstract away the implementation details of a model and serve pipelines composing models running in different frameworks and programming languages. In addition, one of InferLine's key enabling observations was that heterogeneous models can be empirically profiled as black boxes to understand their performance characteristics – specifically latency and throughput – under different system configurations including heterogeneous hardware accelerators. Each model could be profiled independently and the performance profiles composed in simulation to accurately estimate end-to-end pipeline latencies without ever running the pipeline in that configuration. These performance profiles enabled the InferLine configuration optimizer to be fully decoupled from the underlying serving system. As a result, InferLine can be applied to configure and autoscale any prediction serving system as long as it can provide accurate performance profiles.

5.1 Future Directions

While the systems in this thesis provide a substantial step forward in the state of the art of prediction serving, there is still much work to be done. Next, I discuss three research directions that build on this work.

Automatically Navigating the Performance-Accuracy Tradeoff

Each system I have described explores some aspect of the accuracy-latency-throughput-cost tradeoff space. For example, Clipper's partial prediction latency bounding technique trades prediction accuracy for latency, while InferLine's cost-minimizing optimizer finds the lowest cost configuration for a specific workload and latency constraint. However, none of these systems attempt to navigate the full 4-dimensional tradeoff space. Yet model developers are constantly making performance/accuracy tradeoff decisions when developing models. If these performance-accuracy decisions could instead be deferred to prediction time and made on a per-prediction basis, machine learning applications could use spare compute resources to improve prediction accuracy during non-peak load while degrading gracefully rather than becomining unavailable or missing SLOs during unexpected load spikes. Furthermore, prediction accuracy – which often has a measurable monetary value – could be more directly balanced against the measurable resource cost of evaluating a prediction and the monetary cost of missing an SLA to ensure the machine learning application is profitable.

Exploring this space requires models with a tunable accuracy-latency configuration knob. There has already been some work in this area. For example, anytime algorithms [65, 64] quickly provide an initial prediction then use any remaining time and computational resources to refine this prediction. Recent work on Neural ODEs [30] indicates the possibility for infinitely deep neural networks in the future, where the number of layers (and therefore the latency and accuracy) could be decided dynamically at prediction time. And there are kernel methods with dynamically tunable kernel sizes. However, these methods currently only work for specific kinds of models or are still highly speculative. And further analysis techniques need to be developed to reason about how accuracy changes as a function of computational resources in order for a system to automatically make these tradeoff decisions.

To further extend the generality of these ideas, one could imagine constructing a kind of type system or annotation system for models, where any model with the appropriate type could be used to make a prediction. This would allow for generic, or possibly even declarative, prediction serving pipelines where the system could automatically decide which specific model to use to render a prediction. While a few general-purpose models already exist, they are limited to very specific applications such as object detection or machine translation. And even these models typically get fine-tuned when being used in new settings or domains. But as machine learning continues to be more mainstream better techniques will be developed for characterizing and explaining model performance and understanding the impact on accuracy of substituting one model for another.

Generalizing Beyond Machine Learning

The adoption of a decoupled systems architecture can be extended beyond just prediction serving. While Clipper and InferLine leverage some of the performance characteristics known to hold for many machine learning models, the serving systems themselves are low-latency function serving systems for computationally expensive functions. Both systems rely on the functions being stateless, as inference tends to be, to enable fast horizontal scaling. InferLine further relies on the function having predictable performance characteristics that can be profiled. But neither of these properties is necessarily unique to machine learning inference, and they may be able to be adapted to a much larger class of microservices. Understanding what other types of applications the InferLine profiling and configuration optimization can be applied to presents the opportunity for better automatic management of micro-service based serving applications with end-to-end latency constraints. These applications may have different configuration knobs that must be tuned, and will require further study to understand how to extend InferLine to incorporate them. Even the bandit methods explored in Clipper are used to optimize many kinds of decision systems and are commonly used to evaluate user interface and user experience design decisions, and could be easily adapted to optimize rewards beyond prediction accuracy. This work is aligned with the recent interest in serverless computing [75, 69] or Functions-as-a-Service, and could be used to enable serverless systems for low-latency serving, a current pain point with these systems.

Increasing Efficiency for Next Generation ML

Finally, as machine learning becomes more ubiquitous, the amount and scale of machine learning applications will continue to increase by orders of magnitude. Always-on applications will continue to become more and more intelligent, using better models more frequently. As a result, these applications will need to have much higher cost efficiency and power efficiency than the current state of the art.

Addressing these challenges will require innovations at all levels of the stack, from hardware to systems to application design to models. One central element of these improved designs will be doing the bare minimum amount of work possible to solve the problem. For example, recent work on model cascades [152, 80] seeks to use the least accurate (and therefore cheapest) model needed to make an accurate prediction on a per-prediction basis.

In machine learning applications today, system efficiency is often sacrificed for ease and speed of development. The result is a proliferation of machine learning applications, few of which efficiently use the resources available. This includes everything from using over-powered models for simple tasks to sending all inputs to a centralized data center to render predictions. In order to achieve the order-of-magnitude increases in efficiency needed for the next-generation of applications, each of these design decisions will have to be carefully analyzed and reconsidered. And it is the role of system designers to help users analyze and build these applications efficiently. This means building new and more flexible serving systems that can serve next generation machine learning models, run on new specialized hardware, and be more intelligent about how and where computation runs.

5.2 Challenges and Lessons Learned

All research comes with its own challenges, and studying ML serving systems was no exception.

The first challenge we had to overcome was convincing the research community that ML serving was even a systems problem in the first place. When we began work on Velox in 2014, the rise of ML was still in its early days. TensorFlow did not exist (at least not publicly), AlexNet was only two years old, and all the work on systems for ML was focused on training models on increasingly larger datasets. Many of our early conversations with other researchers or even people in industry consisted of us convincing them that ML inference and serving was non-trivial.

This extended into our publishing efforts as well. When writing a paper that solves an existing problem better (i.e. faster, more efficiently, simpler), you can rely on an existing body of literature that describes the problem, and an existing set of systems to benchmark against. In some cases there may even be widely adopted benchmarks that you can use to evaluate your work, such as the TPC benchmarks [149] or the ImageNet [124] competition. In contrast, when writing a paper – particularly a systems paper – that solves a problem that no one else has, you must first convince the reader that the problem is real and describe the nature of it, and then convince them that your solution is novel and solves the problem better than existing approaches. As a result, we had to be creative in coming up with our own benchmarks and points of comparison. As an example, there are no publicly available ML serving traces that include both the request timestamps (to reflect the query arrival process) and the model inputs, much less the models themselves. We therefore had to craft our own ML serving benchmarks using widely adopted models, sample inputs from benchmark training datasets, and query traces from other serving workloads. While this made publishing papers more challenging, it also forced us to deeply understand the problems we worked on and why they were important, which I think can get lost when working in more trendy areas.

One tactic that helped to better understand the problems involved in ML serving was to get out of the lab and talk to industry practitioners. I was lucky to have the semi-annual AMPLab and RISELab retreats to provide both formal and informal venues for these conversations. The lab retreats are full of people with PhDs that are currently working in industry at the largest scales. They understand both research and the limitations inherent in deploying real software in production. Furthermore, as lab sponsors they are deeply invested in the lab working on big and important problems. Having conversations and debates about my research, particularly when it was in its early stages but more than just an idea on a whiteboard, helped steer the research towards solving both more important and more interesting problems.

At the same time, I think that there is a common pitfall that grad students should be aware of when discussing their work with industry practitioners. Industry feedback can be useful for better understanding the contours of a problem. However (and this depends on the nature of the research you want to do as well), getting the feedback that people in industry do not care about your problem does not mean that you should not work on it. Academic research, particularly systems research, should be working on problems farther out on the horizon than those facing industry. In these situations, it can be useful to distinguish between feedback that your problem *will never matter*, and feedback that your problem *is not a pain point right now*. For example, in the first few years of studying ML serving systems, no one in industry cared that much because they were all still

struggling to get data and train models. They had not reached the point where serving was painful, but we knew that they would reach that point eventually. And now ML inference and serving is a huge pain point for production ML.

The last set of lessons I want to discuss are those I learned from developing a research prototype into an active open-source serving system for use in production. The UC Berkeley Computer Science department has a long history of developing open-source software stretching all the way back to the Berkeley Software Distribution (BSD). More recently, Spark [11], Mesos [10], and Caffe [26] are examples of extremely successful and widely adopted open-source systems that have started as research projects in the UC Berkeley CS department. Clearly, open-source systems that start as academic research can be very successful and have a lot of impact. However, there are some things that I wish I had known or considered before making the decision to devote a large chunk of time to developing Clipper into an open-source project.

The first is that Clipper as a serving system needed to be fairly robust and mature before it being considered for use in production, as its primary value was for serving predictions to user-facing applications. In contrast, data analysis and model training systems can afford to be used in production while being less mature, as systems can be restarted in case of unexpected failures without impacting customers. Another early advantage that systems like Spark and Caffe had was finding good early users within the university. One of Spark's first users was a fellow student in the Berkeley CS department who had some large scale data analysis use cases. And Caffe was initially developed by the UC Berkeley Computer Vision group to facilitate their own research. PowerGraph [61] is another example of a system initially developed to facilitate the groups own research, in that case studying graphical models. Having other researchers be first users provides a tight loop for early feedback and helps develop some initial advocates for the system all without the need to leave campus. However, there are not as many uses for serving systems within a university research environment, particularly not the type of high-performance novel serving systems that make for good research.

Another thing that I think is important is to be clear upfront about the goals and success criteria of an open-source effort, and how those goals fit into the goals for your PhD. Creating an open-source project is an extremely labor-intensive and long process. It can take several years for even a very successful open-source project like Spark to build enough momentum to be self-sustaining. It requires a significant amount of engineering time, but you also need to spend time writing documentation, responding to questions and bug requests, and providing extremely hands-on support to your first few users. Doing all this while also trying to write systems research papers is nearly impossible, and there will come times when you need to prioritize one to the detriment of the other. On the other hand, if you have entrepreneurial interests and are towards the end of your PhD, grad school can be a great place to incubate this work and develop some momementum (or experiment with different ideas until you find one that works), before starting to raise money and deal with investors and customers.

Bibliography

[1] Martın Abadi et al. “TensorFlow: Large-scale machine learning on heterogeneous systems, 2015”. In: *Software available from* tensorflow.org ().

[2] Alekh Agarwal et al. “A Multiworld Testing Decision Service”. In: *arXiv preprint arXiv:1606.03966* (2016).

[3] Deepak Agarwal et al. “LASER: A Scalable Response Prediction Platform for Online Advertising”. In: *WSDM*. 2014, pp. 173–182.

[4] Sharad Agarwal and Jacob R Lorch. “Matchmaking for online games and other latency-sensitive P2P systems”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 39. 4. ACM. 2009, pp. 315–326.

[5] Amr Ahmed et al. “Scalable inference in latent variable models”. In: *WSDM*. 2012, pp. 123–132.

[6] Mert Akdere et al. “The Case for Predictive Database Systems: Opportunities and Challenges”. In: *CIDR*. 2011.

[7] *Amazon SageMaker: Developer Guide*. http://docs.aws.amazon.com/sagemaker/latest/dg/sagemaker-dg.pdf.

[8] Jacob Andreas et al. “Deep Compositional Question Answering with Neural Module Networks”. In: *CoRR* abs/1511.02799 (2015). arXiv: 1511.02799. URL: http://arxiv.org/abs/1511.02799.

[9] Anelia Angelova et al. “Real-Time Pedestrian Detection with Deep Network Cascades.” In: *BMVC* (2015), pp. 32.1–32.12.

[10] *Apache Mesos*. http://mesos.apache.org/.

[11] *Apache Spark*. https://spark.apache.org/.

[12] Peter Auer et al. “The Nonstochastic Multiarmed Bandit Problem”. In: *SIAM J. Comput.* 32.1 (Jan. 2003), pp. 48–77. ISSN: 0097-5397. DOI: 10.1137/S0097539701398375. URL: http://dx.doi.org/10.1137/S0097539701398375.

[13] Shivnath Babu and Herodotos Herodotou. “Massively Parallel Databases and MapReduce Systems”. In: *Foundations and Trends in Databases* 5.1 (2013).

[14] Denis Baylor et al. “TFX: A TensorFlow-Based Production-Scale Machine Learning Platform”. In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’17. ACM, 2017, pp. 1387–1395.

[15] Andrew Beck. “Simulation: the practice of model development and use”. In: *Journal of Simulation* 2.1 (Mar. 2008), pp. 67–67. ISSN: 1747-7786. DOI: 10.1057/palgrave.jos.4250031. URL: https://doi.org/10.1057/palgrave.jos.4250031.

[16] Yoshua Bengio. “Learning Deep Architectures for AI”. In: *Found. Trends Mach. Learn.* 2.1 (Jan. 2009), pp. 1–127. ISSN: 1935-8237. DOI: 10.1561/2200000006. URL: http://dx.doi.org/10.1561/2200000006.

[17] James Bergstra et al. “Theano: a CPU and GPU math expression compiler”. In: *Proceedings of the Python for scientific computing conference (SciPy)*. Vol. 4. Austin, TX. 2010, p. 3.

[18] Philip A Bernstein. “Applying Model Management to Classical Meta Data Problems.” In: *CIDR*. 2003.

[19] Muhammad Bilal and Marco Canini. “Towards Automatic Parameter Tuning of Stream Processing Systems”. In: *Proceedings of the 2017 Symposium on Cloud Computing*. SoCC ’17. Santa Clara, California: ACM, 2017, pp. 189–200. ISBN: 978-1-4503-5028-0. DOI: 10.1145/3127479.3127492. URL: http://doi.acm.org/10.1145/3127479.3127492.

[20] C. M. Bishop. *Pattern Recognition and Machine Learning*. Secaucus, NJ, USA: Springer-Verlag, 2006.

[21] *BLAS (Basic Linear Algebra Subprograms)*. http://www.netlib.org/blas/.

[22] Leo Breiman. “Bagging Predictors”. In: *Mach. Learn.* 24.2 (Aug. 1996), pp. 123–140. ISSN: 0885-6125.

[23] Leo Breiman et al. *Classification and Regression Trees*. Statistics/Probability Series. Belmont, California, U.S.A.: Wadsworth Publishing Company, 1984.

[24] Jake Brutlag. *Speed Matters*. https://ai.googleblog.com/2009/06/speed-matters.html. 2009.

[25] Christopher J. C. Burges. “A Tutorial on Support Vector Machines for Pattern Recognition”. In: *Data Min. Knowl. Discov.* 2.2 (June 1998), pp. 121–167. ISSN: 1384-5810. DOI: 10.1023/A:1009715923555. URL: http://dx.doi.org/10.1023/A:1009715923555.

[26] *Caffe*. https://caffe.berkeleyvision.org/.

[27] Eleftherios Charalambous et al. *AI in production: A game changer for manufacturers with heavy assets*. https://www.mckinsey.com/business-functions/mckinsey-analytics/our-insights/ai-in-production-a-game-changer-for-manufacturers-with-heavy-assets. 2019.

[28] Ciprian Chelba et al. “Large scale language modeling in automatic speech recognition”. In: *arXiv preprint arXiv:1210.8440* (2012).

[29] Jianmin Chen et al. “Revisiting Distributed Synchronous SGD”. In: *arXiv.org* (Apr. 2016). arXiv: 1604.00981v1 [cs.LG].

[30] Ricky T. Q. Chen et al. “Neural Ordinary Differential Equations”. In: *Proceedings of the 32Nd International Conference on Neural Information Processing Systems*. NIPS’18. Montrèal, Canada: Curran Associates Inc., 2018, pp. 6572–6583. URL: http://dl.acm.org/citation.cfm?id=3327757.3327764.

[31] Tianqi Chen and Carlos Guestrin. “XGBoost: A Scalable Tree Boosting System”. In: *arXiv.org* (Mar. 2016). arXiv: 1603.02754v1 [cs.LG].

[32] Tianqi Chen et al. “MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems”. In: *arXiv preprint arXiv:1512.01274* (2015).

[33] Trishul Chilimbi et al. “Project adam: Building an efficient and scalable deep learning training system”. In: *OSDI*. 2014, pp. 571–582.

[34] Rada Chirkova and Jun Yang. “Materialized Views”. In: *Foundations and Trends in Databases* 4.4 (2012), pp. 295–405. ISSN: 1931-7883. DOI: 10.1561/1900000020. URL: http://dx.doi.org/10.1561/1900000020.

[35] Dah-Ming Chiu and Raj Jain. “Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks”. In: *Comput. Netw. ISDN Syst.* 17.1 (June 1989), pp. 1–14. ISSN: 0169-7552. DOI: 10.1016/0169-7552(89)90019-6. URL: http://dx.doi.org/10.1016/0169-7552(89)90019-6.

[36] Eric Chung et al. “Serving DNNs in Real Time at Datacenter Scale with Project Brainwave”. In: *IEEE Micro* 38 (Mar. 2018), pp. 8–20. URL: https://www.microsoft.com/en-us/research/publication/serving-dnns-real-time-datacenter-scale-project-brainwave/.

[37] *Clipper*. http://clipper.ai/.

[38] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. “Torch7: A matlab-like environment for machine learning”. In: *BigLearn, NIPS Workshop*. EPFL-CONF-192376. 2011.

[39] F. J. Corbato. “A Paging Experiment With the MULTICS System”. In: (1968).

[40] *Microsoft Cortana*. https://www.microsoft.com/en-us/mobile/experiences/cortana/.

[41] Daniel Crankshaw et al. “Clipper: A Low-Latency Online Prediction Serving System”. In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 613–627. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw.

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

[43] *NVIDIA cuBLAS*. https://developer.nvidia.com/cublas.

[44] *NVIDIA cuDNN*. https://developer.nvidia.com/cudnn.

[45] James Davidson et al. "The YouTube video recommendation system." In: *RecSys* (2010), pp. 293–296.

[46] Jeffrey Dean et al. "Large Scale Distributed Deep Networks". In: *NIPS*. 2012, pp. 1223–1231. URL: http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf.

[47] Amol Deshpande and Samuel Madden. "MauveDB: Supporting Model-based User Views in Database Systems". In: *SIGMOD*. 2006.

[48] *Docker*. https://www.docker.com/.

[49] Jeff Donahue. *CaffeNet*. https://github.com/BVLC/caffe/tree/master/models/bvlc_reference_caffenet.

[50] Kevin Lee, Vijay Rao, and William Christie Arnold. *Accelerating Facebook's Infrastructure With Application-Specific Hardware*. https://code.fb.com/data-center-engineering/accelerating-infrastructure/.

[51] Whitney Zhao. *Sharing a Common Form Factor for Accelerator Modules*. https://code.fb.com/data-center-engineering/accelerator-modules/.

[52] Xixuan Feng et al. "Towards a Unified Architecture for in-RDBMS Analytics". In: *SIGMOD*. 2012.

[53] *Apache Flink*. https://flink.apache.org.

[54] Avrilia Floratou et al. "Dhalion: Self-regulating Stream Processing in Heron". In: *Proc. VLDB Endow.* 10.12 (Aug. 2017), pp. 1825–1836. ISSN: 2150-8097. DOI: 10.14778/3137765.3137786. URL: https://doi.org/10.14778/3137765.3137786.

[55] Tom Z. J. Fu et al. "DRS: Auto-Scaling for Real-Time Stream Analytics". In: *IEEE/ACM Trans. Netw.* 25.6 (Dec. 2017), pp. 3338–3352. ISSN: 1063-6692. DOI: 10.1109/TNET.2017.2741969. URL: https://doi.org/10.1109/TNET.2017.2741969.

[56] João Gama et al. "A Survey on Concept Drift Adaptation". In: *ACM Comput. Surv.* 46.4 (Mar. 2014), 44:1–44:37. ISSN: 0360-0300. DOI: 10.1145/2523813. URL: http://doi.acm.org/10.1145/2523813.

[57] Anshul Gandhi et al. "AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers". In: *ACM Trans. Comput. Syst.* 30.4 (Nov. 2012), 14:1–14:26. ISSN: 0734-2071. DOI: 10.1145/2382553.2382556. URL: http://doi.acm.org/10.1145/2382553.2382556.

[58] Aditya Ganjam et al. "C3: Internet-Scale Control Plane for Video Quality Optimization." In: *NSDI '15* (2015), pp. 131–144.

[59] J. S. Garofolo et al. *DARPA TIMIT Acoustic Phonetic Continuous Speech Corpus CDROM*. 1993.

[60] Joseph E. Gonzalez. "Special Issue on Machine Learning Life-cycle Management". In: *IEEE Data Engineering Bulletin* 41.4 (2018), pp. 4–5. URL: http://sites.computer.org/debull/A18dec/issue1.htm.

[61] Joseph E. Gonzalez et al. "PowerGraph: Distributed Graph-parallel Computation on Natural Graphs". In: OSDI. Hollywood, CA, USA, 2012, pp. 17–30. ISBN: 978-1-931971-96-6.

[62] *Google Now*. https://www.google.com/landing/now/.

[63] Thore Graepel et al. "Web-Scale Bayesian Click-Through rate Prediction for Sponsored Search Advertising in Microsoft's Bing Search Engine." In: *ICML* (2010), pp. 13–20.

[64] Alexander Grubb. "Anytime Prediction: Efficient Ensemble Methods for Any Computational Budget". PhD thesis. May 2014.

[65] Alexander Grubb and J. Andrew (Drew) Bagnell. "SpeedBoost: Anytime Prediction with Uniform Near-Optimality". In: *Fifteenth International Conference on Artificial Intelligence and Statistics*. Apr. 2012.

[66] Jiaqi Guan et al. "Energy-efficient Amortized Inference with Cascaded Deep Classifiers". In: *arXiv.org* (Oct. 2017). arXiv: 1710.03368v1 [cs.LG].

[67] *h20*. http://www.h2o.ai.

[68] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *arXiv preprint arXiv:1512.03385* (2015).

[69] Joseph M. Hellerstein et al. "Serverless Computing: One Step Forward, Two Steps Back". In: *CIDR*. 2019.

[70] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. "Distilling the Knowledge in a Neural Network". In: *arXiv.org* (Mar. 2015). arXiv: 1503.02531v1 [stat.ML].

[71] *AWS Inferentia*. https://aws.amazon.com/machine-learning/inferentia/.

[72] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *arXiv.org* (Feb. 2015). arXiv: 1502.03167v3 [cs.LG].

[73] Pooyan Jamshidi and Giuliano Casale. "An Uncertainty-Aware Approach to Optimal Configuration of Stream Processing Systems." In: *MASCOTS* (2016), pp. 39–48.

[74] Yangqing Jia et al. "Caffe: Convolutional architecture for fast feature embedding". In: *Proceedings of the ACM International Conference on Multimedia*. ACM. 2014, pp. 675–678.

[75] Eric Jonas et al. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. 2019. arXiv: 1902.03383 [cs.OS].

[76] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. 2017, pp. 1–12.

[77] Sangeetha Abdu Jyothi et al. "Morpheus: Towards Automated SLOs for Enterprise Clusters". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Savannah, GA, USA: USENIX Association, 2016, pp. 117–134. ISBN: 978-1-931971-33-1. URL: http://dl.acm.org/citation.cfm?id=3026877.3026887.

[78] Vasiliki Kalavri et al. "Three Steps is All You Need: Fast, Accurate, Automatic Scaling Decisions for Distributed Streaming Dataflows". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'18. Carlsbad, CA, USA: USENIX Association, 2018, pp. 783–798. ISBN: 978-1-931971-47-8. URL: http://dl.acm.org/citation.cfm?id=3291168.3291226.

[79] *Kaldi*. http://kaldi-asr.org/.

[80] Daniel Kang et al. "NoScope: Optimizing Neural Network Queries over Video at Scale". In: *Proc. VLDB Endow.* 10.11 (Aug. 2017), pp. 1586–1597. ISSN: 2150-8097. DOI: 10.14778/3137628.3137664. URL: https://doi.org/10.14778/3137628.3137664.

[81] Meggin Kearney et al. *Measure Performance with the RAIL Model*. https://developers.google.com/web/fundamentals/performance/rail.

[82] Yehuda Koren, Robert Bell, and Chris Volinsky. "Matrix Factorization Techniques for Recommender Systems". In: *IEEE Computer* 42.8 (Aug. 2009), pp. 30–37. ISSN: 0018-9162. DOI: 10.1109/MC.2009.263. URL: http://dx.doi.org/10.1109/MC.2009.263.

[83] Tim Kraska et al. "MLbase: A Distributed Machine-learning System." In: *CIDR*. 2013.

[84] Alex Krizhevsky and Geoffrey Hinton. *CIFAR-10 Dataset*. https://www.cs.toronto.edu/~kriz/cifar.html.

[85] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *NIPS*. 2012, pp. 1097–1105.

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

[87] *LAPACK - Linear Algebra PACKage*. http://www.netlib.org/lapack/.

[88] Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Berlin, Heidelberg: Springer-Verlag, 2001. ISBN: 3-540-42184-X.

[89] Yann LeCun, Corinna Cortes, and Christopher JC Burges. "MNIST handwritten digit database". In: (1998).

[90] Xin Lei et al. "Accurate and compact large vocabulary speech recognition on mobile devices." In: *INTERSPEECH* (2013), pp. 662–665.

[91] Romain Lerallut, Diane Gasselin, and Nicolas Le Roux. "Large-Scale Real-Time Product Recommendation at Criteo". In: *RecSys*. 2015, pp. 232–232.

[92] Gideon Lewis-Kraus. "The Great A.I. Awakening". In: *The New York Times* (2016). URL: %5Curl%7Bhttps://www.nytimes.com/2016/12/14/magazine/the-great-ai-awakening.html%7D.

[93] Boduo Li, Sandeep Tata, and Yannis Sismanis. "Sparkler: Supporting Large-scale Matrix Factorization". In: *EDBT*. 2013.

[94] Haoyuan Li et al. "Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks". In: *SOCC*. 2014.

[95] Lihong Li et al. "A Contextual-bandit Approach to Personalized News Article Recommendation". In: *WWW*. 2010.

[96] Mu Li et al. "Scaling Distributed Machine Learning with the Parameter Server". In: *OSDI*. 2014, pp. 583–598.

[97] B. Lohrmann, P. Janacik, and O. Kao. "Elastic Stream Processing with Latency Guarantees". In: *2015 IEEE 35th International Conference on Distributed Computing Systems*. June 2015, pp. 399–410. DOI: 10.1109/ICDCS.2015.48.

[98] Björn Lohrmann, Daniel Warneke, and Odej Kao. "Nephele Streaming: Stream Processing Under QoS Constraints At Scale". In: *Cluster Computing* 17 (Aug. 2013). DOI: 10.1007/s10586-013-0281-8.

[99] Mateusz Malinowski, Marcus Rohrbach, and Mario Fritz. "Ask Your Neurons: A Neural-Based Approach to Answering Questions about Images". In: *2015 IEEE International Conference on Computer Vision (ICCV)* (2015), pp. 1–9.

[100] Christopher D. Manning et al. "The Stanford CoreNLP Natural Language Processing Toolkit". In: *Association for Computational Linguistics (ACL) System Demonstrations*. 2014, pp. 55–60. URL: http://www.aclweb.org/anthology/P/P14/P14-5010.

[101] Mason McGill and Pietro Perona. "Deciding How to Decide: Dynamic Routing in Artificial Neural Networks". In: *arXiv.org* (Mar. 2017). arXiv: 1703.06217v2 [stat.ML].

[102] H Brendan McMahan et al. "Ad click prediction: a view from the trenches". In: *KDD*. 2013, p. 1222.

[103] Raghu Meka et al. "Matrix Completion from Power-Law Distributed Samples". In: *NIPS*. 2009.

[104] Xiangrui Meng et al. "MLlib: Machine Learning in Apache Spark". In: *Journal of Machine Learning Research* 17.34 (2016), pp. 1–7. URL: http://jmlr.org/papers/v17/15-237.html.

[105] *ML.NET*. https://dot.net/ml.

[106] Volodymyr Mnih, Nicolas Heess, Alex Graves, et al. "Recurrent models of visual attention". In: *NIPS*. 2014, pp. 2204–2212.

[107] *Deep MNIST for Experts*. https://www.tensorflow.org/versions/r0.10/tutorials/mnist/pros/index.html.

[108] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[109] *MXNet*. https://mxnet.apache.org/.

[110] John Nagle. "Congestion Control in IP/TCP Internetworks". In: *SIGCOMM Comput. Commun. Rev.* 14.4 (Oct. 1984), pp. 11–17. ISSN: 0146-4833. DOI: 10.1145/1024908.1024910. URL: http://doi.acm.org/10.1145/1024908.1024910.

[111] *Netflix Prize*. https://www.netflixprize.com/.

[112] *nginx [engine x]*. http://nginx.org/en/.

[113] Jakob Nielsen. *Response Times: The 3 Important Limits*. https://www.nngroup.com/articles/response-times-3-important-limits/. 2014.

[114] *NVIDIA Volta*. https://www.nvidia.com/en-us/data-center/volta-gpu-architecture/.

[115] *Open ALPR*. https://www.openalpr.com/.

[116] Vivek S Pai, Peter Druschel, and Willy Zwaenepoel. "Flash: An efficient and portable Web server." In: *USENIX Annual Technical Conference, General Track* (1999), pp. 199–212.

[117] Adam Paszke et al. "Automatic differentiation in PyTorch". In: (2017).

[118] *Portable Format for Analytics (PFA)*. http://dmg.org/pfa/index.html.

[119] *PMML 4.2*. http://dmg.org/pmml/v4-2-1/GeneralStructure.html.

[120] Joseph Redmon et al. "You only look once: Unified, real-time object detection". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 779–788.

[121] Shaoqing Ren et al. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". In: *CoRR* abs/1506.01497 (2015). arXiv: 1506.01497. URL: http://arxiv.org/abs/1506.01497.

[122] Gonzalo P. Rodrigo et al. "Enabling Workflow-Aware Scheduling on HPC Systems". In: *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '17. Washington, DC, USA: ACM, 2017, pp. 3–14. ISBN: 978-1-4503-4699-3. DOI: 10.1145/3078597.3078604. URL: http://doi.acm.org/10.1145/3078597.3078604.

[123] *ROSS Intelligence*. https://rossintelligence.com/.

[124] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: 10.1007/s11263-015-0816-y.

[125] *Introducing Salesforce Einstein – AI for Everyone*. https://www.salesforce.com/blog/2016/09/introducing-salesforce-einstein.html.

[126] *Apache Samza*. http://samza.apache.org.

[127] Douglas C. Schmidt. "Pattern Languages of Program Design". In: 1995. Chap. Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching, pp. 529–545.

[128] *Scikit-Learn Machine Learning in Python*. http://scikit-learn.org.

[129] D Sculley et al. "Hidden Technical Debt in Machine Learning Systems." In: *NIPS* (2015).

[130] Alexander Sergeev and Mike Del Balso. "Horovod: fast and easy distributed deep learning in TensorFlow". In: *CoRR* abs/1802.05799 (2018). arXiv: 1802.05799. URL: http://arxiv.org/abs/1802.05799.

[131] Dinggang Shen, Guorong Wu, and Heung-Il Suk. "Deep Learning in Medical Image Analysis". In: *Annual review of biomedical engineering* 19 (June 2017). PMC5479722[pmcid], pp. 221–248. ISSN: 1545-4274. DOI: 10.1146/annurev-bioeng-071516-044442. URL: https://www.ncbi.nlm.nih.gov/pubmed/28301734.

[132] Haichen Shen et al. "Nexus: A GPU Cluster Engine for Accelerating DNN-based Video Analysis". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP '19. Huntsville, Ontario, Canada: ACM, 2019, pp. 322–337. ISBN: 978-1-4503-6873-5. DOI: 10.1145/3341301.3359658. URL: http://doi.acm.org/10.1145/3341301.3359658.

[133] Konstantin Shvachko et al. "The hadoop distributed file system". In: *MSST*. IEEE. 2010.

[134] J Sill et al. *Feature-weighted linear stacking*. 2009.

[135] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014).

[136] *Apple Siri*. http://www.apple.com/ios/siri/.

[137] *Skype real time translator*. https://www.skype.com/en/features/skype-translator/.

[138] Evan Sparks. "End-to-End Large Scale Machine Learning with KeystoneML". PhD thesis. EECS Department, University of California, Berkeley, Dec. 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-200.html.

[139] Evan R Sparks et al. "MLI: An API for distributed machine learning". In: *ICDM*. 2013.

[140] *Apache Storm*. http://storm.apache.org.

[141] Dan Suciu et al. *Probabilistic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

[142] Yi Sun, Xiaogang Wang, and Xiaoou Tang. "Deep Convolutional Network Cascade for Facial Point Detection." In: *CVPR* (2013), pp. 3476–3483.

[143] Christian Szegedy et al. "Going deeper with convolutions". In: *CVPR*. 2015, pp. 1–9.

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

[145] *TensorFlow*. https://www.tensorflow.org.

[146] *TensorRT Inference Server*. https://github.com/NVIDIA/tensorrt-inference-server.

[147] *TensorFlow Serving*. https://tensorflow.github.io/serving.

[148] *Timely Dataflow*. https://github.com/TimelyDataflow/timely-dataflow.

[149] *Active TPC Benchmarks*. http://www.tpc.org/information/benchmarks.asp.

[150] *Turi*. https://turi.com.

[151] Daisy Zhe Wang et al. "BayesStore: Managing Large, Uncertain Data Repositories with Probabilistic Graphical Models". In: *VLDB*. 2008.

[152] Xin Wang et al. "IDK Cascades: Fast Deep Learning by Learning not to Overthink". In: *UAI*. 2017.

[153] C. Watkins. "Learning from Delayed Rewards". PhD thesis. Cambridge, UK: King's College, May 1989.

[154] Matt Welsh, David E Culler, and Eric A Brewer. "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services." In: *SOSP* (2001), pp. 230–243.

[155] Eric P. Xing et al. "Petuum: A New Platform for Distributed Machine Learning on Big Data". In: *KDD*. ACM, 2015, pp. 1335–1344.

[156] Jason Yosinski et al. "How Transferable Are Features in Deep Neural Networks?" In: *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'14. Montreal, Canada: MIT Press, 2014, pp. 3320–3328. URL: http://dl.acm.org/citation.cfm?id=2969033.2969197.

[157] S. J. Young et al. *The HTK Book, version 3.4*. Cambridge, UK: Cambridge University Engineering Department, 2006.

[158] Jeong-Min Yun et al. "Optimal Aggregation Policy for Reducing Tail Latency of Web Search". In: *SIGIR*. 2015, pp. 63–72.

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

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

[161] Haoyu Zhang et al. "Live Video Analytics at Scale with Approximation and Delay-Tolerance". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 377–392. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zhang.

[162] Martin Zinkevich. "Rules of Machine Learning: Best Practices for ML Engineering". In: (). URL: https://developers.google.com/machine-learning/guides/rules-of-ml.