Clipper: A Low-Latency Online Prediction Serving System

Daniel Crankshaw*, Xin Wang*, Giulio Zhou*
Michael J. Franklin*†, Joseph E. Gonzalez*, Ion Stoica*
*UC Berkeley †The University of Chicago

Abstract

Machine learning is being deployed in a growing number of applications which demand real-time, accurate, and robust predictions under heavy query load. However, most machine learning frameworks and systems only address model training and not deployment.

In this paper, we introduce Clipper, a general-purpose low-latency prediction serving system. Interposing between end-user applications and a wide range of machine learning frameworks, Clipper introduces a modular architecture to simplify model deployment across frameworks and applications. Furthermore, by introducing caching, batching, and adaptive model selection techniques, Clipper reduces prediction latency and improves prediction throughput, accuracy, and robustness without modifying the underlying machine learning frameworks. We evaluate Clipper on four common machine learning benchmark datasets and demonstrate its ability to meet the latency, accuracy, and throughput demands of online serving applications. Finally, we compare Clipper to the Tensorflow Serving system and demonstrate that we are able to achieve comparable throughput and latency while enabling model composition and online learning to improve accuracy and render more robust predictions.

1 Introduction

The past few years have seen an explosion of applications driven by machine learning, including recommendation systems [28, 60], voice assistants [18, 26, 55], and ad-targeting [3, 27]. 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) [64], 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.

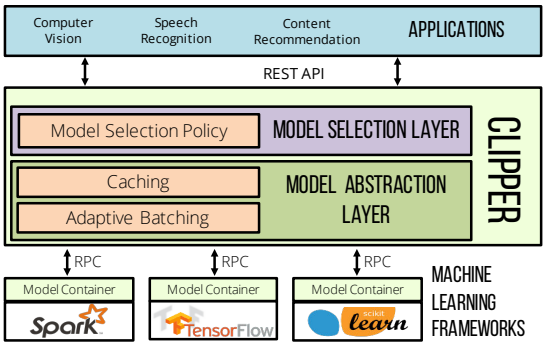


Figure 1: The Clipper Architecture.

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-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 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 [40], Scikit-Learn [51], Caffe [31], TensorFlow [1], and HTK [63]. 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 [59], 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 (§3).
- A set of novel techniques to reduce and bound latency while maximizing throughput that generalize across machine learning frameworks (§4).
- A model selection layer that enables online model selection and composition to provide robust and accurate predictions for interactive applications (§5).

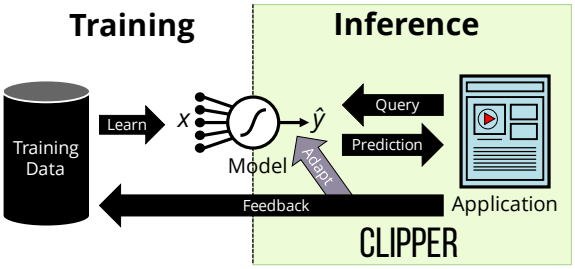


Figure 2: Machine Learning Lifecycle.

2 Applications and Challenges

The machine learning life-cycle (Figure 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 [11, 29, 41]. Much of the innovation in systems for machine learning has focused on model training with the development of systems like Apache Spark [65], the Parameter Server [38], PowerGraph [25], and Adam [14].

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 [34] for large linear models. Others are specialized for specific application domains such as Caffe [31] for computer vision or HTK [63] 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 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*.

2.1 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 prob-

lems – 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 [35], CIFAR-10 [32], and ImageNet [49] 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 [10] and are often composed of many complex sub-models trained using specialized speech recognition frameworks (e.g., HTK [63]). 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 [56]. However, inference in speech models can be costly [10] 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 [24] and the HTK [63] 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.

2.2 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 [1,7,13,16,31]. 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 [47, 48], 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 (§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 [64]. 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) [13] (see Figure 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. More-

over, 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 [3, 23].

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 (§4.3). 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 (§5.2.2).

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 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 [52]. 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.

2.3 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 [63] 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 1.

Dataset	Type	Size	Features	Labels
MNIST [35]	Image	70K	28x28	10
CIFAR [32]	Image	60k	32x32x3	10
ImageNet [49]	Image	1.26M	299x299x3	1000
Speech [24]	Sound	6300	5 sec.	39

Table 1: Datasets. The collection of real-world benchmark datasets used in the experiments.

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 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 System Architecture

Clipper is divided into *model selection* and *model abstraction* layers (see Figure 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.

4 Model Abstraction Layer

The **Model Abstraction Layer** (Figure 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.

4.1 Overview

At the top of the model abstraction layer is the prediction cache (§4.2). The prediction caches provides a partial pre-materialization mechanism for frequent queries and accelerates the adaptive model selection techniques described in §5 by enabling efficient joins between recent predictions and feedback.

The batching component (§4.3) 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 (§4.4), 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.

4.2 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 (§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 [39], 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 [17] 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.

4.3 Batching

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

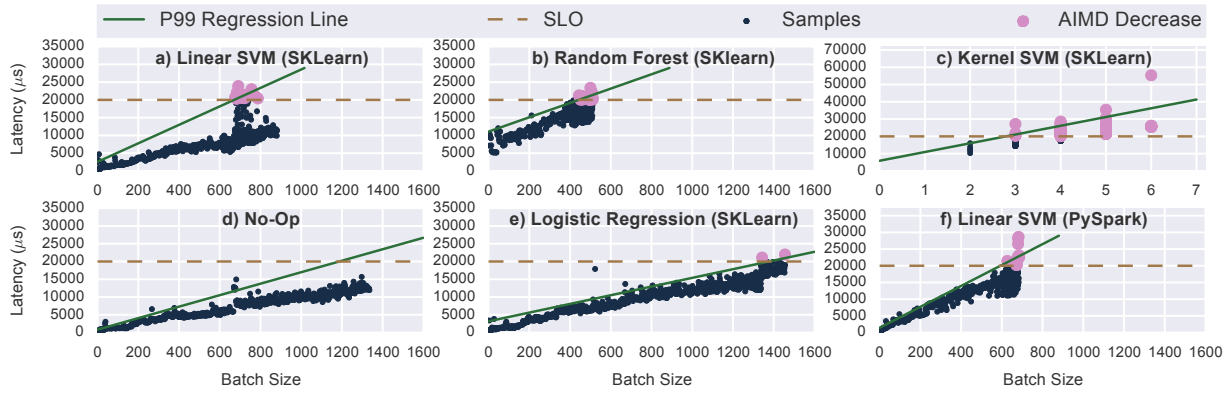


Figure 3: Model Container Latency Profiles. We measured the batching latency profile of several models trained on the MNIST benchmark dataset. The models were trained using Scikit-Learn (SKLearn) or Spark and were chosen to represent several of the most widely used types of models. The No-Op Container measures the system overhead of the model containers and RPC system.

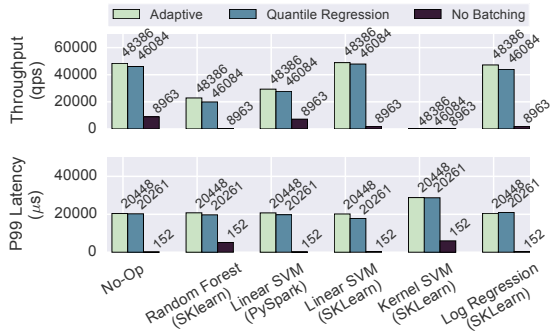


Figure 4: Comparison of Dynamic Batching Strategies.

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 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.

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 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.

4.3.1 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 automati-

cally 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 [15].

Early performance measurements (Figure 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 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.

4.3.2 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 [44], 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 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.

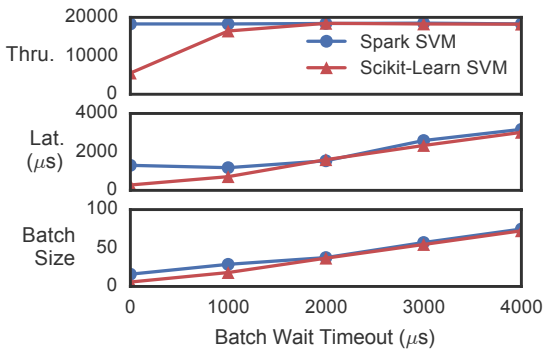


Figure 5: Throughput Increase from Delayed Batching.

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

Listing 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.

4.4 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 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.

4.4.1 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

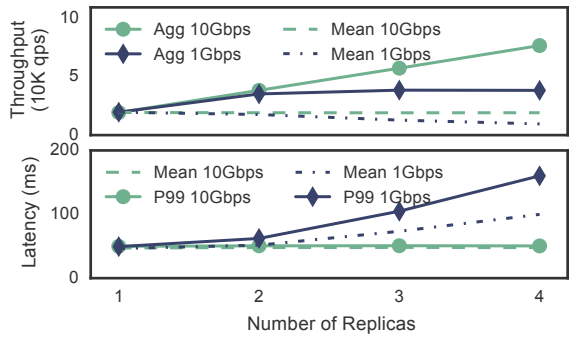


Figure 6: Scaling the Model Abstraction Layer Across a GPU Cluster. The solid lines refer to aggregate throughput of all the model replicas and the dashed lines refer to the mean per-replica throughput.

each replica.

In Figure 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.

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 2) which governs the behavior of the model selection layer and allows users to introduce new model selection techniques themselves.

```
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 2: Model Selection Policy Interface.

The model selection policy (Listing 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 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 (§5.3). 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 §5.2.1 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. [6]. These algorithms span a trade-off between computation overhead and accuracy. The single model selection policy (§5.1) leverages the Exp3 algorithm to optimally *select* the best model based on noisy feedback with minimal computational overhead. The ensemble model selection policy (§5.2) 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.

Framework	Model	Size (Layers)
Caffe	VGG [54]	13 Conv. and 3 FC
Caffe	GoogLeNet [57]	96 Conv. and 5 FC
Caffe	ResNet [29]	151 Conv. and 1 FC
Caffe	CaffeNet [22]	5 Conv. and 3 FC
TensorFlow	Inception [58]	6 Conv, 1 FC, & 3 Incept.

Table 2: Deep Learning Models. The set of deep learning models used to evaluate the ImageNet ensemble selection policy.

5.1 Single Model Selection Policy

We can cast the model-selection process as a multi-armed bandit problem [43]. 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 [6] 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.

5.2 Ensemble Model Selection Policies

It is a well-known result in machine learning [8,12,30,43] that prediction accuracy can be improved by combining predictions from multiple models. For example, bootstrap aggregation [9] (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 [53], 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 [30]. The ensemble model selection policies adaptively combine the predictions from *all* available models to improve accuracy, rather than select individual models.

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

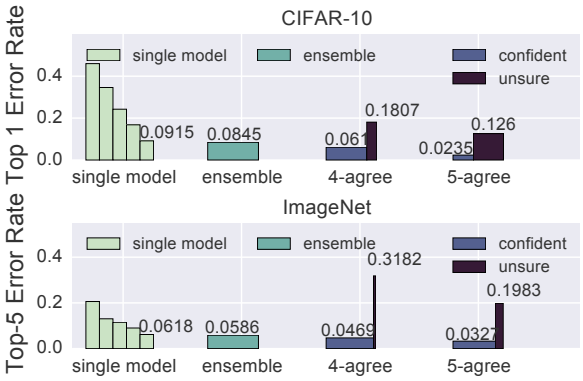


Figure 7: Ensemble Prediction Accuracy. The linear ensembles are composed of five computer vision models (Table 2) applied to the CIFAR and ImageNet benchmarks. The 4-agree and 5-agree groups correspond to ensemble predictions in which the queries have been separated by the ensemble prediction confidence (four or five models agree) and the width of each bar defines the proportion of examples in that category.

In Clipper we use linear ensemble methods which compute a weighted average of the base model predictions. In Figure 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 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 [49].

There are many methods for estimating the ensemble weights including linear regression, boosting [43], and bandit formulations. We adopt the bandits approach and use the Exp4 algorithm [6] 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.

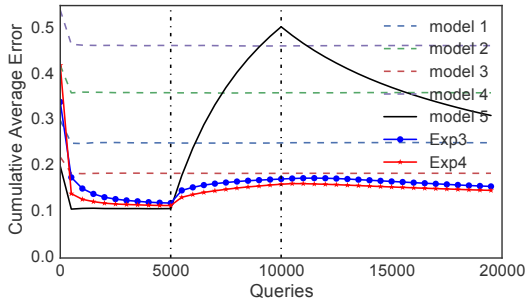


Figure 8: Behavior of Exp3 and Exp4 Under Model Failure. After 5K queries the performance of the lowest-error model is severely degraded, and after 10k queries performance recovers. Exp3 and Exp4 quickly compensate for the failure and achieve lower error than any static model selection.

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.

In Figure 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.

5.2.1 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.

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.

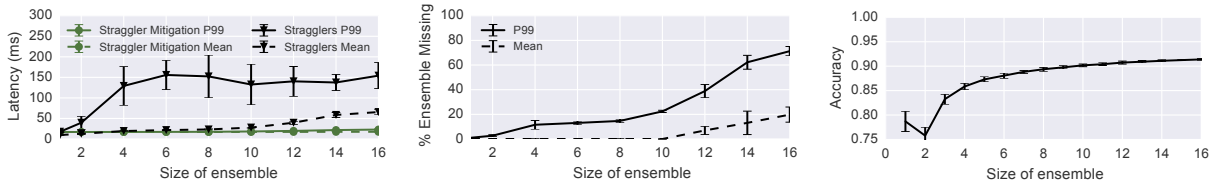
By evaluating predictions from multiple competing models concurrently we can obtain an estimator of the confidence in our predictions. In settings where models have high variance or are trained on random samples from the training data (e.g., bagging), agreement in model predictions is an indicator of 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 7) while declining to predict a small fraction of queries.

5.2.2 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 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



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

Figure 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.

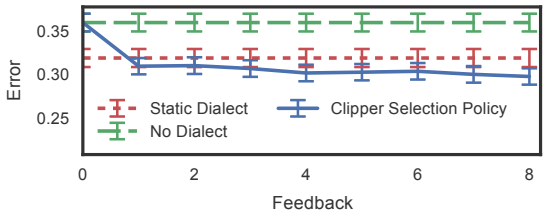


Figure 10: Personalized Model Selection. Accuracy of the ensemble selection policy on the speech recognition benchmark.

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 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 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.

5.3 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 [24] voice recognition models each trained for a different dialect. We then evaluated (Figure 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.

6 System Comparison

In addition to the microbenchmarks presented in §4 and §5, we compared Clipper's performance to TensorFlow Serving and evaluate latency and throughput on three object recognition benchmarks.

TensorFlow Serving [59] 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 intro-

duced 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 [42], the 8-layer AlexNet [33] architecture trained on CIFAR-10 [32], and Google's 22-layer Inception-v3 network [58] 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 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 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.

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.

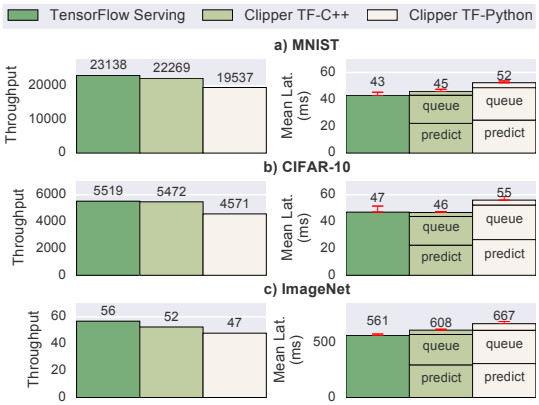


Figure 11: TensorFlow Serving Comparison. Comparison of peak throughput and latency (p99 latencies shown in error bars) on three TensorFlow models of varying inference cost. TF-C++ uses TensorFlow's C++ API and TF-Python the Python API.

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 retraining 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.

8 Related Work

The closest projects to Clipper are LASER [3], Velox [19], and TensorFlow Serving [59]. 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 [20], ad-targeting [27, 39], and product-recommendations [37]. Outside of content recommendation, there has been recent success in speech recognition [36, 55] and internet-scale resource allocation [23]. 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 [5, 21, 38, 62]. 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 [45, 46, 50, 61]. The division of functionality into vertical stages introduced by [61] 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.

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.

Acknowledgments

We would like to thank Peter Bailis, Alexey Tumanov, Noah Fiedel, Chris Olston, our shepherd Mike Dahlin, and the anonymous reviewers for their feedback. This research is supported in part by DHS Award HSHQDC-16-3-00083, DOE Award SN10040 DE-SC0012463, NSF CISE Expeditions Award CCF-1139158, and gifts from Ant Financial, Amazon Web Services, CapitalOne, Ericsson, GE, Google, Huawei, Intel, IBM, Microsoft and VMware.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous systems, 2015. *Software available from tensorflow. org*.

[2] A. Agarwal, S. Bird, M. Cozowicz, L. Hoang, J. Langford, S. Lee, J. Li, D. Melamed, G. Oshri, O. Ribas, et al. A multiworld testing decision service. *arXiv preprint arXiv:1606.03966*, 2016.

[3] D. Agarwal, B. Long, J. Traupman, D. Xin, and L. Zhang. Laser: A scalable response prediction platform for online advertising. In *WSDM*, pages 173–182, 2014.

[4] S. Agarwal and J. R. Lorch. Matchmaking for online games and other latency-sensitive p2p systems. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 315–326. ACM, 2009.

[5] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola. Scalable inference in latent variable models. In *WSDM*, pages 123–132, 2012.

[6] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire. The nonstochastic multiarmed bandit problem. *SIAM J. Comput.*, 32(1):48–77, Jan. 2003.

[7] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: a cpu and gpu math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, volume 4, page 3. Austin, TX, 2010.

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

[9] L. Breiman. Bagging predictors. *Mach. Learn.*, 24(2):123–140, Aug. 1996.

[10] C. Chelba, D. Bikel, M. Shugrina, P. Nguyen, and S. Kumar. Large scale language modeling in automatic speech recognition. *arXiv preprint arXiv:1210.8440*, 2012.

[11] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz. Revisiting Distributed Synchronous SGD. *arXiv.org*, Apr. 2016.

[12] T. Chen and C. Guestrin. XGBoost: A Scalable Tree Boosting System. *arXiv.org*, Mar. 2016.

[13] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[14] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, pages 571–582, 2014.

[15] D.-M. Chiu and R. Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Comput. Netw. ISDN Syst.*, 17(1):1–14, June 1989.

[16] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.

[17] F. J. Corbato. A paging experiment with the multics system. 1968.

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

[19] D. Crankshaw, P. Bailis, J. E. Gonzalez, H. Li, Z. Zhang, M. J. Franklin, A. Ghodsi, and M. I. Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with velox. In *CIDR 2015*, 2015.

[20] J. Davidson, B. Liebald, J. Liu, P. Nandy, T. Van Vleet, U. Gargi, S. Gupta, Y. He, M. Lambert, B. Livingston, and D. Sampath. The YouTube video recommendation system. *RecSys*, pages 293–296, 2010.

[21] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. aurelio Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng. Large scale distributed deep networks. In *NIPS*, pages 1223–1231. 2012.

[22] J. Donahue. Caffenet. `https://github.com/BVLC/caffe/tree/master/models/bvlc_reference_caffenet`.

[23] A. Ganjam, F. Siddiqui, J. Zhan, X. Liu, I. Stoica, J. Jiang, V. Sekar, and H. Zhang. C3: Internet-Scale Control Plane for Video Quality Optimization. *NSDI '15*, pages 131–144, 2015.

[24] J. S. Garofolo, L. F. Lamel, W. M. Fisher, J. G. Fiscus, D. S. Pallett, and N. L. Dahlgren. Darpa timit acoustic phonetic continuous speech corpus cdrom, 1993.

[25] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. OSDI, pages 17–30, 2012.

[26] Google Now. `https://www.google.com/landing/now/`.

[27] T. Graepel, J. Q. Candela, T. Borchert, and R. Herbrich. Web-Scale Bayesian Click-Through rate Prediction for Sponsored Search Advertising in Microsoft's Bing Search Engine. *ICML*, pages 13–20, 2010.

[28] h20. `http://www.h2o.ai`.

[29] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[30] G. Hinton, O. Vinyals, and J. Dean. Distilling the Knowledge in a Neural Network. *arXiv.org*, Mar. 2015.

[31] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia*, pages 675–678. ACM, 2014.

[32] A. Krizhevsky and G. Hinton. Cifar-10 dataset. `https://www.cs.toronto.edu/~kriz/cifar.html`.

[33] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, pages 1097–1105, 2012.

[34] J. Langford, L. Li, and A. Strehl. Vowpal wabbit online learning project, 2007.

[35] Y. LeCun, C. Cortes, and C. J. Burges. MNIST handwritten digit database. 1998.

[36] X. Lei, A. W. Senior, A. Gruenstein, and J. Sorensen. Accurate and compact large vocabulary speech recognition on mobile devices. *INTERSPEECH*, pages 662–665, 2013.

[37] R. Lerallut, D. Gasselin, and N. Le Roux. Large-Scale Real-Time Product Recommendation at Criteo. In *RecSys*, pages 232–232, 2015.

[38] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, pages 583–598, 2014.

[39] H. B. McMahan, G. Holt, D. Sculley, M. Young, D. Ebner, J. Grady, L. Nie, T. Phillips, E. Davydov, D. Golovin, S. Chikkerur, D. Liu, M. Wattenberg, A. M. Hrafnkelsson, T. Boulos, and J. Kubica. Ad click prediction: a view from the trenches. In *KDD*, page 1222, 2013.

[40] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17(34):1–7, 2016.

[41] V. Mnih, N. Heess, A. Graves, et al. Recurrent models of visual attention. In *NIPS*, pages 2204–2212, 2014.

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

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

[44] J. Nagle. Congestion control in ip/tcp internetworks. *SIGCOMM Comput. Commun. Rev.*, 14(4):11–17, Oct. 1984.

[45] nginx [engine x]. `http://nginx.org/en/`.

[46] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. *USENIX Annual Technical Conference, General Track*, pages 199–212, 1999.

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

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

[49] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[50] D. C. Schmidt. Pattern languages of program design. chapter Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching, pages 529–545. 1995.

[51] Scikit-Learn machine learning in python. `http://scikit-learn.org`.

[52] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison. Hidden Technical Debt in Machine Learning Systems. *NIPS*, 2015.

[53] J. Sill, G. Takács, L. Mackey, and D. Lin. Feature-weighted linear stacking, 2009.

[54] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[55] Apple Siri. `http://www.apple.com/ios/siri/`.

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

[57] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *CVPR*, pages 1–9, 2015.

[58] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. *arXiv preprint arXiv:1512.00567*, 2015.

[59] TensorFlow Serving. `https://tensorflow.github.io/serving`.

[60] Turi. `https://turi.com`.

[61] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. *SOSP*, pages 230–243, 2001.

[62] E. P. Xing, Q. Ho, W. Dai, J.-K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. Petuum: A new platform for distributed machine learning on big data. In *KDD*, pages 1335–1344. ACM, 2015.

[63] S. J. Young, G. Evermann, M. J. F. Gales, T. Hain, D. Kershaw, G. Moore, J. Odell, D. Ollason, D. Povey, V. Valtchev, and P. C. Woodland. *The HTK Book, version 3.4*. Cambridge University Engineering Department, Cambridge, UK, 2006.

[64] J.-M. Yun, Y. He, S. Elnikety, and S. Ren. Optimal aggregation policy for reducing tail latency of web search. In *SIGIR*, pages 63–72, 2015.

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