InferLine: ML Inference Pipeline Composition Framework

Corey Zumar

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2018-76
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-76.html

May 18, 2018

Copyright © 2018, by the author(s).
All rights reserved.

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

Acknowledgement

I would like to thank the rest of the InferLine team: Dan Crankshaw, Alexey Tumanov, Eyal Sela, Joseph Gonzalez, and Ion Stoica.

InferLine: ML Inference Pipeline Composition Framework

*Corey Zumar**

Abstract

Model composition in the form of prediction pipelines is an emerging pattern in the design of machine learning applications that offers the opportunity to substantially simplify development, improve accuracy, and reduce cost. However, in low-latency settings spanning multiple machine learning frameworks with varying resource requirements , prediction pipelines are challenging and expensive to provision and execute.

In this paper we address the challenges of allocating resources and efficiently and reliably executing prediction pipelines spanning multiple machine learning models and frameworks. We exploit the reproducible performance characteristics of individual models and monotonic performance scaling of prediction workloads to decompose the resource allocation and performance tuning problem along model boundaries. Consequently, we are able to estimate and optimize end-to-end system performance. Our proposed system—InferLine—leverages these insights and instantiates a general-purpose framework for serving prediction pipelines. We demonstrate that InferLine is able to configure and execute prediction pipelines across a wide range of throughput and latency goals and achieve over a 6x reduction in cost when compared to a hand-tuned and horizontally scaled single process pipeline.

1 Introduction

Machine learning (ML) is rapidly maturing from an academic field to an engineering discipline at the center of many production software systems. As a consequence, the need for systems in machine learning is shifting from the development and training of individual models [14,36,42] to the composition [5,27,32] and serving of prediction pipelines spanning multiple models and frameworks.

*Based on OSDI 2018 conference submission written with Daniel Crankshaw, Alexey Tumanov, Eyal Sela, Joseph E. Gonzalez, and Ion Stoica

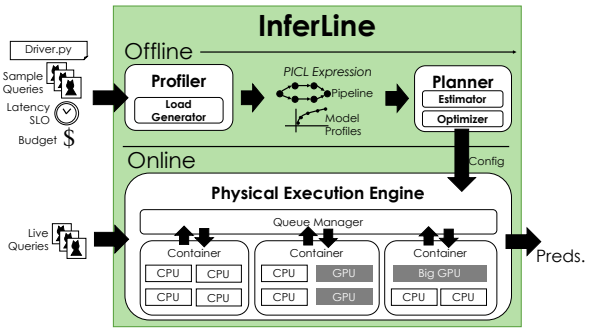


Figure 1: Architecture Diagram. InferLine consists of three main components: the *Profiler*, the *Planner*, and the *Physical Execution Engine*. The Planner builds on the interaction between the Optimizer and the Estimator. Given a pipeline driver program, a representative workload, and end-to-end latency SLO, the Profiler produces pipeline lineage and model profiles, used by the Planner to find an optimal pipeline configuration. The Physical Execution Engine is then used to configure and serve the specified pipeline on a heterogeneous mix of hardware.

Prediction pipelines offer the opportunity to improve accuracy [4,7], increase throughput [17,40], and simplify model development [5,31]. For example, by decomposing complex prediction tasks (e.g., speech translation) into a sequence of simpler prediction tasks (e.g., speech recognition and text translation) we can reuse established task-specific models to develop new functionality in a modular fashion. Moreover, model compositions yields the familiar benefits of modularity by amortizing the cost, both time and money, of developing and maintaining models across a wide range of prediction pipelines.

Problem Statement. The transition from learning singular models to composing and serving increasingly sophisticated prediction pipelines presents fundamental new challenges around how we provision, manage, and serve these new prediction workloads. Pipeline developers must navigate a complex trade-off space spanned by latency,

throughput, accuracy, and monetary costs that depends on the type and configuration of each model and choice of hardware. These choices can have significant consequences, with prediction performance and costs often varying by orders-of-magnitude across configurations. Prediction pipelines are often deployed in latency-critical applications, imposing latency constraints on the end-to-end performance of prediction pipelines. Failing to meet these latency service level objectives can also have significant consequences (e.g., lost revenue).

Complex Configuration Space. The combination of types of models and parallel hardware, in conjunction with model specific replication and batching parameters, creates a combinatorially complex search space of options for deploying prediction pipelines. Each model can be configured to leverage a wide range of parallel hardware including multi-core processors, multiple GPU generations (e.g., K80, V100), and specialized accelerators (e.g., TPUs [18]). The resulting throughput and latency for a given hardware configuration depends heavily on the choice of model and query batching. As a consequence, it is also necessary to specify a batching parameter for each model and hardware configuration.

The configuration of each model interacts with other models in a pipeline requiring global reasoning. For example, tuning the batch size of an upstream model to maximize parallelism may tighten the latency requirements of a downstream model and change its optimal hardware configuration. Moreover, optimally configuring prediction pipelines requires both a local understanding of the performance characteristics of each model and globally reasoning about each model configuration to match latency and throughput requirements.

Proposed Solution. To address these challenges we propose InferLine—a high-performance, general purpose system for provisioning and serving prediction pipelines. InferLine takes as input a pipeline driver program (e.g., python script), a collection of model containers, and example queries. InferLine automatically extracts the pipeline structure from execution traces and constructs performance profiles for each model as a function of hardware and batch size. InferLine combines the individual profiles to construct an end-to-end pipeline performance estimator and optimize the system configuration to maximize throughput subject to end-to-end latency and cost constraints. Finally, InferLine serves the prediction pipeline by adopting a decentralized container oriented architecture similar to the design of Clipper [12].

Key Insight and Contributions. The key insight in the design of InferLine is that the complex but deterministic performance characteristics of each model can be accurately characterized using offline profiling and then composed to optimally configure end-to-end prediction pipelines. The resulting contributions of InferLine are: (1) a single model profiler that empirically extracts the performance characteristics of each model in the pipeline as a function of hardware allocation and batch size, (2) a workload-aware estimator that uses the individual model profiles to *estimate* end-to-end pipeline throughput, latency, and cost as a function of a given pipeline configuration, and (3) a pipeline optimizer that uses the performance estimates to efficiently maximize throughput per dollar subject to SLO constraints.

We use InferLine to configure a range of realistic prediction pipelines that capture common pipeline designs. We evaluate the configured pipelines using query traces with a multiple query arrival distributions. We compare InferLine's optimizer-produced plans to two standard methods for serving prediction pipelines: a hand tuned single process query driver that directly invokes each parallel modeling framework avoiding any RPC overheads, as well as TensorFlow Serving [38], an open-source model serving system developed at Google. We show that InferLine can reliably estimate end-to-end performance and construct configurations that achieve cost reductions that exceed the hand tuned single process design by more than a factor of $6\times$ while meeting latency requirements.

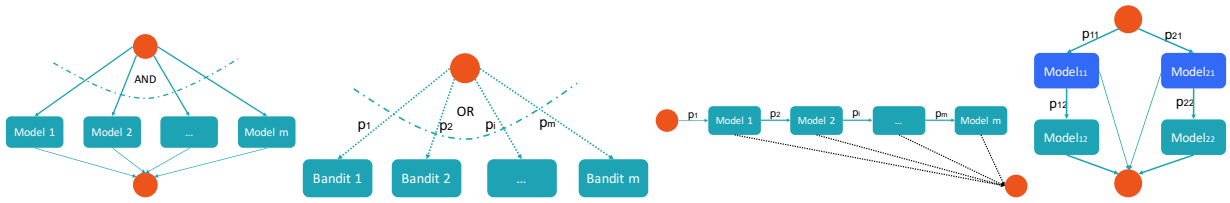
2 Prediction Pipelines

Model composition is the process of composing multiple models to complete a single prediction task. Model composition can improve prediction accuracy, reduce the cost of serving predictions, and simplify model development. Thus, model composition is rapidly becoming standard practice in machine learning [32]. For instance, visual models are combined with language models for visual Q&A [2, 24]. Recent work [43] on stream processing for video surveillance combined models for vehicle detection with models for license plate decoding.

In the following we outline the advantages of model composition and then characterize several of the key system challenges in supporting model composition. We offer a high-level taxonomy of common composition patterns (Figure 2) informed by interviews with industrial and academic ML practitioners.

2.1 Opportunities

Improved Accuracy. Model composition can be used to boost prediction accuracy using a wide range of established techniques. Ensemble methods (e.g., boosting [30], and bagging [7]) are used to combine predictions from multiple models (e.g., by averaging) to obtain more accurate predictions. Bandit methods [4, 22], which dynam-



(a) ensemble (b) bandit/selection (c) cascade (d) cascade ensemble

Figure 2: Common composition patterns that arise in practice include ensembles of models, bandit algorithms, and cascading machine ML inference. There's closure under some of these patterns (e.g., an ensemble of cascades).

ically select and average one or more models on a per query basis, are widely used to personalize predictions.

Faster Inference. Model composition can actually reduce the cost of rendering predictions. Prediction cascades, introduced by Viola and Jones [17] to enable face detection on embedded devices, reduce prediction costs by only invoking more accurate and expensive models when necessary. Typically models are invoked in sequence of increasing cost until their is sufficient confidence in the prediction. However, to retain the full accuracy some queries must still pass through all the models in the cascade. With the increasing computational cost of deep learning, there has been a resurgence of interest [3,15,25,34] in cascaded model design. These newer cascades present the opportunity to substantially reduce prediction costs, but introduce prediction pipelines with varying latencies that span models with different hardware scaling characteristics.

Faster Development. Developing and training new deep learning models requires substantial training data, compute resources, and expertise in the brittle process of hyperparameter tuning [6,33]. As a consequence, model developers commonly reuse models [41] that have been pre-trained on large well studied benchmark datasets or developed internally and used across the organization (e.g., customer churn). In many cases, a single model (e.g., ResNet152 [16]) may be re-used as a feature function for a wide range of prediction tasks (e.g., product recommendation and medical imaging).

While pre-trained versions of these models are freely available [10,37], they are often coded and optimized for a particular machine learning framework. The outputs of these models are then used as inputs to more robust and easier to train models (e.g., nearest neighbor and linear models) or as input to subsequent deep learning pipelines. By avoiding the need to go through the costly model design and training process, this form of model reuse can substantially accelerate application development.

2.2 Model Composition Challenges

Rendering predictions for many machine learning workloads can be computationally intensive with substantial opportunities for internal and external parallelism. A single deep neural network may have many stages that can benefit from parallel hardware. 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 191 QPS on an NVIDIA Tesla V100 GPU (a 300x difference in throughput). However, to fully use the available parallel hardware, queries must be processed in batches (e.g., ResNet152 required a batch size of 32 to maximize throughput on the V100). The optimal batch size depends on the model and hardware configuration as well as latency goals.

Not all models benefit equally from hardware accelerators. The diversity of models results in heterogeneous compute and hardware requirements. In contrast to deep neural networks which substantially benefit from parallel accelerators, many widely used classical models (e.g., support vector machines [9] and decision trees [8]) can be difficult to parallelize on GPUs. Similarly, many of the necessary pre-processing and post-processing operations (e.g., error checking or beam search) have limited use of parallel hardware. As a consequence, 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 is combinatorially more complex for prediction pipelines.

In high query load settings it is necessary to leverage parallelism at the level of the pipeline to scale-out individual models and high-fanout stages. Identifying and scaling the bottleneck models is critical to achieving high-throughput at low cost. Pipelines with high-fanout stages (e.g., ensembles of models) can often be accelerated by dividing compute resources across parallel paths in the pipeline. However, the optimal placement of resources

depends heavily on the performance characteristics of each model and the optimal choice of batch size. Furthermore, as we scale the compositions of models, we quickly discover that individual components scale differently.

Dynamicaly Changing Configuration Space. The optimal point in the trade-off space can change with new hardware, changes in the underlying software, and updates to the model. Over the course of writing this paper, the optimal configuration of the benchmark pipelines changed due to upgrades in the low-level CUDA drivers, TensorFlow library, and even the Meltdown [23]/Spectre [20] kernel patches. As a consequence, exhaustively optimized configurations would need to be rerun with each update to any part of the pipeline or system.

3 Design and Architecture

In this section we provide a high-level overview of the three parts of InferLine system (see Figure 1): the *Profiler*, the *Planner*, and the *Physical Execution Engine*. The Profiler takes user-provided machine learning pipelines written with the light-weight InferLine model composition API and extracts the logical pipeline dependency structure and individual model performance profiles. The Planner takes the dependency graph and profiles expressed in the PICL declarative language, along with latency and cost constraints for the end-to-end pipeline and produces a configuration for the physical execution engine. At serving time, the physical execution engine provisions container resources according to the configuration produced by the planner. Queries are then executed through an RPC interface between the driver program an the execution engine.

3.1 Pipeline Composition API

Every prediction pipeline in InferLine consists of a single, stateless driver function that takes an input (the *query*) and returns an output (the *prediction*). Within this function, developers interleave application-specific code executed in the driver with asynchronous RPC calls to models hosted in InferLine. This flexible programming model addresses the fundamental need for an expressive mechanism to compose models in a prediction pipeline.

To support user defined logic and flexible model composition, InferLine allows users to specify their pipeline in a fully featured Python driver rather than a restrictive domain specific language [26, 32]. However, to scale out, support query planning, and to enable model execution on different hardware architectures, InferLine needs a way to capture statistics about the pipeline call graph and move model invocation off the driver hardware.

The current implementation of the InferLine prediction pipeline API exposes a Python client API wrapping a C++ RPC client that issues queries to the InferLine serving system. To query a model, the driver program makes an RPC request to InferLine specifying the model name and providing the input. This RPC call is asynchronous and returns a future so that pipelines can evaluate multiple models in parallel. InferLine's futures can be chained, composed, and transformed to maximize external parallelism. Finally, the futures are used to extract statistics about function invocations needed for planning.

Before the Planner can optimize a prediction pipeline, all of the models referenced in the pipeline must be registered with the system. InferLine employs a distributed, containerized architecture (see §3.2) for the physical execution engine. Registering a model in InferLine simply requires associating a name (the model name) and optional version with a Docker image. Models can then be associated with multiple pipelines.

3.2 Physical Execution Engine

Similar to [12], the InferLine online physical execution engine adopts a distributed micro-service architecture consisting of three components, a pipeline driver embedded in the client application (e.g., an application server), a centralized queuing system, and a set of replicated model containers each hosted in their own Docker container.

InferLine maintains a centralized queue for all replicas of each type of model. Every RPC requests to a model is placed in the queue corresponding to that model. The replicated model containers then request bounded size batches of queries from their respective centralized queue. The batch size is determined by the Planner.

Each model queue is an EDF priority queue ordered by request deadline to ensure that InferLine is always processing the queries that will expire first. When a model replica worker is ready to process a new batch of inputs, it requests a new batch from the queue. The size of the batch is bounded by the maximum *batch size* for the model, one of the three configuration parameters that the InferLine planner configures. By employing a pull-based queuing strategy and imposing a maximum batch size, InferLine places an upper bound on the time that a query will spend in the model container itself after leaving the queue. This upper bound is critical for enabling the Estimator to accurately estimate the inference time of each model.

Because inference is a compute-intensive operation, it is critical that each container has exclusive access to its own set of compute resources to minimize interference. As a consequence, each replica for a model runs in its own Docker container, allocated an exclusive set of computational resources: the *resource bundle*, the second configuration parameter set by the Planner. All repli-

cas of a model are allocated the same *type* of resource bundle and each individual replica is given its own distinct bundle. Each resource bundle is pinned to a unique set of physical CPUs and GPUs based on the Planner configuration. Because GPUs intensive models still require substantial CPU resources to mediate network and GPU data transfers we use Docker's `cpuset-cpus` option to pin containers to cores at runtime, and set the `CUDA_VISIBLE_DEVICES` environment variable for each container to control GPU access.

When a model container is started, it is provided the address of the InferLine queue. It registers itself with the queuing system, informing the queue which model it is running and requesting a first batch. Once the model receives a batch, it processes it fully before returning the predictions to the queueing system (to be forwarded back to the client) and requesting another batch. Model containers communicate with the queue over an RPC system (note that this RPC system is separate from the one the driver clients use to communicate with InferLine), allowing InferLine to scale its model workers across a cluster. We refer to the number of replicas of a model as the model's *replication factor*, the third configuration parameter set by the Planner.

3.2.1 Implementation

The InferLine physical execution engine is an extension of the Clipper [12] prediction serving system. Clipper provides a latency-aware model serving platform that leverages query batching to improve system throughput. In order to meet the high performance demands associated with serving large volumes of pipeline requests, InferLine's physical execution layer extends Clipper in the following ways:

1. **High-performance RPC frontend:** In order to support query ingest rates of up to 10Gps, InferLine replaces Clipper's client-facing REST frontend with an RPC implementation based on ZeroMQ [1]. The client-side module for querying the frontend supports the asynchronous sending of requests and the execution of user-defined callbacks upon request completion.

2. **Improved memory management:** Clipper's query frontend allocates memory to store model input data on a per-request basis. Unfortunately, in scenarios where the query ingest rate is high, performing thousands of heap allocations per second imposes a significant latency overhead. Accordingly, the InferLine physical execution engine preallocates a large memory buffer for receiving requests. Requests are read from the frontend RPC socket into this buffer at the smallest unoccupied index, thus avoiding additional heap allocations.

3. **Lock-free queueing:** Clipper's critical query evaluation path relies on several highly-utilized queues that are accessed by multiple threads. Lock-based synchronization is employed to prevent concurrency errors. Based on the observation of significant lock contention associated with queue accesses, InferLine replaces the lock-based implementation with lock-free, concurrent queues [13] in the task execution and RPC subsystems.

4. **Removal of redundant copies:** Ideally, request data should only be copied in or out of socket buffers associated with Clipper's frontend or model-facing RPC systems. Because inputs often exceed 1MB in size, as is the case with ImageNet samples, It is critical that no additional copies occur. InferLine heavily modifies Clipper's input data representation format and critical query evaluation path to ensure that this is the case.

These performance-oriented improvements are slated for incorporation into the open source implementation available on GitHub [11].

4 The Profiler

The first step in the *offline* planning process is to run the Profiler and extract the logical pipeline structure as a directed acyclic graph (DAG) of data dependencies among models. We employ an empirical procedure to infer the pipeline structure directly from the pipeline driver program. Along with the pipeline program, users must provide sample queries that characterize the model inputs. The profiler evaluates the pipeline on every sample query, using the RPC and futures to collect the pipeline graph of each individual query through the pipeline. Because the first stage of profiling is primarily focused with extracting the *pipeline structure* and not *pipeline performance*, it can be run on an un-tuned deployment.

Once all the sample queries have been evaluated, the Profiler has a set of pipeline graphs, each of which contains some subset of the entire pipeline structure that resulted from a real evaluation of the pipeline. The Profiler takes the union of all the pipeline graphs to form a single pipeline graph which contains every observed path of a query through the pipeline.

During pipeline aggregation, the Planner counts the number of pipeline graphs each model appears in. The Planner can use these frequencies to detect models that are only queried under certain conditions (e.g., a face-recognition model that is only used if an earlier model in

the pipeline detects a person) and allocate proportionally less resources to them, as described in §4.1. Finally, the aggregated pipeline graph is analyzed to ensure it does not contain any cycles. If a cycle is detected, an error is returned to the developer identifying the models contained in the cycle.

4.1 Single Model Profiling

We make the key observation that end-to-end throughput and latency for a pipeline configuration can be accurately estimated as a function of the throughput and latency statistics of the individual models. This property allows the Planner to effectively search this combinatorially large space by exploiting monotonicity (see §5.2) in throughput and latency as a function of configuration control parameters without reprofiling end-to-end pipelines for each point in this multi-dimensional configuration space.

There are three parameters that control model performance: compute resource bundle, maximum batch size, and replication factor. An individual model configuration corresponds to a specific value for each of these parameters. The Profiler characterizes the model's latency and throughput as a function of these parameters.

Both the batch size and the compute resource bundle are mechanisms for exploiting a model's parallelism and therefore offer diminishing returns. The profiler exploits this property in two ways. We restrict our search over resource bundles to exploring different kinds of resources (e.g. different hardware accelerators) but only allocating at most a single instance of each resource type to a model. And we explore batch size in powers of two to efficiently detect the maximum performance to be gained from batching without having to try every batch size in sequence.

After profiling has been completed, any configurations that result in a performance regression along any of the three performance metrics of cost, latency, or throughput, are eliminated. This pruning step primarily removes sub-optimal batch sizes – batch sizes that do not improve throughput over a lower batch size but lead to higher-latency predictions. Because models are inherently stateless and therefore enjoy perfect horizontal scaling we only need to profile under a replication factor of one.

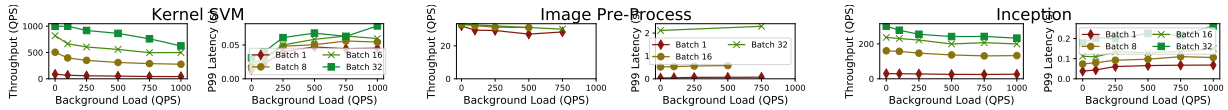
For each configuration, the Planner measures the maximum throughput and 99th percentile latency the model can sustain with the model queue at a steady state. However, the throughput and latency of a model depend on the batch size, which in turn depends on the state of the model queue. Naively profiling models by sending queries at the maximum sustainable rate conflates time spent in the model queue with inference latency. Instead, we employ a two stage approach to profiling that measures throughput and latency separately. In the first stage, the Planner sends requests to the model at a high rate, ensuring that the queue is always larger than the maximum batch size. As a result, the profiler can measure throughput with the assurance that the model is always using the configured batch size. But because the queue is diverging, end-to-end latency at this stage are meaningless.

In the second stage, the planner sends a batch of queries at a time, waiting for the entire previous batch to return before sending the next batch. This eliminates all the pipeline parallelism in InferLine but provides accurate end to end latency measurements without including any time spent in the queue. The Planner repeats this two-stage over-saturation/under-saturation profiling process for each candidate configuration.

While we ensure that models have exclusive access to the CPUs and GPUs allocated to them in their resource bundles there are still sources of kernel level contention (e.g., memory allocation or GPU calls). As a consequence, we also simulate a load on the remaining cores by invoking other models. In Figure 3, we plot the throughput and latency under various levels of simulated load on the remaining cores within a machine. We observe that for some models contention can have a measurable impact on latency and throughput. We therefore profile all models under heavy background load.

5 Planner

The purpose of the Planner is to determine the best possible pipeline configuration subject to the given latency SLO and cost constraints. The Planner operates with profiler-provided per-model information that precisely captures the latency and throughput of a model as a function of its batch size, resource bundle and replication factor. The empirically captured structure of the pipeline, along with individual model profiles is fused into an expression encoded in our declarative intermediate language representation (ILR) called PICL (§5.1). This PICL expression can be used for (1) throughput and latency *estimation*, given a point in the configuration search space, (2) automatically formulating a quadratic programming (QP) problem that mathematically captures the throughput *optimization* problem, (3) capturing the optimization structure of the pipeline and guiding the iterative greedy algorithm through this structure. We introduce InferLine ILR formulation in §5.1 and show how it succinctly encodes pipeline structure, model configurations, conditional probabilities, and the structure and direction of the optimization search procedure. We conclude this section with the discussion of a greedy iterative algorithm leveraging PICL ILR.



(a) Kernel SVM (CPU Only) 1 (b) Image Pre-processing (CPU Only) 2 (c) Inception (V100 GPU) 3

Figure 3: Sensitivity of model profiles to background contention in the host environment. We found that this slowdown occurred inside the model inference computation itself, not outside of the model's container in the network or the queuing system. Each model has exclusive access to its own compute resource bundle, but cannot prevent contention caused by the operating system kernel.

5.1 PICL ILR

Here we introduce our intermediate representation language (ILR) called Pipelined Inference Composition Language (PICL). We design PICL to serve as the intermediate language representation (ILR) for declaratively specifying inference pipeline structure and configuration options associated with its individual models.

We make an observation that searching the space of pipeline configurations can be reduced to bin-packing a space-time rectangle defined by the set of available resources in one dimension and time in the other. Latency SLO bounds the latter, while the provided cost budget constrains the former. We take inspiration from the Space-Time Request Language [39], providing additional support for end-to-end latency SLOs.

Any pipeline graph can be represented as an algebraic expression tree composed of non-leaf operators and leaf operands. For the leafs we introduce a Linear `nCk` primitive that captures model's throughput and latency as a function of its batch size, replication factor, and resource bundle. Note, that PICL is a *declarative* language, not procedural. *It declares a relationship between a (throughput,latency) output tuple and a (replication factor, batch size, resource bundle) input tuple.* As such, the `LnCk` primitive encodes this relationship for each profiled (batch size, resource bundle) input tuple. The effect of the replication factor is captured as the *linear* part of this primitive. We leverage the fact that model throughput scales linearly with the number of replicas. Once each profiled model's configuration choice is specified as an `LnCk` operand they can be composed with PICL operators to encode ensembles (with a min), mutually exclusive choices (with a max), and conditional probabilities (with scale). We provide formal specification of the `LnCk` primitive and the rest of PICL in §5.1.1.

5.1.1 PICL specification

PICL consists of primitives and operators that allow arbitrary composition of subexpressions. Primitives form base expressions, and composite expressions are formed by applying operators to other subexpressions. This results in a natural algebraic tree structure that enables the flow of *value* from its primitives up to the root of the tree, modified by the operators along the way. Primitives always form the leaves of this tree, while operators are always the non-leaf nodes.

- $\texttt{LnCk}(K, dur, v, config = [batch, \vec{b}])$ is an expression that evaluates to $k * v$ if $k \leq K$ replicas were allocated to this model. It establishes a relationship between a configuration tuple (k, batch $\vec{b}$) and the performance tuple ($k * v$, dur) for all replication factors $k \leq K$ without enumerating them! Each replica is $\vec{b}$-sized, so the total cost of this `LnCk` is computed as $k * \vec{b} \cdot \vec{1}$. Visualized, `LnCk` expresses a space-time rectangle, described by a choice of k $\vec{b}$-sized replicas used for duration dur, and associates this elastic rectangle with a value of $(k * v, dur)$. In this paper, $\vec{b} = [cores, gpus]$. Parameter v is a constant that reflects the throughput obtained with configuration $config$. $config$ is extensible, but currently includes the batch size $batch$ and a resource bundle $\vec{b}$ used by the profiler. Each `LnCk` primitive captures exactly one model configuration for exactly one model. Each model in the logical plan produced by the Profiler is associated with at least one `LnCk` primitive.

- $max(e_1, e_2, ...e_n)$ – a max expression that evaluates to the value produced by the maximum-valued child expression. Immediate utility of this operator is evident as multiple model configurations can be grouped by joining them with a max operator. For example, two profiles corresponding to two different batch sizes would result in an expression that looks like this : max(`LnCk` (..., v = 100, config=[batch=32,...]), `LnCk` (..., v = 120, config=[batch=64, ...])). Composing these two options with a max operator, declaratively specifies to the Planner the available options as well as quantifies their absolute throughput value.

- $min(e_1, e_2, ...e_n)$ – a min expression that evaluates to the value produced by the minimum-valued child subexpression. This operator is key to declaratively specifying ensembles of models (Figure 2(a)). For an ensemble of two models m_1 and m_2 an expression would be : $min(\texttt{LnCk}(k_1, v_1 = throughput_1), \texttt{LnCk}(k_2, v_2 = throughput_2))$ and, if satisfied, would evaluate to the $min(v_1, v_2)$.

- $scale(e, f)$ – a scale operator that amplifies the flow of value from its child expression. It's essential for expressing probabilities of taking a particular branch. SIt allows throughput value adjustment on the Forced Flow Law (§4.1).

It is important to note that the invariant of this language is that any subexpression of any PICL expression necessarily evaluates to an output tuple (throughput, latency) that corresponds to the model pipeline subgraph encoded/captured by that subexpression. This is a powerful invariant that simultaneously allows InferLine to use these expressions for automatically generating a QP formulation, using it to evaluate a configuration point in the search space, and guide an iterative algorithm through that search space greedily.

5.1.2 PICL Example: Cascade Pipeline

To illustrate how PICL expressions can be used to represent and optimize pipeline graphs, we implement the cascade pipeline using PICL (Figure 2(c)).

It consists of model m_1, with a possibility of an early exit, and model m_2 reached with probability p_2. The entry and the exit point are included in the figure, but don't need to be captured in the PICL, as they don't contribute to the throughput, cost, or latency we aim to model.

For each model, the batch is specified as a vector $\vec{b} = [1,0]$, indicating that a single replica of this model uses a single CPU core and no GPUs. Maximum external parallelism (constrained by the cost budget) is specified as K The `LnCk` also specifies the throughput v_1^1 associated with model m_1 and attained using batch size bs_1^1. This captures the declaration of a single model configuration as follows: $\texttt{LnCk}(K, dur, v_1^1, bs_1^1, \vec{b} = [1,0])$. We drop dur for convenience of notation.

We create one such `LnCk` for each meaningful configuration we wish to consider. In this example, model m_1 can run with and without a GPU, each with two possible batch sizes, totaling 4 `LnCk` leaves to represent this model. We then attach them to the max operator, which yields the following complete expression for model $m_1 : e_1 =$

$max(\texttt{LnCk}(v_1^1, K, bs_1^1, \vec{b} = [1,0]), \texttt{LnCk}(v_1^2, K, bs_1^2, \vec{b} = [1,0]),$
$\texttt{LnCk}(v_1^3, K, bs_1^1, \vec{b} = [1,1]), \texttt{LnCk}(v_1^4, K, bs_1^2, \vec{b} = [1,1]))$

The second model is a CPU model, has two possible batch sizes, and is reached with probability p_2. We use the *scale* operator to reflect the probabilistic/conditional model access. This yields the following complete PICL expression for $m_2 : e_2 = scale(\frac{1}{p_2}, max($
$\texttt{LnCk}(v_1^1, K, bs_1^1, \vec{b} = [1,0]), \texttt{LnCk}(v_1^1, K, bs_1^1, \vec{b} = [1,0])))$

Finally, we impose the latency SLO with the *window* operator to get the final expression: $e = win(deadline, e) = win(..., max(\texttt{LnCk}(), ...), scale(max(\texttt{LnCk}(), ...)))$.

5.2 Greedy Planner

We describe the Optimizer part of InferLine Planner in this section. The key enabling insight for the greedy optimizer is our observation that model throughput is monotonic in all the three control parameters we consider: batch size, replication factor, and resource bundles. Note that resource bundles can always be ordered such that it is true.

The Profiler (§4) provides the Planner (§5) with a discrete representation of this 4D[1] surface for each of the models in a pipeline. Note that overlaying these convex 4D surfaces (by applying PICL min, max, or sum) creates a convex search space. This means that PICL expressions are convexity-preserving—the key property suitable for a greedy coordinate descent search. Given the discrete nature of the search space, we thus implement an iterative greedy search algorithm that maximizes throughput subject to the specified latency and cost constraints.

Algorithm. At each iteration, the algorithm identifies a throughput bottleneck and alleviates it. To do that, every step of the algorithm sorts all models in the pipeline in the order of increasing adjusted throughput (§4.1). Then the gradient of the model's throughput is computed and a step is taken in the direction of the steepest partial derivative. The cycle is concluded by a constraint check. If the latency or the cost is violated by the update step, we attempt a step along the other partial derivative. If both partial derivatives yield a step that violates SLO or cost constraint, we backtrack, and try the next model in the sorted list. We evaluate this algorithm in §8.

Initialization. Given the monotonicity of the search space, initializing the search algorithm with a feasible solution is trivial. All models are configured with the smallest batch size, bundle, and the number of replicas. We're guaranteed that this configuration is minimal in terms of throughput, latency, and cost. The infeasibility of this configuration implies the infeasibility of the problem, thus serving as the termination condition. The algorithm then proceeds to iteratively improve the solution by taking throughput maximizing steps along one of the three control dimensions.

Termination Condition. Visiting each model at least once constitutes a single full configuration pass through the pipeline. Save for the immediate termination, a full

[1] A choice of a resource bundle makes it 4D, but is easily handled by taking a greedy step to always try GPU models on a GPU and fall back to CPU if need to backtrack. So the order of throughput maximizing steps for a given model are: (a) cpu/gpu bundle choice, (b) highest derivative w.r.t. number of replicas or batch size, (c) second highest derivative.

pipeline pass guarantees existence of a candidate solution. Importantly, the candidate solution forms a subtree $E' \subset E$ of the PICL expression E representing this pipeline. Specifically, each *max* is left with an exactly one chosen `LnCk` attached to it, each `LnCk`—evaluates to a concrete number of replicas k and, by extension, throughput $k * V$. Thus, PICL is a general framework for *expressing*, *evaluating*, and *navigating* the configuration search space either with a solver or greedily.

Constraints. Another enabling insight in the greedy algorithm is the ability to quickly check specified SLO and cost constraints in constant time per each iteration. Indeed, given a candidate solution, which consists of a set of models, each configured with a concrete resource bundle, batch size, and replication factor, we can use the PICL-based estimator by evaluating the PICL expression, given the current algorithm state that consists of a (batch size, replication factor, resource bundle) input tuple for each model in the pipeline. Recall that it evaluates to an output tuple (throughput,latency).

6 System Comparisons

We compare InferLine against a hand-tuned, replicated single process prediction service, as well as Tensorflow Serving - a commercial grade prediction service.

6.1 Single Process Drivers

Today's most common pipeline serving technique treats the entire pipeline as a single black-box model and deploys it within a single process or Docker container. This is equivalent to replacing all of the RPC calls in InferLine with local function calls and embedding the model inference code within the driver program. We refer to this design as the single process driver (SPD).

The coarse-grained configuration control of the SPD design offers a contrast with the fine-grained control afforded by the InferLine architecture. Specifically, because SPD treats a pipeline as a single black box function, it cannot do any per-model configuration. We configure the SPD implementations to use a single batch size, replication factor, and GPU type for the entire pipeline, while InferLine offers the ability to control these parameters on a per-model basis.

It is worth noting that various machine learning frameworks have limited capacity for cross-compatibility, which restricts the flexibility of the SPD approach. For example, some deep learning frameworks, such as TensorFlow, attempt to proactively requisition all available GPU resources available to the process in order to eliminate memory allocation overheads during execution; this makes it difficult to compose models across frameworks using the SPD approach. Furthermore, the depth of a model pipeline is limited by the end-to-end latency constraints of an application. Therefore, as model pipelines grow, developers will be forced to explore methods that afford more parallelism during evaluation in order to meet their latency demands. One consequence of this observation is that wider pipelines substantially complicate the problem of bin-packing SPD replicas onto physical nodes with limited slots for hardware accelerators. More fundamentally, this class of systems lacks the fine-grained control over per-model configuration parameters that yields fine-tuned end-to-end pipeline balance — ideal for maximizing throughput per dollar.

6.1.1 Implementation

The single process driver is implemented using a queue-based request submission and processing scheme in Python. There are two major components of this design: a query generator and a query processor. Given a pre-specified batch size and arrival process consisting of inter-request delays, query generator performs the routine described in algorithm 1. This query generation design is motivated by the need to avoid explicit waiting. On many Unix-based systems, syscall-based sleeps initiated in Python exhibit a significant lack of granularity; guaranteeing sleep duration at the millisecond level is not possible. Additionally, busy waiting is not viable due to performance limitations imposed by Python's Global Interpreter Lock (GIL). We find that the stated procedure is substantially more effective than an approach based on explicit weighting at realizing the inter-request delays specified by the supplied arrival process. Following an

Algorithm 1: SPD query generation

```
Data: ap = arrival process, ap_idx = 0
arrival_window =
  get_time_elapsed_since_last_query_generation();
requests_delay = 0;
while True do
    requests_delay += ap[ap_idx];
    if requests_delay <= arrival_window then
        new_query = create_query();
        request_queue.submit(new_query);
        ap_idx += 1;
    else
        remaining_delay = requests_delay -
          arrival_window;
        ap[ap_idx] = remaining_delay;
        break;
GOTO query processing;
```

iteration of query generation, the SPD control flow proceeds to process queries as in algorithm 2. The *evaluate*

Algorithm 2: SPD query processing

```
Data: bs = batch size
batch_inputs =
  request_queue.get_batch(max_size=bs);
batch_outputs, batch_latency =
  evaluate_inputs(batch_inputs);
GOTO query generation;
```

inputs function is a pipeline-specific batch query evaluator that queries models instantiated in the same process. This evaluator leverages multithreading to perform parallel computations on distinct hardware resources where possible.

6.2 TensorFlow-Serving

The second comparison point is a pipeline manually composed of individual calls to an existing prediction serving system. We use the TensorFlow-Serving (TFS) system to construct this pipeline, as it is the most widely adopted open-source prediction serving system. The TFS comparison highlights the need to perform global reasoning about the end-to-end performance of a pipeline when configuring its constituent models. In particular, without the ability to reason about how changing a single model's configuration will affect the end-to-end latency of the pipeline, one must employ a greedily latency-minimizing approach to ensure that SLOs will be met. Therefore, in order to minimize pipeline latency, batch size is held fixed at a value of one for all models. However, we tune the replication factor on a per-model basis because adding replicas does not impact end-to-end latency. Note that TFS cannot adjust the replication factor to account for models that are conditionally executed and, therefore, do not need to be provisioned for the full workload.

6.2.1 Challenges

Implementing a scalable, high-performance pipeline atop TFS is difficult due to the synchronous query evaluation abstraction provided by the system's user-facing RPC layer. In order to evaluate n queries concurrently, a client must manage n threads - one for each query. This task is complicated by the performance-impacting restrictions imposed Python's GIL; namely, only a single Python thread can execute per quantum.

6.2.2 Implementation

Fortunately, the decision to fix a query batch size of one per model replica due to a lack of global reasoning afforded by TFS substantially simplifies the pipeline implementation: only a single thread is necessary to fully saturate a given replica. Additionally, these per-replica threads spend an appreciable amount of time waiting for the delivery of a prediction response, reducing the aforementioned GIL-related inefficiencies. Nevertheless, based on empirically observed performance degradation when managing more than ten replica threads in a given process, we partition the pool of threads associated with all pipeline model replicas into groups. Each group contains at most ten threads. Pipeline queries are submitted to a global request queue. A master process is responsible for polling this request queue and forwarding queries to the pipeline-appropriate series of model replica queues. This design successfully avoids GIL contention and extracts sufficient parallelism from Tensorflow Serving's RPC implementation.

7 Experimental Setup

To evaluate InferLine we constructed two representative prediction pipelines that span the fundamental model composition patterns. We configure each pipeline with varying cost and latency budgets, and then evaluate the latency SLO attainment and cost-effectiveness measured in queries-per-second-per-dollar (QPSD) under a range of workloads, varying mean throughput and coefficient of variation (CV). We compare the performance of serving each pipeline with InferLine against SPD and TensorFlow serving. In the following, we describe both the prediction pipelines and the comparison points in more detail.

We include two image processing pipelines (Pipeline 1 and Pipeline 2) that take dense, uniformly sized images as input and perform inference using a combination of state-of-the-art convolutional neural networks and classical machine learning models.

The structure of the pipelines were selected to simultaneously reflect realistic application scenarios and include commonly occurring composition patterns such as model sequences, joins, and preprocessing. The models themselves are drawn from a variety of widely used machine learning frameworks to reflect the rich ecosystem of machine learning software in use today, although we primarily use models trained in TensorFlow to compare InferLine's approach to TensorFlow-Serving, which only supports TensorFlow models (see §6). In addition to providing a broad coverage of use cases and types of applications, this also serves to exercise many of the InferLine primitives and operators introduced in §5.

DNN Ensemble. The first pipeline is an example of the ensemble composition pattern. It performs inference in parallel along two separate paths and then aggregates the results of the independent paths together to return a final

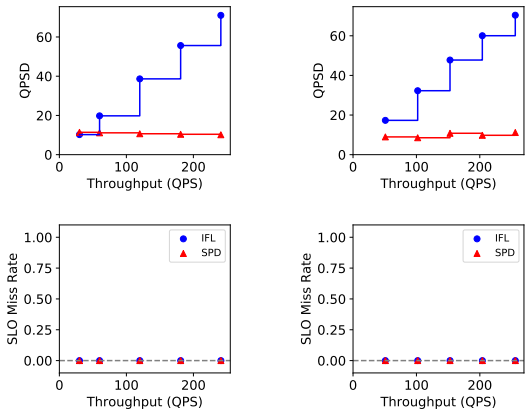
prediction. The first path uses a ResNet152 convolutional neural network model [16] to extract 2048-dimensional features from a 224x224x3 dimensional image. This is a standard technique in machine-learning [28] to extract useful generic features for images. The neural features are subsequently sent to a kernel SVM [9] for binary classification. The second path follows a similar pattern, using an InceptionV3 model [35] to extract 2048-dimensional neural features and a logistic regression model to render a final prediction. All four models in this pipeline were implemented in TensorFlow [36].

Photo Analysis. The second pipeline illustrates a common real-world method of using machine learning. It performs some complex but deterministic pre-processing on input images to transform dirty and mixed format photos into the canonical RGB pixel format that models are typically trained on. It then uses the relatively simple and well-understood AlexNet [21] architecture for classifying the image. The image pre-processing was implemented in the Python Image Library (PIL) and we used the AlexNet model from the PyTorch model zoo.

7.1 Physical Setup

We ran all experiments in a distributed cluster on Amazon EC2. We consider two GPU types in this paper. When using NVIDIA's K80s we use a `p2.8xlarge` instance, which has 32 vCPUs, 8 GPUs, 488.0 GiB of memory and 10Gbps networking all within a single NUMA node. When using NVIDIA's V100s we use a `p3.8xlarge` which has 32 vCPUs, 4 GPUs, 244 GiB of memory, and 10Gbps networking in a single NUMA node. Amazon also offers a `p3.16xlarge` instance which has 8 V100 GPUs, twice as many cores, and two NUMA nodes, but we found that the performance to price ratio of `p3.8xlarge` to be more favorable.

We ran the benchmark client driver on a separate instance for all three systems. For InferLine and TFS, this is the driver program making the individual RPC calls to models, while for SPD this is a batching queue that dispatches a batch of queries at a time to one of the SPD workers. Queries were spread evenly over the workers. We use an `m4.16xlarge` instance for the client, 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 in the experiments. All instances were running Ubuntu 16.04 with Linux Kernel version 4.4.0 and used Amazon's Enhanced Networking on Linux features to utilize the full network capacity.



(a) cv=0.1,SLO=300ms (b) cv=0.1,SLO=500ms

Figure 4: This pipeline consists of a CPU-intensive pre-processing stage followed by a lightweight AlexNet GPU model.

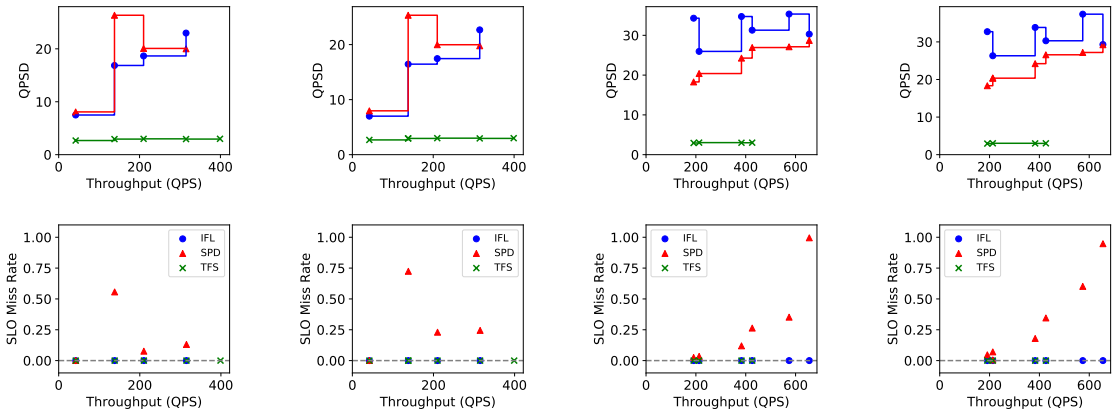
7.2 Workload Setup

We evaluate InferLine on a range of realistic serving workloads. The contents of the queries were randomly sampled with replacement from the validation datasets used to evaluate the component machine learning models in the pipeline. Both pipelines used images from the ImageNet benchmark dataset, although in some cases the model could operate on these images directly (Inception), while in others the image needed to be resized (Resnet152) and transformed (AlexNet) to be compatible.

The workload traces range across different ingest rates and burstiness. We generated the traces by sampling the inter-arrival time for the queries from a Gamma distribution with differing mean (β) to vary the ingest rate, and coefficient of variation (CV) to vary the workload burstiness. When reporting performance on a specific workload as characterized by β and CV, a trace for that workload was generated once and reused across all experiments to provide a more direct comparison of performance.

8 Experimental Evaluation

In the first part of the experimental evaluation, we compare InferLine's performance to the Single Process Driver and TensorFlow-Serving in an end-to-end system comparison. Next, we evaluate InferLine on a series of microbenchmarks that evaluate the sensitivity of both the Planner and the physical execution engine to changes in model, workload, and system behavior.



(a) throughput,cv=0.1,SLO=300ms (b) throughput,cv=4,SLO=300ms (c) throughput,cv=0.1,SLO=500ms (d) throughput,cv=4,SLO=500ms

Figure 5: DNN Ensemble System Comparison: End-to-end performance comparison across all three systems for the DNN Ensemble.

8.1 End-to-end Evaluation

We start with a direct comparison of the performance of all three systems on Pipeline One. InferLine configures throughput maximizing configurations that still meet the specified cost and latency constraints. We compare the maximum throughput that each system can provide under the same cost and latency constraints.

To find the maximum throughput for InferLine a given cost and latency constraint pair, we generate workload traces with a range of mean throughputs (by varying the Gamma distribution β) and then perform a binary search over these traces with the InferLine Planner to find the trace with the highest mean throughput that the Planner determines that InferLine can support.

For SPD and TFS, we first configure the systems to meet the latency and cost constraints. For TFS, this simply means replicating the bottleneck model iteratively until the cost budget is reached. For SPD, we first profile the system under increasing batch sizes to find the maximum batch size the application can support that is still under the latency SLO, then replicating the entire pipeline until the cost budget is reached.

We evaluated two types of pipelines: one balanced Figure 5 and one unbalanced Figure 5 pipeline. We expect our approach to fine-grain configuration control to work best for unbalanced pipelines. We vary the costs and latency SLOs to analyze the behavior of the systems under different application settings. We also varied the burstiness of the workloads, considering workloads with a CV of 0.1 (very little burstiness), CV of 1.0 (moderate burstiness, corresponding to an arrival process described by a Poisson distribution), and a CV of 4.0 (high burstiness). In Figure 4, we find that InferLine significantly outperforms an expertly tuned, monolithic single process driver by 7× on the number of queries it can sustain per second per dollar(QPSD), while simultaneously meeting its 300ms SLO objective. SPD SLO miss rater is higher, as it is provisioned for the mean throughput. For a tighter SLO objective of 200ms, InferLine achieves a 6.3x improvement over SPD with a similarly favorable SLO miss rate.

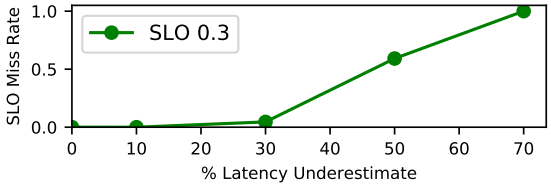


Figure 6: Sensitivity of SLO miss rate to profiled P99 latency.

For a more balanced pipeline that consists of a preprocessing stage followed by a GPU model stage, we find that we still match the throughput performance of the expertly tuned SPD, however we significantly outperform SPD with a 0% SLO miss rate, while SPD lets too many queries miss their SLO. In this pipeline, InferLine and SPD provision the same amount of resources because it's balanced. However, InferLine does so in a way that satisfies the latency SLO.

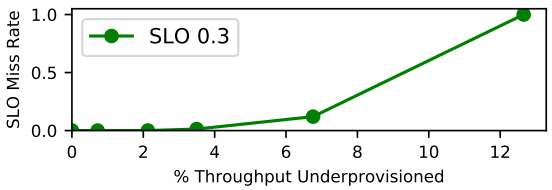


Figure 7: Sensitivity of SLO miss rate to profiled throughput.

8.2 Sensitivity Analysis

The InferLine planner is designed to provision model pipelines to peak capacity while still ensuring that the pipeline can meet its latency objective. It relies on the fidelity of the individual model profiles to do this provisioning safely. We next evaluate the effect of mis-estimated model profiles on the SLO rate. The optimizer relies on both the profiled throughput and latency.

We first evaluate the impact of mis-estimated P99 latency. In Figure 6 we jointly reduced the P99 latency of each model profile in the DNN Ensemble pipeline and re-running the optimizer on the new profiles. Up until the P99 latency is underestimated by 30% *for every model in the pipeline*, the SLO miss rate remains close to zero. This is because in order for the optimizer to change plans based on the p99 latency, it needs to be low enough for the estimator to accept the higher latencies associated with larger batch sizes as being within the latency object. Because increasing the batch size results in a significant increase in end-to-end latency, the optimizer is robust to mis-estimates in P99 latency of up to 20%. By provisioning for P99 latency – a measure of worst-case behavior that is much more pessimistic than the average case of the running system – the profiling itself builds robustness into the optimizer and ensures that transient changes in performance do not affect the SLO miss rate.

In contrast to the P99 latency, throughput is a much less noisy estimator and therefore the Planner is able to provision to the max throughput capacity of a pipeline, maximizing resource utilization, while meeting the latency SLOs. As we see in the Figure 5 and Figure 4, the optimizer properly provisions the pipeline to achieve the SLO. To evaluate the extent to which the optimizer overprovisions the pipeline and as a result increasing the cost unnecessarily, in Figure 7 we increased the load on the pipelin above what the optimizer claimed it could support. Here, we see that as soon as the pipeline is underprovisioned by as little 4%, the SLO miss rate starts to increase, and by the time the pipeline is underprovisioned by 10%, the queues start to diverge, filling up faster than they can be drained, and the SLO miss rate goes to 1.0. This demonstrates that the Planner is provisioning the pipelines just to their peak capacity but not beyond, extracting the maximum value out of the expensive hardware resources.

9 Related Work

There have been a number of recent efforts to study the design of generic prediction serving systems. TensorFlow Serving [38] is a commercial grade prediction serving system primarily designed to support prediction pipelines implemented using the TensorFlow APIs. Unlike InferLine, TensorFlow Serving adopts a more monolithic design with the pipeline orchestration living within a single process. As a consequence, TensorFlow Serving is able to introduce important performance optimizations like operator fusion across computation stages to reduce coordination between the CPU and GPU. More recently, Baylor et al. [5] extended TensorFlow to support the larger machine learning life-cycle. They added more support for basic data transformations and multitenancy in the serving system and found that model loading can interfere with low-latency predictions and used large thread pools to help alleviate this contention resulting in an order-of-magnitude reduction in the P99 serving latency.

Alternatively, Clipper [12] adopts a more distributed design, similar to InferLine. Like InferLine, each model in Clipper is placed in a separate Docker container. This design improves isolation across individual models at the expense of greater data movement. In addition, Clipper does not directly support prediction pipelines.

The Zhang et al. [43] explored the design of a streaming video processing system in the VideoStorm project. VideoStorm shares several common goals and architectural decisions with InferLine. Similar to InferLine, VideoStorm adopts a distributed design with pipeline operators provisioned across compute nodes. VideoStorm also focus on optimizing the combinatorial search space of hardware and model configurations. However, rather than adjusting resource type and batching, VideoStorm focuses on replication, frame skipping, and re-sizing to trade-off throughput and accuracy. Like InferLine, VideoStorm also introduces an offline profiling stage to estimate the trade-off space for individual models and then uses a greedy algorithm similar to the one described in §5.2 to search the configuration space. However, unlike InferLine VideoStorm focuses on provisioning a fixed pool of resources across a changing set of queries.

The PICL intermediate representation used in InferLine was inspired by TetriSched's [39] space-time request language (STRL). We augment PICL with additional operators to support end-to-end latency SLO specification and partial order constraints. PICL's main primitive, `LnCk` deviates from STRL, though, as it supports arbitrary resource bundles. The main contribution of PICL is demon-

strating how common inference pipelines, including with conditionals, can be declaratively captured and how this ILR representation (a) generalizes and (b) formalizes the search space for both the optimizer and the greedy heuristic we propose.

A large body of prior work leveraged profiling information to inform better scheduling. Workflow-aware scheduling, however, is a relatively recent phenomenon, including SLURM-integrated HPC scheduler published in 2017 [29] and Morpheus [19]. In this paper, we concretely exploit the compute-intensive and side-effect free nature of machine learning models to estimate end-to-end pipeline performance given individual model profiles. Note that we do not and cannot leverage prior runs of the *entire* pipeline to estimate its throughput and latency, as they change with different latency SLO and cost constraints. Rather, we use individual model performance estimates PICL to extrapolate the pipeline performance.

10 Conclusion

In this paper we exploit unique machine learning model properties to address the challenges of configuring and provisioning inference pipelines. The key insight is that fine-grain control of per-model configurations can be achieved with a greedy search policy that leverages the monotonicity of a model's throughput as a function of its internal and external replication factors. We instantiate these ideas in a system prototype, InferLine, which builds on three principal contributions. First, we develop a pipelined inference composition language (PICL) —the intermediate language representation for arbitrary model pipelines. PICL enables a declarative specification of a large configuration space with a handful of arbitrarily composable primitives and operators. Second, we introduce a greedy algorithm used to efficiently navigate this combinatorial configuration space, formalized by PICL and produce throughput optimal pipeline configurations that satisfy latency and cost constraints. Third, we build an optimizer that extracts per-model profiling information and pipeline structure empirically. As a result, we achieve 6x improvement in cost for the same throughput and latency objectives over single process driver pipeline. InferLine straddles a wide opportunity gap in the throughput/cost trade-off space and efficiently gains throughput per dollar by greedily maintaining a balanced end-to-end pipeline through its fine-grain control of each model's internal and external paralellism.

References

[1] F. Akgul. *ZeroMQ*. Packt Publishing, 2013.

[2] J. Andreas, M. Rohrbach, T. Darrell, and D. Klein. Deep compositional question answering with neural module networks. *CoRR*, abs/1511.02799, 2015.

[3] A. Angelova, A. Krizhevsky, V. Vanhoucke, A. S. Ogale, and D. Ferguson. Real-Time Pedestrian Detection with Deep Network Cascades. *BMVC*, pages 32.1–32.12, 2015.

[4] 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.

[5] D. Baylor, E. Breck, H.-T. Cheng, N. Fiedel, C. Y. Foo, Z. Haque, S. Haykal, M. Ispir, V. Jain, L. Koc, C. Y. Koo, L. Lew, C. Mewald, A. N. Modi, N. Polyzotis, S. Ramesh, S. Roy, S. E. Whang, M. Wicke, J. Wilkiewicz, X. Zhang, and M. Zinkevich. 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, pages 1387–1395. ACM, 2017.

[6] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13:281–305, Feb. 2012.

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

[8] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Statistics/Probability Series. Wadsworth Publishing Company, Belmont, California, U.S.A., 1984.

[9] C. J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Min. Knowl. Discov.*, 2(2):121–167, June 1998.

[10] Caffe Model Zoo. `https://github.com/BVLC/caffe/wiki/Model-Zoo`.

[11] D. Crankshaw. Clipper, oct 2016.

[12] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, Boston, MA, 2017. USENIX Association.

[13] C. Desrochers. A fast lock-free queue for c++, 2013.

[14] 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.

[15] J. Guan, Y. Liu, Q. Liu, and J. Peng. Energy-efficient Amortized Inference with Cascaded Deep Classifiers. *arXiv.org*, Oct. 2017.

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

[17] M. J. Jones and P. Viola. *Robust real-time object detection*. Workshop on Statistical and Computational Theories . . . , 2001.

[18] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, 2017.

[19] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. n. Goiri, S. Krishnan, J. Kulkarni, and S. Rao. Morpheus: Towards automated slos for enterprise clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 117–134, Berkeley, CA, USA, 2016. USENIX Association.

[20] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, Jan. 2018.

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

[22] L. Li, W. Chu, J. Langford, and R. E. Schapire. A contextual-bandit approach to personalized news article recommendation. In *WWW*, 2010.

[23] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. *ArXiv e-prints*, Jan. 2018.

[24] M. Malinowski, M. Rohrbach, and M. Fritz. Ask your neurons: A neural-based approach to answering questions about images. *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1–9, 2015.

[25] M. McGill and P. Perona. Deciding How to Decide: Dynamic Routing in Artificial Neural Networks. *arXiv.org*, Mar. 2017.

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

[27] N. Polyzotis, S. Roy, S. E. Whang, and M. Zinkevich. Data management challenges in production machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1723–1726. ACM, 2017.

[28] A. S. Razavian, H. Azizpour, J. Sullivan, and S. Carlsson. Cnn features off-the-shelf: An astounding baseline for recognition. In *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, CVPRW '14, pages 512–519, Washington, DC, USA, 2014. IEEE Computer Society.

[29] G. P. Rodrigo, E. Elmroth, P.-O. Östberg, and L. Ramakrishnan. Enabling workflow-aware scheduling on hpc systems. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '17, pages 3–14, New York, NY, USA, 2017. ACM.

[30] R. E. Schapire. The strength of weak learnability. *Mach. Learn.*, 5(2):197–227, July 1990.

[31] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, and M. Young. Machine learning: The high interest credit card of technical debt. In *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*, 2014.

[32] E. Sparks. *End-to-End Large Scale Machine Learning with KeystoneML*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2016.

[33] E. R. Sparks, A. Talwalkar, D. Haas, M. J. Franklin, M. I. Jordan, and T. Kraska. Automating model search for large scale machine learning. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 368–380, New York, NY, USA, 2015. ACM.

[34] Y. Sun, X. Wang, and X. Tang. Deep Convolutional Network Cascade for Facial Point Detection. *CVPR*, pages 3476–3483, 2013.

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

[36] TensorFlow. `https://www.tensorflow.org`.

[37] TensorFlow Models. `https://github.com/tensorflow/models`.

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

[39] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger. TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proc. of the 11th European Conference on Computer Systems*, EuroSys '16. ACM, Apr 2016.

[40] X. Wang, Y. Luo, D. Crankshaw, A. Tumanov, and J. E. Gonzalez. IDK cascades: Fast deep learning by learning not to overthink. *CoRR*, abs/1706.00885, 2017.

[41] Y. Yang, D.-C. Zhan, Y. Fan, Y. Jiang, and Z.-H. Zhou. Deep learning for fixed model reuse, 2017.

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

[43] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman. Live video analytics at scale with approximation and delay-tolerance. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 377–392, Boston, MA, 2017. USENIX Association.