Towards High-Performance Prediction Serving Systems

Yunseong Lee
Seoul National University
Seoul, South Korea
yunseong@snu.ac.kr

Alberto Scolari
Politecnico di Milano
Milano, Italy
alberto.scolari@polimi.it

Matteo Interlandi
Microsoft
Redmond, WA
mainterl@microsoft.com

Markus Weimer
Microsoft
Redmond, WA
mweimer@microsoft.com

Byung-Gon Chun
Seoul National University
Seoul, South Korea
bgchun@snu.ac.kr

Abstract

Machine Learning models are often composed of sequences of transformations. While this design makes it easy to decompose and efficiently execute single model components at training time, predictions require low latency and high-performance predictability whereby end-to-end and multi-model runtime optimizations are needed to meet such goals. This paper sheds some light on the problem by introducing a new system design for high-performance prediction serving. We report some preliminary results showing how our system design is able to improve performance over several dimensions with respect to current state-of-the-art approaches.

1 Introduction

Many Machine Learning (ML) frameworks such as Google TensorFlow [3], Facebook Caffe2 [2], Scikit-learn [8], or Microsoft's Internal ML Toolkit (IMLT) allow data scientists to declaratively author sequences of transformations to train models from large-scale multi-dimensional input datasets. The sequences internally are represented as Directed Acyclic Graphs (DAGs) of operators comprising data transformations and featurizers (e.g., string tokenization, hashing, etc.), and ML models (e.g., Neural networks, Linear models, etc.).[1] Figure 1a shows an example DAG for text analysis whereby input sentences are classified according to the expressed sentiment.

When trained DAGs are served for prediction, the full set of operators is deployed altogether to massage and featurize the raw input data points before ML model scoring. Training and prediction DAGs have however different system characteristics: for instance ML models at training time have to scale over large datasets, while, once trained, they can behave as other regular featurizers and data transformations; furthermore, prediction DAGs are often surfaced for direct users' access and therefore require low latency, high throughput, and high predictability. Specifically, prediction systems have three main performance requirements in order to be usable by consumers and be profitable for ML-as-a-service providers: (**R1**) *latency has to be minimal*—in the order of milliseconds—and *predictable* because scoring is often one segment in more complex services (e.g., smart phone or web applications) which potentially provide a Service Level Agreement (SLA); (**R2**) *small resource usage*—such as memory and CPU—to save operational costs; and (**R3**) *high throughput* to handle as many concurrent requests as possible. Existing prediction serving systems, such as Clipper [1]

[1] IMLT implements dozens of pre-defined operators and ML algorithms; IMLT is extensible and users implement their own custom operators and ML algorithms in C#.

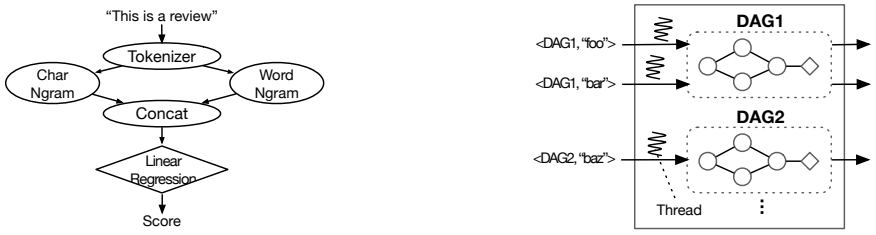
31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

and IMLT itself, focus mainly on ease of deployment, whereby model DAGs are considered as *black box* and therefore only certain "DAG-agnostic" set of optimizations such as caching and buffering are possible. The black box approach works well when the models to be served are small in number, while our experiments show that there is a limit to the number of models that can be served on a single machine (related to **R2**) without loosing on throughput and latency (requirements **R1** and **R3**). Section 2 contains a more detailed description of the limitations of existing systems.

To address the aforementioned performance requirements, in this paper we propose a system for scoring models authored in IMLT[2], borrowing ideas from the database and systems community. Starting from the observation that trained DAGs often share operators and parameters (such as weights and dictionaries used within operators), we introduce a *Parameter Store* where operators' parameters are consolidated and shared in order to minimize memory footprint (**R2**). A *logical representation* of the DAG of operators composing the model is saved along with the related parameter mappings. To address **R1**, logical representations of models are compiled into *stages*: single scheduling units where multiple operators are executed together to reduce overheads such as memory allocation and chains of virtual function invocations. Lastly, event-based scheduling of stages [10] is used to increase throughput through DAGs (**R3**) while maintaining target latency and memory footprint.

Using 300 DAGs used internally at Microsoft, and implementing different versions of the sentiment analysis models of Figure 1a, we show the impact of the above design choices with respect to baseline IMLT. Our experiments mirror production-like settings where many customers train sentiment models (with similar structures) on their data. Compared to IMLT, we are able to improve the performance over different dimensions; specifically, we improve the memory footprint by 43.1 times and reduce the latency by up to 87 times.

All the experiments reported in the paper were carried out on a Windows 10 machine with 2 $\times$ 8-core Intel Xeon CPU E5-2620 v4 processors at 2.10GHz and 32GB of RAM.



(a) A DAG for sentimental analysis from text inputs consisting of operators for featurization (ellipses), followed by a ML model (diamond).

(b) Each DAG is compiled as a chain of virtual function calls. When a prediction request is issued, a thread is dispatched to execute the chain as a single (black box) function call.

Figure 1: An example of a model DAG and how existing systems handle the prediction requests.

2 Limitations of State-of-the-Art Prediction Serving Systems

Nowadays, several "intelligent" services such as Microsoft Cortana speech recognition, Netflix movie recommender or Gmail spam detector depend on ML model scoring capabilities and are currently experiencing a growing demand. This fosters demand for research in prediction serving systems in a cloud setting, where trained models coming from data science experts are operationalized.

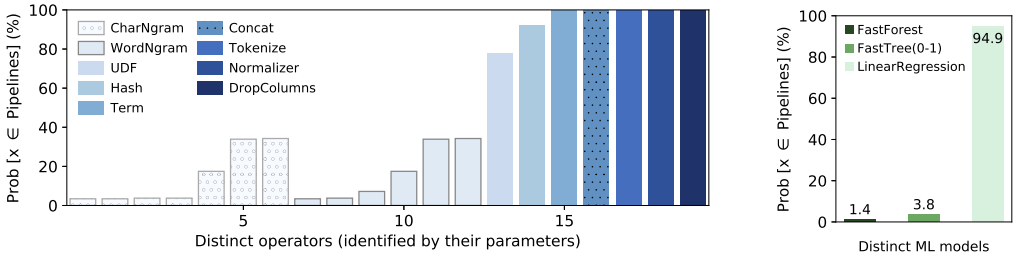
Data scientists prefer to use high-level declarative tools such as IMLT (or TensorFlow and Scikit-learn) for better productivity and easy operationalization. IMLT, as other systems, aims to minimize the overhead of deploying trained DAGs in production by serving trained DAGs into (black box) *containers*[3]. This design obfuscates the semantics of each served model and prevents the system from controlling and optimizing the internal execution of the DAG and its operators. Therefore, under this approach, there is no principled way neither for sharing optimizations between DAGs, nor to improve DAGs execution end-to-end. As depicted in Figure 1b, inside each container, model DAGs are compiled Just-In-Time (JIT) as a chain of virtual functions calls; invoking the function chain

[2]IMLT is a C# library that runs on a managed runtime (e.g., garbage collection and Just-In-Time (JIT) compilation). Unmanaged code can be employed to speed up processing when possible.

[3]Note that while TensorFlow Serving [4] is slightly more flexible since users are allowed to split model DAGs and serve them into different containers (called *servables*), this process is manual and not automatic.

(e.g., `predict()`) returns the result of the prediction, by pulling the input through the operators in the call chain, similarly to the well-known Volcano-style iterator model of databases [5]. To optimize the performance, IMLT (and systems such as Clipper among others [1]) apply techniques such as handling multiple requests in batches, and caching the results of the prediction if some queries are frequently issued for the same DAG. These techniques assume no knowledge and no control over the DAG, and are unaware of its internal structure. However, if we consider a large-scale service where the prediction requests are made over thousands of models (e.g., a web-service with personalized ML models), we observe that existing systems can hardly achieve the performance goals mentioned in Section 1 for the following reasons:

Memory waste: Containerization of DAGs disallows any sharing of resources between DAGs, therefore only a few (tens of) models can be served per machine (violation of **R2**). Conversely, ML frameworks such as IMLT have a known set of operators to start with, and models trained over similar datasets have a high likelihood of also sharing parameters. For example, IMLT suggests default training pipelines to users given a task and a dataset, which leads to many DAGs with similar structure and common objects and parameters such as dictionaries. To better illustrate such scenario, we pick a sentiment analysis task with 300 different versions of the DAG. Figure 2 shows how many different (parametrized) operators are used, and how often are they used within the 300 DAGs. Many operators can be shared between DAGs, therefore, allowing more aggressive packing of models: 5 operators are present in all DAGs, whereas the remaining are used from 40% to 90% of DAGs with only a few DAGs having peculiar operators. This suggests us that the resource utilization of current black box approaches can be largely improved.



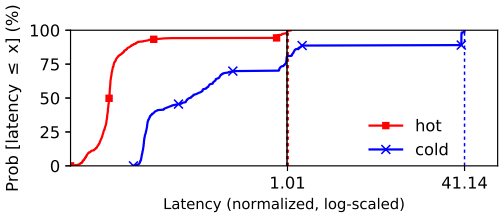
(a) Ngram operators are categorized into 2 groups, each of which has multiple versions (i.e., operators with different parameters). Operators in the same group and the same version can be shared by multiple DAGs.

(b) 3 different type of ML models are used, each of which has distinct parameters.

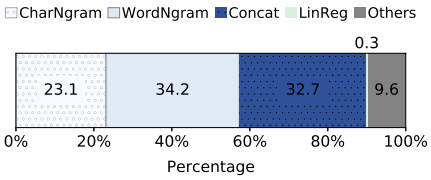
Figure 2: Probability for operators/models to be the same between the 300 different DAGs for sentimental analysis trained on similar datasets.

Lazy materialization: IMLT employs a pull-based execution model that lazily materializes input vectors, and tries to reuse existing data buffers for intermediate transformations. This largely decreases the memory footprint and the pressure on garbage collection at training time. Conversely, this design forces memory allocation along the data path, therefore making latency of predictions suboptimal and hard to predict. Additionally, DAGs are typically authored in a high-level language with generics. At prediction time, IMLT deploys the DAGs as in the training phase, which requires reflection for type inference and JIT compilation. In general, the above problems result in difficulties in providing meaningful SLAs by ML-as-a-service providers (violation of **R1**). Figure 3a describes this situation, where the performance of *hot* predictions over a DAG with memory already allocated and JIT-compiled code is more than an order of magnitude faster then the *cold* version for the same DAG.

Coarse-grained scheduling: Scheduling CPU resources carefully is essential to serve highly concurrent requests. Under the black box approach: (1) a thread pool can be used to serve multiple concurrent requests to the same DAG; (2) for each request, one thread is responsible for the execution of a full DAG sequentially, where one operator is active at each point in time; (3) shared operators might be instantiated and evaluated multiple times independently; and (4) when single operators are multi-threaded, resource contention situations can arise because of poor coordination between (independent) instantiations of the same operators. Furthermore, DAGs are composed of operators with different performance characteristics as depicted in Figure 3b where the latency breakdown of the sentiment analysis DAG of Figure 1a is presented. These considerations lead us to the conclusion that in order to achieve better throughput (requirement **R3**), better scheduling decisions have to be



(a) Average CDF of latency of prediction requests of 300 DAGs; we denote the first execution as *cold*, the subsequents as *hot*. The solid and dotted vertical lines mark the 99% pctl. The plot is normalized over the latency goal (black vertical line).



(b) Latency breakdown of a sentiment analysis DAG: each frame represents the relative wall clock time spent on an operator.

Figure 3: Execution of single pipelines in IMLT.

implemented at the (shared) operator level instead of at the DAG level. To further emphasize this point, Figure 4 shows how, due to resource contention, throughput decreases in current IMLT as we load more DAGs for prediction.

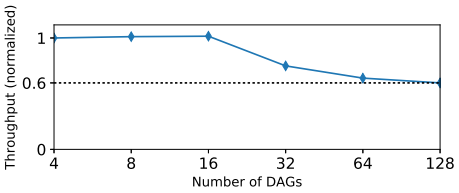


Figure 4: Fixed the total amount of threads, as the number of concurrently served DAGs increases, resources become saturated and the throughput (normalized by the $y|_{x=4}$) starts to drop ($x \geq 16$).

Now that we have highlighted several inefficiencies gripping the current prediction serving systems, we are going to introduce our system design.

3 System Design

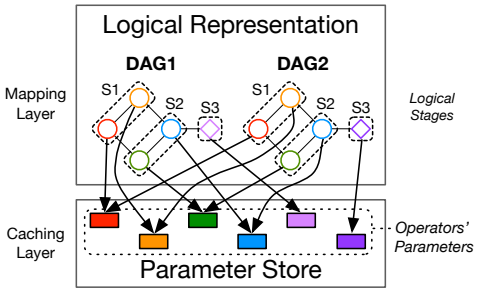
Based on the observations of Section 2, we argue that the three system requirements **R1**, **R2**, and **R3** can be met if we optimize the execution of prediction both horizontally *end-to-end* and vertically *among multiple model DAGs*.

End-to-end Optimizations: The operationalization of models for prediction should focus on execution units making optimal decisions on how data is processed while maintaining low and predictable latency (**R1**). Such execution units should: (1) avoid memory allocation on the data path; (2) avoid creating separate routines per operator as possible, which are sensible to costly branch mis-prediction and poor data locality [7]; and (3) avoid reflection and JIT compilation at prediction time.

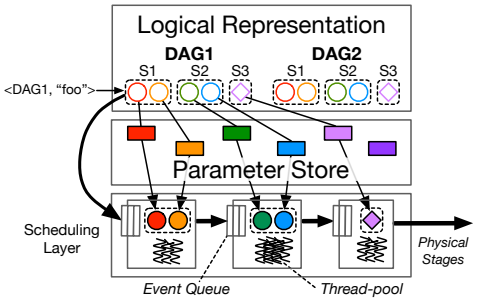
Multi-model Optimizations: To take full advantage of the fact that pipelines often use similar operators and parameters (Figure 2), shareable components have to be uniquely stored in memory and reused as much as possible to achieve optimal memory usage (**R2**). Similarly, scheduling units should be shared at run-time and resources properly managed, such that multiple prediction requests can be evaluated in a pipelined fashion (**R3**).

Following the above guidelines, we have designed a prototypical prediction system composed of the following layers: a *caching layer* where operators and parameters are globally maintained into a *parameter store* and shared among DAGs; a *mapping layer* where a *logical representation* of operators composing DAGs, and related parameters, is kept. Logical representations are analyzed and compiled Ahead-Of-Time (AOT) into efficient physical units called *stages* where memory resources and threads are pooled. Finally, a *scheduling layer* is in charge of the execution of each stage using an event-based approach, where prediction requests are pushed to each stage composing the model. Figure 5 pictorially summarizes the above description; note that only the latter part is executed at prediction time, whereas the parameters and logical-to-physical mapping are computed offline.

Next, we will describe each layer composing our envisioned high-performance prediction system.



(a) Before serving predictions, DAGs are converted to sequences of stages. Operators' parameters are cached into the Parameter Store. The Logical Representation keeps all the mapping information above.



(b) When a prediction request is issued, physical stages are assembled from the Logical Representation. Each stage is composed of the parameters fetched from the Parameter Store, an event queue, and a thread-pool.

Figure 5: System design optimized for prediction serving.

3.1 Caching Layer

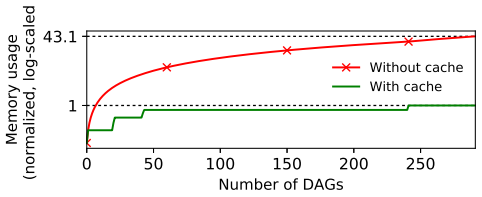


Figure 6: Cumulative memory usage of the model DAGs with and without caching layer.

The Caching Layer design is based on the insights of Figure 2: since many DAGs have similar structures, sharing operators and, when possible, also operators' state (parameters) can considerably improve memory footprint, and consequently the number of predictions served per machine. An example is language dictionaries used for input text featurization, which are often in common among many models and use a relatively large amount of memory.

The Parameter Store is populated offline: when a new model DAG is deployed, the operators involved and their parameters are identified; new parameters are kept in the Parameter Store, while parameters that already exist are ignored and the DAG is rewritten to reuse the previously loaded one. The caching layer enables considerable memory savings, represented in Figure 6: for the 300 example DAGs we analyzed, the Caching Layer reduced the memory consumption by a factor of 43.1x.

3.2 Mapping Layer

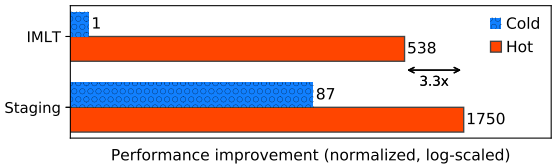


Figure 7: Performance improvement (in terms of latency, higher is better) to execute a single DAG with the staging approach.

While the Parameter Store is populated, the mapping layer builds a logical representation of each input model DAG composed of operators' metadata (e.g., type) and links to related parameters (state) in the Caching Layer. Offline, the logical representation is AOT-compiled into parameterizable execution units called stages.[4] Inside each stage, (logical) operators are fused together when possible to improve memory locality and end-to-end performance. For the moment, we use simple rules to compile set of operators into stages; in the future, we will explore how further optimizations can be

[4]Our prototype currently applies heuristics in compiling stages into optimized code; automatic compilation with general rules is in progress.

added to the compilation phase. Each stage is designed so that no memory allocation occurs along the data path: when instantiated (by the Scheduling Layer described next), each stage is dynamically fed with a set of pre-allocated buffers and the model-specific parameters stored in the Caching Layer.

By compiling operators into stages and by sharing common state, the Mapping Layer is able to obtain a considerable reduction in latency as shown in Figure 7: namely a 87 times speedup in latency for the *cold* case and a 3.3 times speedup for the *hot* case. This design and related optimizations are currently limited to DAGs written in IMLT. Our goal is, however, to target unified model formats such as [6]; this will allow us to apply the discussed techniques to models from other ML frameworks.

3.3 Scheduling Layer

Once model DAGs are assembled and compiled into stages (offline), they are deployed for execution in an environment where they share resources with other DAGs. Therefore, scheduling them appropriately is essential to ensure scalability and optimal machine utilization while guaranteeing the performance requirements.

The Scheduling Layer coordinates the execution of multiple stages via an event-based scheduling mechanism similar to SEDA [10]: each stage is equipped with an input buffer and a thread pool; intermediate results are wrapped as events that are then routed through the proper set of stages together with related parameters (as shown in Figure 5b). Benefits of this mechanism are that stages can now be pipelined, and the Scheduling Layer can assign more resources to slow stages/operators (as from Figure 3b). For further optimization, though not added yet, orthogonal techniques such as batching at the level of stages or DAGs, can be employed as in other prediction serving systems.

The drawback of this approach is that overheads due to buffering and context switching can be introduced on the data path. Such overheads are, however, related to the system load and therefore, controllable by the scheduler. Exploring this trade-off is in progress.

4 Conclusion and Future Work

Inspired by the growth of ML applications and ML-as-a-service platforms, this paper identified the key requirements for ML prediction-serving systems, namely *low latency*, *high throughput*, and *high resource utilization*, which existing systems fall short in guaranteeing as they focus on the ease of deployment rather than on optimizing the model execution. Conversely, this work focuses on laying down optimizations for the end-to-end execution of prediction DAGs by sharing parts of the execution pipeline through pre-optimized stages and through the shared model state. To achieve this result, we designed three layers, namely the Caching layer, the Mapping layer and the Scheduling layer, which inspect each incoming DAG to identify shared and exclusive state and allow optimized execution on the available resources. Experiments with production-like DAGs show the validity of our approach in achieving an optimized execution.

As a future work, we are considering to expand the Scheduling layer over heterogeneous hardware, augmented with GPUs and FPGAs for a more efficient and predictable computation. Recent work [9] has started exploring the acceleration of ML DAGs on FPGA, highlighting potential gains of employing such accelerators, whose characteristics are well suited to some ML operators. Therefore, in a heterogeneous system, the Scheduling layer can be augmented with FPGA control and decide to schedule some stages on such resources. In this scenario, the Scheduling layer must take into account two major costs. The first cost, which incurs only at the beginning, is programming the FPGA logic with the stage, which is in the order of few seconds; this cost must be accurately accounted, as it may impact the latency of the whole system. The second cost is the data movement cost, as current FPGA chips are accessible only through a PCIe bus with a latency in the order of microseconds and good performance only in case of sequential memory accesses. Therefore, batching is essential in this scenario to achieve good performance through an FPGA-based accelerator.

Acknowledgments

Yunseong Lee and Byung-Gon Chun were partly supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No.2017-0-01772, Development of QA systems for Video Story Understanding to pass the Video Turing Test).

References

[1] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. In *NSDI*, 2017.

[2] Facebook. Caffe2, 2017.

[3] Google. TensorFlow, 2016.

[4] Google. TensorFlow serving, 2016.

[5] G. Graefe. Volcano: An extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng.*, 6(1):120–135, Feb. 1994.

[6] Microsoft and Facebook. Open Neural Network Exchange (ONNX), 2017.

[7] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011.

[8] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, Nov. 2011.

[9] A. Scolari, Y. Lee, M. Weimer, and M. Interlandi. Towards accelerating generic machine learning prediction pipelines. In *IEEE ICCD*, 2017.

[10] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *SOSP*, 2001.