Towards High-Performance Prediction Serving Systems

Yunseong Lee
Seoul National University
yunseong@snu.ac.kr

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

Matteo Interlandi
Microsoft
mainterl@microsoft.com

Markus Weimer
Microsoft
mweimer@microsoft.com

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

1 INTRODUCTION

Many Machine Learning (ML) frameworks such as Google TensorFlow [3], Facebook Caffe2 [2], Scikit-learn [5], 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]

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] 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**).

To address the aforementioned performance requirements, in this paper we sketch the design of a system, currently under development at Microsoft, 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 [6] is used to increase throughput through DAGs (**R3**) while maintaining target latency and memory footprint.

Initial results are encouraging: over 300 DAGs used internally at Microsoft, compared to IMLT our prototype is able to improve the memory footprint by 43.1× and reduce the latency by up to 87×.

2 INITIAL DESIGN AND EVALUATION

Based on the above observations, 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.*

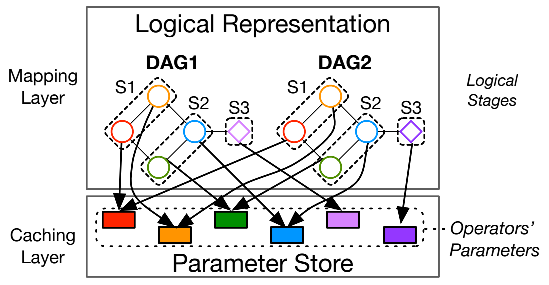
End-to-end Optimizations: The operationalization of models for prediction should focus on execution units making optimal decisions on how data is processed while maintaining low and predictable latency (**R1**). Such execution units should: (1) avoid memory allocation on the data path; (2) avoid creating separate routines per operator, which are sensible to costly branch mis-prediction and poor data locality [4]; and (3) avoid reflection and JIT compilation.

Multi-model Optimizations: ML frameworks such as IMLT have a known set of operators, and models trained over similar datasets have a high likelihood of also sharing parameters. To take full advantage of this, 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**).

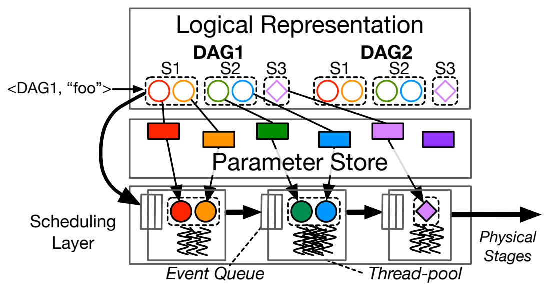
[1]IMLT implements dozens of pre-defined operators and ML algorithms; IMLT is extensible and users implement their own custom operators and ML algorithms in C#.

[2]IMLT is a C# library that runs on a managed runtime (e.g., garbage collection and Just-In-Time (JIT) compilation).

SysML, Feb 2018, Stanford, CA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn



(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) At prediction time, 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 1: System design optimized for prediction serving.

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* (PS) and shared among DAGs; a *Mapping Layer* where a *Logical Representation* (LR) 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. Figure 1 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.

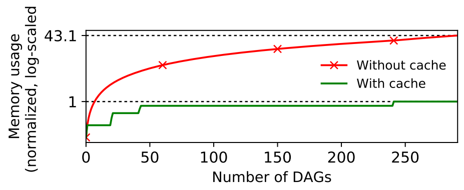


Figure 2: Cumulative memory usage of the model DAGs with and without caching layer.

Caching Layer: The Caching Layer design is based on the insights that 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 PS is populated offline: when a new model DAG is deployed, the operators involved and their parameters are identified; new parameters are kept in the PS, 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 2: for the 300 example DAGs we analyzed, the Caching Layer reduced the memory consumption by 43.1×.

Mapping Layer: 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 stages. Inside each stage, (logical)

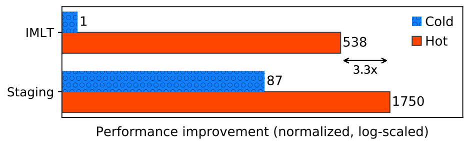


Figure 3: Performance improvement (in terms of latency, higher is better) to execute a single DAG.

operators are fused together when possible to improve memory locality and end-to-end performance. Each stage is designed so that no memory allocation occurs along the data path: when instantiated, 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 3: namely a 87× speedup in latency for the *cold* case and a 3.3× speedup for the *hot* case.

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. The Scheduling Layer coordinates the execution of multiple stages via an event-based scheduling mechanism similar to SEDA [6]: 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 1b). Benefits of this mechanism are that the Scheduling Layer can assign more resources to slow stages/operators. Orthogonal techniques such as batching at the level of stages or DAGs, can be also 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.

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 and No.R0126-17-1093, (SW Star Lab) Development of a Unified High-Performance Stack for Diverse Big Data Analytics).

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

[5] 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.

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