Evaluating Model Serving Strategies over Streaming Data

[†]Sonia Horchidan sfhor@kth.se
[*]Emmanouil Kritharakis ekrithar@bu.edu
[*]Vasiliki Kalavri vkalavri@bu.edu
[†]Paris Carbone parisc@kth.se
[†]KTH Royal Institute Of Technology [*]Boston University

ABSTRACT

We present the first performance evaluation study of model serving integration tools in stream processing frameworks. Using Apache Flink as a representative stream processing system, we evaluate alternative Deep Learning serving pipelines for image classification. Our performance evaluation considers both the case of embedded use of Machine Learning libraries within stream tasks and that of external serving via Remote Procedure Calls. The results indicate superior throughput and scalability for pipelines that make use of embedded libraries to serve pre-trained models. Whereas, latency can vary across strategies, with external serving even achieving lower latency when network conditions are optimal due to better specialized use of underlying hardware. We discuss our findings and provide further motivating arguments towards research in the area of ML-native data streaming engines in the future.

CCS CONCEPTS

• **Information systems** → **Data streams**; • **Computing methodologies** → **Machine learning**.

KEYWORDS

data streams, machine learning inference

ACM Reference Format:
Sonia Horchidan, Emmanouil Kritharakis, Vasiliki Kalavri, Paris Carbone. 2022. Evaluating Model Serving Strategies over Streaming Data. In *Data Management for End-to-End Machine Learning (DEEM'22), June 12, 2022, Philadelphia, PA, USA*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3533028.3533308

1 INTRODUCTION

Stream processing systems like Apache Flink [7] and Spark Streaming [17] enable continuous analytics on data streams and facilitate applications with low-latency requirements, such as anomaly detection, recommendations, and fault diagnosis in large data centers. Even though many streaming applications rely on Deep Learning (DL) models to make predictions, modern stream processing systems provide limited support for model serving. Stream processing systems are traditionally written in JVM languages and have no support for GPUs, which makes them a hostile environment for handling DL models. On the other hand, model serving frameworks, like TorchServe [5] or TensorFlow Serving [12], are ill-suited for building and operating end-to-end streaming analytics pipelines due to the lack of basic event-based semantics, such as event-time windows, and exactly-once processing guarantees. As a result, combining ML and business logic necessitates manual configuration.

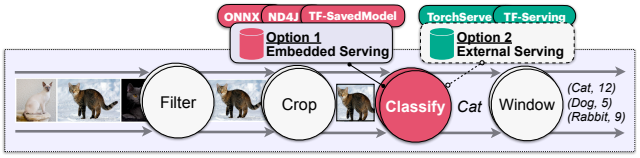

Figure 1: Apache Flink pipeline that integrates a DL classification operator. The workflow features data engineering tasks (e.g., filtering), and post-inference steps (e.g., time windows).

Currently, there exist two alternative options for developers to fuse general event-based logic and ML: (i) use an interoperability library to load a pre-trained model into a streaming operator's memory (Fig. 1, Option 1) or (ii) deploy a model serving system alongside the stream processor and perform inference via RPC (Fig. 1, Option 2). Despite advice offered by vendors and experience reports by industry experts [10, 13, 14], the performance characteristics and trade-offs of these two architectures are not well understood due to a lack of a quantitative study to support the claims [16].

This paper is a first step towards a systematic performance study of model serving integration in modern stream processing systems. We select Apache Flink as a representative streaming engine and describe template DL serving pipelines with embedded and external models. We evaluate the streaming pipelines using an image classification task implemented with a feed-forward neural network. We observe that embedded models for online serving most often offer a clear advantage for achieving high throughput, exploiting local state access and vertical scalability already present in modern stream processors. However, contrary to intuition, our results also showcase that external serving options can exhibit comparable or lower inference latency than that of embedded alternatives, when the hardware capabilities are properly exploited and the cost of RPC calls does not become a bottleneck. Our study's source code and evaluation configurations are all publicly available [1].

2 MODEL SERVING STRATEGIES

Due to the lack of unified system support to handle the complexity of deploying and serving DL models, ML engineers often fall back to custom solutions, tailored to their specific use-cases. As a consequence, a large number of frameworks and libraries exist that can facilitate the integration between stream processing engines and ML serving platforms. In this particular study, we narrow down our search to the most popular tools used in each respective category for online model serving.

DEEM'22, June 12, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9375-1/22/06.
https://doi.org/10.1145/3533028.3533308

[1] https://github.com/soniahorchidan/streaming-model-serving

Experiment \ Parameter	Input Rate	Batch Size	Parallelism
Throughput Exp.	$10, 100, \ldots, 10K$	1	1
Latency Exp.	10	$1, 10, \ldots 1K$	1
Vertical Scalability Exp.	$30K$	1	$1, 2, \ldots, 16$

Table 1: Experimental parameters. The input rate is measured in requests per second. The batch size refers to the number of images per request. The parallelism specifies the number of model servers (i.e. operator parallelism for embedded serving, number of serving workers for external serving).

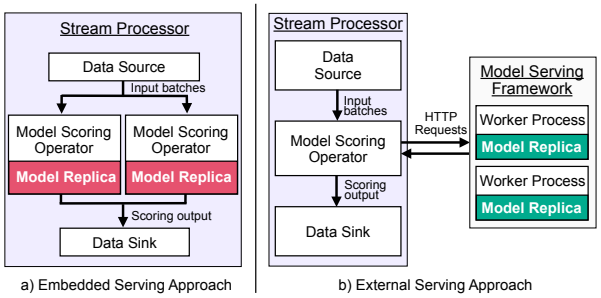

Figure 2: Model Serving Architectures.

2.1 Tools for model serving on data streams

Apache Flink [7] is a widely-used, open-source framework tailored to executing long-running dataflow streaming applications with exactly-once processing guarantees. Flink's execution model makes use of data-parallel task processing based on consistent hashing to distribute events and states across parallel stream sub-tasks. These characteristics serve as a good foundation for low-latency and high-throughput execution of business logic, however, no native support is provided for incorporating ML-based tasks within stream pipelines (i.e., model training and serving). Contrary to training that is catered by specialized ML frameworks, model serving is a particularly good fit for data streaming applications due to its online and often time-critical nature. Currently, model serving logic in Flink can only be configured manually through either task-embedded or external use of third-party ML libraries (see Sec. 2.2).

Machine Learning Frameworks. ML Frameworks aim to cater to all the steps of developing a model and allow developers to exploit the available hardware for efficient data pre-processing, training, and serving. We limit the scope of this study to the most widely-adopted ML Frameworks, PyTorch and TensorFlow, and their corresponding offerings for production-level model deployment, **Torch Serve** [5] and **TensorFlow Serving** [12] respectively. Torch Serve and TensorFlow Serving are managed solutions for scalable model serving. Their operation mode is straightforward: the client issues remote procedure calls (RPCs) containing the test data points and the model server sends back a response with the model inference result. The systems provide a handful of essential features, such as batching, model scaling, and continuous monitoring. Harnessing the hardware capabilities, the frameworks enable powerful accelerators, such as GPU support or SIMD operations for CPU-based inference optimization.

Interoperability Libraries. Integrating model inference into a stream processor typically requires the aid of external frameworks to provide interoperability, since ML models are commonly trained using Python-based ML training frameworks, whereas stream-processing frameworks are commonly implemented in Java or Scala. For this study, we choose three interoperability formats: **TensorFlow SavedModel** [4], **Open Neural Network Exchange** (ONNX) [3], and **ND4J** [2].

2.2 Model deployment strategies

Figure 2 depicts the two alternative deployment schemes we consider in this work. In *"embedded serving"*, the model is integrated natively into the streaming framework. The pre-trained model has to be converted into a standardized format that allows for interoperability between the stream processor and the training tool. Alternatively, in *"external serving"*, the models are wrapped inside micro-services that are external to the stream processing framework. The communication between the stream processor and the serving micro-service happens via RPC-type APIs. Unlike the embedded serving approach, the model does not need to be converted into other formats. The model serving framework is responsible for loading, using, and scaling the model to handle the workload.

3 EVALUATION METHODOLOGY

Before presenting our results, we describe the baseline ML task, the setup for all the selected frameworks and libraries, and the chosen quality metrics. Lastly, we list the machine specifications.

3.1 Machine Learning task

We evaluate the model deployment alternatives using a classic image classification task trained on the Fashion MNIST Dataset [15]. We employ a simple feed-forward, fully-connected neural network with three hidden layers. Each hidden layer contains 32 neurons and is followed by a `ReLU` activation function.

Since this study only targets the inference phase, the evaluation is agnostic to the chosen ML task. Thus, the intrinsics of the pre-processing and training steps are omitted from this section. We only document the factors that will further play a role in the performance of the serving approaches: the model and input/output sizes. The input data is represented as images of size `28x28`. The model outputs a vector of size 10 corresponding to the probability of the image to be assigned to one of the 10 base classes. The model is trained using TensorFlow and PyTorch. The resulting files are $488KB$ and $116KB$ in size, respectively. The PyTorch model is then converted to ONNX ($113KB$) and Numpy ($128KB$) formats.

3.2 Implementation setup

The streaming pipelines we implemented follow closely the depicted workflows in Figure 2. They consist of only three operators: (1) a data source operator that generates random input images and sends them downstream in batches, (2) a model scoring (map) operator that receives the input batches, runs model prediction, and sends the scorings downstream, and (3) a sink operator that writes the results to persistent storage.

Embedded Serving Tools. The embedded serving scenario is implemented by loading the model in one of the chosen formats

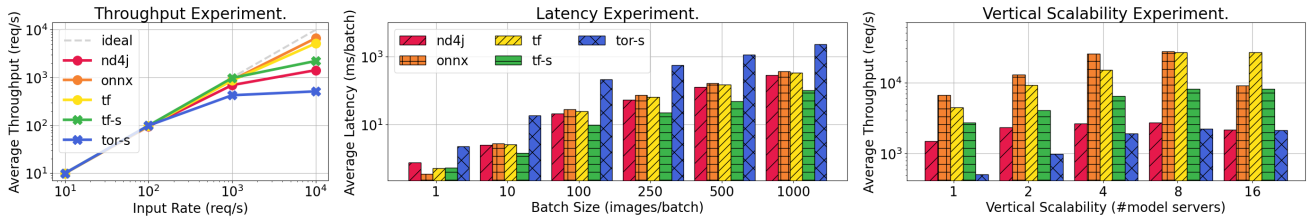

Figure 3: Experiment results. `nd4j`, `onnx` and `tf` correspond to ND4J, ONNX, and TensorFlow SavedModel, respectively. `tf-s` and `tor-s` correspond TensorFlow Serving and TorchServe.

(ONNX/ND4J/TensorFlow SavedModel) on the operator initialization. ND4J provides no API for arbitrary pre-trained models and, thus, we write a custom class responsible for loading the weights, re-constructing, and applying the model. For that, we decompose the pre-trained model and manually save the layers as *numpy* arrays. Regarding the other two formats, we use their corresponding APIs for model loading and scoring.

External Serving Tools. We spawn one TorchServe/TensorFlow Serving instance collocated on the same node with the Apache Flink process. Each request received from the Flink process will contain one batch of images and will be sent asynchronously to avoid blocking I/O operations. TensorFlow enables support for SIMD instructions by default, while TorchServe has no such support at the time of this study.

3.3 Performance metrics

We evaluate the serving pipelines in terms of *throughput* and *latency*. The throughput is defined as the number of completed predictions per second, while the latency is measured as the time required for one prediction to complete. We measure the throughput in an *open-loop scenario*, in which the Flink source sends requests at a predefined input rate asynchronously, thus relying on the system to handle the workload. The latency is measured in a *closed-loop experiment*, where the producer waits for a request to be processed before sending the next one. The latency measurements include the time requests spend waiting to be processed alongside the inference time. Specifically, for embedded serving, the latency includes the time spent in network buffers, while in the external case, it includes the request round-trip times.

3.4 Environment

We performed all experiments on the `Google Cloud Platform`, on a VM equipped with `Intel(R) Xeon(R) @2.00GHz 32-cores Processor` and `120GB RAM`. We used Apache Flink `1.12`, TorchServe `0.5.2`, TensorFlow Serving `2.8`, ONNX `1.10`, ND4J `1.0`, and TensorFlow Java `1.15`. The external serving approach entails sending images over the network. In our setup, all the systems are collocated on the same node. The average time of sending one ping packet as large as one image (3.16 KB) to `localhost` is `0.047ms`.

4 EXPERIMENTS

We measure the throughput of the various pipelines under increasing load and the latency for varying batch sizes. We also perform a scalability experiment by increasing the parallelism of model serving. Table 1 includes all the configuration parameters.

Throughput. Figure 3 (left) depicts the average throughput for input rates up to $10k$ requests/s. As expected, embedded serving achieves higher throughput for standardized formats (`onnx`, `tf`) compared to custom solutions (`nd4j`). When it comes to external services, Tensorflow Serving (`tf-s`) outperforms TorchServe (`tor-s`) for high input rates. Comparing deployment alternatives, it may come as no surprise that embedded model serving achieves higher throughput than external serving. Yet, we also observe that `tf-s` outperforms `nd4j` for input rates higher than 100 requests/s. This result indicates that, under certain conditions, a highly-optimized ML serving framework may outperform a naive embedded implementation. Moreover, we note that `tor-s`'s throughput is an order of magnitude lower than that of the best-performing embedded model serving pipeline, `onnx`.

Latency. In Figure 3 (center) we plot the average latency for increasing batch sizes. The three embedded serving alternatives achieve similar results, with very little variation. TensorFlow Serving `tf-s`, on the other hand, presents lower latency than all the embedded serving tasks, with latency approximately 3× lower than that of `onnx`, `tf`, and `nd4j`. TorchServe (`tor-s`) has the highest latency among all tested tools, with a maximum latency of 2.2s for a batch size of 1000 images. Compared to its competitor (`tf-s`), `tor-s` is between 4× to 24× slower.

Vertical Scalability. Figure 3 (right) shows the vertical scalability experiment results. The results show that the embedded serving pipelines harness data-parallel execution in Flink to scale up to $27K$ requests/s under an input rate of $30K$ requests/s when setting the model scoring operator parallelism to 8. However, this trend is observable only for optimized model formats (`onnx` and `tf`), while the custom `nd4j` implementation peaks at only $2.7K$ requests/s with 8 parallel serving operators. We also observe that `onnx` achieves its maximum performance at parallelism 4, while `tf` can scale up to 8 parallel serving operators. This is due to extra resources allocated by the ONNX Runtime compared with TensorFlow SavedBundle. ONNX allocates a separate thread for each ONNX session, corresponding to each Flink operator instance, whereas the TensorFlow session runs the inference on the caller thread (i.e. the same thread allocated to the corresponding Flink scoring operator). Lastly, TensorFlow Serving (`tf-s`) achieves superior results against TorchServe (`tor-s`), showcasing a maximum throughput of $8.1K$ requests/s with 8 parallel servers, while `tor-s` peaks at only $2.2K$ requests/s under the same configuration.

5 LESSONS LEARNED

Embedded options are comparable. Our experimental study shows that embedded alternatives do not differ significantly in terms of performance. The custom ND4J-based serving pipeline ranks the lowest, yet, this is expected since it currently lacks common optimization strategies. ND4J further requires the highest engineering effort, compared to the rest of the alternatives which provide intuitive APIs to handle loading and serving. ONNX and TensorFlow SavedModel both show impressive results in all our experiments and seem to be suitable for achieving vertical scalability.

Exploiting hardware with TensorFlow Serving. When comparing the two external model servers, TorchServe and TensorFlow Serving, the latter consistently exhibits better performance. We attribute these results to TensorFlow's SIMD acceleration support to speed-up CPU-based predictions. The results clearly show that this optimization gives TensorFlow Serving a considerable advantage against alternatives that currently do not exploit vectorized instructions out-of-the-box. However, we found TorchServe to be more flexible since it allows users to write custom model handlers which can be used to encapsulate arbitrary business logic. Whereas, TensorFlow Serving is geared towards serving the specified model and cannot be further configured.

Embedded for high throughput, yet, not always for low latency. Our experiments show that there is no clear advantage in embedded model serving in all situations and that the performance widely depends on the environment, i.e., hardware and network latency conditions. Surprisingly, our evaluation reveals that the TensorFlow Serving approach can achieve lower latency than all the embedded deployments. This shows that external inference can be a viable option, even under low-latency constraints if the hardware is exploited properly. In our testbeds, the client and the model server are collocated on the same node and, therefore, the latency of sending HTTP requests is minimal. In various real-world environments, however, this cost can play a decisive role in the overall latency of the system which could potentially lead to insufficient amortization of performance gains in hardware acceleration. Lastly, the lack of communication over the network in the embedded serving scenario enables offline real-time predictions, which makes it an ideal candidate for model serving on edge devices.

A promising direction forward for data streaming systems is to examine native support for ML model serving. Existing efforts in GraalVM and TornadoVM [6] showcase that different types of HW acceleration are possible even within the execution of a stream processor. Besides hardware acceleration, an evident need is also to offer off-the-shelf support for different model formats as well as loading and serving capabilities within existing stream processors.

6 RELATED WORK

In this section, we examine promising model serving systems and directions. Clipper [9] and InferLine [8] are general-purpose managed inference systems that natively enable features such as model composition, online learning, caching, adaptive batching, or automatic resource allocation. Clipper and InferLine both handle model prediction via RPC calls to containers with pre-trained models, which makes them fall into the category of external model servers.

A different direction to achieve real-time model predictions is Deep Learning on Flink [1], which is an attempt to run usual DL training and inference natively inside Flink operators, thus making use of the Apache Flink cluster to provide distributed execution. Finally, we mention current efforts to bring stream processing capabilities to Python-based ecosystems. One approach is to provide Python APIs for the most popular stream processors (e.g. PyFlink for Apache Flink). Ray [11] shows another rising direction, which consists of implementing a flexible framework directly in Python targeted to ML-centric applications that can also handle streaming data.

7 CONCLUSION AND FUTURE WORK

Our preliminary results can guide vendors to integrate serving into data pipelines and developers to deploy new managed solutions. Nonetheless, we plan to extend our evaluation study as follows. First, we would like to study the impact of model size and understand whether external architectures can compete with small embedded models that can fit in memory. Second, we are interested in evaluating how the request size might affect the latency of external serving. Third, it is crucial to compare existing tools in terms of their support for fast model updates, without disrupting the performance of the streaming application. Fourth, we hope to evaluate model composition and the requirements to deploy, manage, and scale multiple models at different stages in a serving pipeline. Further, we are interested in exploring the impact of HW accelerator extensively (e.g., inference backed by GPUs), especially for batched, parallel serving of large models. Finally, remote service placement should be investigated to evaluate scenarios where the external model server is not collocated with the client application.

ACKNOWLEDGMENTS

This work was partially supported by the RISE AI Research Center, the Swedish Foundation for Strategic Research under Grant No.: BD15-0006, a Red Hat Collaboratory Research Incubation Award (ID: 2022-01-RH08), a Samsung MSL UR collaboration grant, and a Google DAPA award. Emmanouil Kritharakis is also supported by the Onassis Scholarship [Scholarship ID: F ZR 030/1-2021/2022].

REFERENCES

[1] 2022 (accessed). *Deep Learning on Flink*. https://github.com/flink-extended/dl-on-flink
[2] 2022 (accessed). *DeepLearning4J*. https://github.com/eclipse/deeplearning4j
[3] 2022 (accessed). *ONNX Runtime*. https://github.com/microsoft/onnxruntime
[4] 2022 (accessed). *TensorFlow SavedModel*. https://www.tensorflow.org/guide/saved_model
[5] 2022 (accessed). *TorchServe*. https://github.com/pytorch/serve
[6] 2022 (accessed). *TornadoVM*. https://github.com/beehive-lab/TornadoVM
[7] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38.
[8] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. 2020. InferLine: latency-aware provisioning and scaling for prediction serving pipelines. In *SoCC '20: ACM Symposium on Cloud Computing*. ACM, 477–491. https://doi.org/10.1145/3419111.3421285
[9] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017*. 613–627.
[10] Boris Lublinsky. 2017. *Serving Machine Learning Models: A Guide to Architecture, Stream Processing Engines, and Frameworks*. O'Reilly Media.
[11] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications.

In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018*. 561–577.

[12] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. TensorFlow-Serving: Flexible, High-Performance ML Serving. *CoRR* abs/1712.06139 (2017). arXiv:1712.06139 http://arxiv.org/abs/1712.06139

[13] Javier Ramos. 2020. *Machine Learning Model Serving Options*. https://itnext.io/machine-learning-model-serving-options-1edf790d917

[14] Kai Waehner. 2019. *Machine Learning and Real-Time Analytics in Apache Kafka Applications*. https://www.confluent.io/blog/machine-learning-real-time-analytics-models-in-kafka-applications/

[15] Han Xiao, Kashif Rasul, and Roland Vollgraf. 2017. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. *CoRR* abs/1708.07747 (2017). arXiv:1708.07747 http://arxiv.org/abs/1708.07747

[16] Doris Xin, Hui Miao, Aditya G. Parameswaran, and Neoklis Polyzotis. 2021. Production Machine Learning Pipelines: Empirical Analysis and Optimization Opportunities. In *SIGMOD '21: International Conference on Management of Data*.

[17] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. 2012. Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters. In *4th USENIX Workshop on Hot Topics in Cloud Computing*.