From the Edge to the Cloud: Model Serving in ML.NET

Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Markus Weimer and Matteo Interlandi
{yunseong, bgchun}@snu.ac.kr, alberto.scolari@polimi.it, {mweimer, mainterl}@microsoft.com

Abstract

As Machine Learning (ML) is becoming ubiquitously used within applications, developers need effective solutions to build and deploy their ML models across a large set of scenarios, from IoT devices to the cloud. Unfortunately, the current state of the art in model serving suggests to deliver predictions by running models in containers. While this solution eases the operationalization of models, we observed that it is not flexible enough to address the variety of ML scenarios encountered in large companies such as Microsoft. In this paper, we will overview ML.NET—a recently open sourced ML pipeline framework—and describe how ML models written in ML.NET can be seamlessly integrated into applications. Finally, we will discuss how model serving can be cast to a database problem, and provide insights on our recent experience in building a database optimizer for ML.NET pipelines.

1 Introduction

Machine Learning (ML) is transitioning from an art and science into a technology readily available to every developer. In the near future, every application on every platform will rely on trained models for functionalities that evade traditional programming due to their complex statistical nature. This unfolding future—where most applications make use of at least one model—profoundly differs from the current practice in which data science and software engineering are performed in separate and different processes and sometimes even by different teams and organizations. Furthermore, in current practice, models are routinely deployed and managed in completely distinct ways from other software artifacts: while typical software libraries are seamlessly compiled and run on a myriad of heterogeneous devices, ML models are often implemented in high-level languages (e.g., Python) and relegated to be run as web services in remotely hosted containers [3, 10, 12, 15, 22]. Ad-hoc solutions or bespoken re-engineering strategies can be pursued to address specific applications (e.g., low latency scenarios as in obstacle detection for self-driving cars), but these efforts are not scalable in general. Therefore they are inappropriate for enterprise-scale ML needs as the one that can be observed in large companies. This pattern not only severely limits the kinds of applications one can build with ML capabilities, but also discourages developers from embracing ML as a core component of applications.

ML.NET [9] is the end-to-end solution provided by Microsoft to address the above problems. ML.NET is an open source ML framework allowing developers to author and deploy in their applications complex ML pipelines composed of data featurizers and state of the art ML models. Pipelines implemented and trained using ML.NET can be seamlessly surfaced for prediction without any modification, and adding a model into an application is as easy as importing the ML.NET runtime and binding the input/output data sources. ML.NET's

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

ability to capture full, end-to-end pipelines has been demonstrated by the fact that thousands of Microsoft's data scientists and developers have been using ML.NET over the past decade, infusing hundreds of products and services with ML models used by hundreds of millions of users worldwide.

In this paper, we will give an overview of current state of the art practices for surfacing ML pipelines predictions into applications, and we will highlight the limitations of using containers to operationalize models for application consumption. We will then introduce how ML.NET allows developers to design their data-driven applications end-to-end without having to rely on any external resource. Finally, we will present few challenges we have observed in running ML.NET models in production, and how these can be addressed by considering models as Direct Acyclic Graphs (DAGs) of operators instead of black-box executable code. Specifically, we will describe a new ML.NET runtime for model scoring called PRETZEL. PRETZEL treats model scoring as a database problem and, as such, it employs database techniques to optimize the performance of predictions.

2 Background: ML Pipelines

Many ML frameworks such as Spark MLlib [2], H2O [6], Scikit-learn [23], or Microsoft ML.NET [9] allow data scientists to declaratively author pipelines of transformations for better productivity and easy operationalization. Model pipelines are internally represented as DAGs of pre-defined operators [1] comprising *data transformations* and *featurizers* (e.g., string tokenization, hashing, etc.), and *ML models* (e.g., decision trees, linear models, SVMs, etc.). Figure 1 shows an example pipeline for text analysis whereby input sentences are classified according to the expressed sentiment.

ML.NET is an open-source C# library running on a managed runtime with garbage collection and Just-In-Time (JIT) compilation [2]. ML.NET's main abstraction is called DataView, which borrows ideas from the database community. Similarly to (intensional) database relations, the DataView abstraction provides compositional processing of schematized data, but specializes it for ML pipelines. In relational databases, the term *view* typically indicates the result of a query on one or more tables (base relations) or views, and is generally immutable [18]. Views have interesting properties that differentiate them from tables and make them appropriate abstractions for ML: (1) views are *composable*—new views are formed by applying transformations (queries) over other views; (2) views are *virtual*, i.e., they can be lazily computed on demand from other views or tables without having to materialize any partial results; and (3) since a view does not contain values but merely computes values from its source views, it is *immutable* and *deterministic*: the exact same computation applied over the same input data always produces the same result. Immutability and deterministic computation enables transparent data caching (for speeding up iterative computations such as ML algorithms) and safe parallel execution. DataView inherits the aforementioned database view properties, namely: composability, lazy evaluation, immutability, and deterministic execution.

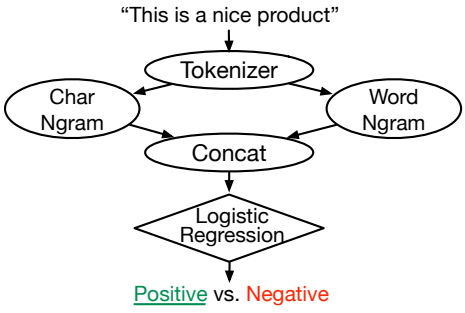


Figure 1: A Sentiment Analysis (SA) pipeline consisting of operators for featurization (ellipses), followed by a ML model (diamond). *Tokenizer* extracts tokens (e.g., words) from the input string. *Char* and *Word Ngrams* featurize input tokens by extracting n-grams. *Concat* generates a unique feature vector which is then scored by a *Logistic Regression* predictor. This is a simplification: the actual DAG contains about 12 operators.

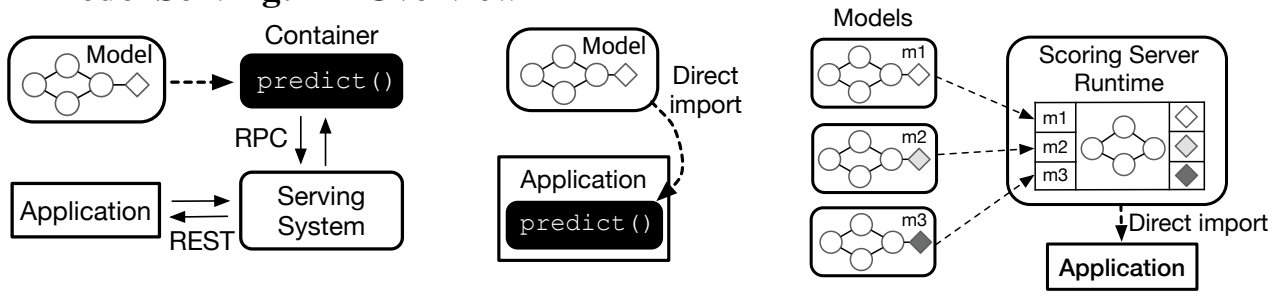
In ML.NET, pipelines are represented as DAGs of operators, each of them implementing the DataView

[1] Note that user-defined code can still be executed through a second order operator accepting arbitrary UDFs.

[2] Unmanaged C/C++ code can also be employed to speed up processing when possible.

interface and executing a featurization step or a ML model. Upon pipeline initialization, the operators composing the model DAG are analyzed and arranged to form a chain of function calls which, at execution time, are JIT-compiled to form a unique function executing the whole DAG on a single call. Operators are able to gracefully and efficiently handle high-dimensional and large datasets thanks to *cursoring*, which resembles the well-known iterator model of databases [17]: within the execution chain, inputs are pulled through each operator to produce intermediate vectors that are input to the following operators, until a prediction or a trained model is rendered as the final output of the pipeline. We refer readers to [13] for further details on ML.NET.

3 Model Serving: An Overview



(a) Models are deployed into containers, connected to a Serving System via RPC. To score the models, applications must call into the Web Server hosted on the Serving System using a REST API.

(b) ML.NET allows developers to deploy models directly into their applications and without any additional custom engineering effort.

(c) With the white-box approach, models' structural information can be used to optimize the execution. As in (b), applications can directly access the models by importing the Scoring Server Runtime.

Figure 2: Three different ways to deploy models into applications. (a) and (b) represent two variations of the black-box approach where the invocation of the function chain (e.g., `predict()`) on a pipeline returns the result of the prediction. (c) shows the white-box approach.

In this Section, we survey how models are commonly operationalized in industry. The most popular (and easiest) method to deploy ML models (in general, and pipelines in particular) is what we refer to as *black box*. Under this approach, internal pipelines' information and structures are not considered inasmuch as pipelines are opaque executable code accepting some input record(s) and producing a prediction. Within the black box approach, there are two possible ways for a developer to deploy models, and consequently for an application to request and consume predictions. The first option (à la Clipper [3], depicted in Figure 2(a) and further described in Section 3.1) is to ship models into containers (e.g., Docker [4]) wired with proper Remote Procedure Calls (RPCs) to a Web Server. With this approach, predictions have to go through the network and be rendered on the cloud: low latency or edge scenarios are therefore out of scope. The second option (Figure 2(b) and detailed in Section 3.2) is to integrate the model logic directly into the application (à la ML.NET: the model is a dynamic library the application can link). This approach is suitable for the cloud as well as for edge devices and it unlocks low latency scenarios. However, we still find this approach sub-optimal with respect to customized solutions because it ships the same training pipeline code for prediction. In fact, while using the same code is a great advantage because it removes the need for costly translation work, it implicitly assumes that training and prediction happen in the same regime. However, prediction serving is much more latency sensitive. The *white box* approach (Section 3.3) depicted in Figure 2(c) tackles the aforementioned problem by considering ML pipelines not anymore as black-box artifacts, but as DAGs of operators, and therefore it tries to rewrite them using optimizations specifically tailored to prediction-time scenarios. We next provide additional details on each of the three possibilities.

3.1 Deploying Models into Containers

Most serving systems in the state of the art [3, 8, 10, 12, 15, 22] aim to minimize the burden of deploying trained pipelines in production by serving them in containers, where the same code is used for both training and inference [3]. This design allows decoupling models from serving system development, and eases the implementation of mechanisms and policies for fault tolerance and scalability. Furthermore, hardware acceleration can be exploited when available. A typical container-based, model serving system follows the design depicted in Figure 2(a): containers are connected to a Serving System (e.g., Clipper) via RPC, and, to score models, applications should contact the Serving System by invoking a Web Server through a REST API. Developers are responsible for setting up the communication between their applications and the Serving System, but this is in general an easy task as most Serving Systems provide convenient libraries (e.g., Microsoft ML Server [8]). Implementing model containers for new ML frameworks and integrating them with the Serving System requires a reasonable amount of effort: for example, a graduate student spent a couple of weeks to implement the protocol for integrating an ML.NET container into Clipper.

Limitations. While serving models via containers greatly eases operationalization, we found though that it is not flexible enough to accommodate the requirements stemming from running ML models at Microsoft scale. For instance, containers allow resource isolation and thus achieve effective multitenancy, but each container comes with its own runtime (e.g., an ML.NET instance) and set of processes, thus introducing memory overheads that can possibly be higher than the actual model size. Additionally, the RPC layer and REST API introduce network communication costs, which are especially relevant for models that have millisecond-level prediction latency. Finally, only a restricted set of optimizations are available, specifically those that do not require any knowledge of the internals of the pipelines; examples are handling multiple requests in batches and caching prediction results if some inputs queries are frequently issued for the same pipeline. Instead, any optimization specific to the model is out of scope, as from the inscrutable nature of its container [4].

3.2 Importing Models Directly into Applications

At Microsoft, we have encountered the problem of model deployment across a wide spectrum of applications ranging from Bing Ads to Excel, PowerPoint and Windows 10, and running over diverse hardware configurations ranging from desktops, to custom hardware (e.g., XBox and IoT devices) and to high performance servers [1, 5, 7]. To allow such diverse use cases, an ML toolkit deeply embedded into applications should not only satisfy several intrinsic constraints (e.g., scale up or down based on the available main memory and number of cores) but also preserve the benefits commonly associated with model containerization, i.e., (1) it has to capture the full prediction pipeline that takes a test example from a given domain (e.g., an email with headers and body) and to produce a prediction that can often be structured and domain-specific (e.g., a collection of likely short responses); and (2) it has to allow to seamlessly carry the complete train-time pipeline into production for model inference. This later requirement is the keystone for building effective, reproducible pipelines [27].

ML.NET is able to implement all the above desiderata. Once a model is trained in ML.NET, the full training pipeline can be saved and directly surfaced for prediction serving without any external modification. Figure 2(b) depicts the ML.NET solution for black-box model deployment and serving: models are integrated into application logic natively and predictions can be served in any OS (Linux, Windows, Android, MacOS) or device supported by the .NET Core framework. This approach removes the overhead of managing containers and implementing RPC functionalities to communicate with the Serving System. In this way, application developers

[3] Note that TensorFlow Serving [12] is slightly more flexible since users are allowed to split model pipelines and serve them into different containers (called *servables*). However, this process is manual and occurs when building the container image, ignoring the final running environment.

[4] In the case of ML.NET pipelines, the C# runtime in the container can optimize *the code* of the model, but not the model itself as we propose in the following.

are facilitated for writing applications with ML models inside. Nevertheless, models can still be deployed in the cloud if suggested by the application domain (e.g., because of special hardware requirements).

Limitations. ML.NET assumes no knowledge and no control over the pipeline inasmuch as the same code is executed both for training and prediction. [5] This is in general good practice because it simplifies the process of training-inference skew debugging [27]. Nevertheless, we found that such approach is sub-optimal from a performance perspective. For instance, transfer learning, A/B testing and model personalization are getting popular. Such trends produce models DAGs with high chance of overlapping structure and similar parameters, but these similarities cannot be recognized nor exploited using a black-box approach. Furthermore, it is common practice for in-memory data-intensive systems to pipeline operators in order to minimize memory accesses for memory-intensive workloads, and to vectorize compute-intensive operators in order to minimize the number of instructions per data item [16, 28]. ML.NET's operator-at-a-time model [28] is sub-optimal because computation is organized around logical operators, ignoring how those operators behave together: in the example of the sentiment analysis pipeline of Figure 1, logistic regression is commutative and associative (e.g., dot product between vectors) and can be pipelined with Char and WordNgram, eliminating the need for the Concat operation and the related buffers for intermediate results. Note that this optimization is applicable only at prediction-time whereas at training-time logistic regression runs the selected optimization algorithm. We refer readers to [19] for further limitations arising when serving models for prediction using the black-box approach.

3.3 White Box Model Serving

As we have seen so far, black-box approaches disallow any optimization and sharing of resources among models. Such limitations are overcome by the *white box approach* embraced by systems such as TVM [14] and PRETZEL [19]. Figure 2(c) sketches white-box prediction serving. Models are registered to a Runtime that considers them not as mere executable code but as DAGs of operators. Applications can request predictions by directly including the Runtime in their logic (similarly to how SQLite databases [11] can be integrated into applications), or by submitting a REST request to a cloud-hosted Runtime. The white box approach enables the Runtime to apply optimizations over the models such as operator reordering to improve latency or operator and sub-graph sharing to improve memory consumption and computation reuse (through caching). Thorough scheduling of pipelines' components can be managed within the Runtime, which controls the whole workload so that optimal allocation decisions can be made for running machines to high utilization and avoid many of the aforementioned overheads. In general, we have identified the following optimization opportunities for white-box model serving.

End-to-end Optimizations: The operationalization of models for prediction should focus on computation units making optimal decisions on how data are processed and results are computed, to keep low latency and graceful degradation of performance with increasing load. Such computation units should: (1) avoid memory allocation on the data path; (2) avoid creating separate routines per operator when possible, which are sensitive to branch mis-prediction and poor data locality [21]; and (3) avoid reflection and JIT compilation at prediction time. Optimal computation units can be compiled Ahead-Of-Time (AOT) since pipeline and operator characteristics are known upfront, and often statistics from training are available. The only decision to make at runtime is where to allocate the computation units based on available resources and constraints.

Multi-model Optimizations: To take full advantage of the fact that pipelines often use similar operators and parameters, shareable components have to be uniquely stored in memory and reused as much as possible to achieve optimal memory usage. Similarly, execution units should be shared at runtime and resources should be properly pooled and managed, so that multiple prediction requests can be evaluated concurrently. Partial results, for example outputs of featurization steps, can be saved and re-used among multiple similar pipelines.

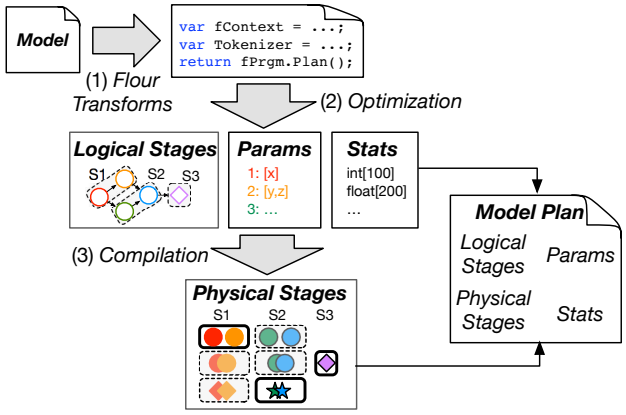
Out of these guidelines, the next Section describes a prototype runtime for ML.NET enabling white-box model serving.

[5] Indeed, already trained operators will bypass the execution of the learning algorithm and directly apply the previously learned parameters.

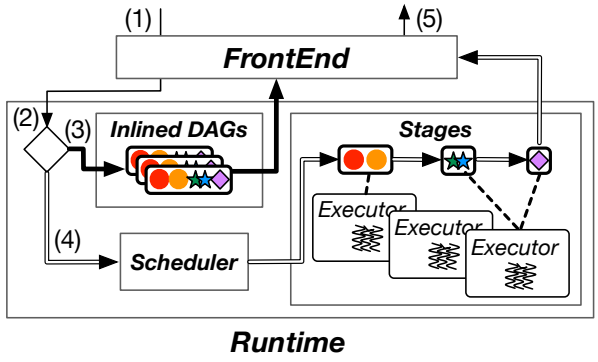
4 A Database Runtime for Model Serving

Following the guidelines of white-box model serving, we implemented PRETZEL [19, 24], a new runtime for ML.NET specifically tailored for high-performance prediction serving. PRETZEL views models as database queries and employs database techniques to optimize DAGs and to improve end-to-end performance. The problem of optimizing co-located pipelines is casted as a multi-query optimization and techniques such as view materialization are employed to speed up pipeline execution. Memory and CPU resources are shared in the form of vector and thread pools, such that overheads for instantiating memory and threads are paid only upfront at initialization time. PRETZEL is organized in 6 main components. A *data-flow-style language integrated API* called Flour with related *compiler* and *optimizer* called Oven are used in concert to convert ML.NET pipelines into *model plans*. An Object Store saves and shares parameters among plans. A Runtime manages compiled plans and their execution, while a Scheduler manages the dynamic decisions on how to schedule plans based on machine workload. Finally, a FrontEnd is used to submit prediction requests to the system.

In PRETZEL, deployment and serving of model pipelines follow a two-phase process as illustrated in Figure 3(a) and 3(b). During the *off-line phase*, pre-trained ML.NET pipelines are translated into Flour transformations. Oven optimizer re-arranges and fuses transformations into model plans composed of parameterized logical units called *stages*. Each logical stage is then AOT-compiled into physical computation units. Logical and physical stages together with model parameters and training statistics form a model plan. Model plans are registered for prediction serving in the Runtime where physical stages and parameters are shared among pipelines with similar model plans. In the *on-line phase*, when an inference request for a registered model plan is received, physical stages are parameterized dynamically with the proper values maintained in the Object Store. The Scheduler is in charge of binding physical stages to shared execution units.



(a) Model optimization and compilation in PRETZEL: (1) A model is translated into a Flour program. (2) Oven Optimizer generates a DAG of logical stages from the program. Additionally, parameters and statistics are extracted. (3) A DAG of physical stages is generated by the Oven Compiler using logical stages, parameters, and statistics. A model plan is the union of all the elements.



(b) (1) When a prediction request is issued, (2) the Runtime determines whether to serve the prediction using (3) the request/response engine or (4) the batch engine. In the latter case, the Scheduler takes care of properly allocating stages over the Executors running concurrently on CPU cores. (5) The FrontEnd returns the result to the Client once all stages are complete.

Figure 3: How PRETZEL system works in two phases: (a) Offline and (b) Online.

PRETZEL compiles a model pipeline into an optimized, executable plan following 3 steps:

1. **Model conversion:** Flour is an an intermediate representation, which makes PRETZEL's optimizations applicable to various ML frameworks (although the currently implementation is for ML.NET). Once a pipeline is ported into Flour, it can be optimized and compiled into a model plan. Flour provides a language-integrated API similar to KeystoneML [25] or LINQ [20], where sequences of transformations

are chained into DAGs and lazily compiled for execution.

2. **Optimization:** Oven's rule-based Optimizer rewrites a model pipeline represented in Flour into a graph of logical stages. Initially, the Optimizer applies rules for schema propagation, schema validation and graph validation. The rules check whether the input schema of each data transformation is valid (e.g., WordNgram in Figure 1 takes a text as input) and the structure of the graph is well-formed (e.g., only one final predictor model at the end). The next rules are used to build stages by traversing the entire graph to find *pipeline-breaking* transformations that require all inputs to be fully materialized (e.g., normalizer): all transformations up to that point are then grouped into a stage. By leveraging stage graphs similar to Spark [26], PRETZEL can run computations more efficiently than the operator-at-a-time strategy of ML.NET. The stage graph is then optimized by recursively applying rules such as (1) removing unnecessary branches (similar to common sub-expression elimination); (2) merging stages containing identical transformations; (3) inlining stages that contain only one transformation.

3. **Compilation:** After building the stage graph, a *Model Plan Compiler* (MPC) translates the graphs into physical stages, which are AOT-compiled, parameterized, and result in lock-free computation unit. Logical and physical stages have a 1-to-n mapping, and MPC selects the most efficient physical implementation given the logical stage's parameters (e.g., the maximum length of n-grams) and statistics (e.g., whether the vector is dense or sparse). During the translation process, MPC saves the additional parameters required for running stage code (e.g., a dictionary consisting of frequency of n-grams) into the ObjectStore in order to share them with other stages with the same parameters. Finally, model plans are registered into the Runtime. Model plans consist of mappings between logical representations, physical implementations and the associated parameters. Upon registration, physical stages composing a plan are loaded into a system catalog. When a prediction request is submitted to the system, the AOT-compiled physical stages are initialized with the parameters from the mapping in the model plan, which allows PRETZEL Runtime to share the same physical implementation among multiple pipelines.

5 Conclusion

Inspired by the growth of ML applications and ML-as-a-service platforms, this paper identifies three strategies for operationalizing trained models: container-based, direct import into applications, and the white-box approach. Using ML.NET as use case, we listed a set of limitations on how existing systems fall short in key requirements for ML prediction-serving, disregarding the optimization of model execution in favor of ease of deployment. Finally, we describe how the problem of serving predictions can be casted as a database problem, whereby end-to-end and multi-query optimization strategies are applied to ML pipelines.

We recognize that much work remains to be done for achieving a seamless and efficient integration of ML models with applications and development processes. While we believe that ML.NET and PRETZEL are a step in the right direction, equivalents in data science for common tools and techniques in software development (e.g., unit/integration test, build server, code review, versioning, backward compatibility, and lifecycle management) are not defined yet. We encourage the community to engage in the work towards closing those gaps.

References

[1] AI Platform for Windows Developers. `https://blogs.windows.com/buildingapps/2018/03/07/ai-platform-windows-developers`

[2] Apache Spark MLlib. `https://spark.apache.org/mllib`

[3] Clipper. `http://clipper.ai`

[4] Docker. https://docker.com

[5] The future of AI marketing: human ingenuity amplified. https://advertise.bingads.microsoft.com/en-us/insights/g/artificial-intelligence-for-marketing

[6] H2O.ai. https://www.h2o.ai

[7] Microsoft Looks To Patent AI For Detecting Video Game Cheaters. https://www.cbinsights.com/research/microsoft-xbox-machine-learning-cheat-detection-gaming-patent

[8] Microsoft Machine Learning Server. https://docs.microsoft.com/en-us/machine-learning-server

[9] ML.Net. https://dot.net/ml

[10] MXNet Model Server (MMS). https://github.com/awslabs/mxnet-model-server

[11] SQLite. https://www.sqlite.org

[12] TensorFlow Serving. https://www.tensorflow.org/serving

[13] Z. Ahmed, and et al. Machine Learning for Applications, not Containers. *Under Submission*, 2018.

[14] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. *OSDI*, 2018

[15] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A Low-Latency Online Prediction Serving System. *NSDI*, 2017.

[16] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Cetintemel, and S. Zdonik. An Architecture for Compiling UDF-centric Workflows. *VLDB*, 2015.

[17] G. Graefe. Volcano: An Extensible and Parallel Query Evaluation System. *IEEE Trans. on Knowl. and Data Eng.*, February 1994.

[18] A. Y. Halevy Answering Queries Using Views: A Survey. *VLDB Journal*, December 2001.

[19] Y. Lee, A. Scolari, B. -G. Chun, M. D. Santambrogio, M. Weimer, and M. Interlandi. PRETZEL: Opening the black box of Machine Learning Prediction Serving. *OSDI*, 2018.

[20] E. Meijer, B. Beckman, and G. Bierman LINQ: Reconciling Object, Relations and XML in the .NET Framework. *SIGMOD*, 2006.

[21] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *VLDB Endowment*, June 2011.

[22] C. Olston, F. Li, J. Harmsen, J. Soyke, K. Gorovoy, L. Lao, N. Fiedel, S. Ramesh, and V. Rajashekhar. TensorFlow-Serving: Flexible, High-Performance ML Serving. *Workshop on ML Systems at NIPS*, 2017.

[23] 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 É. Duchesnay. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.*, 2011.

[24] A. Scolari, Y. Lee, M. Weimer, and M. Interlandi. Towards Accelerating Generic Machine Learning Prediction Pipelines. *IEEE ICCD*, 2017.

[25] E. R. Sparks, S. Venkataraman, T. Kaftan, M. J. Franklin, and B. Recht. KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics. *ICDE*, 2017

[26] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. *NSDI*, 2012

[27] M. Zinkevich. Rules of Machine Learning: Best Practices for ML Engineering. https://developers.google.com/machine-learning/rules-of-ml

[28] M. Zukowski, P. A. Boncz, N. Nes, and S. Héman. MonetDB/X100 - A DBMS in the CPU Cache. *IEEE Data Eng. Bull.*, 2005.