PRETZEL: Opening the Black Box of Machine Learning Prediction Serving Systems

Yunseong Lee, ***Seoul National University;*** **Alberto Scolari,** ***Politecnico di Milano;***
Byung-Gon Chun, ***Seoul National University;*** **Marco Domenico Santambrogio,**
Politecnico di Milano; **Markus Weimer and Matteo Interlandi,** ***Microsoft***

https://www.usenix.org/conference/osdi18/presentation/lee

This paper is included in the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18).

October 8–10, 2018 • Carlsbad, CA, USA

ISBN 978-1-939133-08-3

Open access to the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

PRETZEL: Opening the Black Box of Machine Learning Prediction Serving Systems

Yunseong Lee
Seoul National University

Alberto Scolari
Politecnico di Milano

Byung-Gon Chun
Seoul National University

Marco Domenico Santambrogio
Politecnico di Milano

Markus Weimer
Microsoft

Matteo Interlandi
Microsoft

Abstract

Machine Learning models are often composed of pipelines of transformations. While this design allows to efficiently execute single model components at training-time, prediction serving has different requirements such as low latency, high throughput and graceful performance degradation under heavy load. Current prediction serving systems consider models as black boxes, whereby prediction-time-specific optimizations are ignored in favor of ease of deployment. In this paper, we present PRETZEL, a prediction serving system introducing a novel white box architecture enabling both end-to-end and multi-model optimizations. Using production-like model pipelines, our experiments show that PRETZEL is able to introduce performance improvements over different dimensions; compared to state-of-the-art approaches PRETZEL is on average able to reduce 99th percentile latency by 5.5× while reducing memory footprint by 25×, and increasing throughput by 4.7×.

1 Introduction

Many Machine Learning (ML) frameworks such as Google TensorFlow [4], Facebook Caffe2 [6], Scikit-learn [48], or Microsoft ML.Net [14] allow data scientists to declaratively author pipelines of transformations to train models from large-scale input datasets. Model pipelines are internally represented as Directed Acyclic Graphs (DAGs) of operators 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 is usually conceptualized as a two-steps process: first, during *training* model parameters are estimated from large datasets by running computationally intensive iterative algorithms; successively, trained pipelines are used for *inference* to generate predictions through the estimated model parameters. When trained pipelines are served for inference, the full set of operators is deployed altogether. However, pipelines have different system characteristics based on the phase in which they are employed: for instance, at training time ML models run complex algorithms to scale over large datasets (e.g., linear models can use gradient descent in one of its many flavors [52, 50, 54]), while, once trained, they behave as other regular featurizers and data transformations; furthermore, during inference pipelines are often surfaced for direct users' servicing and therefore require low latency, high throughput, and graceful degradation of performance in case of load spikes.

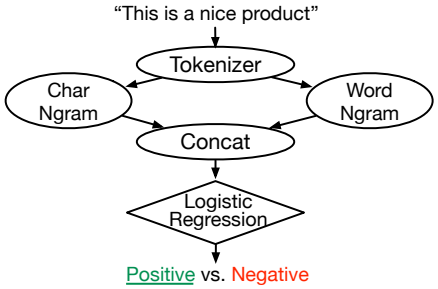

Figure 1: A Sentiment Analysis (SA) pipeline consisting of operators for featurization (ellipses), followed by a ML model (diamond). *Tokenizer* extracts tokens (e.g., words) from the input string. *Char* and *Word Ngrams* featurize input tokens by extracting n-grams. *Concat* generates a unique feature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

Existing prediction serving systems, such as Clipper [9, 32], TensorFlow Serving [5, 46], Rafiki [59], ML.Net [14] itself, and others [17, 18, 43, 15] focus mainly on ease of deployment, where pipelines are con-

sidered as *black boxes* and deployed into *containers* (e.g., Docker [11] in Clipper and Rafiki, *servables* in TensorFlow Serving). Under this strategy, only "pipeline-agnostic" optimizations such as caching, batching and buffering are available. Nevertheless, we found that black box approaches fell short on several aspects. For instance, prediction services are profitable for ML-as-a-service providers only when pipelines are accessed in batch or frequently enough, and may be not when models are accessed sporadically (e.g., twice a day, a pattern we observed in practice) or not uniformly. Also, increasing model density in machines, thus increasing utilization, is not always possible for two reasons: first, higher model density increases the pressure on the memory system, which is sometimes dangerous—we observed (Section 5) machines swapping or blocking when too many models are loaded; as a second reason, co-location of models may increase tail latency especially when seldom used models are swapped to disk and later re-loaded to serve only a few users' requests. Interestingly enough, model pipelines often share similar structures and parameters inasmuch as A/B testing and customer personalization are often used in practice in large scale "intelligent" services; operators could therefore be shared between "similar" pipelines. Sharing among pipelines is further justified by how pipelines are authored in practice: ML pipelines are often produced by fine tuning pre-existing or default pipelines and by editing parameters or adding/removing steps like featurization, etc.

These and other limitations of existing black box systems (further described in Section 2) inspired us for developing PRETZEL: a system for serving predictions over trained pipelines originally authored in ML.Net and that borrows ideas from the Database and System communities. Starting from the above observation that trained pipelines often share operators and parameters (such as weights and dictionaries used within operators, and especially during featurization [64]), we propose a *white box* approach for model serving whereby end-to-end and multi-pipeline optimization techniques are applied to reduce resource utilization while improving performance. Specifically, in PRETZEL deployment and serving of model pipelines follow a two-phase process. During an *off-line phase*, statistics from training and state-of-the-art techniques from in-memory data-intensive systems [33, 66, 26, 40, 45] are used in concert to optimize and compile operators into *model plans*. Model plans are white box representations of input pipelines such that PRETZEL is able to store and re-use parameters and computation among similar plans. In the *on-line phase*, memory (data vectors) and CPU (thread-based execution units) resources are pooled among plans. When an inference request for a plan is received, an event-based scheduling [60] is used to bind computation to execution units.

Using 500 different production-like pipelines used internally at Microsoft, we show the impact of the above design choices with respect to ML.Net and end-to-end solutions such as Clipper. Specifically, PRETZEL is on average able to improve memory footprint by 25×, reduce the 99th percentile latency by 5.5×, and increase the throughput by 4.7×.

In summary, our contributions are:

- A thorough analysis of the problems and limitations burdening black box model serving approaches;
- A set of design principles for white box model serving allowing pipelines to be optimized for inference and to share resources;
- A system implementation of the above principles;
- An experimental evaluation showing order-of-magnitude improvements over several dimensions compared to previous black box approaches.

The remainder of the paper is organized as follows: Section 2 identifies a set of limitations affecting current black box model serving approaches; the outcome of the enumerated limitations is a set of design principles for white box model serving, described in Section 3. Section 4 introduces the PRETZEL system as an implementation of the above principles. Section 5 contains a set of experiments validating the PRETZEL performance, while Section 6 lists the limitations of current PRETZEL implementation and future work. The paper ends with related work and conclusions, respectively in Sections 7 and 8.

2 Model Serving: State-of-the-Art and Limitations

Nowadays, "intelligent" services such as Microsoft Cortana speech recognition, Netflix movie recommender or Gmail spam detector depend on ML scoring capabilities, which are currently experiencing a growing demand [31]. This in turn fosters the research in prediction serving systems in cloud settings [5, 46, 9, 32], where trained models from data science experts are operationalized.

Data scientists prefer to use high-level declarative tools such as ML.Net, Keras [13] or Scikit-learn for better productivity and easy operationalization. These tools provide dozens of pre-defined operators and ML algorithms, which data scientists compose into sequences of operators (called *pipelines*) using high-level APIs (e.g., in Python).

ML.Net, the ML toolkit used in this paper, is a C# library that runs on a managed runtime with garbage collection and Just-In-Time (JIT) compilation. Unmanaged C/C++ code can also be employed to speed up processing when possible. Internally, ML.Net operators consume data vectors as input and produce one (or more) vectors as output. [1] Vectors are immutable whereby multiple downstream operators can safely consume the same input without triggering any re-execution. Upon pipeline initialization, 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. Although ML.Net supports Neural Network models, in this work we only focus on pipelines composed by featurizers and classical ML models (e.g., trees, logistic regression, etc.).

Pipelines are first trained using large datasets to estimate models' parameters. ML.Net models are exported as compressed files containing several directories, one per pipeline operator, where each directory stores operator parameters in either binary or plain text files. ML.Net, as other systems, aims to minimize the overhead of deploying trained pipelines in production by serving them into black box containers, where the same code is used for both training and inference. Figure 2 depicts a set of black box models where the invocation of the function chain (e.g., `predict()`) on a pipeline returns the result of the prediction: throughout this execution chain, inputs are pulled through each operator to produce intermediate results that are input to the following operators, similarly to the well-known Volcano-style iterator model of databases [36]. To optimize the performance, ML.Net (and systems such as Clipper among others) applies techniques such as handling multiple requests in batches and caching the results of the inference if some predictions are frequently issued for the same pipeline. However, these techniques assume no knowledge and no control over the pipeline, and are unaware of its internal structure. Despite being regarded as a good practice [65], the black box, container-based design hides the structure of each served model and prevents the system from controlling and optimizing the pipeline execution. Therefore, under this approach, there is no principled way neither for sharing optimizations between pipelines, nor to improve the end-to-end execution of individual pipelines. More concretely, we observed the following limitations in current state-of-the-art prediction serving systems.

Memory Waste: Containerization of pipelines disallows

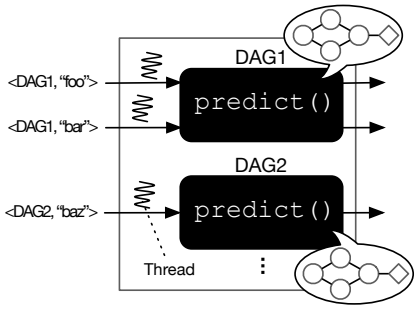

Figure 2: A representation of how existing systems handle prediction requests. Each pipeline is surfaced externally as a black box function. When a prediction request is issued (`predict()`), a thread is dispatched to execute the chain as a single function call.

any sharing of resources and runtimes [2] between pipelines, therefore only a few (tens of) models can be deployed per machine. Conversely, ML frameworks such as ML.Net have a known set of operators to start with, and featurizers or models trained over similar datasets have a high likelihood of sharing parameters. For example, transfer learning, A/B testing, and personalized models are common in practice; additionally, tools like ML.Net suggest default training configurations to users given a task and a dataset, which leads to many pipelines with similar structure and common objects and parameters. To better illustrate this scenario, we pick a Sentiment Analysis (SA) task with 250 different versions of the pipeline of Figure 1 trained by data scientists at Microsoft.

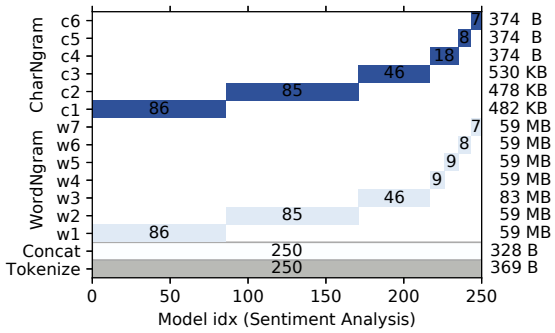

Figure 3: How many identical operators can be shared in multiple SA pipelines. CharNgram and WordNgram operators have variations that are trained on different hyper-parameters. On the right we report operators sizes.

Figure 3 shows how many different (parameterized) operators are used, and how often they are used within the 250 pipelines. While some operators like linear regression (whose weights fit in ~15MB) are unique to each pipeline,

[1] Note that this is a simplification. ML.Net in fact support several data types. We refer readers to [23] for more details.

[2] One instance of model pipeline in production easily occupies 100s of MB of main memory.

and thus not shown in Figure 3, many other operators can be shared among pipelines, therefore allowing more aggressive packing of models: Tokenize and Concat are used with the same parameters in all pipelines; Ngram operators have only a handful of versions, where most pipelines use the same version of the operators. This suggests that the resource utilization of current black box approaches can be largely improved.

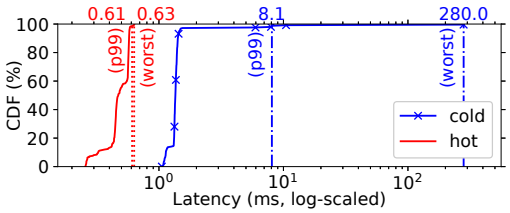

Figure 4: CDF of latency of prediction requests of 250 DAGs. We denote the first prediction as *cold*; the *hot* line is reported as average over 100 predictions after a warm-up period of 10 predictions. We present the 99th percentile and worst case latency values.

Prediction Initialization: ML.Net employs a pull-based execution model that lazily materializes input feature vectors, and tries to reuse existing vectors between 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, thus making latency of predictions sub-optimal and hard to predict. Furthermore, at prediction time ML.Net deploys pipelines as in the training phase, which requires initialization of function chain call, reflection for type inference and JIT compilation. While this composability conveniently hides complexities and allows changing implementations during training, it is of little use during inference, when a model has a defined structure and its operators are fixed. In general, the above problems result in difficulties in providing strong tail latency guarantees by ML-as-a-service providers. Figure 4 describes this situation, where the performance of *hot* predictions over the 250 sentiment analysis pipelines with memory already allocated and JIT-compiled code is more than two orders of magnitude faster than the worst *cold* case version for the same pipelines.

To drill down more into the problem, we found that 57.4% of the total execution time for a single cold prediction is spent in pipeline analysis and initialization of the function chain, 36.5% in JIT compilation and the remaining is actual computation time.

Infrequent Accesses: In order to meet milliseconds-level latencies [61], model pipelines have to reside in main memory (possibly already warmed-up), since they can have MBs to GBs (compressed) size on disk, with loading and initialization times easily exceeding several seconds. A common practice in production settings is to unload a pipeline if not accessed after a certain period of time (e.g., a few hours). Once evicted, successive accesses will incur a model loading penalty and warming-up, therefore violating Service Level Agreement (SLA).

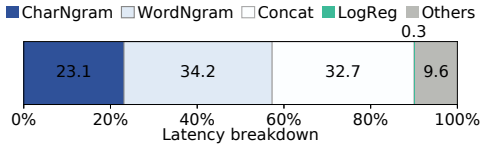

Figure 5: Latency breakdown of a sentiment analysis pipeline: each frame represents the relative wall clock time spent on an operator.

Operator-at-a-time Model: As previously described, predictions over ML.Net pipelines are computed by pulling records through a sequence of operators, each of them operating over the input vector(s) and producing one or more new vectors. While (as is common practice for in-memory data-intensive systems [45, 58, 24]) some interpretation overheads are eliminated via JIT compilation, operators in ML.Net (and in other tools) are "logical" entities (e.g., linear regression, tokenizer, one-hot encoder, etc.) with diverse performance characteristics. Figure 5 shows the latency breakdown of one execution of the SA pipeline of Figure 1, where the only ML operator (linear regression) takes two orders-of-magnitude less time with respect to the slowest operator (WordNgram). 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 [33, 66]. ML.Net operator-at-a-time model [66] (as other libraries missing an optimization layer, such as Scikit-learn) is therefore sub-optimal in that computation is organized around logical operators, ignoring how those operators behave together: in the example of the sentiment analysis pipeline at hand, linear 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. As we will see in the following sections, PRETZEL's optimizer is able to detect this situation and generate an execution plan that is several times faster than the ML.Net version of the pipeline.

Coarse Grained Scheduling: Scheduling CPU resources carefully is essential to serve highly concurrent requests and run machines to maximum utilization. Under

the black box approach: (1) a thread pool is used to serve multiple concurrent requests to the same model pipeline; (2) for each request, one thread handles the execution of a full pipeline sequentially [3], where one operator is active at each point in time; (3) shared operators/parameters are instantiated and evaluated multiple times (one per container) independently; (4) thread allocation is managed by the OS; and (5) load balancing is achieved "externally" by replicating containers when performance degradation is observed. We found this design sub-optimal, especially in heavily skewed scenarios where a small amount of popular models are scored more frequently then others: indeed, in this setting the popular models will be replicated (linearly increasing the resources used) whereas containers of less popular pipelines will run underutilized, therefore decreasing the total resource utilization. The above problem is currently out-of-scope for black box, container-based prediction serving systems because they lack visibility into pipelines execution, and they do not allow models to properly share computational resources.

After highlighting the major inefficiencies of current black box prediction serving systems, we discuss a set of design principles for white box prediction serving.

3 White Box Prediction Serving: Design Principles

Based on the observations of Section 2, we argue that all previously mentioned limitations can be overcome by embracing a *white box approach* allowing to optimize the execution of predictions both horizontally *end-to-end* and vertically *among multiple model pipelines*.

White Box Prediction Serving: Model containerization disallows any sharing of optimizations, resources, and costs between pipelines. By choosing a white box architecture, pipelines can co-exist on the same runtime; unpopular pipelines can be maintained up and warm, while popular pipelines pay the bills. Thorough scheduling of pipelines' components can be managed within the runtime so that optimal allocation decisions can be made for running machines to high utilization. Nevertheless, if a pipeline requires exclusive access to computational or memory resources, a proper reservation-based allocation strategy can be enforced by the scheduler so that container-based execution can be emulated.

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 gracefully degrade with load increase. 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 [45]; 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 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 (Figure 3), 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 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.

4 The Pretzel System

Following the above guidelines, we implemented PRETZEL, a novel white box system for cloud-based inference of model pipelines. PRETZEL views models as database queries and employs database techniques to optimize DAGs and improve end-to-end performance (Section 4.1.2). The problem of optimizing co-located pipelines is casted as a multi-query optimization and techniques such as view materialization (Section 4.3) 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 upfront at initialization time.

PRETZEL is organized in several components. A *dataflow-style language integrated API* called Flour (Section 4.1.1) with related *compiler* and *optimizer* called Oven (Section 4.1.2) are used in concert to convert ML.Net pipelines into *model plans*. An Object Store (Section 4.1.3) saves and shares parameters among plans. A Runtime (Section 4.2.1) manages compiled plans and their execution, while a Scheduler (Section 4.2.2) 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. During the *offline phase* (Section 4.1), ML.Net's pre-trained pipelines

[3] Certain pipelines allow multi-threaded execution, but here we evaluate only single-threaded ones to estimate the per-thread efficiency.

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 where memory resources and threads are pooled at runtime. Model plans are registered for prediction serving in the Runtime where physical stages and parameters are shared between pipelines with similar model plans. In the *on-line phase* (Section 4.2), 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.

Figures 6 and 7 pictorially summarize the above descriptions; note that only the on-line phase is executed at inference time, whereas the model plans are generated completely off-line. Next, we will describe each layer composing the PRETZEL prediction system.

4.1 Off-line Phase

4.1.1 Flour

The goal of Flour is to provide an intermediate representation between ML frameworks (currently only ML.Net) and PRETZEL, that is both easy to target and amenable to optimizations. Once a pipeline is ported into Flour, it can be optimized and compiled (Section 4.1.2) into a model plan before getting fed into PRETZEL Runtime for on-line scoring. Flour is a language-integrated API similar to KeystoneML [55], RDDs [63] or LINQ [42] where sequences of *transformations* are chained into DAGs and lazily compiled for execution.

Listing 1 shows how the sentiment analysis pipeline of Figure 1 can be expressed in Flour. Flour programs are composed by transformations where a one-to-many mapping exists between ML.Net operators and Flour transformations (i.e., one operator in ML.Net can be mapped to many transformations in Flour). Each Flour program starts from a `FlourContext` object wrapping the Object Store. Subsequent method calls define a DAG of transformations, which will end with a call to `Plan` to instantiate the model plan before feeding it into PRETZEL Runtime. For example, in lines 2 and 3 of Listing 1 the `CSV.FromText` call is used to specify that the target DAG accepts as input text in CSV format where fields are comma separated. Line 4 specifies the schema for the input data, where `TextReview` is a class whose parameters specify the schema fields names, types, and order. The successive call to `Select` in line 5 is used to pick the `Text` column among all the fields, while the call to `Tokenize` in line 6 is used to split the input fields into tokens. Lines 8 and 9 contain the two branches defining the char-level and word-level n-gram transformations, which are then merged with the `Concat` transform in lines 10/11 before the linear binary classifier of line 12. Both char and word n-gram transformations are parameterized by the number of n-grams and maps translating n-grams into numerical format (not shown in the Listing). Additionally, each Flour transformation accepts as input an optional set of statistics gathered from training. These statistics are used by the compiler to generate physical plans more efficiently tailored to the model characteristics. Example statistics are max vector size (to define the minimum size of vectors to fetch from the pool at prediction time, as in Section 4.2), dense/sparse representations, etc.

We have instrumented the ML.Net library to collect statistics from training and with the related bindings to the Object Store and Flour to automatically extract Flour programs from pipelines once trained.

Listing 1: Flour program for the SA pipeline. Parameters are extracted from the original ML.Net pipeline.

```
1  var fContext = new FlourContext(objectStore, ...)
2  var tTokenizer = fContext.CSV
3                   .FromText(',')
4                   .WithSchema<TextReview>()
5                   .Select("Text")
6                   .Tokenize();
7
8  var tCNgram = tTokenizer.CharNgram(numCNgrms, ...);
9  var tWNgram = tTokenizer.WordNgram(numWNgrms, ...);
10 var fPrgrm = tCNgram
11              .Concat(tWNgram)
12              .ClassifierBinaryLinear(cParams);
13
14 return fPrgrm.Plan();
```

4.1.2 Oven

With Oven, our goal is to bring query compilation and optimization techniques into ML.Net.

Optimizer: When `Plan` is called on a Flour transformation's reference (e.g., `fPrgrm` in line 14 of Listing 1), all transformations leading to it are wrapped and analyzed. Oven follows the typical rule-based database optimizer design where operator graphs (query plans) are transformed by a set of rules until a fix-point is reached (i.e., the graph does not change after the application of any rule). The goal of Oven Optimizer is to transform an input graph of Flour transformations into a stage graph, where each stage contains one or more transformations. To group transformations into stages we used the Tupleware's hybrid approach [33]: memory-intensive transformations (such as most featurizers) are pipelined together in a single pass over the data. This strategy achieves best data locality because records are likely to reside in

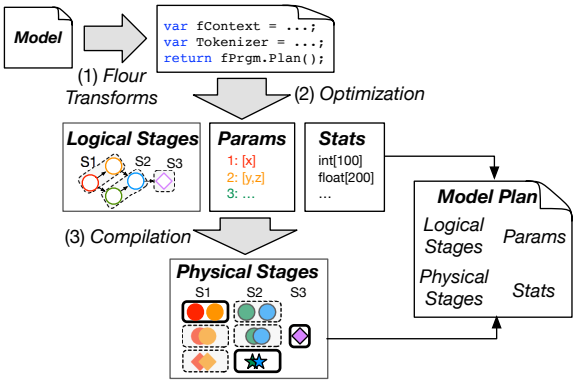

Figure 6: Model optimization and compilation in PRETZEL. In (1), a model is translated into a Flour program. (2) Oven Optimizer generates a DAG of logical stages from the program. Additionally, parameters and statistics are extracted. (3) A DAG of physical stages is generated by the Oven Compiler using logical stages, parameters, and statistics. A model plan is the union of all the elements.

CPU L1 caches [40, 45]. Compute-intensive transformations (e.g., vector or matrix multiplications) are executed one-at-a-time so that Single Instruction, Multiple Data (SIMD) vectorization can be exploited, therefore optimizing the number of instructions per record [66, 26]. Transformation classes are annotated (e.g., 1-to-1, 1-to-n, memory-bound, compute-bound, commutative and associative) to ease the optimization process: no dynamic compilation [33] is necessary since the set of operators is fixed and manual annotation is sufficient to generate properly optimized plans [4].

Stages are generated by traversing the Flour transformations graph repeatedly and applying rules when matching conditions are satisfied. Oven Optimizer consists of an extensible number of *rewriting steps*, each of which in turn is composed of a set of rules performing some modification on the input graph. Each rewriting step is executed sequentially: within each step, the optimizer iterates over its full set of rules until an iteration exists such that the graph is not modified after all rules are evaluated. When a rule is active, the graph is traversed (either top-down, or bottom up, based on rule internal behavior; Oven provides graph traversal utilities for both cases) and the rewriting logic is applied if the matching condition is satisfied over the current node. In its current implementation, the Oven Optimizer is composed of 4 rewriting steps:

InputGraphValidatorStep: This step comprises three rules, performing schema propagation, schema validation and graph validation. Specifically, the rules propagate schema information from the input to the final transformation in the graph, and validate that (1) each transformation's input schema matches with the transformation semantics (e.g., a WordNgram has a string type as input schema, or a linear learner has a vector of floats as input), and (2) the transformation graph is well-formed (e.g., a final predictor exists).

StageGraphBuilderStep: It contains two rules that rewrite the graph of (now schematized) Flour transformations into a stage graph. Starting with a valid transformation graph, the rules in this step traverse the graph until a pipeline-breaking transformation is found, i.e., a Concat or an n-to-1 transformation such as an aggregate used for normalization (e.g., L2). These transformations, in fact, require data to be fully scanned or materialized in memory before the next transformation can be executed. For example, operations following a Concat require the full feature vector to be available, or a Normalizer requires the L2 norm of the complete vector. The output of the `StageGraphBuilderStep` is therefore a stage graph, where each stage internally contains one or more transformations. Dependencies between stages are created as aggregation of the dependencies between the internal transformations. By leveraging the stage graph, PRETZEL is able to considerably decrease the number of vectors (and as a consequence the memory usage) with respect to the operator-at-a-time strategy of ML.Net.

StageGraphOptimizerStep: This step involves 9 rules that rewrite the graph in order to produce an optimal (logical) plan. The most important rules in this step rewrite the stage graph by (1) removing unnecessary branches (similar to common sub-expression elimination); (2) merging stages containing equal transformations (often generated by traversing graphs with branches); (3) inlining stages that contain only one transform; (4) pushing linear models through Concat operations; and (5) removal of unnecessary stages (e.g., when linear models are pushed through Concat operations, the latter stage can be removed if not containing any other additional transformation).

OutputGraphValidatorStep: This last step is composed of 6 rules. These rules are used to generate each stage's schema out of the schemas of the single internal transformations. Stage schema information will be used at runtime to request properly typed vectors. Additionally, some training statistics are applied at this step: transformations are labeled as sparse or dense, and dense compute-bound operations are labeled as vectorizable. A final validation check is run to ensure that the stage graph is well-formed.

In the example sentiment analysis pipeline of Figure

[4]Note that ML.Net does provide a second order operator accepting arbitrary code requiring dynamic compilation. However, this is not supported in our current version of PRETZEL.

1, Oven is able to recognize that the Linear Regression can be pushed into CharNgram and WordNgram, therefore bypassing the execution of Concat. Additionally, Tokenizer can be reused between CharNgram and WordNgram, therefore it will be pipelined with CharNgram (in one stage) and a dependency between CharNgram and WordNgram (in another stage) will be created. The final plan will therefore be composed of 2 stages, versus the initial 4 operators (and vectors) of ML.Net.

Model Plan Compiler: Model plans have two DAGs: a DAG of *logical stages*, and a DAG of *physical stages*. Logical stages are an abstraction of the results of the Oven Optimizer; physical stages contain the actual code that will be executed by the PRETZEL runtime. For each given DAG, there is a 1-to-n mapping between logical to physical stages so that a logical stage can represent the execution code of different physical implementations. A physical implementation is selected based on the parameters characterizing a logical stage and available statistics.

Plan compilation is a two step process. After the stage DAG is generated by the Oven Optimizer, the Model Plan Compiler (MPC) maps each stage into its logical representation containing all the parameters for the transformations composing the original stage generated by the optimizer. Parameters are saved for reuse in the Object Store (Section 4.1.3). Once the logical plan is generated, MPC traverses the DAG in topological order and maps each logical stage into a physical implementation. Physical implementations are AOT-compiled, parameterized, lock-free computation units. Each physical stage can be seen as a parametric function which will be dynamically fed at runtime with the proper data vectors and pipeline-specific parameters. This design allows PRETZEL runtime to share the same physical implementation between multiple pipelines and no memory allocation occurs on the prediction path (more details in Section 4.2.1). Logical plans maintain the mapping between the pipeline-specific parameters saved in the Object Store and the physical stages executing on the Runtime as well as statistics such as maximum vector size (which will be used at runtime to request the proper amount of memory from the pool). Figure 6 summarizes the process of generating model plans out of ML.Net pipelines.

4.1.3 Object Store

The motivation behind Object Store is based on the insights of Figure 3: since many DAGs have similar structures, sharing operators' state (parameters) can considerably improve memory footprint, and consequently the number of predictions served per machine. An example

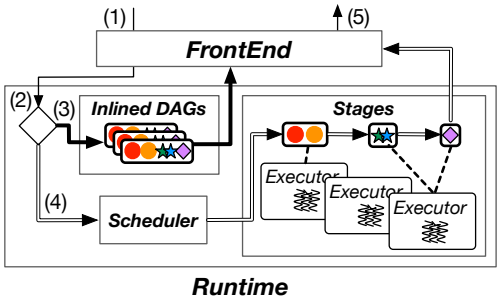

Figure 7: (1) When a prediction request is issued, (2) the Runtime determines whether to serve the prediction using (3) the request/response engine or (4) the batch engine. In the latter case, the Scheduler takes care of properly allocating stages over the Executors running concurrently on CPU cores. (5) The FrontEnd returns the result to the Client once all stages are complete.

is language dictionaries used for input text featurization, which are often in common among many models and are relatively large. The Object Store is populated off-line by MPC: when a Flour program is submitted for planning, new parameters are kept in the Object Store, while parameters that already exist are ignored and the stage information is rewritten to reuse the previously loaded one. Parameters equality is computed by looking at the checksum of the serialized version of the objects.

4.2 On-line Phase

4.2.1 Runtime

Initialization: Model plans generated by MPC are registered in the PRETZEL Runtime. Upon registration, a unique pipeline ID is generated, and physical stages composing a plan are loaded into a system *catalog*. If two plans use the same physical stage, this is loaded only once in the catalog so that similar plans may share the same physical stages during execution. When the Runtime starts, a set of vectors and long-running thread pools (called *Executors*) are initialized. Vector pools are allocated per Executor to improve locality [35]; Executors are instead managed by the Scheduler to execute physical stages (Section 4.2.2) or used to manage incoming prediction requests by the FrontEnd. Allocations of vector and thread pools are managed by configuration parameters, and allow PRETZEL to decrease the time spent in allocating memory and threads during prediction time.

Execution: Inference requests for the pipelines registered into the system can be submitted through the FrontEnd by specifying the pipeline ID, and a set of input records. Figure 7 depicts the process of on-line inference. PRET-

ZEL comes with a *request-response engine* and a *batch engine*. The request-response engine is used by single predictions for which latency is the major concern whereby context-switching and scheduling overheads can be costly. Conversely, the batch engine is used when a request contains a batch of records, or when the prediction time is such that scheduling overheads can be considered as negligible (e.g., few hundreds of microseconds). The request-response engine inlines the execution of the prediction within the thread handling the request: the pipeline physical plan is JIT-compiled into a unique function call and scored. Instead, by using the batch engine requests are forwarded to the Scheduler that decides where to allocate physical stages based on the current runtime and resource status. Currently, whether to use the request-response or batch engine is set through a configuration parameter passed when registering a plan. In the future we plan to adaptively switch between the two.

4.2.2 Scheduler

In PRETZEL, model plans share resources, thus scheduling plans appropriately is essential to ensure scalability and optimal machine utilization while guaranteeing the performance requirements.

The Scheduler coordinates the execution of multiple stages via a late-binding event-based scheduling mechanism similar to task scheduling in distributed systems [47, 63, 60]: each core runs an Executor instance whereby all Executors pull work from a shared pair of queues: one *low priority* queue for newly submitted plans, and one *high priority* queue for already started stages. At runtime, a scheduling event is generated for each stage with related set of input/output vectors, and routed over a queue (low priority if the stage is the head of a pipeline, high priority otherwise). Two queues with different priorities are necessary because of memory requirements. Vectors are in fact requested per pipeline (not per stage) and lazily fulfilled when a pipeline's first stage is being evaluated on an Executor. Vectors are then utilized and not re-added to the pool for the full execution of the pipeline. Two priority queues allow started pipelines to be scheduled earlier and therefore return memory quickly.

Reservation-based Scheduling: Upon model plan registration, PRETZEL offers the option to reserve memory or computation resources for exclusive use. Such resources reside on different, pipeline-specific pools, and are not shared among plans, therefore enabling container-like provision of resources. Note however that parameters and physical stage objects remain shared between pipelines even if reservation-based scheduling is requested.

4.3 Additional Optimizations

Sub-plan Materialization: Similarly to materialized views in database multi-query optimization [37, 29], results of installed physical stages can be reused between different model plans. When plans are loaded in the runtime, PRETZEL keeps track of physical stages and enables caching of results when a stage with the same parameters is shared by many model plans. Hashing of the input is used to decide whether a result is already available for that stage or not. We implemented a simple Least Recently Used (LRU) strategy on top of the Object Store to evict results when a given memory threshold is met.

External Optimizations: While the techniques described so far focus mostly on improvements that other prediction serving systems are not able to achieve due to their black box nature, PRETZEL FrontEnd also supports "external" optimizations such as the one provided in Clipper and Rafiki. Specifically, the FrontEnd currently implements prediction results caching (with LRU eviction policy) and delayed batching whereby inference requests are buffered for a user-specified amount of time and then submitted in batch to the Runtime. These external optimizations are orthogonal to PRETZEL's techniques, so both are applicable in a complementary manner.

5 Evaluation

PRETZEL implementation is a mix of C# and C++. In its current version, the system comprises 12.6K LOC (11.3K in C#, 1.3K in C++) and supports about two dozens of ML.Net operators, among which linear models (e.g., linear/logistic/Poisson regression), tree-based models, clustering models (e.g., K-Means), Principal Components Analysis (PCA), and several featurizers.

Scenarios: The goals of our experimental evaluation are to evaluate how the white box approach performs compared to black box. We will use the following scenarios to drive our evaluation:

- *memory*: in the first scenario, we want to show how much memory saving PRETZEL's white box approach is able to provide with respect to regular ML.Net and ML.Net boxed into Docker containers managed by Clipper.
- *latency*: this experiment mimics a request/response pattern (e.g., [19]) such as a personalized web-application requiring minimal latency. In this scenario, we run two different configurations: (1) a micro-benchmark measuring the time required by a system to render a prediction; and (2) an experiment measuring the total end-to-end latency observed by

Table 1: Characteristics of pipelines in experiments.

Type	Sentiment Analysis (SA)	Attendee Count (AC)
Input	Plain Text (variable length)	Structured Text (40 dimensions)
Size	50MB - 100MB (Mean: 70MB)	10KB - 20MB (Mean: 9MB)
Featurizers	N-gram with dictionaries (∼1M entries)	PCA, KMeans, Ensemble of multiple models

a client submitting a request.

- *throughput*: this scenario simulates a batch pattern (e.g., [8]) and we use it to assess the throughput of PRETZEL compared to ML.Net.
- *heavy-load*: we finally mix the above experiments and show PRETZEL's ability to maintain high throughput and graceful degradation of latency, as load increases. To be realistic, in this scenario we generate skewed load across different pipelines. As for the *latency* experiment, we report first the PRETZEL's performance using a micro-benchmark, and then we compare it against the containerized version of ML.Net in an end-to-end setting.

Configuration: All the experiments reported in the paper were carried out on a Windows 10 machine with 2 × 8-core Intel Xeon CPU E5-2620 v4 processors at 2.10GHz with Hyper Threading disabled, and 32GB of RAM. We used .Net Core version 2.0, ML.Net version 0.4, and Clipper version 0.2. For ML.Net, we use two black box configurations: a non-containerized one (1 ML.Net instance for all models), and a containerized one (1 ML.Net instance for each model) where ML.Net is deployed as Docker containers running on Windows Subsystem for Linux (WSL) and orchestrated by Clipper. We commonly label the former as just ML.Net; the latter as ML.Net + Clipper. For PRETZEL we AOT-compile stages using CrossGen [16]. For the end-to-end experiments comparing PRETZEL and ML.Net + Clipper, we use an ASP.Net FrontEnd for PRETZEL; the Redis front-end for Clipper. We run each experiment 3 times and report the median.

Pipelines: Table 1 describes the two types of model pipelines we use in the experiments: 250 unique versions of Sentiment Analysis (SA) pipeline, and 250 different pipelines implementing Attendee Count (AC): a regression task used internally to predict how many attendees will join an event. Pipelines within a category are similar: in particular, pipelines in the SA category benefit from sub-plan materialization, while those in the AC category are more diverse and do not benefit from it. These latter pipelines comprise several ML models forming an ensemble: in the most complex version, we have a dimensionality reduction step executed concurrently with a KMeans clustering, a TreeFeaturizer, and multi-class tree-based classifier, all fed into a final tree (or forest) rendering the prediction. SA pipelines are trained and scored over Amazon Review dataset [38]; AC ones are trained and scored over an internal record of events.

5.1 Memory

In this experiment, we load all models and report the total memory consumption (model + runtime) per model category. SA pipelines are large and therefore we expect memory consumption (and loading time) to improve considerably within this class, proving that PRETZEL's Object Store allows to avoid the cost of loading duplicate objects. Less gains are instead expected for the AC pipelines because of their small size.

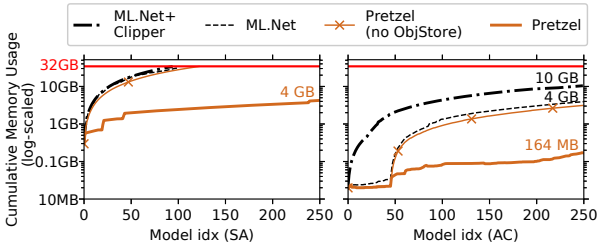

Figure 8: Cumulative memory usage (log-scaled) of the pipelines in PRETZEL, ML.Net and ML.Net + Clipper. The horizontal line represents the machine's physical memory (32GB). Only PRETZEL is able to load all SA pipelines within the memory limit. For AC, PRETZEL uses one order of magnitude less memory than ML.Net and ML.Net + Clipper. The memory usage of PRETZEL without Object Store is almost on par with ML.Net.

Figure 8 shows the memory usage for loading all the 250 model pipelines in memory, for both categories. For SA, only PRETZEL with Object Store enabled can load all pipelines. [5] For AC, all configurations are able to load the entire working set, however PRETZEL occupies only 164MBs: about 25× less memory than ML.Net and 62× less than ML.Net + Clipper. Given the nature of AC models (i.e., small in size), from Figure 8 we can additionally notice the overhead (around 2.5×) of using a container-based black box approach vs regular ML.Net.

[5] Note that for ML.Net, ML.Net + Clipper and PRETZEL without Object Store configurations we can load more models and go beyond the 32GB limit. However, models are swapped to disk and the whole system becomes unstable.

Keeping track of pipelines' parameters also helps reducing the time to load models: PRETZEL takes around 2.8 seconds to load 250 AC pipelines while ML.Net takes around 270 seconds. For SA pipelines, PRETZEL takes 37.3 seconds to load all 250 pipelines, while ML.Net fills up the entire memory (32GB) and begins to swap objects after loading 75 pipelines in around 9 minutes.

5.2 Latency

In this experiment we study the latency behavior of PRETZEL in two settings. First, we run a micro-benchmark directly measuring the latency of rendering a prediction in PRETZEL. Additionally, we show how PRETZEL's optimizations can improve the latency. Secondly, we report the end-to-end latency observed by a remote client submitting a request through HTTP.

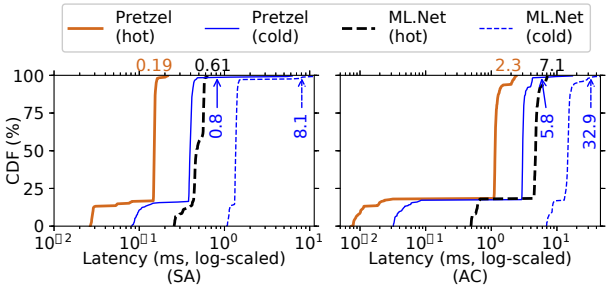

Figure 9: Latency comparison between ML.Net and PRETZEL. The accompanying blue lines represent the *cold* latency (first execution of the pipelines). On top are the P99 latency values: the hot case is above the horizontal line and the cold case is annotated with an arrow.

5.2.1 Micro-benchmark

Inference requests are submitted sequentially and in isolation for one model at a time. For PRETZEL we use the request-response engine over one single core. The comparison between PRETZEL and ML.Net for the SA and AC pipelines is reported in Figure 9. We start with studying *hot* and *cold* cases while comparing PRETZEL and ML.Net. Specifically, we label as cold the first prediction requested for a model; the successive 10 predictions are then discarded and we report hot numbers as the average of the following 100 predictions.

If we directly compare PRETZEL with ML.Net, PRETZEL is 3.2× and 3.1× faster than ML.Net in the 99th percentile latency in hot case (denoted by $P99_{hot}$), and about 9.8× and 5.7× in the $P99_{cold}$ case, for SA and AC pipelines, respectively. If instead we look at the difference between cold and hot cases relative to each system, PRETZEL again provides improvements over ML.Net. The $P99_{cold}$ is about 13.3× and 4.6× the $P99_{hot}$ in ML.Net, whereas in PRETZEL $P99_{cold}$ is around 4.2× and 2.5× from the $P99_{hot}$ case. Furthermore, PRETZEL is able to mitigate the long tail latency (worst case) of cold scoring. In SA pipelines, the worst case latency is 460.6× off the $P99_{hot}$ in ML.Net, whereas PRETZEL shows a 33.3× difference. Similarly, in AC pipelines the worst case is 21.2× $P99_{hot}$ for ML.Net, and 7.5× for PRETZEL.

To better understand the effect of PRETZEL's optimizations on latency, we turn on and off some optimizations and compare the performance.

AOT compilation: This options allows PRETZEL to preload all stage code into cache, removing the overhead of JIT compilation in the cold cases. Without AOT compilation, latencies of cold predictions increase on average by 1.6× and 4.2× for SA and AC pipelines, respectively.

Vector Pooling: By creating pools of pre-allocated vectors, PRETZEL can minimize the overhead of memory allocation at prediction time. When we do not pool vectors, latencies increase in average by 47.1% for hot and 24.7% for cold, respectively.

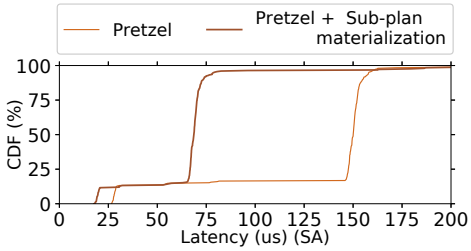

Figure 10: Latency of PRETZEL to run SA models with and without sub-plan materialization. Around 80% of SA pipelines show more than 2× speedup. Sub-plan materialization does not apply for AC pipelines.

Sub-plan Materialization: If different pipelines have common featurizers (e.g., SA as shown in Figure 3), we can further apply sub-plan materialization to reduce the latency. Figure 10 depicts the effect of sub-plan materialization over prediction latency for hot requests. In general, for the SA pipelines in which sub-plan materialization applies, we can see an average improvement of 2.0×, while no pipeline shows performance deterioration.

5.2.2 End-to-end

In this experiment we measure the end-to-end latency from a client submitting a prediction request. For PRETZEL, we use the ASP.Net FrontEnd, and we compare against ML.Net + Clipper. The end-to-end latency con-

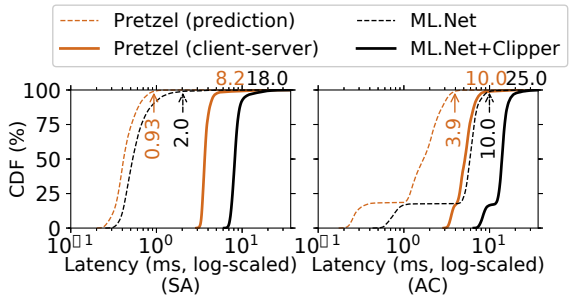

Figure 11: The latency comparison between ML.Net + Clipper and PRETZEL with ASP.Net FrontEnd. The overhead of client-server communication compared to the actual prediction is similar in both PRETZEL and ML.Net: the end-to-end latency compared to the just prediction latency is 9× slower in SA and 2.5× in AC, respectively.

siders both the prediction latency (i.e., Figure 9) as well as any additional overhead due to client-server communication. As shown in Figure 11, the latter overhead in both PRETZEL and ML.Net + Clipper is in the milliseconds range (around 4ms for the former, and 9 for the latter). Specifically, with PRETZEL, clients observe a latency of 4.3ms at *P*99 for SA models (vs. 0.56ms *P*99 latency of just rendering a prediction) and a latency of 7.3ms for AC models (vs. 3.5ms). In contrast, in ML.Net + Clipper, clients observe 9.3ms latency at *P*99 for SA models, and 18.0ms at *P*99 for AC models.

5.3 Throughput

In this experiment, we run a micro-benchmark assuming a batch scenario where all 500 models are scored several times. We use an API provided by both PRETZEL and ML.Net, where we can execute prediction queries in batches: in this experiment we fixed the batch size at 1000 queries. We allocate from 2 up to 13 CPU cores to serve requests, while 3 cores are reserved to generate them. The main goal is to measure the maximum number of requests PRETZEL and ML.Net can serve per second.

Figure 12 shows that PRETZEL's throughput (queries per second) is up to 2.6× higher than ML.Net for SA models, 10× for AC models. PRETZEL's throughput scales on par with the expected ideal scaling. Instead, ML.Net suffers from higher latency in rendering predictions and from lower scalability when the number of CPU cores increases. This is because each thread has its own internal copy of models whereby cache lines are not shared, thus increasing the pressure on the memory subsystem: indeed, even if the parameters are the same, the model objects are allocated to different memory areas.

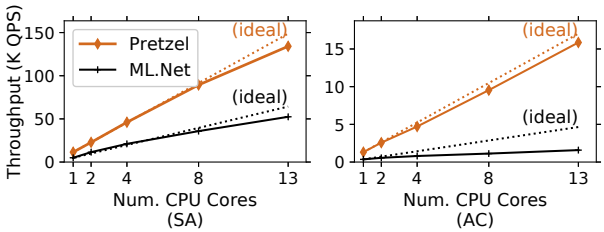

Figure 12: The average throughput computed among the 500 models to process one million inputs each. We scale the number of CPU cores on the x-axis and the number of prediction queries to be served per second on the y-axis. PRETZEL scales linearly to the number of CPU cores.

5.4 Heavy Load

In this experiment, we show how the performance changes as we change the load. To generate a realistic load, we submit requests to models by following the Zipf distribution ($\alpha = 2$).[6] As in Section 5.2, we first run a micro-benchmark, followed by an end-to-end comparison.

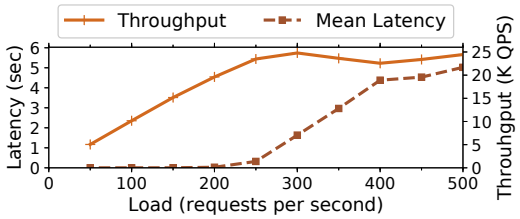

Figure 13: Throughput and latency of PRETZEL under the heavy load scenario. We maintain all 500 models in-memory within a PRETZEL instance, and we increase the load by submitting more requests per second. We report latency measurements from latency-sensitive pipelines, and the total system throughput.

5.4.1 Micro-benchmark

We load all 500 models in one PRETZEL instance. Among all models, we assume 50% to be "latency-sensitive" and therefore we set a batch size of 1. The remaining 50% models will be requested with 100 queries in a batch. As in the throughput experiment, we use the batch engine with 13 cores to serve requests and 3 cores to generate load. Figure 13 reports the average latency of latency-sensitive models and the total system throughput under different load configurations. As we increase the number of requests, PRETZEL's throughput increases linearly until it stabilizes at about 25k queries per second. Similarly, the average latency of latency-sensitive pipelines gracefully increases linearly with the load.

[6]The number of requests to the *i*th most popular models is proportional to $i^{-\alpha}$, where α is the parameter of the distribution.

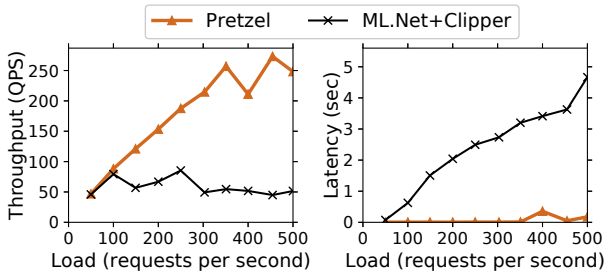

Figure 14: Throughput and latency of PRETZEL and ML.Net + Clipper under the end-to-end heavy load scenario. We use 250 AC pipelines to allow both systems to have all pipelines in memory.

Reservation Scheduling: If we want to guarantee that the performance of latency-critical pipelines is not degrading excessively even under high load, we can enable reservation scheduling. If we run the previous experiment reserving one core (and related vectors) for one model, this does not encounter any degradation in latency (max improvement of 3 orders of magnitude) as the load increases, while maintaining similar system throughput.

5.4.2 End-to-end

In this setup, we periodically send prediction requests to PRETZEL with the ASP.Net FrontEnd and ML.Net + Clipper. We assume all pipelines to be latency-sensitive, thus we set a batch of 1 for each request. As we can see in Figure 14, PRETZEL's throughput keeps increasing up to around 300 requests per second. If the load exceeds that point, the throughput and the latency begin to fluctuate. On the other hand, the throughput of ML.Net + Clipper is considerably lower than PRETZEL's and does not scale as the load increases. Also the latency of ML.Net + Clipper is several folds higher than with PRETZEL. The difference is due to the overhead of maintaining hundreds of Docker containers; too many context switches occur across/within containers.

6 Limitations and Future Work

Off-line Phase: PRETZEL has two limitations regarding Flour and Oven design. First, PRETZEL currently has several logical and physical stages classes, one per possible implementation, which make the system difficult to maintain in the long run. Additionally, different back-ends (e.g., PRETZEL currently supports operators implemented in C# and C++, and experimentally on FPGA [53]) require all specific operator implementations. We are however confident that this limitation will be overcome once code generation of stages will be added (e.g., with hardware-specific templates [41]). Secondly, Flour and Oven are currently limited to pipelines authored in ML.Net, and porting models from different frameworks to the white box approach may require non-trivial work. On the long run our goal is, however, to target unified formats such as ONNX [7]; this will allow us to apply the discussed techniques to models from other ML frameworks as well.

On-line Phase: PRETZEL's fine-grained, stage-based scheduling may introduce additional overheads in contrasts to coarse-grained whole pipeline scheduling due to additional buffering and context switching. However, such overheads are related to the system load and therefore controllable by the scheduler. Additionally, we found GC overheads to introduce spikes in latency. Although our implementation tries to minimize the number of objects created at runtime, in practice we found that long tail latencies are common. On white box architectures, failures happening during the execution of a model may jeopardize the whole system. We are currently working on isolating model failures over the target Executor. Finally, PRETZEL runtime currently runs on a single-node. An experimental scheduler adds Non Uniform Memory Access (NUMA) awareness to scheduling policies. We expect this scheduler to bring benefits for models served from large instances (e.g., [12]). We expect in the future to be able to scale the approach over distributed machines, with automatic scale in/out capabilities.

7 Related Work

Prediction Serving: As from the Introduction, current ML prediction systems [9, 32, 5, 46, 17, 30, 18, 43, 59, 15] aim to minimize the cost of deployment and maximize code re-use between training and inference phases [65]. Conversely, PRETZEL casts prediction serving as a database problem and applies end-to-end and multi-query optimizations to maximize performance and resource utilization. Clipper and Rafiki deploy pipelines as Docker containers connected through RPC to a front end. Both systems apply external model-agnostic techniques to achieve better latency, throughput, and accuracy. While we employed similar techniques in the FrontEnd, in PRETZEL we have not yet explored "best effort" techniques such as ensembles, straggler mitigation, and model selection. TensorFlow Serving deploys pipelines as *Servable*s, which are units of execution scheduling and version management. One Servable is executed as a black box, although users are allowed to split model pipelines and surface them into different Servables, similarly to PRETZEL's stage-based execution. Such optimization is however not automatic. LASER [22] enables large scale

training and inference of logistic regression models, applying specific system optimizations to the problem at hand (i.e., advertising where multiple ad campaigns are run on each user) such as caching of partial results and graceful degradation of accuracy. Finally, runtimes such as Core ML [10] and Windows ML [21] provide on-device inference engines and accelerators. To our knowledge, only single operator optimizations are enforced (e.g., using target mathematical libraries or hardware), while neither end-to-end nor multi-model optimizations are used. As PRETZEL, TVM [20, 28] provides a set of logical operators and related physical implementations, backed by an optimizer based on the Halide language [49]. TVM is specialized on neural network models and does not support featurizers nor "classical" models.

Optimization of ML Pipelines: There is a recent interest in the ML community in building languages and optimizations to improve the execution of ML workloads [20, 44, 27, 3, 39]. However, most of them exclusively target Neural Networks and heterogeneous hardware. Nevertheless, we are investigating the possibility to substitute Flour with a custom extension of Tensor Comprehension [57] to express featurization pipelines. This will enable the support for Neural Network featurizers such as word embeddings, as well as code generation capabilities (for heterogeneous devices). We are confident that the set of optimizations implemented in Oven generalizes over different intermediate representations.

Uber's Michelangelo [2] has a Scala DSL that can be compiled into bytecode which is then shipped with the whole model as a zip file for prediction. Similarly, H2O [1] compiles models into Java classes for serving. This is exactly how ML.Net currently works. Conversely, similar to database query optimizers, PRETZEL rewrites model pipelines both at the logical and at the physical level. KeystoneML [55] provides a high-level API for composing pipelines of operators similarly to Flour, and also features a query optimizer similar to Oven, albeit focused on distributed training. KeystoneML's cost-based optimizer selects the best physical implementation based on runtime statistics (gathered via sampling), while no logical level optimizations is provided. Instead, PRETZEL provides end-to-end optimizations by analyzing logical plans [33, 40, 45, 26], while logical-to-physical mappings are decided based on stage parameters and statistics from training. Similarly to the SOFA optimizer [51], we annotate transformations based on logical characteristics. MauveDB [34] uses regression and interpolation models as database views and optimizes them as such. MauveDB models are tightly integrated into the database, thus only a limited class of declaratively definable models is efficiently supported. As PRETZEL, KeystoneML and MauveDB provide sub-plan materialization.

Scheduling: Both Clipper [9] and Rafiki [59] schedule inference requests based on latency targets and provide adaptive algorithms to maximize throughput and accuracy while minimizing stragglers, for which they both use ensemble models. These techniques are external and orthogonal to the ones provided in PRETZEL. To our knowledge, no model serving system explored the problem of scheduling requests while sharing resource between models, a problem that PRETZEL addresses with techniques similar to distributed scheduling in cloud computing [47, 62]. Scheduling in white box prediction serving share similarities with operators scheduling in stream processing systems [25, 56] and web services [60].

8 Conclusion

Inspired by the growth of ML applications and ML-as-a-service platforms, this paper identified how existing systems fall short in key requirements for ML prediction-serving, disregarding the optimization of model execution in favor of ease of deployment. Conversely, this work casts the problem of serving inference as a database problem where end-to-end and multi-query optimization strategies are applied to ML pipelines. To decrease latency, we have developed an optimizer and compiler framework generating efficient model plans end-to-end. To decrease memory footprint and increase resource utilization and throughput, we allow pipelines to share parameters and physical operators, and defer the problem of inference execution to a scheduler that allows running multiple predictions concurrently on shared resources.

Experiments with production-like pipelines show the validity of our approach in achieving an optimized execution: PRETZEL delivers order-of-magnitude improvements on previous approaches and over different performance metrics.

Acknowledgments

We thank our shepherd Matei Zaharia and the anonymous reviewers for their insightful comments. Yunseong Lee and Byung-Gon Chun were partly supported by the MSIT (Ministry of Science and ICT), Korea, under the SW Starlab support program (IITP-2018-R0126-18-1093) supervised by the IITP (Institute for Information & communications Technology Promotion), and by the ICT R&D program of MSIT/IITP (No.2017-0-01772, Development of QA systems for Video Story Understanding to pass the Video Turing Test).

References

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

[2] Michelangelo. https://eng.uber.com/michelangelo/.

[3] TensorFlow XLA. https://www.tensorflow.org/performance/xla/.

[4] TensorFlow. https://www.tensorflow.org, 2016.

[5] TensorFlow serving. https://www.tensorflow.org/serving, 2016.

[6] Caffe2. https://caffe2.ai, 2017.

[7] Open Neural Network Exchange (ONNX). https://onnx.ai, 2017.

[8] Batch python API in Microsoft machine learning server, 2018.

[9] Clipper. http://clipper.ai/, 2018.

[10] Core ML. https://developer.apple.com/documentation/coreml, 2018.

[11] Docker. https://www.docker.com/, 2018.

[12] Ec2 large instances and numa. https://forums.aws.amazon.com/thread.jspa?threadID=144982, 2018.

[13] Keras. https://www.tensorflow.org/api_docs/python/tf/keras, 2018.

[14] ML.Net. https://dot.net/ml, 2018.

[15] MXNet Model Server (MMS). https://github.com/awslabs/mxnet-model-server, 2018.

[16] .Net Core Ahead of Time Compilation with CrossGen. https://github.com/dotnet/coreclr/blob/master/Documentation/building/crossgen.md, 2018.

[17] PredictionIO. https://predictionio.apache.org/, 2018.

[18] Redis-ML. https://github.com/RedisLabsModules/redis-ml, 2018.

[19] Request response python API in Microsoft machine learning server. https://docs.microsoft.com/en-us/machine-learning-server/operationalize/python/how-to-consume-web-services, 2018.

[20] TVM. https://tvm.ai/, 2018.

[21] Windows ml. https://docs.microsoft.com/en-us/windows/uwp/machine-learning/overview, 2018.

[22] D. Agarwal, B. Long, J. Traupman, D. Xin, and L. Zhang. LASER: A scalable response prediction platform for online advertising. In *WSDM*, 2014.

[23] Z. Ahmed and et al. Machine learning for applications, not containers (under submission), 2018.

[24] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational data processing in spark. In *SIGMOD*, 2015.

[25] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas. Operator scheduling in data stream systems. *The VLDB Journal*, 13(4):333–353, Dec. 2004.

[26] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. pages 225–237, 2005.

[27] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, 2015.

[28] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Q. Yan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: end-to-end optimization stack for deep learning. *CoRR*, 2018.

[29] R. Chirkova and J. Yang. Materialized views. *Foundations and Trends in Databases*, 4(4):295–405, 2012.

[30] D. Crankshaw, P. Bailis, J. E. Gonzalez, H. Li, Z. Zhang, M. J. Franklin, A. Ghodsi, and M. I. Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with Velox. In *CIDR*, 2015.

[31] D. Crankshaw and J. Gonzalez. Prediction-serving systems. *Queue*, 16(1):70:83–70:97, Feb. 2018.

[32] 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.

[33] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Cetintemel, and S. Zdonik. An architecture for compiling UDF-centric workflows. *PVLDB*, 8(12):1466–1477, Aug. 2015.

[34] A. Deshpande and S. Madden. MauveDB: Supporting model-based user views in database systems. In *SIGMOD*, 2006.

[35] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *OSDI*, 1999.

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

[37] A. Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, Dec. 2001.

[38] R. He and J. McAuley. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *WWW*, 2016.

[39] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia. NoScope: Optimizing neural network queries over video at scale. *PVLDB*, 10(11):1586–1597, Aug. 2017.

[40] A. Kemper, T. Neumann, J. Finis, F. Funke, V. Leis, H. Mühe, T. Mühlbauer, and W. Rödiger. Processing in the hybrid OLTP & OLAP main-memory database system hyper. *IEEE Data Eng. Bull.*, 36(2):41–47, 2013.

[41] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE*, 2010.

[42] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling object, relations and XML in the .NET framework. In *SIGMOD*, 2006.

[43] A. N. Modi, C. Y. Koo, C. Y. Foo, C. Mewald, D. M. Baylor, E. Breck, H.-T. Cheng, J. Wilkiewicz, L. Koc, L. Lew, M. A. Zinkevich, M. Wicke, M. Ispir, N. Polyzotis, N. Fiedel, S. E. Haykal, S. Whang, S. Roy, S. Ramesh, V. Jain, X. Zhang, and Z. Haque. TFX: A TensorFlow-based production-scale machine learning platform. In *SIGKDD*, 2017.

[44] G. Neubig, C. Dyer, Y. Goldberg, A. Matthews, W. Ammar, A. Anastasopoulos, M. Ballesteros, D. Chiang, D. Clothiaux, T. Cohn, K. Duh, M. Faruqui, C. Gan, D. Garrette, Y. Ji, L. Kong, A. Kuncoro, G. Kumar, C. Malaviya, P. Michel, Y. Oda, M. Richardson, N. Saphra, S. Swayamdipta, and P. Yin. DyNet: The dynamic neural network toolkit. *ArXiv e-prints*, 2017.

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

[46] 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. In *Workshop on ML Systems at NIPS*, 2017.

[47] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *SOSP*, 2013.

[48] 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.

[49] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI*, 2013.

[50] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *NIPS*. 2011.

[51] A. Rheinländer, A. Heise, F. Hueske, U. Leser, and F. Naumann. SOFA: an extensible logical optimizer for udf-heavy data flows. *Inf. Syst.*, 52:96–125, 2015.

[52] S. Ruder. An overview of gradient descent optimization algorithms. *CoRR*, 2016.

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

[54] S. Shalev-Shwartz and T. Zhang. Stochastic dual coordinate ascent methods for regularized loss. *J. Mach. Learn. Res.*, 14(1):567–599, Feb. 2013.

[55] E. R. Sparks, S. Venkataraman, T. Kaftan, M. J. Franklin, and B. Recht. KeystoneML: Optimizing pipelines for large-scale advanced analytics. In *ICDE*, 2017.

[56] T. Um, G. Lee, S. Lee, K. Kim, and B.-G. Chun. Scaling up IoT stream processing. In *APSys*, 2017.

[57] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR*, 2018.

[58] S. Wanderman-Milne and N. Li. Runtime code generation in cloudera impala. *IEEE Data Eng. Bull.*, 37:31–37, 2014.

[59] W. Wang, S. Wang, J. Gao, M. Zhang, G. Chen, T. K. Ng, and B. C. Ooi. Rafiki: Machine Learning as an Analytics Service System. *ArXiv e-prints*, Apr. 2018.

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

[61] J.-M. Yun, Y. He, S. Elnikety, and S. Ren. Optimal aggregation policy for reducing tail latency of web search. In *SIGIR*, 2015.

[62] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, 2010.

[63] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[64] C. Zhang, A. Kumar, and C. Ré. Materialization optimizations for feature selection workloads. *ACM Trans. Database Syst.*, 41(1):2:1–2:32, Feb. 2016.

[65] M. Zinkevich. Rules of machine learning: Best practices for ML engineering. https://developers.google.com/machine-learning/rules-of-ml.

[66] M. Zukowski, P. A. Boncz, N. Nes, and S. Héman. MonetDB/X100 - a DBMS in the CPU cache. *IEEE Data Eng. Bull.*, 28(2):17–22, 2005.