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

Wenyi Wu (wenyiwu), Alexandra Spence (aspe)

Problem and Motivation

The motivation behind this paper is to build a prediction serving system that has low latency, high throughput and graceful performance degradation under heavy load.

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), 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.

Hypothesis

Existing prediction serving systems focus mainly on ease of deployment, where pipelines are considered as black boxes and deployed into containers. Black box approaches fell short on several aspects. For instance, prediction services are profitable for ML-as-aservice providers only when pipelines are accessed in batch or frequently enough, and may be not when models are accessed sporadically. Also, increasing model density in machines, thus increasing utilization, is not always possible.

In addition to that, trained pipelines often share operators and parameters (such as weights and dictionaries used within operators, and especially during featurization.

Solution Overview

The paper proposes a white box approach, PRETZEL, for model serving whereby end-to-end and multi-pipeline optimization techniques are applied to reduce resource utilization while improving performance.

PRETZEL is organized in several components. A dataflow-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.

During the offline phase, ML.Net's pre-trained 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 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.

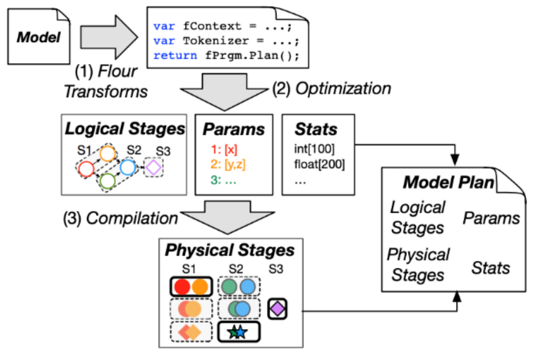


Figure 6: Model optimization and compilation in PRETZEL. In (1), a model is translated into a Flour program. (2) Oven Optimizer generates a DAG of logical stages from the program. Additionally, parameters and statistics are extracted. (3) A DAG of physical stages is generated by the Oven Compiler using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

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.

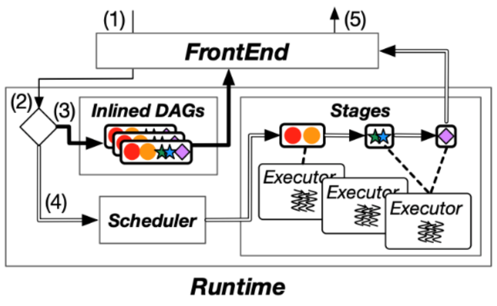


Figure 7: (1) When a prediction request is issued, (2) the Runtime determines whether to serve the prediction using (3) the request/response engine or (4) the batch engine. In the latter case, the Scheduler takes care of properly allocating stages over the Executors running concurrently on CPU cores.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

Limitations and Possible Improvements

- PRETZEL currently has several logical and physical stages classes, one per possible implementation, which make the system difficult to maintain in the long run.

- PRETZEL is currently limited to pipelines authored in ML.Net, and porting models from different frameworks to the white box approach may require non-trivial work.

Summary of Class Discussion

Q: During the offline phase, ML.Net's pre-trained pipelines are translated into Flour transformations. How many resources are needed for this process?

A: This paper only talks about simple models such as logistical regression and no neural network models are involved, so the transformation should be fairly light weight. The example in the paper also shows that the Flour program only does simple manipulation of the machine learning algorithms. Besides that, since this process is in the offline phase, this overhead should not affect the inference part at all.

Q: The paper talks about using 2 priority queues, one low priority queue for newly submitted plans, and one high priority queue for already started stages. How / when does the pipeline switch between the queues?

A: The pipeline will start with the low priority one and then switch to the high priority one. The purpose of the queues are also to keep track how late is each request in the process.

Q: The online phase still looks like a black box. The decision process of whether (3) or (4) in the online phase will be taken is not explicitly explained.

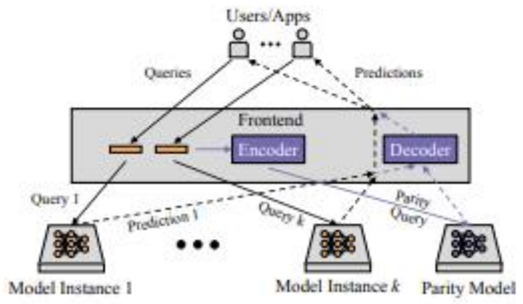
A: The offline phase analyzes the ML algorithms used and use that understanding to help build the pipeline for the online phase. Once in the online pahse, how to choose from (3) or (4) is not every well explained in the paper.

Summary of "Parity Models: Erasure-Coded Resilience for Prediction Serving Systems"

Wenyi Wu (wenyiwu), Alexandra Spence (aspe)

Problem and Motivation

With the increasing prominence of neural networks, inference has become an important task. However, prediction often suffers from tail latency inflation. The standard structure for a prediction serving system is shown in the figure below.



Prediction services have two components, the frontend and the model instance. The model instance contains the model and performs inference. There will be multiple model instances on separate servers, and the frontend distributes queries using load-balancing.

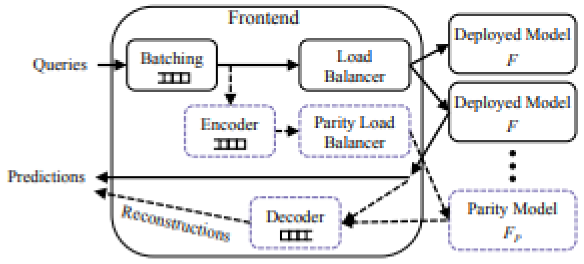
Current approaches to mitigate tail latency inflation either have high overhead due to proactively issuing redundant requests to multiple servers or high latency due to waiting until they are confident a failure has occurred. Erasure codes use less resource-overhead than replication-based techniques. However, the current approaches to coded-computation only support a few computations, so they are not good enough for prediction serving systems. While erasure codes work well for linear functions, they do not work well for non-linear functions. Because of this, current approaches to coded-computation are not adequate for prediction.

Hypothesis

Using a learning-based approach, it is possible to use coded-computation for prediction services. It is possible to learn erasure codes, so to design erasure codes all that is needed is to train an encoder and decoder model. However, using neural network encoders and decoders limits the ability to reduce tail latency, so parity models are introduced. Instead of learning encoders and decoders, using simple and fast encoders and decoders and then learning a parity model will reduce computational burden for prediction frontend and reduce latency of reconstructions, while also using the resilience of erasure-coding. Since predictions are already approximations, returning approximations of failed predictions will be good enough as a solution to slow or failed predictions.

Solution Overview

Instead of designing new encoders and decoders, they used parity models, which use simple and fast encoders and decoders. This was built as ParM. The architecture of ParM is shown in the figure below.



As before, queries are batched and dispatched to different model instances using load-balancing. However, ParM also adds an encoder and decoder to the frontend and adds m/k parity models, where m is the number of instances of the deployed model and k is the number of data units encoded by an erasure code. Query batches are encoded to create a “parity batch”, which are dispatched to parity models for inference and then returned to the frontend. The decoder is only used if a prediction batch is unavailable. When a prediction is unavailable, the output of the parity model and the (k-1) available model instances are used to reconstruct on approximation of the unavailable prediction.

For this paper, a simple addition/subtraction encoder and decoder were used, but ParM supports many different designs for encoders and decoders. It uses neural networks to learn a model to transform parities into a decodable form. To keep runtime of the parity model similar to runtime of the deployed model, the parity model uses the same number and size of layers as the deployed model. This is not a requirement in general, but was used for the paper.

To train the parity model, they use parities generated by the encoder as training data and the transformations expected by the decoder as training labels. Mean-squared-error is used as the loss function. When queries are dispatched to model instances, they are encoded to generate a parity, such as $P=(X_1+X_2+X_3)$. If the model instance for one of these is slow, the decoder will reconstruct the prediction using the parity model for the parity and each of the other two Xs.

Limitations and Possible Improvements

-The time spent training the parity model can be much longer than that to train the deployed model. This could be reduced by sampling from the deployed model’s dataset in more sophisticated ways to help reduce the number of possible combinations of k samples.

-ParM requires 1/k of all model instances for parity models, which makes the maximum possible throughput lower than other systems.

-ParM tends to have lower accuracy than systems without ParM because of its method of approximating the failed predictions.

-There is a tradeoff between accuracy and the parameter k, where larger values of k are more resource-efficient but lead to lower accuracy.

Summary of Class Discussion

Q: How should the parameters r and k should be chosen?

A: In general, r determines the number of failures that ParM can handle, but the choice of r and k are very task reliant.

Q: Do the servers need to wait for each other? What should the system do if there is a slower model?

A: Since the parity model is designed to have a similar runtime as the deployed model, the servers should generally be able to finish work at similar times. In the case of a slowdown, determined the slowdown could be treated as a failure and then recovered from the parity model.

Q: Did the paper talk about non-CNN models? How would ParM work for other types of models?

A: The current system requires numerical values, so wouldn’t work well for certain models such as RNN with text outputs. With a more sophisticated encoder and decoder, tasks such as this may be possible.