PRETZEL: Opening the Black Box of ML Prediction Serving Systems

Yunseong Lee\textsuperscript{s}, Alberto Scolari\textsuperscript{p}, Byung-Gon Chun\textsuperscript{s}, Marco Domenico Santambrogio\textsuperscript{p}, Markus Weimer\textsuperscript{m}, Matteo Interlandi\textsuperscript{m}
Machine Learning Prediction Serving

1. Models are learned from data
2. Models are deployed and served together

Performance goal:
1) Low latency
2) High throughput
3) Minimal resource usage
ML Prediction Serving Systems: State-of-the-art

- Assumption: models are black box
  - Re-use the same code in training phase
  - Encapsulate all operations into a function call (e.g., predict())
  - Apply external optimizations

剪贴板
How do Models Look inside Boxes?

Pretzel is tasty
{text}

Model

😊 VS. 😞
(positive vs. negative)

<Example: Sentiment Analysis>
How do Models Look inside Boxes?

DAG of Operators

Featurizers

Char Ngram

Word Ngram

Tokenizer

Concat

Predictor

Logistic Regression

Pretzel is tasty

<Example: Sentiment Analysis>
How do Models Look inside Boxes?

Tokenize

DAG of Operators

- Extract N-grams
- Char N-gram
- Word N-gram
- Concat
- Merge two vectors
- Logistic Regression
- Compute final score

Example: Sentiment Analysis

Pretzel is tasty

Split text into tokens

VS.

ML.NET
Many Models Have Similar Structures

• Many part of a model can be re-used in other models

• Customer personalization, Templates, Transfer Learning

• Identical set of operators with different parameters
Outline

• Prediction Serving Systems
• Limitations of Black Box Approaches
• PRETZEL: White-box Prediction Serving System
• Evaluation
• Conclusion
Limitation 1: Resource Waste

• Resources are isolated across Black boxes

1. Unable to share memory space
   ➔ Waste memory to maintain duplicate objects
      (despite similarities between models)

2. No coordination for CPU resources between boxes
   ➔ Serving many models can use too many threads
Limitation 2: Inconsideration for Ops’ Characteristics

1. Operators have different performance characteristics
   • Concat materializes a vector
   • LogReg takes only 0.3% (contrary to the training phase)

2. There can be a better plan if such characteristics are considered
   • Re-use the existing vectors
   • Apply in-place update in LogReg
Limitation 3: Lazy Initialization

- ML.Net initializes code and memory lazily (efficient in training phase)
- Run 250 Sentiment Analysis models 100 times
  ➔ **cold**: first execution / **hot**: average of the rest 99
- Long-tail latency in the **cold** case
  - Code analysis, Just-in-time (JIT) compilation, memory allocation, etc
  - Difficult to provide strong Service-Level-Agreement (SLA)

![Graph showing latency comparison between cold and hot cases]
Outline

• (Black-box) Prediction Serving Systems
• Limitations of Black Box Approaches
• PRETZEL: White-box Prediction Serving System
• Evaluation
• Conclusion
PRETZEL: White-box Prediction Serving

• We analyze models to optimize the internal execution

• We let models co-exist on the same runtime, sharing computation and memory resources

• We optimize models in two directions:
  1. End-to-end optimizations
  2. Multi-model optimizations
End-to-End Optimizations

Optimize the execution of **individual** models from start to end

1. [Ahead-of-time Compilation]
   Compile operators’ code in advance
   → No JIT overhead

2. [Vector pooling]
   Pre-allocate data structures
   → No memory allocation on the data path
Multi-model Optimizations

Share computation and memory across models

1. [Object Store]
   - Share Operators parameters/weights
     → Maintain only one copy

2. [Sub-plan Materialization]
   - Reuse intermediate results computed by other models
     → Save computation
System Components

1. Flour: Intermediate Representation

   ```javascript
   var fContext = ...;
   var Tokenizer = ...;
   return fPrgm.Plan();
   ```

2. Oven: Compiler/Optimizer

3. Runtime: Execute inference queries

4. FrontEnd: Handle user requests

Runtime

Object Store

Scheduler

...
Prediction Serving with PRETZEL

1. Offline
   • Analyze structural information of models
   • Build \textit{ModelPlan} for optimal execution
   • Register \textit{ModelPlan} to Runtime

2. Online
   • Handle prediction requests
   • Coordinate CPU & memory resources
1. Translate Model into Flour Program

```
var fContext = new FlourContext(...)  
var tTokenizer = fContext.CSV     
  .FromText(fields, fieldsType, sep)  
  .Tokenize();

var tCNgram = tTokenizer.CharNgram(numCNgrms, ...);  
var tWNgram = tTokenizer.WordNgram(numWNgrms, ...);  
var fPrgrm = tCNgram     
  .Concat(tWNgram)  
  .ClassifierBinaryLinear(cParams);

return fPrgrm.Plan();
```
System Design: Offline Phase

2. Oven optimizer/compiler build Model Plan

```javascript
var fContext = new FlourContext(...)
var tTokenizer = fContext.CSV
  .FromText(fields, fieldsType, sep)
  .Tokenize();

var tCNgram = tTokenizer.CharNgram(numCNgrms, ...);
var tWNgram = tTokenizer.WordNgram(numWNgrms, ...);
var fPrgrm = tCNgram
  .Concat(tWNgram)
  .ClassifierBinaryLinear(cParams);

return fPrgrm.Plan();
```
System Design: Offline Phase

2. Oven optimizer/compiler build Model Plan

```javascript
var fContext = new FlourContext(...)
var tTokenizer = fContext.CSV
  .FromText(fields, fieldsType, sep)
  .Tokenize();

var tCNgram = tTokenizer.CharNgram(numCNgrms, ...);
var tWNgram = tTokenizer.WordNgram(numWNgrms, ...);
var fPrgrm = tCNgram
  .Concat(tWNgram)
  .ClassifierBinaryLinear(cParams);

return fPrgrm.Plan();
```

Rule-based optimizer

Push linear predictor & Remove Concat

Group ops into stages

Stage 1

Stage 2

Parameters e.g., Dictionary, N-gram Length

Statistics e.g., dense vs. sparse, maximum vector size
System Design: Offline Phase

3. Model Plan is registered to Runtime

1. Store parameters & mapping between logical stages
2. Find the most efficient physical impl. using params & stats
System Design: Offline Phase

3. Model Plan is registered to Runtime

1. Store parameters & mapping between logical stages

2. Find the most efficient physical impl. using \texttt{params} & \texttt{stats}

3. Register selected physical stages to Catalog

- Logical Stages
  - Model1
  - S1
  - S2

- Physical Stages
  - S1
  - S2

- Catalog
  - Sparse vs. Dense

- Object Store

- N-gram length 1 vs. 3
System Design: Online Phase

1. When a prediction request arrives

2. Instantiate physical stages along with parameters

3. Execute stages using thread-pools, managed by Scheduler

4. Send result back to Client

<Model1, “Pretzel is tasty”>
Outline

• (Black-box) Prediction Serving Systems
• Limitations of Black box approaches
• PRETZEL: White-box Prediction Serving System

• Evaluation
• Conclusion
Evaluation

• Q. How PRETZEL improves performance over black-box approaches?
  • in terms of latency, memory and throughput

• 500 Models from Microsoft Machine Learning Team
  • 250 Sentiment Analysis (Memory-bound)
  • 250 Attendee Count (Compute-bound)

• System configuration
  • 16 Cores CPU, 32GB RAM
  • Windows 10, .Net core 2.0
Evaluation: Latency

- Micro-benchmark (No server-client communication)
  - Score 250 Sentiment Analysis models 100 times for each
  - Compare ML.Net vs. PRETZEL

<table>
<thead>
<tr>
<th></th>
<th>ML.Net</th>
<th>PRETZEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>P99 (hot)</td>
<td>0.6</td>
<td>0.2</td>
</tr>
<tr>
<td>P99 (cold)</td>
<td>8.1</td>
<td>0.8</td>
</tr>
<tr>
<td>Worst (cold)</td>
<td>280.2</td>
<td>6.2</td>
</tr>
</tbody>
</table>

Latency comparison:
- ML.Net: 0.2, 0.6, 280.2 ms
- PRETZEL: 0.2, 0.8, 6.2 ms

Comparison graphs show:
- 3x faster
- 10x faster
- 45x faster
- Better result

CDF (%)

Latency (ms, log-scaled)
Evaluation: Memory

- Measure Cumulative Memory Usage after loading 250 models
  - Attendee Count models (smaller size than Sentiment Analysis)
  - 4 settings for Comparison

<table>
<thead>
<tr>
<th>Settings</th>
<th>Shared Objects</th>
<th>Shared Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>ML.Net + Clipper</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML.Net</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PRETZEL without ObjectStore</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>PRETZEL</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

![Cumulative Memory Usage Graph](image-url)

- ML.Net + Clipper: 9.7GB
- ML.Net: 3.7GB
- PRETZEL (w.o. ObjStore): 2.9GB
- PRETZEL: 164MB

Number of pipelines: 25x 62x

Better

Cumulative Memory Usage (log-scaled) vs. Number of pipelines
Evaluation: Throughput

• Micro-benchmark
  • Score 250 Attendee Count models 1000 times for each
  • Request 1000 queries in a batch
  • Compare ML.Net vs. PRETZEL

More results in the paper!
Conclusion

• PRETZEL is the first white-box prediction serving system for ML pipelines

• By using models’ structural info, we enable two types of optimizations:
  • End-to-end optimizations generate efficient execution plans for a model
  • Multi-model optimizations let models share computation and memory resources

• Our evaluation shows that PRETZEL can improve performance compared to Black-box systems (e.g., ML.Net)
  • Decrease latency and memory footprint
  • Increase resource utilization and throughput
PRETZEL: a White-Box ML Prediction Serving System

Thank you!
Questions?