OPTIMIZING TERASCALE MACHINE LEARNING PIPELINES WITH Keystone ML

Evan R. Sparks, UC Berkeley AMPLab
with Shivaram Venkataraman, Tomer Kaftan, Michael Franklin, Benjamin Recht
WHAT'S A MACHINE LEARNING PIPELINE?
A STANDARD MACHINE LEARNING PIPELINE

Right?
A STANDARD MACHINE LEARNING PIPELINE

That’s more like it!
A REAL PIPELINE FOR IMAGE CLASSIFICATION

Inspired by Coates & Ng, 2012

Data → Image Parser → Normalizer → Convolver → Symmetric Rectifier → Zipper → Linear Solver

Feature Extractor

Pooler
- sqrt, mean
- ident, abs
- ident, mean
- Global Pooling

Label Extractor

Test Data → Feature Extractor → Linear Mapper

Model

Error Computer → Test Error

Error
Data Image Parser Normalizer Convolver Symmetric Rectifier Pooler sqrt, mean ident, abs ident, mean Global Pooling Zipper Linear Solver

Label Extractor

Test Data

Feature Extractor

Linear Mapper

Label Extractor

Error Computer

Embarrassingly Parallel
Requires Coordination
Tricky to Scale

Model

Test Error
ABOUT KEystoneML

- Software framework for building **scalable end-to-end** machine learning pipelines on **Apache Spark**.
- Helps us understand what it means to build systems for **robust**, **scalable**, end-to-end **advanced analytics** workloads and the **patterns** that emerge.
- Example pipelines that achieve **state-of-the-art** results on **large scale datasets** in computer vision, NLP, and speech - **fast**.
SIMPLE EXAMPLE:
TEXT CLASSIFICATION

Once estimated - apply these steps to your production data in an online or batch fashion.
NOT SO SIMPLE EXAMPLE: IMAGE CLASSIFICATION

5,000 examples, 40,000 features, 20 classes
Pleasantly parallel featurization and evaluation.

7 minutes on a modest cluster.
Achieves performance of Chatfield et. al., 2011
EVEN LESS SIMPLE: IMAGENET

Color
- Resize
- LCS
- PCA
- Fisher Vector
- Weighted Block Linear Solver
- Top 5 Classifier

Edges
- Grayscale
- SIFT
- PCA
- Fisher Vector

Texture
- Gabor
- Wavelets
- PCA
- Fisher Vector

<100 SLOC

1000 class classification.
1,200,000 examples
64,000 features.
90 minutes on 100 nodes.
Upgrading the solver for higher precision means changing 1 LOC.
Adding 100,000 more texture features is easy.
Optimizing KeystoneML Pipelines

High-level API enables rich space of optimizations

Automated ML operator selection.

Auto-caching for iterative workloads.
**KeystoneML Optimizer**

- Sampling-based *cost model* projects resource usage
  - CPU, Memory, Network
- Utilization tracked through pipeline.
  - Decisions made to minimize total cost of execution.
- Catalyst-based optimizer does the heavy lifting.

<table>
<thead>
<tr>
<th>Stage</th>
<th>n</th>
<th>d</th>
<th>size (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>5000</td>
<td>1m pixel JPEG</td>
<td>0.4</td>
</tr>
<tr>
<td>Resize</td>
<td>5000</td>
<td>260k pixels</td>
<td>3.6</td>
</tr>
<tr>
<td>Grayscale</td>
<td>5000</td>
<td>260k pixels</td>
<td>1.2</td>
</tr>
<tr>
<td>SIFT</td>
<td>5000</td>
<td>65000x128</td>
<td>309</td>
</tr>
<tr>
<td>PCA</td>
<td>5000</td>
<td>65000x80</td>
<td>154</td>
</tr>
<tr>
<td>FV</td>
<td>5000</td>
<td>256x64x2</td>
<td>1.2</td>
</tr>
<tr>
<td>Linear Regression</td>
<td>5000</td>
<td>20</td>
<td>0.0007</td>
</tr>
<tr>
<td>Max Classifier</td>
<td>5000</td>
<td>1</td>
<td>0.00009</td>
</tr>
</tbody>
</table>
CHOOSING A SOLVER

• Datasets have a number of interesting degrees of freedom.
  • Problem size (n, d, k)
  • Sparsity (nnz)
  • Condition number
• Platform has degrees of freedom:
  • Memory, CPU, Network, Nodes
• Solvers are predictable!

Objective:
\[
min_X |AX - B|^2_2 + \lambda |X|^2_2
\]

Where:
\[
A \in \mathbb{R}^{n \times d},
X \in \mathbb{R}^{d \times k},
B \in \mathbb{R}^{n \times k}
\]
CHOOSING A SOLVER

• Three Solvers
  • Exact, Block, LBFGS
• Two datasets
  • Amazon - >99% sparse, n=65m
  • TIMIT - dense, n=2m
• Exact solve works well for small # features.
• Use LBFGS for sparse problems.
• Block solver scales well to big dense problems.
  • Hundreds of thousands of features.
**SOLVER PERFORMANCE**

- Compared KeystoneML with:
  - VowpalWabbit - specialized system for large, sparse problems.
  - SystemML - general purpose, optimizing ML system.
- Two problems:
  - Amazon - Sparse text features.
  - Binary TIMIT - Dense phoneme data.
- High Order Bit:
  - KeystoneML *pipelines featurization* and *adapts* to workload changes.

<table>
<thead>
<tr>
<th>Features</th>
<th>Time (s)</th>
<th>System</th>
<th>KeystoneML</th>
<th>Vowpal Wabbit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2048</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4096</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8192</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16384</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Graphs comparing performance](chart.png)
DECIDING WHAT TO SAVE

• Pipelines Generate Lots of intermediate state.
  • E.g. SIFT features blow up a 0.42GB VOC dataset to 300GB.
• Iterative algorithms —> state needed many times.
• How do we determine what to save for later and what to reuse, given fixed resource budget?
• Can we adapt to workload changes?
CACHING PROBLEM

- Output is computed via depth-first execution of DAG.
  - Caching “truncates” a path after first visit.
- Want to minimize execution time.
  - Subject to memory constraints.
- Picking optimal set is hard!

<table>
<thead>
<tr>
<th>Cache set</th>
<th>Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABCDE</td>
<td>140s</td>
<td>340g</td>
</tr>
<tr>
<td>B</td>
<td>140s</td>
<td>200g</td>
</tr>
<tr>
<td>A</td>
<td>180s</td>
<td>50g</td>
</tr>
<tr>
<td>{}</td>
<td>240s</td>
<td>0g</td>
</tr>
</tbody>
</table>
# END-TO-END PERFORMANCE

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Training Examples</th>
<th>Features</th>
<th>Raw Size (GB)</th>
<th>Feature Size (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon</td>
<td>65 million</td>
<td>100k (sparse)</td>
<td>14</td>
<td>89</td>
</tr>
<tr>
<td>TIMIT</td>
<td>2.25 million</td>
<td>528k</td>
<td>7.5</td>
<td>8800</td>
</tr>
<tr>
<td>ImageNet</td>
<td>1.28 million</td>
<td>262k</td>
<td>74</td>
<td>2500</td>
</tr>
<tr>
<td>VOC</td>
<td>5000</td>
<td>40k</td>
<td>0.43</td>
<td>1.5</td>
</tr>
</tbody>
</table>
# End-to-End Performance

<table>
<thead>
<tr>
<th>Dataset</th>
<th>KeystoneML Accuracy</th>
<th>Reported Accuracy</th>
<th>KeystoneML Time (m)</th>
<th>Reported Time (m)</th>
<th>Speedup over Reported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon</td>
<td>91.6%</td>
<td>N/A</td>
<td>3.3</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>TIMIT</td>
<td>66.1%</td>
<td>66.3%</td>
<td>138</td>
<td>120</td>
<td>0.87x</td>
</tr>
<tr>
<td>ImageNet</td>
<td>67.4%</td>
<td>66.6%</td>
<td>270</td>
<td>5760</td>
<td>21x</td>
</tr>
<tr>
<td>VOC</td>
<td>57.2%</td>
<td>59.2%</td>
<td>7</td>
<td>87</td>
<td>12x</td>
</tr>
</tbody>
</table>
END-TO-END PERFORMANCE
END-TO-END PERFORMANCE

• Tested three levels of optimization
  • None
  • Auto-caching only
  • Auto-caching and operator-selection.
• 7x to 15x speedup
QUESTIONS?

Project Page
http://keystone-ml.org/

Code
http://github.com/amplab/keystone

Training
http://goo.gl/axbkkc
BACKUP SLIDES
SOFTWARE FEATURES

• Data Loaders
  • CSV, CIFAR, ImageNet, VOC, TIMIT, 20 Newsgroups

• Transformers
  • NLP - Tokenization, n-grams, term frequency, NER*, parsing*
  • Images - Convolution, Grayscaling, FisherVector*, Pooling, Windowing, HOG, Daisy
  • Speech - MFCCs*

• Stats - Random Features, Normalization, Scaling*, Signed Hellinger Mapping, FFT

• Utility/misc - Caching, Top-K classifier, indicator label mapping, sparse/dense encoding transformers.

• Estimators
  • Learning - Block linear models, Linear Discriminant Analysis, PCA, ZCA Whitening, Naive Bayes*, GMM*

• Example Pipelines
  • NLP - Amazon Product Review Classification, 20 Newsgroups, Wikipedia Language model
  • Images - MNIST, CIFAR, VOC, ImageNet
  • Speech - TIMIT

• Evaluation Metrics
  • Binary Classification
  • Multiclass Classification
  • Multilabel Classification

* - Links to external library

Just 11k Lines of Code, 5k of which are Tests or JavaDoc.
KEY API CONCEPTS
abstract class Transformer[In, Out] {
  def apply(in: In): Out
  def apply(in: RDD[In]): RDD[Out] = in.map(apply)
  ...
}

TYPE SAFETY HELPS ENSURE ROBUSTNESS
abstract class Estimator[In, Out] {
  def fit(in: RDD[In]): Transformer[In, Out]
  ...
}

RDD[Input] ← Estimator .fit() → Transformer
val featurizer: Transformer[String, Vector] = NGrams(2) \textit{then} Vectorizer
val pipeline = (featurizer thenLabelEstimator LinearModel).fit(data, labels)