CSE 232A
Database System Implementation

Arun Kumar

Topic 8: Data Systems for ML Workloads

Book: “Data Management in ML Systems”
by Morgan & Claypool Publishing
“Big Data” Systems

- Parallel RDBMSs and Cloud-Native RDBMSs
- Beyond RDBMSs: A Brief History
- “Big Data” Systems
  - The MapReduce/Hadoop Craze
  - Spark and Other Dataflow Systems
  - Key-Value NoSQL Systems
  - Graph Processing Systems
  - Advanced Analytics/ML Systems
Lifecycle/Tasks of ML-based Analytics

Data acquisition
Data preparation
Feature Engineering
Training
Model Selection
Inference
Monitoring

ML/Al + Data Systems Implementations

Source → Build → Deploy

Data Scientist/ML Engineer
ML 101: Popular Forms of ML

Generalized Linear Models (GLMs); from statistics

Bayesian Networks; inspired by causal reasoning

Decision Tree-based: CART, Random Forest, Gradient-Boosted Trees (GBT), etc.; inspired by symbolic logic

Support Vector Machines (SVMs); inspired by psychology

Artificial Neural Networks (ANNs): Multi-Layer Perceptrons (MLPs), Convolutional NNs (CNNs), Recurrent NNs (RNNs), Transformers, etc.; inspired by brain neuroscience
Q: What is a Machine Learning (ML) System?

❖ A data processing system (aka data system) for mathematically advanced data analysis ops (inferential or predictive), i.e., beyond just SQL aggregates

❖ Statistical analysis; ML, deep learning (DL); data mining (domain-specific applied ML + feature eng.)

❖ High-level APIs for expressing statistical/ML/DL computations over large datasets
Key concerns in ML:
- Accuracy
- Runtime efficiency (sometimes)

Additional key *practical* concerns in ML Systems:
- Scalability (and efficiency at scale)
- Usability
- Manageability
- Developability

Q: How do “ML Systems” relate to ML?

Q: What if the dataset is larger than single-node RAM?
Q: How are the features and models configured?
Q: How does it fit within production systems and workflows?
Q: How to simplify the implementation of such systems?

Long-standing concerns in the DB systems world!

Can often trade off accuracy a bit to gain on the rest!
## Conceptual System Stack Analogy

<table>
<thead>
<tr>
<th>Relational DB Systems</th>
<th>ML Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Theory</strong></td>
<td></td>
</tr>
<tr>
<td>First-Order Logic</td>
<td>Learning Theory</td>
</tr>
<tr>
<td>Complexity Theory</td>
<td>Optimization Theory</td>
</tr>
<tr>
<td><strong>Program Formalism</strong></td>
<td></td>
</tr>
<tr>
<td>Relational Algebra</td>
<td>Matrix Algebra</td>
</tr>
<tr>
<td><strong>Program Specification</strong></td>
<td></td>
</tr>
<tr>
<td>Declarative Query Language</td>
<td>Gradient Descent</td>
</tr>
<tr>
<td><strong>Program Modification</strong></td>
<td></td>
</tr>
<tr>
<td>Query Optimization</td>
<td>TensorFlow?</td>
</tr>
<tr>
<td><strong>Execution Primitives</strong></td>
<td></td>
</tr>
<tr>
<td>Parallel Relational Operator Dataflows</td>
<td>R? Scikit-learn?</td>
</tr>
<tr>
<td><strong>Hardware</strong></td>
<td>Depends on ML Algorithm</td>
</tr>
<tr>
<td>CPU, GPU, FPGA, NVM, RDMA, etc.</td>
<td></td>
</tr>
</tbody>
</table>
Categorizing ML Systems

❖ Orthogonal Dimensions of Categorization:

1. **Scalability**: In-memory libraries vs Scalable ML system (works on larger-than-memory datasets)

2. **Target Workloads**: General ML library vs Decision tree-oriented vs Deep learning, etc.

3. **Implementation Reuse**: Layered on top of scalable data system vs Custom from-scratch framework
Major Existing ML Systems

General ML libraries:

- In-memory:
  - mllearn
  - R
  - WEKA
  - SAS
  - DASK

- Disk-based files:
  - MADlib
  - Spark
  - MLlib

- Layered on RDBMS/Spark:
  - Azure Machine Learning
  - Amazon SageMaker

Cloud-native:

- "AutoML" platforms:
  - DataRobot
  - H2O.ai

Decision tree-oriented:

- XGBoost
- LightGBM

Deep learning-oriented:

- TensorFlow
- PyTorch
Recall that an ML model is a parametric function:

\[ f : \mathcal{D}_W \times \mathcal{D}_X \rightarrow \mathcal{D}_Y \]

Training: Process of fitting model parameters from data

Training can be expressed in this form for many ML models; aka “empirical risk minimization” (ERM) aka “loss” function:

\[ L(W) = \sum_{i=1}^{n} l(y_i, f(W, x_i)) \]

\( l() \) is a differentiable function; can be compositions

GLMs, linear SVMs, and ANNs fit the above template
Key Algo. For ML: Gradient Descent

❖ Goal of training is to find minimizer: \( W^* = \text{argmin} \ L(W) \)
❖ Not possible to solve for optimal in closed form usually
❖ Gradient Descent (GD) is an iterative procedure to get to an optimal solution:

Gradient \( \nabla L(W) = \sum_{i=1}^{n} \nabla l(y_i, f(W, x_i)) \)

Update rule at iteration t:
\[
W^{(t+1)} \leftarrow W^{(t)} - \eta \nabla L(W^{(t)})
\]
Learning rate hyper-parameter

❖ Each iteration/pass aka epoch akin to a SQL aggregate!
❖ Typically, multiple of epochs needed for convergence
GD vs Stochastic GD (SGD)

❖ Disadvantages of GD:
❖ An update needs pass over whole dataset; inefficient for large datasets (> millions of examples)
❖ Gets us only to a local optimal; ANNs are “non-convex”

❖ Stochastic GD (SGD) resolves above issues:
❖ Popular for large-scale ML, especially ANNs/deep learning
❖ Updates based on samples aka mini-batches; batch sizes: 10s to 1000s; OOM more updates per epoch than GD!

\[
W^{(t+1)} \leftarrow W^{(t)} - \eta \nabla \tilde{L}(W^{(t)}) \quad \nabla \tilde{L}(W) = \sum_{i \in B} \nabla l(y_i, f(W, x_i))
\]

Sample mini-batch from dataset without replacement
Data Access Pattern of SGD

\[
W^{(t+1)} \leftarrow W^{(t)} - \eta \nabla \tilde{L}(W^{(t)}) \quad \nabla \tilde{L}(W) = \sum_{i \in B} \nabla l(y_i, f(W, x_i))
\]

Sample mini-batch from dataset without replacement

Original dataset

Random “shuffle”

Randomized dataset

Epoch 1

\[W^{(0)}\]

Seq. scan

\[W^{(1)}\]

(Optional)

New Random Shuffle

\[W^{(2)}\]

\[W^{(3)}\]

Epoch 2 ...

\[W^{(3)}\]

\[W^{(4)}\]

…

ORDER BY RAND()
An SGD epoch is similar to SQL aggs but also different:

- More complex agg. state (running info): model param. $W^{(t)}$
- Multiple mini-batch updates to model param. within a pass
- Sequential dependency across mini-batches in a pass!
- Need to keep track of model param. across epochs
- (Optional) New random shuffling before each epoch
- Not an algebraic aggregate; hard to parallelize!
- Not even commutative: different random shuffle orders give different results (very unlike relational ops)!

Q: How to implement **scalable** SGD in an ML system?
Bismarck: Single-Node Scalable SGD

- An SGD epoch runs in RDBMS process space using its User-Defined Aggregate (UDA) abstraction with 4 functions:
  - **Initialize**: Run once; set up info; alloc. memory for $W(t)$
  - **Transition**: Run per-tuple; compute gradient as running info; track mini-batches and update model param.
  - **Merge**: Run per-worker at end; “combine” model param.
  - **Finalize**: Run once at end; return final model param.
- Commands for shuffling, running multiple epochs, checking convergence, and validation/test error measurements issued from an external controller written in Python

https://adalabucsd.github.io/papers/2012_Bismarck_SIGMOD.pdf
Combining SGD Model Parameters

- RDBMS takes care of scaling UDA code to larger-than-RAM data and distributed execution across workers
- Tricky part: How to “combine” model params. from workers, given that an SGD epoch is not an algebraic agg.?
  - Master typically performs “Model Averaging”

\[ \mathbf{W}(t) = \frac{1}{n} \sum_{i=1}^{n} \mathbf{W}_i(t) \]

- A bizarre heuristic!
- Works OK for GLMs
- Terrible for ANNs!
Disadvantages of Model Averaging for distributed SGD:
- Poor convergence for non-convex/ANN models; leads to too many epochs and typically poor ML accuracy
- UDA merge step is choke point at scale (n in 100s); model param. size can be huge (even GBs); wastes resources

ParameterServer is a more flexible from-scratch design of an ML system specifically for distributed SGD:
- Break the synchronization barrier for merging: allow asynchronous updates from workers to master
- Flexible communication frequency: at mini-batch level too

ParameterServer for Distributed SGD

Multi-sever “master”; each server manages a part of $W^{(t)}$

<table>
<thead>
<tr>
<th>PS 1</th>
<th>PS 2</th>
<th>...</th>
<th>PS 2</th>
</tr>
</thead>
</table>

No sync. for workers or servers

Push / Pull when ready/needed

Workers send gradients to master for updates at each mini-batch (or lower frequency)

$\nabla \tilde{L}(W_1^{(t)})$  $\nabla \tilde{L}(W_2^{(t-1)})$  $\nabla \tilde{L}(W_n^{(t+1)})$

❖ Model params may get out-of-sync or stale; but SGD turns out to be remarkably robust—multiple updates/epoch really helps
❖ Communication cost per epoch is higher (per mini-batch)
Deep Learning Systems

❖ Offer 3 crucial new capabilities for DL users:
  ❖ High-level APIs to easily construct complex neural $L(\cdot)$ architectures; aka differentiable programming
  ❖ Automatic differentiation (“autodiff”) to compute $\nabla \tilde{L}(\cdot)$
  ❖ Automatic backpropagation: chain rule to propagate gradients and update model params across ANN layers
  ❖ “Neural computational graphs” compiled down to low-level hardware-optimized physical matrix ops (CPU, GPU, etc.)
  ❖ In-built support for many variants of GD and SGD
  ❖ Wider tooling infra. for saving models, plotting loss, etc.
Takeaway: Scalable, efficient, and usable systems for ML have become critical for unlocking the value of “Big Data”

If you are interested in learning more about this topic, read my recent book “Data management ML Systems”:

PDF: https://www.morganclaypool.com/doi/10.2200/S00895ED1V01Y201901DTM057


Advertisement: I will offer CSE 291/234 titled “Data Systems for Machine Learning” in Fall 2020
Switch to Krypton Slides