Classical Machine Learning At Scale

Thomas Parnell
Research Staff Member, Data and AI Systems
IBM Research - Zurich
Motivation

1. Why do classical machine learning models dominate in many applications?

2. Which classical machine learning workloads might benefit from being deployed in HPC-like environments.

Source: Kaggle Data Science Survey, November 2019
Why is classical ML still popular?

- Deep neural networks dominate machine learning research, and have achieved state-of-the-art accuracy on a number of different tasks.
  - Image Classification
  - Natural language processing
  - Speech recognition

- However, in many industries such as finance and retail, classical machine learning techniques are still widely used in production. Why?

- The reason is primarily the data itself.

- Rather than images, natural language or speech, real world data often looks like…. 
Datasets have a **tabular structure** and contain a lot of categorical variables.

DNNs require feature engineering / embeddings.

Whereas, a number of classical ML models can deal with them “out of the box”.

---

**Source:** [https://towardsdatascience.com/encoding-categorical-features-21a2651a065c](https://towardsdatascience.com/encoding-categorical-features-21a2651a065c)
Classical ML Models
GLMs, Trees, Forests and Boosting Machines
Generalized Linear Models

Pros:
- Simple and fast.
- Scale well to huge datasets.
- Easy to interpret.
- Very few hyper-parameters.

Cons:
- Cannot learn non-linear relationships between features.
- Require extensive feature engineering.
Decision Trees

Pros:
✓ Simple and fast.
✓ Easy to interpret.
✓ Capture non-linear relationships between features.
✓ Native support for categorical variables.

Cons:
✗ Greedy training algorithm.
✗ Can easily overfit the training data.
Random Forests

**Pros:**

- Inherits most benefits of decision trees.
- Improve generalization via bootstrap sampling + averaging.
- Embarrassingly parallel.

**Cons:**

- Somewhat heuristic.
- Computationally intense.
- Harder to interpret.

Source: https://medium.com/@williamkoehrsen/random-forest-simple-explanation-377895a60d2d
Gradient Boosting Machines

Pros:

✓ Inherits most benefits of decision trees.
✓ State-of-the-art generalization.
✓ Theoretically elegant training algorithm.

Cons:

x Computationally intense.
x Inherently sequential.
x Harder to interpret.
x A lot of hyper-parameters.

Distributed Training
Data Parallel vs. Model Parallel
Why scale-out?

1. Very huge data (e.g. 1+ TBs)
   - Dataset does not fit inside the memory of a single machine.
   - The dataset may be stored in a distributed filesystem.
   - Data-parallel training algorithms are a necessity, even for relatively simple linear models.

2. Training acceleration
   - Dataset may fit inside the memory of a single node.
   - However, model may be very complex (e.g. RF with 10k trees)
   - We choose to scale-out using model-parallel algorithms to accelerate training.

We will now consider two examples of the above scenarios.
Training GLMs on Big Data

- Training GLMs involves solving an optimization of the following form:

\[ \min_{\alpha} f(A\alpha) + \sum_{i} g_i(\alpha_i) \]

- Where \( \alpha \) denotes the model we would like to learn, \( A \) denotes the data matrix and \( f \) and \( g_i \) denote convex functions specifying the loss and regularization, respectively.

- We assume that data matrix \( A \) is partitioned across a set of worker machines.

- One way to solve the above is to use the standard mini-batch stochastic gradient descent (SGD) widely used in the deep learning field.

- However, since the cost of computing gradients for linear models is typically relatively cheap relative to the cost of communication over the network, mini-batch SGD often performs poorly.
CoCoA Framework

- Let us assume that the data matrix $A$ is partitioned across workers by column (feature).
- The CoCoA framework (Smith et al. 2018) define a data-local subproblem:

$$\min_{\alpha[k]} F_k(A[k], \alpha[k], v)$$

- Each worker solves its local subproblem with respect to its local model coordinates $\alpha[k]$.
- This subproblem depends only on the local data $A[k]$ as well as some shared state $v$.
- An arbitrary algorithm can be used to solve the sub-problem in an approximate way.
- The shared state is then updated across all workers, and the process repeats.
- This method is theoretically guaranteed to converge to the optimal solution and allows one to trade-off the ratio of computation vs. communication much more effectively.
Distributed Training using CoCoA

Worker 0
Data Partition 0
Local Solver
\( \alpha_{[0]} \) \( \nu^{(0)} \) AllReduce \( \nu \)
Local Solver

Worker 1
Data Partition 1
Local Solver
\( \alpha_{[1]} \) \( \nu^{(1)} \) AllReduce \( \nu \)
Local Solver

Worker 2
Data Partition 2
Local Solver
\( \alpha_{[2]} \) \( \nu^{(2)} \) AllReduce \( \nu \)
Local Solver

Worker 3
Data Partition 3
Local Solver
\( \alpha_{[3]} \) \( \nu^{(3)} \) AllReduce \( \nu \)
Local Solver
Many GLMs admit two equivalent representations: primal and dual.

CoCoA can be applied to either.

**Primal** case:
- Partition the data by column (feature)
- $\alpha$ has dimension $m$
- $\nu$ has dimension $n$
- **Minimal communication when $m >> n$**

**Dual** case:
- Partition the data by row (example)
- $\alpha$ has dimension $n$
- $\nu$ has dimension $m$
- **Minimal communication when $n >> m$**
Real Example

**Dataset:** Criteo TB Click Logs (4 billion examples)

**Model:** Logistic Regression

Snap ML (Dünner et al. 2018) uses a variant of CoCoA + new algorithms for effectively utilizing GPUs + efficient MPI implementation.
Model-parallel Random Forests

- Scenario: the dataset fits in memory of a single node.
- We wish to build a very large forest of trees (e.g. 4000).
- Replicate the training dataset across the cluster.
- Each worker builds a partition of the trees, in parallel.
- Embarrassingly parallel, expect linear speed-up for large enough models.

<table>
<thead>
<tr>
<th>Worker 0</th>
<th>Worker 1</th>
<th>Worker 2</th>
<th>Worker 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataset</td>
<td>Dataset</td>
<td>Dataset</td>
<td>Dataset</td>
</tr>
<tr>
<td>Trees 0-999</td>
<td>Trees 1000-1999</td>
<td>Trees 2000-2999</td>
<td>Trees 3000-3999</td>
</tr>
</tbody>
</table>
Scaling Example
Distributed Tree Building

- What if dataset is too large to fit in memory of a single node?
- Partition dataset across workers in the cluster.
- Build each tree in the forest in a distributed way.
- Tree building requires a lot of communication, scales badly.
- Can we do something truly data parallel?
Data-parallel + model-parallel Random Forest

- In a random forest, each tree is trained on a bootstrap sample of the training data.
- What if we relax this constraint? Instead, we could train each tree on a random partition.
- We can thus randomly partition the data across the workers in the cluster.
- And then train a partition of the trees independently on each worker on a partition of the data.
- This approach can achieve **super-linear scaling**, possibly at the expense of accuracy.
Accuracy Trade-Off

**Dataset:** Rossmann Store Sales (800k examples, 20 features)

**Model:** Random Forest, 100 trees, depth 8, 10 repetitions

- Accuracy degrades quickly as we approach ~100 partitions
- Accuracy degrades fairly slowly up to ~10 partitions
Hyper-parameter tuning
Random Search, Successive Halving and Hyperband
Hyper-parameter Tuning

- GBM-like models have a large number of hyper-parameters:
  - Number of boosting rounds.
  - Learning rate.
  - Subsampling (example and feature) rates.
  - Maximum tree depth.
  - Regularization penalties.

- Standard approach is to split training set into an effective training set and a validation set.
- The validation set is used to evaluate the accuracy for different choices of hyper-parameters.
- Many different algorithms exist for hyper-parameter tuning (HPT).
- However, all involve evaluating a large number (e.g. 1000s) of configurations.
  - HPT can lead to HPC-scale workloads even for relatively small datasets.
- We will now introduce 3 HPT methods that are well-suited for HPC environments.
Random Search

- Random search is perhaps the simplest HPT method.
- It works as follows:
  - Draw $N$ hyper-parameter configurations at random.
  - Train each one on the training set.
  - Evaluate (or score) each one on the validation set (e.g. measure loss or ROC AUC).
  - Select the configuration that provides the lowest score.
- Clearly, random search is *embarrassingly parallel* and can be very effectively parallelized across a large cluster of machines.
- An optimal implementation requires a little thought since different configurations may take much longer to train than others (tip: avoid batch synchronization).
- Despite its simplicity, when used as a baseline, *random search is often very hard to beat*, especially for high-dimensional problems (Li and Talwalkar 2019).
Successive Halving

- Random search may waste a lot of time training and evaluating bad configurations.
- Is there a way to discard bad configurations more quickly?

Successive Halving (Jamieson and Nowak 2014) introduces the notion of a resource:
  - Number of gradient descent steps.
  - Number of boosting rounds.
  - Size of random subsample of training dataset.

Main idea:
  - Evaluate a large number of configurations, quickly, with a small resource.
  - Carry forward only the best ones for evaluation with a larger resource.
Worked Example

Draw \( n_0 \) configurations at random.

Evaluate (in parallel) with resource \( r_0 \).

Take best \( n_1 = \left\lfloor \frac{n_0}{\eta} \right\rfloor \) configurations.

Round 0

Evaluate (in parallel) with resource \( r_1 = r_0 \eta \).

Take best \( n_2 = \left\lfloor \frac{n_1}{\eta} \right\rfloor \) configurations.

Round 1

Evaluate (in parallel) with resource \( r_s = R \).

Take best \( n_s = \left\lfloor \frac{n_{s-1}}{\eta} \right\rfloor \) configurations.

Round \( s \)
When can SH go wrong?

(a) Ideal behaviour: the best configuration in the final round can be identified from the beginning and is not eliminated as the algorithm progresses.

(b) Problematic behaviour: promising configurations in the initial rounds do not correspond to the best configurations in the final round.

Source: Sommer et al. “Learning to tune XGBoost with XGBoost”, MetaLearn 2019
Hyperband (Li et al. 2018)

- **Main idea:** create multiple “brackets” of successive halving, each one getting progressively more exploitative rather than explorative.

<table>
<thead>
<tr>
<th>Bracket 0</th>
<th>Bracket 1</th>
<th>Bracket 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n_i$</td>
<td>$r_i$</td>
<td>$n_i$</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>100</td>
</tr>
</tbody>
</table>

Example with $\eta = 10$ and $R = 100$

- Algorithm is very simple: **run all brackets in parallel** and output best configuration found.
Hyperband applied to HPT task for kernel-based classifier on CIFAR-10

- Hyperband significantly outperforms random search.
- It also out-performs Bayesian optimization techniques such as SMAC and TPE (which are much harder to parallelize)

Source: Li et al., Hyperband: A Novel Bandit-Based Approach for Hyperparameter Optimization", JMLR 2018
Conclusions

- Classical machine learning methods still reign supreme in domains where tabular data is abundant.

- Training may be distributed across a large cluster when either:
  - Dataset is extremely large, but model is simple
  - Dataset is relatively small, but model is complex
  - …or both!

- For models like GBMs, with a large number of hyper-parameters, the HPT task can become extremely computationally intensive, even for small datasets.

- Recent advances in highly-parallel HPT algorithms show a lot of promise and may benefit from being deployed on HPC-scale clusters.