Distributed Learning: Spark for Logistic Regression, Spark MLLib, Parameter Server

Michelle Lim
Yuki de Pourbaix
Wei Tai Ting
March 26, 2019
Review Spark Key Ideas
Review Spark Key Ideas

- Improve MapReduce efficiency: in-memory, multi-step, coarse-grained computation graphs using high-level operators
  - Claims 100x faster

- Improves MapReduce: usability: richer APIs in Scala, Java, Python
  - Claims 2.5 times less code
An RDD is a collection of Java or Python objects partitioned across a cluster.

```scala
val links = spark.textFile(...).map(_.).persist()
var ranks = // RDD of (URL, rank) pairs
for (i <- 1 to ITERATIONS) {
  // Build an RDD of (targetURL, float) pairs
  // with the contributions sent by each page
  val contribs = links.join(ranks).flatMap {
    (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }
  // Sum contributions by URL and get new ranks
  ranks = contribs.reduceByKey((x, y) => x+y)
    .mapValues(sum => a/N + (1-a)*sum)
}
```
Review Spark: Execution Key Ideas

- Transformations are lazy
  - nothing happens when only transformations are called; executions start only when action is called (why?)

- Maximizing pipelining of transformations
  - Q: which type of transformations is easier to pipeline (narrow, or wide)?

- Consider Spark lineage graph as a DAG, what is constraint on execution order?

- Assigning tasks to nodes to optimize locality (how?)
Review Spark: Spark Execution Scheduling

Stage 1
- A: 
- B: groupBy

Stage 2
- C: 
- D: map 
- E: union 

Stage 3
- F: join 
- G: 

Diagram showing the execution and scheduling of Spark tasks.
Discuss Spark for Distributed Learning: Logistic Regression
Why logistic regression?

- To demonstrate its superiority over Hadoop: Logistic regression requires us to reuse variables at every iteration. (Hadoop will be way too slow since it’d save the variables to disk every iteration)
- To show why Spark is great for machine learning: The algorithm used here is used for most of machine learning.
Example of how logistic regression is used: predicting clicks
Notation

\[ \sigma(z) = \frac{1}{1 + e^{-z}} \]

Sigmoid function

\[ \theta^T x = \sum_{i=1}^{n} \theta_i x_i \]

Weighted sum (aka dot product)

\[ = \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n \]

\[ \sigma(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}} \]

Sigmoid function of weighted sum
Spark Exercise for Distributed Learning: Logistic Regression

Goal: learn parameter theta (weights) from samples \((x, y)\),
- where \(x\) is the feature vector
- \(y\) is the outcome (e.g., ad click features: YES or NO)

1. Make logistic regression assumption

\[
P(Y = 1|X = x) = \sigma(\theta^T x) \\
P(Y = 0|X = x) = 1 - \sigma(\theta^T x)
\]
Spark Exercise for Distributed Learning: Logistic Regression

2. Calculate the log probability for all data

\[ LL(\theta) = \sum_{i=0}^{n} y^{(i)} \log \sigma(\theta^T x^{(i)}) + (1 - y^{(i)}) \log[1 - \sigma(\theta^T x^{(i)})] \]

- If we find the theta that maximizes this, we will find the theta that fits the data best!
- How to maximize this? Gradient descent!
Offline: Derivation for step 2

INTUITION BEHIND LIKELIHOOD FUNCTION: $y(i)$ is 1, and we predicted that, we multiply. $y(i)$ is 0, and we predicted 1, we reduce it to 1. Converse is true for $y(i)$ is 0, $y(i)$ is 0, then I'll

For IID data

$$L(\theta) = \prod_{i=1}^{n} P(Y = y(i) | X = x^{(i)})$$

$$= \prod_{i=1}^{n} \sigma(\theta^T x^{(i)})^{y(i)} \cdot [1 - \sigma(\theta^T x^{(i)})]^{(1-y^{(i)})}$$

Take the log

$$LL(\theta) = \sum_{i=0}^{n} y^{(i)} \log \sigma(\theta^T x^{(i)}) + (1 - y^{(i)}) \log [1 - \sigma(\theta^T x^{(i)})]$$
Spark Exercise for Distributed Learning: Logistic Regression

The Log Probability

Gradient/Derivative

$$w \leftarrow w - \alpha \cdot \sum_{i=1}^{n} g(w; x_i, y_i)$$

3 Get derivative of log probability with respect to thetas

$$\frac{\partial LL(\theta)}{\partial \theta_j} = \sum_{i=0}^{n} \left[ y^{(i)} - \sigma(\theta^T x^{(i)}) \right] x_j^{(i)}$$
Offline: Derivation for step 3

\[ \frac{\partial}{\partial \theta_j} \sigma(\theta^T x) = \sigma(\theta^T x) \left[ 1 - \sigma(\theta^T x) \right] \]

True fact about sigmoid functions

\[ \frac{\partial}{\partial \theta_j} \sigma(\theta^T x) = \frac{\partial}{\partial z} \sigma(z) \cdot \frac{\partial z}{\partial \theta_j} \]

Chain rule!

\[ \frac{\partial}{\partial \theta_i} \sigma(\theta^T x) = \sigma(\theta^T x) \left[ 1 - \sigma(\theta^T x) \right] x_j \]

Plug and chug
Spark: Logistic Regression: Putting it all together

This is the variable we need to store between iterations!

\[ w \leftarrow w - \alpha \cdot \sum_{i=1}^{n} g(w; x_i, y_i) \]

```scala
val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.zeros(d)
for (i <- 1 to numIterations) {
  val gradient = points.map { p =>
    (1 / (1 + exp(-p.y * w.dot(p.x)) - 1)) * p.y * p.x
  }.reduce(_ + _)
  w -= alpha * gradient
}```
Logistic Regression Results

Spark is much faster:
Recall: Logistic regression requires us to reuse variables at every iteration. (Hadoop will be way too slow since it’d save the variables to disk every iteration)

100 GB of data on 50 m1.xlarge EC2 machines
the more RAM Spark can use, the faster is the compute. But with lesser and lesser RAM, you become just like Hadoop (since you have to go to disk more).
Distributing computation on Spark

What’s the best way to split up many matrix computations across many nodes?
Distributing Matrix Computation

- How to distribute a matrix across machines?
  - By Entries (CoordinateMatrix)
  - By Rows (RowMatrix)
  - By Blocks (BlockMatrix)

- All of Linear Algebra to be rebuilt using these partitioning schemes

- This adds complexity to operations. E.g. if you multiply two matrices together, you’ll need many matrix multiples (involves a lot of communication).
Single Value Decomposition

Purpose of SVD: reduce a dataset containing a large number of values to a dataset containing significantly fewer values, but which still contains a large fraction of the variability present in the original data.

Possible cases: $A$ is $m \times n$ matrix that represents $m$ rows of data with $n$ dimensions

1. Tall and skinny: $\# \text{ e.g.s} > \# \text{ features}$
2. Short and fat (not really): $\# \text{ features} > \# \text{ e.g.s}$
3. Roughly square

Tall and skinny most likely case.
SVD

Key insight: For tall and skinny, i.e. $m \gg n$, $(A^T) \cdot A$ is $n \times n$, considerably smaller than $A$.

Rank-$r$ SVD is $O(mnr)$
vs. $(A^T) \cdot A$ SVD is $O((n^2) \cdot r)$

$(A^T A)_{jk} = \sum_{i=1}^{m} a_{ij} a_{ik}$

How to design mapper and reducer? (naive)
SVD: Naive mapper-reducer

Mapper-Reducer design (naive)

Input: The \(i\)th row of a matrix, denoted by \(r_i\)

for all (non-zero) pairs \(a_{ij}, a_{ik}\) in \(r_i\) do

\[
\text{Emit} \; 
\langle (j, k) \rightarrow a_{ij} a_{ik} \rangle \\
// \; (j,k) \text{ is the key, the product is the value}
\]

end

Algorithm 2: \(A^TA\) mapper

Input: A coordinate pair as key and a listing of products of scalars: \((\langle j, k \rangle, (v_1, \ldots, v_m))\)

\[
(j, k) \rightarrow \sum_{i=1}^{m} v_i
\]

Algorithm 3: \(A^TA\) reducer

Suppose each row has at most $L$ non-zero entries.

Problems with naive mapper-reducer:

- Shuffle-size: $O(mL^2)$ emissions [too much communication between machines]
- Max reduce-key complexity: $m$ [too much work for single machine]

Solution: DIMSUM Algorithm. Probabilistic sampling to reduce work. Sample columns that have many non-zeros with lower probability, few non-zeros with higher probability.
for all pairs \((a_{ij}, a_{ik})\) in \(r_i\) do
With probability \(\min\left\{1, \frac{\gamma}{\|c_j\|\|c_k\|}\right\}\), emit \(((j, k) \rightarrow a_{ij}a_{ik})\)
end

Algorithm 4: DIMSUMMapper \((r_i)\)

if \(\frac{\gamma}{\|c_i\|\|c_j\|} > 1\) then
return \(b_{ij} \rightarrow \frac{1}{\|c_j\|\|c_j\|} \sum_{i=1}^{R} v_i\)
end

return \(b_{ij} \rightarrow \frac{1}{\gamma} \sum_{i=1}^{R} v_i\)

Algorithm 5: DIMSUMReducer \(((i, j), \langle v_1 \ldots, v_R \rangle)\)

New shuffle-size complexity:
New reduce-key complexity:
Independent of amount of data: \(m\)
Spark MLLib & Discussion
Spark for Distributed Learning: MLlib

- classification: logistic regression, linear SVM, naïve Bayes, least squares, classification tree
- regression: generalized linear models (GLMs), regression tree
- collaborative filtering: alternating least squares (ALS), non-negative matrix factorization (NMF)
- clustering: k-means||
- decomposition: SVD, PCA
- optimization: stochastic gradient descent, L-BFGS
OSDI’14 parameter server design motivation

- How likely is a server crash?
  - More likely with more servers running
  - More likely with longer duration spent computing

- Fault-tolerance in Spark:
  - If a node fails, need to recompute missing or damaged partitions of RDD from scratch according to RDD lineage graph [DAG]
  - What does this mean?
Suppose part of the DAG is X->Y->Z. When a node fails, cluster manager detects failure and then assigns incomplete tasks to another node on the affected RDD partition. That node will perform the assigned series of operations (called lineage) X->Y->Z on that RDD partition.

- Recompute from scratch costly!
  - Nature of ML calculations: iterative -> epochs
- What if we remember intermediate stage results?
  - Motivation for parameter server design
Using OSDI Parameter Server to scale distributed learning

- Partition the data and model into separate nodes, run at scale
- Slides are adapted from: https://www.usenix.org/sites/default/files/conference/protected-files/osdi14_slides_li-mu.pdf
Overview of machine learning

Scale of Industry
100’s of billions examples
10’s of billions of features
1TB-PB of training data
100-1000 machines
Data and model partition

Training data
Data and model partition

Training data

Worker machines
Data and model partition

Model

Training data

Worker machines
Data and model partition

Model

Server machines

Training data

Worker machines
Data and model partition

Model

Server machines

push

Worker machines

Training data
Data and model partition

Model

Server machines

push

Worker machines

pull

Training data
Example: distributed gradient descent

Server machines

Worker machines
Example: distributed gradient descent

Workers **pull** the working set of model

Server machines

Worker machines
Example: distributed gradient descent

Workers **pull** the working set of **model**
Iterate until stop
workers compute **gradients**

Server machines

Worker machines
Example: distributed gradient descent

Workers **pull** the working set of **model**

Iterate until stop

workers **compute** **gradients**

workers **push** **gradients**

Server machines

Worker machines
Example: distributed gradient descent

Workers **pull** the working set of **model**
Iterate until stop
workers **compute** **gradients**
workers **push** **gradients**
update **model**

Server machines

Worker machines
Example: distributed gradient descent

Workers **pull** the working set of **model**

Iterate until stop

Workers **compute** **gradients**

Workers **push** **gradients**

Update **model**

Workers **pull** updated **model**

Server machines

Worker machines
Challenges for data synchronization

✨ Massive communication traffic

★ frequent access to the shared model

● Workers have to read and write to the same model.

✨ Expensive global barriers

★ between iterations

● Worker nodes have to wait for the model to be updated and pulled before they can begin.
Task

- a push / pull / user defined function (an iteration)
- “execute-after-finished” dependency

![Diagram showing synchronization]

iter 0: CPU intensive (gradient) → Network intensive (push & pull)

iter 1: gradient → push & pull
Task

- a push / pull / user defined function (an iteration)
- “execute-after-finished” dependency

```plaintext
iter 0
gradient  push & pull

iter 1
gradient  push & pull
```

- executed asynchronously
Task

- a push / pull / user defined function (an iteration)
- “execute-after-finished” dependency

- executed asynchronously

<table>
<thead>
<tr>
<th>Iteration</th>
<th>CPU intensive</th>
<th>Network intensive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iter 0</td>
<td>gradient</td>
<td>push &amp; pull</td>
</tr>
</tbody>
</table>

- Synchronous

<table>
<thead>
<tr>
<th>Iteration</th>
<th>CPU intensive</th>
<th>Network intensive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iter 1</td>
<td>gradient</td>
<td>push &amp; pull</td>
</tr>
</tbody>
</table>

- Asynchronous
Flexible consistency

- Trade-off between algorithm efficiency and system performance
Flexible consistency

- Trade-off between algorithm efficiency and system performance

Sequential

1 <-> 2 <-> 3 <-> 4
Flexible consistency

- Trade-off between algorithm efficiency and system performance
Flexible consistency

- Trade-off between algorithm efficiency and system performance

Sequential

1 → 2 → 3 → 4

1-bounded delay

1 → 2 → 3 → 4 → 5

Eventual

1 → 2 → 3 → 4
Results for bounded delay

Ad click prediction

<table>
<thead>
<tr>
<th>time (hour)</th>
<th>0.9</th>
<th>0.45</th>
<th>0.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>bounded delay</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

- computing
- waiting
Results for bounded delay

Ad click prediction

- Sequential

- Computing

- Waiting

Time (hour)

Bounded delay
Results for bounded delay

Ad click prediction

Sequential

Time (hour)

Computing

Waiting

Bounded delay
Results for bounded delay

Ad click prediction

Sequential

Computing

Waiting

Best trade-off

Time (hour)

Bounded delay

0 1 2 4 8 16
User-defined filters
User-defined filters

- Selectively communicate (key, value) pairs
User-defined filters

- Selectively communicate (key, value) pairs
- E.g., the KKT filter
  - send pairs if they are likely to affect the model
  - >95% keys are filtered in the ad click prediction task
Industry size machine learning problems

Efficient communication

Fault tolerance
Machine learning job logs in a three-month period:
Machine learning job logs in a three-month period:

<table>
<thead>
<tr>
<th>failure rate %</th>
<th>#machine x time (hour)</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td></td>
</tr>
<tr>
<td>19.5</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>6.5</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Machine learning job logs in a three-month period:

![Graph showing failure rate percentage vs. machine hours](image)
Machine learning job logs in a three-month period:

![Bar chart showing failure rate % against #machine x time (hour)]
Machine learning job logs in a three-month period:

- Failure rate %
  - 6.5
  - 13
  - 26

- Machine x Time (hour)
  - 100
  - 1000
  - 10000
Fault tolerance

- Model is partitioned by consistent hashing
Fault tolerance

- Model is partitioned by consistent hashing
- Default replication: Chain replication (consistent, safe)
Fault tolerance

- Model is partitioned by consistent hashing
- Default replication: Chain replication (consistent, safe)
Fault tolerance

- Model is partitioned by consistent hashing
- Default replication: Chain replication (consistent, safe)
- Option: Aggregation reduces backup traffic (algo specific)
Fault tolerance

- Model is partitioned by consistent hashing
- Default replication: Chain replication (consistent, safe)
- Option: Aggregation reduces backup traffic (algo specific)
Fault tolerance

- Model is partitioned by consistent hashing
- Default replication: Chain replication (consistent, safe)

- Option: Aggregation reduces backup traffic (algo specific)
  implemented by efficient vector clock
Industry size machine learning problems

Efficient communication

Fault tolerance

Easy to use
(Key, value) vectors for the shared parameters

math sparse vector

\[ i_1 \quad i_2 \quad i_3 \]

(key, value) store

\[ (i_1, \text{blue}) \quad (i_2, \text{green}) \quad (i_3, \text{orange}) \]
(Key, value) vectors for the shared parameters

math sparse vector

(key, value) store

(i₁, □) (i₂, □) (i₃, □)

- Good for programmers: Matches mental model
- Good for system: Expose optimizations based upon structure of data
The Parameter Server in Actuality