Distributed Machine Learning

Jinyang Li

Slides of PS are from Mu Li
Some slides of TensorFlow are from Jeff Dean
Distributed Computation

1989: Ivy DSM, Munin DSM

2005: Map Reduce, Hadoop

2010: Power Graph, Spark

2015: TensorFlow

Scientific Computing
On-disk big data analysis
In-memory big data analysis
Large-scale machine learning
Large-scale Machine learning

• Applying ML algorithms to large quantities of data
• More training data $\rightarrow$ Better ML predictions
• More training data $\rightarrow$ Expensive computation $\rightarrow$ needs acceleration via distribution

• e.g.
  – cluster all news articles into different categories
  – recommending products/movies/articles to users
  – vision, machine translation, sentiment analysis
Some ML algorithms work well with MapReduce/Spark

• Example: k-means
  – Given data points \((x_1, x_2, \ldots, x_n)\), partition them into \(k\) clusters with centroids \((u_1, u_2, \ldots, u_k)\), such that the distances to assigned centroids are minimized

• Algorithm:
  – initialize random \(k\) centroids
  – for each data point, assign it to the “closest” centroid
  – for each cluster, compute the average of all assigned points as new centroid
  – iterate
... while tempDist > convergeDist:

    closest = data.map(
        lambda p: (closestPoint(p, kPoints), (p, 1)))

    pointStats = closest.reduceByKey(
        lambda p1_c1, p2_c2: (p1_c1[0] + p2_c2[0], p1_c1[1] + p2_c2[1]))

    newPoints = pointStats.map(
        lambda st: (st[0], st[1][0] / st[1][1])).collect()

    tempDist = sum(numpy.sum(((kPoints[iK] - p)**2) for (iK, p) in newPoints)

    for (iK, p) in newPoints:
        kPoints[iK] = p

Some ML algorithms work well with MapReduce/Spark

• Why k-means work well with MapReduce/Spark?
• Small parameters copied to and from master
  – k centroids
• Each iteration scans large amounts of data
Some ML algorithm works well with Graph frameworks

• Example: Collaborative filtering via ALS

Find U and V such that the difference between R and U*V are minimized
• **ALS: Alternating Minimum Squares**
  - Fix $U$, optimize $V$, then fix $V$, optimize $U$ etc.

\[ u_i = (\sum_{r_{i,j}} v_j v_j^T)^{-1} \sum_{r_{i,j}} r_{i,j} v_j \]

\[ v_j = (\sum_{r_{i,j}} u_i u_i^T)^{-1} \sum_{r_{i,j}} r_{i,j} u_i \]

*I did not include regularization terms*

ALS could be written in Spark too using joins
Beyond MapReduce and Graph Computing

• Example large-scale ML problem: predict whether a user will click an ad
A very large-scale ML problem

![Bar chart showing Ad click prediction from 2010 to 2014]

- **Training data size (TB)**
  - 2010: 50 TB
  - 2011: 100 TB
  - 2012: 200 TB
  - 2013: 400 TB
  - 2014: 800 TB
Ad-click prediction w/ logistic regression

100 billion unique features including query text, ad text, ad meta-data etc.

Predicted clickiness: $f(x \cdot w^T)$

Find $w$ that minimizes loss with regularization:

$$\sum_{i=1}^{n} l(x_i, y_i, w) + ...$$
Solving logistic regression

• Batch gradient descent

\[ \sum_{i=1}^{n} l(x_i, y_i, w) + \ldots \]

Training data  \rightarrow  current weight  \rightarrow  \sum_{i=1}^{n} l(x_i, y_i, w) + \ldots  \rightarrow  gradients  \rightarrow  new weights = current weights – learningRate*gradients
Solving logistic regression

- Batch gradient descent

\[ \sum_{i=1}^{n} l(x_i, y_i, w) + \ldots \]

\[ g: \sum_{i=1}^{n} g_i \]

Training data

Current weight

New weights = current weights – learningRate * gradients

Sum of the gradient contributed by each training example
Distributing logistic regression: data parallelism

$\sum_{i=1}^{\text{subset}} g_i$

$g = \sum \ldots$

update

need to distribute this

subset

subset

subset

a partition of training data
Scaling Distributed Machine Learning with the Parameter Server

Mu Li*†, David G. Andersen*, Jun Woo Park*, Alexander J. Smola*†, Amr Ahmed†,
Vanja Josifovski†, James Long†, Eugene J. Shekita†, Bor-Ying Su†
*Carnegie Mellon University  †Baidu  ‡Google
{muli, dga, junwoo}@cs.cmu.edu, alex@smola.org, {amra, vanjaj, jamlong, shekita, borylingsu}@google.com

Abstract

We propose a parameter server framework for distributed machine learning problems. Both data and workloads are distributed over worker nodes, while the server nodes maintain globally shared parameters, represented as dense or sparse vectors and matrices. The framework manages asynchronous data communication between nodes, and supports flexible consistency models, elastic scalability, and continuous fault tolerance.

To demonstrate the scalability of the proposed framework, we show experimental results on petabytes of real data with billions of examples and parameters on problems ranging from Sparse Logistic Regression to Latent Dirichlet Allocation and Distributed Sketching.

<table>
<thead>
<tr>
<th># of jobs</th>
<th>failure rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 hours</td>
<td>7.8%</td>
</tr>
<tr>
<td>1,000 hours</td>
<td>13.7%</td>
</tr>
<tr>
<td>10,000 hours</td>
<td>24.7%</td>
</tr>
</tbody>
</table>

Table 1: Statistics of machine learning jobs for a three month period in a data center.

The cost of synchronization and machine latency is high. At scale, fault tolerance is critical. Learning tasks are often performed in a cloud environment where machines can be unreliable and jobs can be preempted.

To illustrate the last point, we collected all job logs for...
PS partitions model parameters across machines
PS distributes communication and computation for weight updates
PS distributes communication and computation for weight updates
How data-parallel gradient descent works under PS

Workers pull the working set of model
How data-parallel gradient descent works under PS

Workers pull the working set of model
Iterate until stop

Server machines

Worker machines

workers compute gradients
How data-parallel gradient descent works under PS

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

Server machines

Worker machines
How data-parallel gradient descent works under PS
How data-parallel gradient descent works under PS

Workers pull the working set of model
Iterate until stop
workers compute gradients
workers push gradients
update model
workers pull updated model
Some additional features provided by PS

• Reduce push/pull communication traffic
  – not all parameters are used/updated by all workers

• Global synchronization may not always be needed
Range-based push and pull
Range-based push and pull
Range-based push and pull
Different consistencies for weights

Sequential / BSP

Bounded delay / SSP
[Langford 09, Cipar 13]

Eventual / Total asynchronous
[Smola 10]
Fault tolerance

- Default replication: Chain replication (consistent, safe)
  - worker 0 → server 0 → server 1
  - push x → push f(x)
  - ack

- Option: Aggregation reduces backup traffic (algo specific)
  - worker 0 → server 0 → server 1
  - push y
  - push x
  - ack
  - server 0 → server 1
  - push f(x+y)

implemented by efficient vector clock
Deep Learning and TensorFlow
The Deep learning revolution

• Deep neural network has become state-of-art technique for many AI problems:
  – Image recognition
  – Speech recognition
  – Machine translation
  – Question-answer
  – AlphaGo
  – ...

Why is deep learning so hot/important
A 2-layer Neural networks

\[
\begin{align*}
\text{predicted output } y & = \text{softmax}(h_2) \\
\end{align*}
\]

\[
\begin{align*}
\mathbf{h}_1 & = g(\mathbf{x} \cdot \mathbf{W}_1) \\
\mathbf{h}_2 & = g(\mathbf{h}_1 \cdot \mathbf{W}_2) \\
\end{align*}
\]

vector of 784 (flattened 28x28 image)

\[
\begin{align*}
\mathbf{W}_1 & : 784 \times 15 \\
\mathbf{W}_2 & : 15 \times 10 \\
\end{align*}
\]
DNNs have many layers

single layer
two layer
three layer
DNNs have many layers

Revolution of Depth

ImageNet Classification top-5 error (%)

- ILSVRC'15 ResNet: 3.57
- ILSVRC'14 GoogleNet: 6.7 layers
- ILSVRC'14 VGG: 7.3 layers
- ILSVRC'13: 8 layers
- ILSVRC'12 AlexNet: 8 layers
- ILSVRC'11: 16.4
- ILSVRC'10: 25.8
- ILSVRC'10: 28.2

152 layers
Why deep learning needs its own frameworks

• DNN computation is mostly dense
  – should use tensor (N-dimension array) instead of key-value (Spark) or graph (PowerGraph), to maximize efficiency

• DNN computation is evolving very fast
  – must allow all aspects of computation to be programmable in a high-level language

• DNNs are trained using stochastic gradient descent
  – Calculating gradient by hand is too much, must support auto-differentiation

• GPU acceleration
  – must use fast GPU kernels for convolution, matrix multiplication etc.
Deep learning frameworks

- Torch
- Theano
- Caffe
- Caffe2
- DistBelief (Google)
- MXNet
- CNTK (Microsoft)
- TensorFlow (Google)
- PyTorch (Facebook)
TensorFlow: Computation as a graph
Computation as a dataflow graph

Edges are N-dimensional arrays: Tensors
Example TensorFlow program

```python
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
x = tf.placeholder("float", shape=[None, 784])
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
y = tf.nn.softmax(tf.matmul(x, W) + b)
```
Computation is a dataflow graph with state.
Symbolic Auto Differentiation

• Represent computation as a graph enables auto-differentiation

```python
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = -tf.reduce_sum(y_ * tf.log(y))
opt = tf.train.GradientDescentOptimizer(0.01)
train_op = opt.minimize(cross_entropy)
```
Define a graph and execute it iteratively

```python
init = tf.initialize_all_variables()
sess = tf.Session()
sess.run(init)
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
```

Deep learning uses mini-batch based SGD. Each iteration trains using a mini-batch (100s) instead of the entire training dataset (billions).
TensorFlow’s graph can be distributed

TensorFlow inserts Send/Recv to transfer data across workers/devices
How is DNN training commonly distributed? data parallelism

$$g = \sum \ldots$$

update

$$g_i$$

a mini batch for iter-\textit{i}

a mini batch for iter-\textit{i+1}
TensorFlow has no explicit PS

```python
with tf.device("/job:ps/task:0"):
    weights_1 = tf.Variable(...)
    biases_1 = tf.Variable(...)

with tf.device("/job:ps/task:1"):
    weights_2 = tf.Variable(...)
    biases_2 = tf.Variable(...)

with tf.device("/job:worker/task:7"):
    input, labels = ...
    layer_1 = tf.nn.relu(tf.matmul(input, weights_1) + biases_1)
    logits = tf.nn.relu(tf.matmul(layer_1, weights_2) + biases_2)
    # ...
    train_op = ...

with tf.Session("grpc://worker7.example.com:2222") as sess:
    for _ in range(10000):
        sess.run(train_op)
```

http://tensorflow.org/deploy/distributed
TensorFlow has no explicit PS
Open questions: What TF did not do so well?

• Data parallelism is not always the right answer
  – Requires larger batch size → negatively affects accuracy and convergence

• TensorFlow’s distribution is flexible, but requires a lot of manual efforts
  – Manually partition weights (partition different weight matrices to different devices, partition a single weight matrix to different devices?)
  – How to partition dataflow graph among devices?
Summary: distributed machine learning

- Distributed ML algorithms are diverse
  - some suitable for MapReduce/Spark, GraphLab
  - others require new framework support
- Parameter service is good for ML algorithms optimized with data-parallel gradient descent
- Deep learning frameworks adopt based dataflow graph representation for its efficiency and flexibility
- Distributed training currently rely on data parallelism