Distributed Machine Learning

Jinyang Li

Slides of PS are from Mu Li
Some slides of TensorFlow are from Jeff Dean
Slides of ByteScheduler are based on authors’ SOSP’19 talk
Distributed Computation

- Ivy DSM (1989)
- Munin DSM (1989+)
- Map Reduce (2005)
- Hadoop (2005+)
- Spark (2010+)
- POWER Graph Service (2010+)

- Scientific Computing: e.g. n-body simulation
- On-disk big data analysis: e.g. indexing/sort
- In-memory big data Analysis: e.g. PageRank
- Large-scale machine learning
Large-scale Machine learning

• Applying ML algorithms to large quantities of data
• More training data → Better ML predictions
• More training data → Expensive computation → 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 \((x_1, x_2, ..., x_n)\), partition them into \(k\) clusters with centroids \((u_1, u_2, ..., u_k)\), 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


For each point, output the index of the best centroid, p, 1 (used as count)

For each cluster, add point coordinates and counts together

For each cluster, scale summed point coordinates by counts to compute new centroids

Turn RDD into a normal variable that will be copied to every worker
Some ML algorithms work well with MapReduce/Spark

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

- Example: Collaborative filtering via ALS

  Extremely sparse

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.

\[
\begin{align*}
    u_i &= \left( \sum_{i,j} v_j v_j^T \right)^{-1} \sum_{r_{i,j}} r_{i,j} v_j \\
    v_j &= \left( \sum_{i,j} u_i u_i^T \right)^{-1} \sum_{r_{i,j}} r_{i,j} u_i
\end{align*}
\]

* 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 and training data size (TB) from 2010 to 2014. The chart indicates a significant increase in training data size and ad click prediction over the years.]
Ad-click prediction w/ logistic regression

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

100s of billions

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

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

\[ g: \text{gradients} = \text{current weights} - \text{learningRate} \times \text{gradients} \]

new weights = current weights – learningRate*gradients
Solving logistic regression

• Batch gradient descent

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

Training data

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

sum of the gradient contributed by each training example

new weights = current weights – learningRate*gradients

current weight
Distributing logistic regression: data parallelism

need to distribute this

\[ g = \sum_{i=1}^{\text{subset}} g_i \]

update

\[ W \]

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-Yiing Su†
* Carnegie Mellon University †Baidu ‡Google
{muli, dga, junwoop}@cs.cmu.edu, alex@smola.org, {amra, vanjaj, jamlong, shekita, boryiingsu}@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>≈ #machine × time</th>
<th># of jobs</th>
<th>failure rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 hours</td>
<td>13,187</td>
<td>7.8%</td>
</tr>
<tr>
<td>1,000 hours</td>
<td>1,366</td>
<td>13.7%</td>
</tr>
<tr>
<td>10,000 hours</td>
<td>77</td>
<td>24.7%</td>
</tr>
</tbody>
</table>

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

- 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

Server machines

Worker machines
How data-parallel gradient descent works under PS

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

Server machines

Worker machines
How data-parallel gradient descent works under PS

Workers pull the working set of model
Iterate until stop
workers compute gradients
workers push 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
Update model
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
Different consistencies for weights

Sequential / BSP

Bounded delay / SSP
[Langford 09, Cipar 13]

Eventual / Total asynchronous
[Smola 10]
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

predicted output $y$ is:

$$h_1 = g(x \cdot W_1)$$
$$h_2 = g(h_1 \cdot W_2)$$
$$y = \text{softmax}(h_2)$$

vector of 784
(flattened 28x28 image)
DNNs have many layers
DNNs have many layers

Revolution of Depth

ImageNet Classification top-5 error (%)
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
- MXNet
- DistBelief (Google)
- CNTK (Microsoft)
- TensorFlow (Google)
- PyTorch (Facebook)

Timeline:
- 2002
- 2010
- 2015
- 2017
TensorFlow: Computation as a graph

Graph of Nodes, also called Operations or ops.
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

'Biases' is a variable

Some ops compute gradients

== updates biases
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)
How is DNN training commonly distributed? data parallelism

\[ g = \sum \ldots \]

update

\[ g_i \]

a mini batch for iter-i

a mini batch for iter-i+1
Can we improve upon the basic PS?

A Generic Communication Scheduler for Distributed DNN Training Acceleration

Yanghua Peng*, Yibo Zhu†, Yangrui Chen*, Yixin Bao*, Bairen Yi†, Chang Lan†
Chuan Wu*, Chuanxiong Guo†
{yhpeng, yrchen, yxbao, cwu}@cs.hku.hk*, {zhuyibo, yibairen.byron, lanchang, guochuanxiong}@bytedance.com†
The University of Hong Kong*, ByteDance Inc.†

Abstract
We present ByteScheduler, a generic communication scheduler for distributed DNN training acceleration. ByteScheduler is based on our principled analysis that partitioning and rearranging the tensor transmissions can result in optimal results in theory and good performance in real-world even with scheduling overhead. To make ByteScheduler work generally for various DNN training frameworks, we introduce a unified abstraction and a Dependency Proxy mechanism to enable communication scheduling without breaking the original dependencies in framework engines. We further introduce a Bayesian Optimization approach to auto-tune tensor partition size and other parameters for different training models under various networking conditions. ByteScheduler now supports TensorFlow, PyTorch, and MXNet without modifying their source code, and works well with both Parameter Server (PS) and all-reduce architectures for gradient Natural Language Processing, etc. Training DNNs, however, are time-consuming tasks, mainly due to large volumes of data and growing DNN model sizes. The most common way to scale out and accelerate DNN training is data parallelism (§2). Unfortunately, its performance is often far from linear speed-up, due mainly to the communication overhead. As a large online service provider, in many of our internal and publicly available training workloads, communication often consumes a significant portion of total training time. This is also echoed by recent literature [9, 18, 39].

Consequently, many different communication acceleration approaches have been proposed and integrated into popular frameworks, including TensorFlow [6], PyTorch [28], MXNet [12], with drastically different implementations. For example, one can use RDMA to replace TCP, while the RDMA implementations are quite different among frameworks. Or, one can use ring-based all-reduce, either from one of several
Naïve data parallel training uses FIFO communication.
FIFO communication is sub-optimal

weight tensor of layer $N-1$ is enqueued for the network earlier than layer $1$

But weight tensor of layer $N-1$ is need for computation later than that of for layer $1$
Priority scheduling can improve performance

Partition each tensor so we can pretensor to finish communication entired before we can pre

Better overlap of communication and computation

40% improvement
ByteScheduler’s design

- Partition all gradient/weight tensors
  - $s$: Partition size

- Assign priority to tensors
  - Determine operator execution order based on dataflow graph
    - topological sort
  - assign higher priority to tensors needed by operators with smaller sort orders.

- Simultaneously send up $c$ tensor partitions (credit size)
  - After finish sending a tensor partition, choose the next highest priority one waiting

- Auto-tune $s$ and $c$. 
ByteScheduler Performance

- Works better for DNNs with large weights
  - Communication is the bottleneck
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

• Distributed training currently rely on data parallelism, but
  – DP requires larger batch size → negatively affects accuracy and convergence