MapReduce/Bigtable for Distributed Optimization

Keith Hall, Scott Gilpin, Gideon Mann

presented by: Slav Petrov
Zurich, Thornton, New York
Google Inc.
Outline

• Large-scale Gradient Optimization
  – Distributed Gradient
  – Asynchronous Updates
  – Iterative Parameter Mixtures

• MapReduce and Bigtable

• Experimental Results
Gradient Optimization Setting

Goal: find $\theta^* = \arg\min_{\theta} f(\theta)$

If $f$ is differentiable, then solve via gradient updates:

$$\theta^{i+1} = \theta^i + \alpha \nabla f(\theta^i)$$

Consider case where $f$ is composed of a sum of differentiable functions $f_q$, then the gradient update can be written as:

$$\theta^{i+1} = \theta^i + \alpha \sum_q \nabla f_q(\theta^i)$$

This case is the focus of the talk
Maximum Entropy Models

\( \mathcal{X} \) : input space  \quad \mathcal{Y} : output space

\( \Phi : \mathcal{X} \times \mathcal{Y} \rightarrow \mathbb{R}^n \) : feature mapping

\( S = ((x_1, y_1), \ldots, (x_m, y_m)) \) : training data

\( p_\theta[y|x] = \frac{1}{Z} \exp(\theta \cdot \Phi(x, y)) \) : probability model

\( \theta^* = \arg\min_\theta \frac{1}{m} \sum_i \log p_\theta(y_i|x_i) \)

The objective function is a summation of functions, each of which is computed on one data instance.
Distributed Gradient

Observe that the gradient update can be broken down into three phases:

\[ \theta^{i+1} = \theta^i + \alpha \sum_q \nabla f_q(\theta^i) \]

\( \nabla f_q(\theta^i) \) Can be done in parallel

\[ \sum_q \] Cheap to compute

Update Step Depends on number of features, not data size.

At each iteration, must send complete \( \theta \) to each parallel worker.

[Chu et al. '07]
Stochastic Gradient

Alternatively approximate the sum by a subset of functions $\nabla f_q(\theta^i)$. If the subset is size 1, then the algorithm is an online:

$$\theta^{i+1} = \theta^i + \alpha \sum_q \nabla f_q(\theta^i)$$

$$\theta^{\theta+1} = \theta^i + \alpha \nabla f_q(\theta^i)$$

Stochastic gradient approaches provably converge, and in practice often much quicker than exact gradient calculation methods.
Asynchronous updates are a distributed extension of stochastic gradient.

Each worker:
- Get current $\theta^i$
- Compute $\nabla f_q(\theta^i)$
- Update global parameter vector

Since each worker will not be computing in lock-step, some gradients will be based on old parameters. Nonetheless, this also converges.

[Langford et al. ’09]
Iterative Parameter Mixtures

Separately estimate $\theta$ on different samples, and then combine. Iterative: take resulting $\theta$ as starting point for next round and rerun.

For certain classes of objective functions (e.g. maxent and perceptron), convergence can also be shown.

[Mann et al. ’09, McDonald et al. ’10]

Little communication between workers.
Typical Datacenter (at Google)

Commodity machines, with relatively few cores (e.g. <=4)

Racks of machines connected by commodity ethernet switches.

No GPUs. No Shared Memory. No NAS. No massive multicores machines.

[Holzle, Barroso ’09]
Implications for algorithms

• Communication between workers is costly
  – No shared memory means very hard to maintain global state
  – Even ethernet communication can be a significant source of latency

• Each worker relatively low powered
  – No machines have massive amounts of RAM or disk
  – Simple isolated jobs performed in parallel will have significant gains.
Outline

• Distributed Gradient Optimization
• MapReduce and Bigtable
  – Optimal Configurations
• Experimental Results
1. Backup map workers compensate for failure of map task.

2. Sorting phase requires all map jobs to be completed before starting reduce phase.
Distributed Gradient and Iterative Parameter Mixtures are straight-forward to implement in MapReduce.

Distributed Gradient:
Each iteration has multiple mappers and one reducer.

Iterative Parameter Mixtures:
One mapper/reducer pair, each runs to convergence, and then there’s an outside mixture and loop.

Asynchronous – needs shared memory.
Bigtable

Distributed storage – built off of GFS
Not a database: cells indexed by row/column/time. (e.g. can recover n most recent values)
Multiple layers of caching:
    Machine and network level
Transaction processing is optional
Asynchronous Updates

Map
- Update Parameters, Compute Gradient
- Sorting turned off

Reduce
- Sum and Update Bigtable
- Gradients collected into mini-batches

Bigtable
- Transaction processing turned off: overwrites are possible
Outline

- Distributed Gradient Optimization
- MapReduce and Bigtable
- Experimental Results
Experimental Set-up

Click-through data

- A. 370 million training instances
  - Partitioned into 200 pieces
  - 240 worker machines

- B. 1.6 billion instances
  - Partitioned into 900 pieces
  - 600 worker machines

- Evaluation: 185 million instances (temporally occurring after first instances)
Stopping Criterion Tricky

Complicated because the implementations are all slightly different

• Gradient change:
  – no global gradient in asynchronous or iterative parameter mixtures

• Log-likelihood/objective function value:
  – Asynchronous log-likelihood fluctuating frequently

• Fixed number of iterations
  – Means different things for each implementation

Currently working on more clean performance result estimation – take AUC results with a grain of salt.
370 Million Training Instances

<table>
<thead>
<tr>
<th>Method</th>
<th>AUC</th>
<th>CPU (rel.)</th>
<th>Wall clock (rel.)</th>
<th>Network Usage (rel.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-core SGD</td>
<td>.8557</td>
<td>0.10</td>
<td>9.64</td>
<td>0.03</td>
</tr>
<tr>
<td>Distributed Gradient</td>
<td>.8323</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Asynchronous SGD</td>
<td>.7283</td>
<td>2.17</td>
<td>7.50</td>
<td>82.47</td>
</tr>
<tr>
<td>Iterative Parameter Mixture: MaxEnt</td>
<td>.8557</td>
<td>0.12</td>
<td>0.13</td>
<td>0.10</td>
</tr>
<tr>
<td>Iterative Parameter Mixture: Perceptron</td>
<td>.8870</td>
<td>0.16</td>
<td>0.20</td>
<td>0.22</td>
</tr>
</tbody>
</table>

Observations:
- Even with tuning, Asynchronous Updates are impractical
- Iterator Parameter mixtures perform well
- IPM w/240 machines is ~70x as fast as single core
- w/10 machines is 8x as fast as single core.
## 1.6 Billion Training Instances

<table>
<thead>
<tr>
<th>Method</th>
<th>AUC</th>
<th>CPU (rel.)</th>
<th>Wall clock (rel.)</th>
<th>Network Usage (rel.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributed Gradient</td>
<td>.8117</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Asynchronous SGD</td>
<td>.8525</td>
<td>3.71</td>
<td>14.33</td>
<td>133.93</td>
</tr>
<tr>
<td>Iterative Parameter Mixture : MaxEnt</td>
<td>.8904</td>
<td>0.17</td>
<td>0.15</td>
<td>0.15</td>
</tr>
<tr>
<td>Iterative Parameter Mixture : Perceptron</td>
<td>.8961</td>
<td>0.24</td>
<td>0.20</td>
<td>0.38</td>
</tr>
</tbody>
</table>

**Observations:**

- Network usage much worse with more machines, and asynchronous much slower.
- Iterative parameter mixtures still effective with 900 partitions.
Conclusions

Asynchronous Updates not suited to current datacenter configurations

Iterative parameter mixtures give significant time improvements over distributed gradient.