UBC MLRG (Winter 2018):
Parallel and Distributed Machine Learning
Machine Learning Reading Group (MLRG)

• **Machine learning reading group** (MLRG) format:
  – Each semester we pick a general topic.
  – Each week someone leads us through a tutorial-style lecture/discussion.
  – So it’s organized a bit more like a “topics course” than reading group.

• We use this format because **ML has become a huge field**.
Machine Learning Reading Group (MLRG)

• I’ve tried to pack as much as possible into the two ML courses:
  – CPSC 340 covers most of the most-useful methods.
  – CPSC 540 covers most of the background needed to read research papers.

• This reading group covers topics that aren’t yet in these course.
  – Aimed at people who have taken CPSC 340, and are comfortable with 540-level material.

• This may change now that we have 4 ML faculty.
Recent MLRG History

• Topics covered in recent tutorial-style MLRG sessions:
  – Summer 2015: Probabilistic graphical models.
  – Fall 2015: Convex optimization.
  – Summer 2016: Miscellaneous.
  – Fall 2016: Deep learning.
  – Summer 2017: Online, active, and causal learning.
  – Fall 2017: Deep learning meets graphical models.
Motivation for Parallel and Distributed Systems

• Clock speeds aren’t increasing anymore:
  – Though new tricks like 64-bit vs. 32-bit.

• But datasets keep getting bigger.
  – MNIST: 60k, ImageNet: 1.4M.

• We need to use parallel computation.
  – Use more than 1 CPU to reduce time.
  – Lets you keep pace with growth of data.
Motivation for Parallel and Distributed Systems

- Data might get so big it doesn’t fit on one machine.

- We need to consider distributed data and distributed computation.
  - How can we solve ML problems efficiently in this setting?

https://en.wikipedia.org/wiki/Hard_disk_drive
3 Approaches to Machine Learning

• There are roughly three computational approaches to ML:
  – **Counting** (sufficient statistics, decision trees, naïve Bayes, KNN).
  – **Optimization** (least squares, logistic regression, PCA, deep learning).
  – **Integration** (random forests, graphical models, Bayesian methods).

• Today:
  – **Issues** arising in these settings when you parallelize/distributed.
Counting-Based Learning

• Consider finding the mean of a data matrix ‘\(X\)’:

\[
X = \left[ \begin{array}{c} \vdots \end{array} \right] \in \mathbb{R}^{d \times n}
\]

Compute: \( \mu_j = \frac{1}{n} \sum_{i=1}^{n} x_{ij} \)

for each ‘\(j\)’

• Usual cost with a processor is \(O(nd)\).
  – For each of the ‘\(d\)’ values of ‘\(j\)’, add up the ‘\(n\)’ values of ‘\(x_{ij}\)’.

• Now suppose we have ‘\(p\)’ processors with shared memory:
  – Make each processor each up the number for \(O(n/d)\) examples.
  – So each processor takes \(O(nd/p)\) operations, and total time is \(O(nd/p)\).
Linear Speedup

• This is called a “linear speedup”: 
  – We’re ‘p’-times faster with ‘p’ processors.

• Can we do better? 
  – No! 
  – Superlinear speedups aren’t possible (in standard models of computation). 
    • In practice, issues like caching levels might give superlinear in some situations.

• So a linear speedup is the best case scenario. 
  – Our job is to design methods where speedup isn’t too sublinear.
Embarrassingly Parallel

• We say that computing the mean is “embarrassingly parallel”.
  – We can divide most of work into ‘p’ independent sub-problems.

• You’ll rarely see papers about embarrassingly-parallel methods.
  – It’s not really that interesting.

• But, embarrassingly parallel problems are very common.
  – You should always look for embarrassingly parallel approaches first.
Issues: Lock and Synchronization

• This algorithm may not achieve linear speedup in practice.

• One reason is locking:
  – They can’t all write to the same $\mu_j$ values at once.

• Another is synchronization
  – One processor could take much longer than the others.

• Even with homogeneous hardware, another issue is load balancing:
  – Data could be sparse with most non-zeroes assigned to the same processor.

• For this problem, simple modifications could alleviate these issues.
  – For more complicated problems, we need to think about these issues.
Distributed Computation

• Suppose data was distributed (evenly) on ‘p’ different machines.

• Since they don’t have shared memory, we need to communicate.

• Computing mean in this distributed setting:
  – Each computer computes mean of its own set of examples.
  – Each computer sends its mean to a “master” computer.
  – The “master” computer combines them together to get overall mean.
Map and Reduce Operations

• Computing mean on each computer is called a “map” operation.
  – Each machine computes a simple “value” on its own data.

• Combining means is called a “reduce” operation.
  – The “values” are combined with a simple binary operation.

• Standard distributed frameworks will implement these operations.
  – And usually a few others.
Analysis of Map then Reduce Approach

• The “map” step costs $O(nd/p)$ on each machine.
• The “reduce” step involves each machine sending ‘d’ numbers.
• If they all send to “master”, cost of reduce is $O(dp)$.
  – So total cost is $O(nd/p + dp)$, so for large ‘p’ we won’t have linear speedup.

• You be more clever and organize communication in a binary tree:
  – Cost $O(nd/p + d \log(p))$, so linear speedup if $n/p > \log(p)$.
  – Obviously, won’t be linear with more machines than examples.

• Maybe you want to distribute features rather than examples?
  – Only need to communicate $O(d)$ numbers if each has $O(d/p)$ features.
Issues: Communication Costs

• Communicating among machines adds extra costs.
  – We need to think about if this is worth it.

• Communication is usually expensive compared to computation.
  – Sometimes, some machines can communicate more cheaply than others.

• Also, how did you get data onto ‘p’ machines in the first place?
  – This cost is often ignored in papers, but it matters where the data “starts”.
  – You don’t want to send data to machines just to compute mean!

• If you have huge ‘p’, probability of failure becomes non-trivial.
  – How do you deal with computation or communication failure?
Optimization-Based Learning

• Optimization-based methods minimize average of continuous $f_i$:

$$
\min_{w \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} f_i(w)
$$

• Standard approach is gradient descent (and faster variations):

$$
w^{k+1} = w^k - \frac{\alpha_k}{n} \sum_{i=1}^{n} \nabla f_i(w^k)
$$

• This is often embarrassingly parallel:
  – Dominant cost is computing gradient on each of ‘n’ examples.
  – Each processor can compute gradients for $O(n/p)$ examples.

• Papers look at fancier methods, but if you can do this you should.
Fancier Optimization Methods

- **Stochastic gradient** methods:
  - Not so easy to parallelize, each iteration only uses 1 gradient.
  - You could have each processor compute 1 gradient and use ‘batch’ update.
    - Does not give a linear speedup: just reduces variance of gradient estimate.
  - Asynchronous approach: each processor read/updates “master” vector.
    - Works if you make the step-size smaller.

- **Coordinate optimization** methods:
  - Each machine updates one coordinate.
  - Doesn’t work unless you make the step-size small enough.
Fancier Optimization Methods

• **Decentralized gradient:**
  – Each machine takes a *gradient descent step on its own data*.
  – Parameters are averaged across neighbours in communication graph.

• **Newton’s method:**
  – Newton has memory requirements and iteration cost.
    • But it takes very few iterations.
  – *Cloud computing* allows enormous memory/parallelism.
  – Maybe Newton makes sense again in this setting?
Integration-Based Learning

• Integration-based learning methods need to solve integrals:

\[ \hat{y}_i = \int f(x)p(x) \, dx \]

• Typical approach is Monte Carlo methods:

\[ \hat{y}_i \sim \frac{1}{m} \sum_{i=1}^{m} f(x_m) \text{ where } x_m \text{ are distributed according to } p(x) \]

• Embarrassingly-parallel if you can generate IID samples from \( p(x) \):
  – Have each processor generate its own independent samples.

• Typical cases like MCMC are more complicated:
  – Running independent MCMC chains is embarrassingly-parallel.
  – But speedup could be very sublinear if all chains are in “burn in” phase.
<table>
<thead>
<tr>
<th>Date</th>
<th>Topic</th>
<th>Presenter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jan 30</td>
<td>Motivation/Overview</td>
<td>Mark</td>
</tr>
<tr>
<td>Feb 6</td>
<td>Distributed file systems (MAPREDUCE, HADOOP, Spark, etc.)</td>
<td>Yasha</td>
</tr>
<tr>
<td>Feb 13</td>
<td>Asynchronous stochastic gradient (HOGWILD!, YellowFin, etc.)</td>
<td>Michael</td>
</tr>
<tr>
<td>Feb 27</td>
<td>Synchronous stochastic gradient (&quot;fit then average&quot;, Sync-Opt)</td>
<td>Sharan</td>
</tr>
<tr>
<td>Mar 6</td>
<td>Parallel coordinate optimization</td>
<td>Julie</td>
</tr>
<tr>
<td>Mar 13</td>
<td>Decentralized gradient (EXTRA)</td>
<td>Devon</td>
</tr>
<tr>
<td>Mar 20</td>
<td>Decomposition methods (Elastic-Averaging, ADMM, etc.)</td>
<td>Wu</td>
</tr>
<tr>
<td>Mar 27</td>
<td>Asynchronous/distributed SAG/SDCA/SVRG</td>
<td>Reza</td>
</tr>
<tr>
<td>Apr 3</td>
<td>Randomized Newton and least squares on the cloud</td>
<td>Vaden</td>
</tr>
<tr>
<td>Apr 10</td>
<td>Parallel tempering and distributed particle filtering</td>
<td>Nasim</td>
</tr>
<tr>
<td>Apr 17</td>
<td>Distributed deep networks (DDNNs, Downpour)</td>
<td>Alireza</td>
</tr>
<tr>
<td>Apr 24</td>
<td>Blockchain-based distributed learning</td>
<td>Raunak*</td>
</tr>
</tbody>
</table>