CME 323: Distributed Algorithms and Optimization, Spring 2015
`http://stanford.edu/~rezab/dao`.
Instructor: Reza Zadeh, Matroid and Stanford.

Lecture 8, 4/20/2016. Scribed by Irwan Bello, David Flatow.

Many learning algorithms can be formalized with the following unconstrained optimization problem:

$$\min_w \quad F(w) = \sum_{i=0}^{n} F_i(w, x_i, y_i)$$

where i indexes the data, $x_i \in R^d$ is an input vector, $y_i \in R$ is a label and w is the parameter. There are 2 quantities of interest when scaling such algorithms:

- **Data parallelism**: How does the algorithm scale with n (number of training points)?
- **Model parallelism**: How does the algorithm scale with d (number of parameters)?

The objective function F depends on the problem at hand. Some examples include:

- cross-entropy (e.g: for logistic regression or neural nets)
- hinge loss[1] $F_i(w, x_i, y_i) = \max(1 - y_i(x_i^T w))$ with $y_i \in \{-1, +1\}$ (e.g: for SVM)
- (linear) least squares $F_i(w, x_i, y_i) = ||x_i^T w - y_i||_2^2$

Note, in practice we often have hyper-parameters, but changing them is manual and is typically an embarrassingly parallel problem.

8 Optimization Algorithms

8.1 Gradient Descent

We'll focus our analysis on linear least squares. In particular, we consider the complexity of computing the gradient of the least squares objective (many optimization algorithms rely on computing the gradient). The linear least squares objective:

$$F(w) = \sum_{i=1}^{n} F_i(w, x_i, y_i) = \sum_{i=1}^{n} ||x_i^T w - y_i||_2^2$$

has gradient

$$\Delta_w F = \sum_{i=1}^{n} \Delta_w F_i(w) = \sum_{i=1}^{n} 2(x_i^T w - y_i) x_i \in R^d$$

[1] Non-differentiable but still convex and can have subgradients

Computing the gradient for a single datapoint can be done in $O(\log d)$ depth (the computation being dominated by the dot product operation). The global gradient $\Delta_w F$ then requires $O(\log d) + O(\log n)$ depth.

Assuming gradient descent requires T iterations[2] we have the form:

$$w_{k+1} \leftarrow w_k - \alpha \Delta F(w_k)$$

For $k = 1 \dots T$. We also assume a constant step size and fixed $\alpha < \frac{1}{L}$ where L is the lipschitz constant. Thus, the total depth is $O(T \log d) + O(T \log n)$, which means that our algorithm is slow even with multiple processors. The problem is the sequential nature of the gradient updates that is not easily paralellized.

8.2 Stochastic Gradient Descent

This motivates the use of stochastic gradient descent (SGD) where the gradient is computed on a batch of data sampled uniformly from the training set. This works because the mini-batch gradient is equal in expectation to the full batch gradient, so the number of iterations to converge doesn't necessarily increase linearly to the inverse of the proportion of points being sampled, while the work decreases linearly with this proportion.

The table below summarizes the performance of gradient descent vs stochastic gradient descent[3] where ϵ is our tolerance for convergence.

..	# iterations	work/iter	depth/iter	total work	total depth
GD	$O(\log \frac{1}{\epsilon})$	$O(nd)$	$O(\log d + \log n)$	$O(nd \log \frac{1}{\epsilon})$	$O((\log \frac{1}{\epsilon})(\log d + \log n))$
SGD	$O(\frac{1}{\epsilon})$	$O(d)$	$O(\log d)$	$O(\frac{d}{\epsilon})$	$O(\frac{\log d}{\epsilon})$

A further improvement on scaling gradient descent comes from the Hogwild![1] idea: it turns out that many statistical algorithms can be run in parallel without locks and still converge under certain assumptions[4].

8.3 Hogwild!

Hogwild for SGD is simply to never lock waiting for w_k computations to finish and instead just reuse current w_k.

Alternatively it is possible to assign a single processor to each ΔF_i computation and have the processors write updates to their respectively assigned w_i and move on without waiting / locking. This is more common in practice. When the single processor is finished with it's ΔF_i that update is applied to the current value of w rather than from the weight vector that was the basis for computing ΔF_i.

[2]Here, to simplify the analysis, we assume the algorithm converges in T steps, later we'll address this in more detail

[3]There's a hidden $O(\log(n))$ cost to sampling from a uniform random variable

[4]The assumptions here are on the sparsity of the training data and relative rates of processors; however, in practice Hogwild performs well for many applications

Communication in GD/SGD happens at each iteration: $O(\frac{1}{\varepsilon})$ vs $O(\log \frac{1}{\varepsilon})$. Thus large differences in the cost of communication between a single machine with shared memory and distributed cluster may change the economics of the choice between GD and SGD. In practice we see single machine architectures using SGD on big beefy GPUs for things like deep learning whereas we may see distributed systems running GD.

9 Scheduling

The scheduler assigns sequential computation threads to processors. Brent's theorem assumes an optimal scheduling, where no processor is over/underused. One might think that this will be a hard problem since, for example the related problem, Optimal Makespan Scheduling is is NP-hard. However, we find that a greedy approach can yield a fast and easy algorithm with a competitive approximation ratio.

References

[1] B. Recht, C. Re, S. Wright, F. Niu *Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent.* NIPS, 2011.