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

Lecture 11, 5/4/2015. Scribed by K. Bergen, K. Chavez, A. Ioannidis, and S. Schmit.

10.1 Introduction

In computer science "streaming" has traditionally referred to a situation with continuous read off disk, i.e. no random access, and with little available memory. For "distributed streams," however, we are considering a continuous flow of data that is so high throughput (a "fire hose") that it cannot be handled by a single machine and needs to be split across many.

10.2 Gradient descent

Consider a function $F(w)$ that we seek to optimize, $\min_w F(w)$, which is the sum of constituent functions, $F(w) = \sum_{i=1}^n f_i(w)$. We will be assuming n is large, $w \in \mathbb{R}^d$, and d fits in memory on a single machine. Now, we can calculate the gradient of $F(w)$ as a simple sum of the gradients of the constituent $f_i(w)$ functions, $\nabla F(w) = \sum_{i=1}^n \nabla f_i(w)$, which we can then compute in $O(nd)$. For example, if we have a least squares objective, i.e. $f_i(w) = (x_i^\top w - y_i)^2$, then $\nabla f_i(w) = 2(w^\top x_i - y_i)x_i$, which is just a re-weighting of the original vector $f_i(w)$.

Algorithm 1 Gradient Descent

Initialize w_1
for $k = 1$ **to** K **do**
  Compute $\nabla F(w_k) = \sum_{i=1}^n \nabla f_i(w_k)$
  Update $w_{k+1} \leftarrow w_k - \alpha \nabla F(w_k)$
end for
Return w_K.

We now perform the gradient descent update (stepping 'down the gradient') $w_{k+1} \leftarrow w_k - \alpha \nabla F(w)$, where $\nabla F(w) = \sum_{i=1}^n \nabla f_i(w)$, which takes $O(ndT)$. (The α parameter determines the step-size.) Repeatedly calculating, and then stepping down, an objective function's gradient like this constitutes gradient descent (GD).

10.3 Stochastic gradient descent

The downside of gradient descent is that we have to compute the sum of all the gradients before we update the weights. Stochastic gradient descent (SGD) tries to lower the computation per iteration, at the cost of an increased number of iterations necessary for convergence.

Instead of computing the sum of all gradients, stochastic gradient descent selects an observation uniformly at random, say i and uses $f_i(w)$ as an estimator for $F(w)$. While this is a noisy estimator, we are able to update the weights much more frequency and therefore hope to converge more rapidly.

Algorithm 2 Stochastic Gradient Descent

Initialize w_1
for $k = 1$ **to** K **do**
  Sample an observation i uniformly at random
  Update $w_{k+1} \leftarrow w_k - \alpha \nabla f_i(w_k)$
end for
Return w_K.

Note that this updates takes only $\mathcal{O}(d)$ computation, though the total number of iterations, T, is larger than in the Gradient Descent algorithm. For strongly convex functions, results on the number of iterations and computational cost is summarized in Table 1.

Method	# iterations	cost per iteration	total cost
GD	$\mathcal{O}(\log(1/\epsilon))$	$\mathcal{O}(nd)$	$\mathcal{O}(\frac{n}{d}\log(1/\epsilon))$
SGD	$\mathcal{O}(1/\epsilon)$	$\mathcal{O}(d)$	$\mathcal{O}(d/\epsilon)$

Table 1: Comparison of complexity of GD and SDG.

While the dependence on ϵ is worse for SGD, we note the for n large enough, stochastic gradient descent wins in computational cost. However, parallelizing SGD is not trivial. In GD, there are clear communication barriers between iterations. But SGD can need thousands of iterations, and we can't feasibly communicate that often.

10.4 Hogwild!

Outside of the Spark framework, there are a few approaches that attempt to alleviate the synchronization issue with SGD. Consider the typical set-up for distributed SGD, where we have a driver node which holds a copy of the latest model and various worker nodes which compute stochastic gradient updates.

A technically correct implementation of SGD would have a single worker read the latest copy of the model, compute a stochastic gradient, and update the model, while all other workers wait. Once the model has been updated, the next worker can proceed to read, compute, and update. This does not take advantage of any parallelism.

The Hogwild! paper claims that under certain sparsity assumptions, the workers need not wait for each other. In other words, each worker is free to read the latest model, compute a gradient update, and apply that update to the model, without regard to the updates being done by other workers. In particular, it's possible that by the time a worker applies a gradient update to the latest model, it no longer matches the model that was used to compute the update. However, the

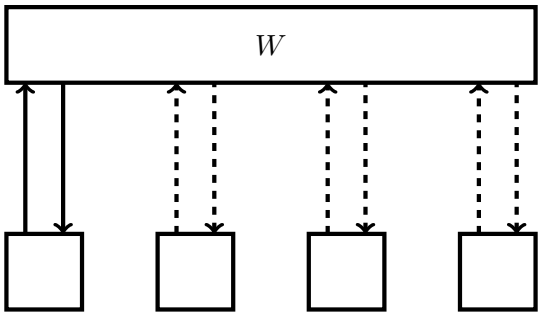

Figure 1: With proper SGD, the updates indicated by dashed lines would have to wait until the bold line update finished. With Hogwild, all four updates can be performed asynchronously.

assumption that a single SGD step only modifies a small portion of the model (i.e. it is a "sparse" update) allows this to work. See [1] for full details.

10.5 Parallel SGD

For strongly convex objective functions, there is another simple algorithm that has been proposed.

Algorithm 3 Parallel SGD

Shuffle the data ▷ this is an expensive operation
for $k = 1$ **to** K **do** ▷ in parallel
  Perform SGD on the k^{th} random partition of the data, producing model w_k.
end for
Compute the average of the models, $w = \frac{1}{K} \sum_k w_k$.
Return w.

Full details available in [2].

10.6 Streaming SGD

So far we have assumed that we have all of the data at our disposal. Instead, we now consider the streaming setting where we don't have access to all the data, and it is potentially infinitely large.

This scenario lends itself nicely to minibatch gradient descent. Over some interval of time we collect B samples of data as an RDD, compute the gradient over this batch,

$$g_k = \sum_{i=1}^{B} \nabla_w f_i(w_k),$$

and then update and broadcast the new model

$$w_{k+1} = w_k + \eta g.$$

The stream of RDDs is created by Spark Streaming and referred to as a DStream. After an RDD is processed, we discard it. Thus each data point is touched only once.

We typically assume that the stream provides us randomly shuffled data. This assumption is generally not true, but it allows us to prove the same theoretical guarantees as minibatch gradient descent. Without it, we cannot make any meaningful guarantees. Note that we cannot shuffle the data explicitly since we do not have access to the full data set.

The streaming SGD algorithm for linear models is currently available in Spark.

10.7 K-means

We now consider the problem of clustering n observations $\{x_1, \ldots, x_n\} \in \mathbb{R}^d$. The goal is to identify a set of k centers, $\{c_1, \ldots, c_k\} \in \mathbb{R}^d$ that minimize the sum of the squared distances between each observation x its closest center $c^\star(x)$:

$$\phi = \min_{\{c_1,\ldots,c_k\}} \sum_{i=1}^{n} \|x_i - c^\star(x_i)\|_2^2 \tag{1}$$

The number of clusters k is small and is assumed to be known. The standard algorithm, called the k-means algorithm or *Lloyd's iteration*[3].

Algorithm 4 Lloyd's iteration (k-means algorithm)

Start with an arbitrary* set of k cluster centers, $c_1^{(1)}, \ldots, c_k^{(1)}$

for $t = 1, \ldots, T$ **do**

1) *Assignment step:* Assign each point to closest center.

$$\mathcal{C}_i^{(t)} = \{x \ : \ \|x - c_i^{(t)}\|_2 \leq \|x - c_\ell^{(t)}\|_2 \ \forall \ell = 1, \ldots, k\}, \quad i = 1, \ldots, k$$

2) *Update step:* Update cluster centers by averaging set of points assigned to each cluster.

$$c_i^{(t+1)} = \frac{1}{|\mathcal{C}_i^{(t)}|} \sum_{x \in \mathcal{C}_i^{(t)}} x, \quad i = 1, \ldots, k$$

end for

The two-step iteration for k-means clustering can be viewed as alternating minimization or expectation maximization (minimization). Lloyd's algorithm is guaranteed to converge to a local minimum which will depend on the initialization of the algorithm. The set of clusters used to initialize the k-means algorithm can be chosen arbitrarily, or the method can be initialized using careful cluster seeding, e.g. *k-means++* [4]. For arbitrarily chosen cluster centers, there is no approximation guarantee for the solution of the algorithm. That is, if ϕ is the objective of the solution returned by Lloyd's iteration, then $\frac{\phi}{\phi^\star}$ can be arbitrarily large, where $\phi^\star$ is the global optimum. In contrast, k-means++ comes with a theoretical guarantee that the solution ϕ produced by the algorithm will be $O(\log k)$-competitive with the optimal solution $\phi^\star$.

10.7.1 Distributed K-means

K-means can be distributed as follows. The cluster centers are broadcast and the assignment step is computed locally. The cluster centers are updated using a `reduceByKey` to average the observations within each cluster.

Algorithm 5 Distributed k-means

Start with an arbitrary* set of k cluster centers, $c_1^{(1)}, \ldots, c_k^{(1)}$
for $t = 1, \ldots, T$ **do**
1) Broadcast current model: $\{c_1, \ldots, c_k\}$ ▷ small: k vectors of length d
2) *Assignment step:* Do an *RDD map* to find closest center to each point; emit key-value pairs (`clusterID`, `point`).
3) *Update step:* `reduceByKey` with `clusterID` as the key to compute average of data points within each cluster.
end for

Although each iteration requires all-to-all communication, k-means typically converges in a small number of iterations.

10.7.2 Streaming k-means

Now we consider the problem of identifying cluster centers from streaming data. To update the model from streaming data, we use weighted averaging. The model, i.e. the k cluster centers $\{c_1^{(t)}, \ldots, c_k^{(t)}\}$, is stored as a state. For each cluster $\mathcal{C}_i$, we keep the current cluster center, $ci^{(t)}$ and the number of points $n_i^{(t)}$ assigned to the cluster through time t. The $n_i^{(t)}$ points that have been assigned to cluster $\mathcal{C}_i$ are not stored and once a point is assigned to a cluster it can not be reassigned in the streaming setting. To update the cluster center $c_i^{(t+1)}$ we take the weighted average of the previous center $c_i^{(t)}$ and the new batch of points from the stream assigned to cluster $\mathcal{C}_i$. The weighted average can also be modified to give higher weight to newer data points than to old data points.

References

[1] B. Recht et al. *Hogwild: A lock-free approach to parallelizing stochastic gradient descent.* Advances in Neural Information Processing Systems, pp. 693–701, 2011.

[2] M. Zinkevich et al. *Parallelized stochastic gradient descent* Advances in Neural Information Processing Systems, pp. 2595–2603, 2010.

[3] S. Lloyd. *Least squares quantization in PCM.* Information Theory, IEEE Transactions on, vol.28, no.2, pp.129,137, Mar 1982.

[4] D. Arthur and S. Vassilvitskii. *k-means++: the advantages of careful seeding.* In Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms (SODA), 2007.