**CME 323: Distributed Algorithms and Optimization, Spring 2017**
`http://stanford.edu/~rezab/dao`.
Instructor: Reza Zadeh, Matroid and Stanford.

**Lecture 8, 4/24/2017. Scribed by Andreas Santucci.**

# 8 Optimization Algorithms

## 8.1 Gradient Descent

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

$$\min_w \quad F(w) = \sum_{i=1}^{n} F_i(w, x_i, y_i)$$

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 we wish to optimize over. The general idea of gradient descent is based on initializing $w$ at random and performing the sequential updates:

$$w_{k+1} \leftarrow w_k - \alpha \nabla f(w_k),$$

where if $\alpha$ small enough, then $F$ is guaranteed to decrease.[1] The objective function F depends on the problem at hand. Some examples include:

- squared error loss function, yields least squares $F_i(w, x_i, y_i) = ||x_i^T w - y_i||_2^2$ (strongly convex)
- a logistic loss leads to logistic regression (convex)
- cross-entropy (e.g: for logistic regression or neural nets) (not convex)
- hinge loss $F_i(w, x_i, y_i) = \max(1 - y_i(x_i^T w))$ with $y_i \in \{-1, +1\}$ (e.g: for SVM).[2] (convex)

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

[1] Want $\alpha < \frac{1}{L}$ where $L$ is the Lipschitz constant for $F$.

[2] This function is non-differentiable but still convex and can have subgradients. It says that we care about the direction in which you get the answer wrong. If $y_i = 1$ and $x_i^T w > 1$, that's a loss of zero. If $x_i^T w < 1$, we incur a penalty proportional to how far off we are.

## 8.2 Analyzing least squares gradient descent

We'll focus our analysis on linear least squares, a strongly convex problem. 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 is given by

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

This has work $O(nd)$ and depth $O(\log n + \log d)$. This object function has gradient

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

Here, we see that the gradient is just a re-scaling of $x_i$. 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 $\nabla_w F$ then requires $O(\log d) + O(\log n)$ depth. Optimization theory informs us that for strongly convex functions, to achieve $\epsilon$ error, gradient descent requires $\mathcal{O}(\log \frac{1}{\epsilon})$ iterations. Thus total depth for gradient descent is $\mathcal{O}(\log \frac{1}{\epsilon} (\log n + \log d))$, 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.

## 8.3 Stochastic Gradient Descent

SGD *only* makes sense for separable objective functions $F = \sum_{i=1}^{n} F_i(w, x_i, y_i)$; it's not even a well formed question to ask whether we can do SGD on a non-separable objective. How does it work?

**General idea** Instead of computing the full gradient, randomly select one training point and apply gradient there. If we sample uniformly at random, we hope that our algorithm will converge (and it does). Let $s_k$ be index uniformly sampled at time step $k$. Then, the SGD iteration as follows:

$$w_{k+1} \leftarrow w_k - \alpha \nabla F_{s_k}(w_k).$$

We can no longer say we'll be done in $\mathcal{O}\left(\log \frac{1}{\epsilon}\right)$ iterations before convergence, because we're only using part of our training data each iterate.

**Theorem 8.1** *SGD achieves $\epsilon$ error after $\mathcal{O}(\frac{1}{\epsilon})$ iterations.*

Notice the $\mathcal{O}(\cdot)$ notation hides constants such as the Lipschitz constant, depending on $F$, which can be pretty large. 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 $\epsilon$ is our tolerance for convergence. (To get this table, we have assumed $F_i$ has a reasonable Lipschitz constant, and that evaluating each $F_i$ requires $O(d)$ work and $\log(d)$ depth.)

| | # 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 | $O(\frac{1}{\epsilon})$ | $O(d)$ | $O(\log d)$ | $O(\frac{d}{\epsilon})$ | $O(\frac{\log d}{\epsilon})$ |

In the case of Gradient Descent, total depth is trapped in log expression; this is powerful. If we want to set $\epsilon = 1e-6$, this becomes problematic in SGD, which requires many more iterations than GD. But what's the tradeoff? With SGD, we have low work; the total work not dependent on $n$. This creates a need to parallelize SGD. On a single machine, it's silly to use Gradient Descent; people typically use Stochastic Gradient Descent in this case.

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].

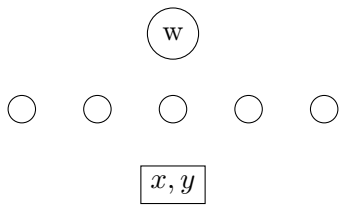
**What are the issues that arise when trying to evaluate SGD iteration?** Suppose we have many cores, and the current model $w$ is on shared memory. The cores all have access to the training data as well. Why can't each core, say, do one sample and then do the update? Without communication between processors, two issues may occur.

1. Model used by core might be obsolete by the time it tries to write the update.

   That is, a core may read the current model, do some computation, and then try to write the model down into shared memory; it's possible by the time it goes back to write it down, the old model may have already been updated by another processor. So, then we overwrite previous work and we have a stale model. No good.

2. The new update may be overwritten. Even if I'm a core and can get my update written, the new update may be overwritten by stale information.

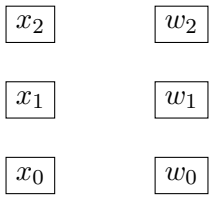
Why can't we just use thread-locking? I.e. we have $w$ sitting somewhere and $p$ processors.




[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

We will rely on the sparsity of the problem to ignore these problems. The reason it will be fine, is that if there is overlap when processors try to write, so long as each processor attempts to write to a *different* location of $w$. This goes with the assumption that each processor only tries to write to a small block of $w$ at any given update.




Each training point $x_i$ in $\mathbb{R}^d$. In a spam filter, for example, our $x_i$ sparse, i.e. each email only contains a small fraction of all words in the English vocabulary. So we can just ignore the dependence between $w_{k+1}$ to $w_k$. There is a giant chunk of memory which stores $w$, and each processor go HOGWILD! with reading and writing to chunks of $w$ concurrently.

### 8.4 Hogwild!

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

```
Initialize w in shared memory // in parallel, do
for i = {1, ..., p} do
    while TRUE do
        if stopping criterion met then
            break
        end
        Sample j from 1, ..., n uniformly at random.
        Compute f_j(w) and ∇f_j(w) using whatever w is currently available.
        Let e_j denote non-zero indices of x_i
        for k ∈ e_j do
            w_(k) ← w_(k) − α [∇F_j(w)]_(k)
        end
    end
end
```

**Algorithm 1:** HOGWILD!

Alternatively it is possible to assign a single processor to each $\nabla F_i$ 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 $\Delta F_i$ that update is applied to the current value of $w$ rather than from the weight vector that was the basis for computing $\Delta F_i$.

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.

**Takeaway lesson** Optimization is about decreasing an objective function. It's the case that we assumed the way to do that is to follow the gradient. In reality, if we have lots of compute power available, we can do other things to try and decrease our objective function value; e.g. we could try a random update, and if it brings the objective down we accept the update.

## References

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