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

Lecture 16, 5/24/2017. Scribed by Andreas Santucci.

Overview We're going to continue with optimization. We'll see how to implement these algorithms on Spark and also how to generalize these algorithms to other platforms.

15.1 Optimization, scaling, and gradient descent in Spark

Separable objective functions If you recall, we have two different optimization algorithms. There's gradient descent, and stochastic gradient descent. In this class, we'll implement gradient descent on Spark, and then we'll see why SGD not suitable for Spark, and then we'll learn how to implement SGD regardless of the data-flow paradigm of the language we're computing in. We're going to do this all on seperable objective functions, of the form

$$\min_w \sum_{i=1}^{n} F_i(w)$$

where $w \in \mathbb{R}^d$ and F_i is often the loss-function applied to a training point i; the loss can be defined with something as fancy as a neural network or as something as simple as least squares. F could be evaluating a linear or non-linear model. We're dealing with a very general framework.

Scaling in various dimensions We're interested in scaling in the following ways.

- Scaling n
- Scaling d

Assumptions of what can fit locally on a single machine In the first part of the lecture, we'll assume n is large but that d numbers can fit on a single machine's RAM. E.g. d is a million, and n is a trillion. Recall what gradient descent looks like. Start with a random vector x_0, say initialized to all zeros. Then $x_{k+1} \leftarrow x_k - \alpha \sum_{i=1}^{n} \nabla F_i(x_k)$, where $\nabla F_i(\cdot)$ is a vector itself.

15.1.1 Implementing Gradient Descent in Spark

Let's now take out our Spark tools and try to implement this algorithm. We have an RDD and its associated operations, where do we start? First off, how are our data stored on the distributed cluster? Look at where we're scaling, and think about how we can shove this data into a cluster. We store one training point on a single machine. The data points are chopped up row-by-row. If we look at our data as if its rows of a matrix,

How are our data partitioned? So, we split up our data row-by-row and store it in an RDD.

```
val points = spark.textFile(...).map(parsePoint).cache()
```

where `parsePoint` is a closure which takes as argument a text-file of training points and their associate labels, and outputs a clean d-dimensional x_i vector and a single label y_i.

A note on transformations and actions The action `cache` tells Spark that we're going to use the RDD of interest a lot, and that each machine should try to hold that its part of the RDD in memory as much as possible rather than flushing data to disk. Since we're going to repeatedly iterate over this RDD in our gradient descent, we wish to store the RDD in memory. Since `cache` is an action, Spark actually kicks off computation and reads in the text file, applying the `parsePoint` closure to each part of the distributed file.[1] Note that the transformation `map(parsePoint)` is a closure which is shipped to a worker, and that worker must be able to perform whatever computations `parsePoint` requires on its input argument; the worker *must* be able to fit the resulting output in memory in order to store it as an entry in the RDD.

```
def parsePoint: textFile("0, 1, 25, 3.14") --> [0, 1, 25] (3.14)

val w = Vector.zeros(d)
```

Gradient descent implementation Now, let's write our code for gradient descent.

```
val points = spark.textFile(...).map(parsePoint).cache()    var w = Vector.zeros(d)
For (i <- 1 to numIterations):
  val gradient = ...
  w -= alpha * gradient
End
```

Think about how to fill in our `gradient` object. We have to perform a summation across machines, where on each machine we compute the gradient $\nabla F_i(\cdot)$. So, first, we compute the individuals gradients.

$$\texttt{points.map(p => } \nabla F_p(w)\texttt{)}$$

where obviously $\nabla F_p(w)$ is the gradient of F_p evaluated at point w. E.g. if $F_i(w) = (w^T x_i - y_i)^2$, then ∇F_i may be calculated symbolically, and we may plug this formula into $\nabla F_p(\cdot)$. Now, we need to compute the summation, for this we use a reduce. Realize that so far no computation has kicked off. When we use a reduce (action) we kick off computation.

$$\texttt{points.map(p => } \nabla F_p(w)\texttt{).reduce(\_+\_)}$$

[1] Suppose we have a 1 terabyte dataset, and we have a small amount of memory, say 10 gigabytes. Suppose we wish to process the large dataset. We can read directly off the large terabyte slowly, or we can read directly off the ram for faster access. The question becomes which part of the data set to store in ram since we can't fit the entire object in memory. We commonly employ Least Recently Used caching, which evicts items from ram according to which piece of data was least recently used. This is a simple heuristic, but works in practice.

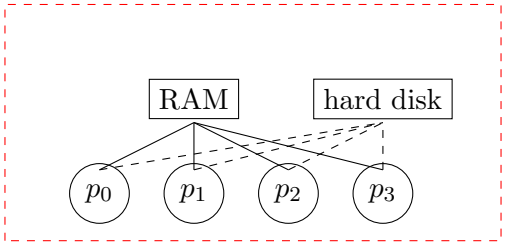
This last reduce is non-trivial, but we know how to implement it. It considers communication cost between machines, scheduling, and executing operations locally in a PRAM model. The map and reduce are all performed in parallel. In all,

```
val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.zeros(d)
For (i <- 1 to numIterations):
  val gradient = points.map(p => \nabla F_p(w)) .reduce(_+_)
  w -= alpha * gradient
End
```

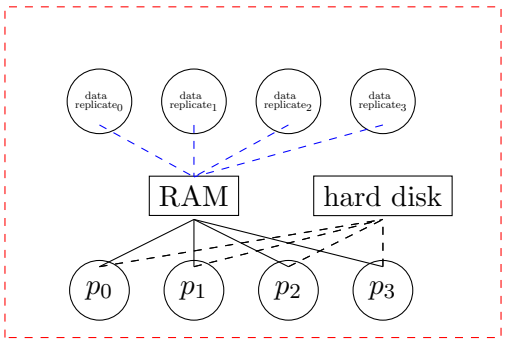
Take a moment to appreciate that we understand each piece of the above code, and all the inner workings of each action and transformation. Gradient descent is very powerful, and can go a long way.

15.1.2 Broadcasting in Spark

There's something particular about the way Spark broadcasts. Suppose we have a machine with four different CPU's on it.



By default, when a `map` happens, anything that is needed for the `map` to happen is shipped out to the workers. So, in our code above w must be shipped to the mappers. Notice that there is a mapper per CPU. So in reality, if each node in our cluster has 4 CPU's, we'll actually send 4 copies of w to each machine.



Why do we do this? By default, we assume w is being modified, in which case we must store w separately since each CPU modifies some part of it, and otherwise we run into concurrent write

issues. Also notice that if w large and we are *not* modifying it (which is true in gradient descent) that we are wasting storage. So, to get around this we can broadcast `w` so that it only gets sent to each machine a single time. To do this, we need to first define `w` as a broadcast variable.

```
var w_br = sc.broadcast(w)
```

and whenever we use this broadcast object, we must refer to its value.

$$\texttt{points.map(p => } \nabla F_p(\texttt{w\_br.value}).\texttt{reduce(\_+\_)})$$

And finally we need to re-broadcast our w after it gets updated. So, the code now looks like

```
val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.zeros(d)
var w_br = sc.broadcast(w)
For (i <- 1 to numIterations):
  val gradient = points.map(p => \nabla F_p(w_br.value)) .reduce(_+_)
  w -= alpha * gradient
  w_br = sc.broadcast(w)
End
```

What do we gain from this broadcast? Aside from being more storage efficient on each machine, i.e. storing a single copy of the data object instead of multiple, we also take advantage of bit-torrent broadcasting when we use Spark's broadcasting. So, broadcast really only makes sense for data objects that are large enough we don't want to store multiple times needlessly, but small enough that we can store the entire object in ram.

15.2 Analysis

The work is straightforward to analyze: we just add up the cost of all matrix operations we're applying. But what about depth? Where is our main bottleneck? The `for` loop is a giant synchronization barrier where we have to wait for all mappers to finish computation before continuing on to the next iteration of our loop. If we have to do too many iterations, we have to wait for a very long time. Thankfully, we can take advantage of our cluster of machines to distribute the work of calculating the gradient and updating w across machines. The outer loop is what is not parallelized.

So, `numIterations` doesn't have to be very large for convex functions. For convex functions, gradient descent converges quickly. But for non-convex problems such as neural nets, we need many iterations.

Communication costs The map is embarrassingly parallel, and requires no communication. The broadcast is a one to all communication. The reduce is an all-to-one communication. So, we

have a one-to-all, an embarassingly parallel, and then an all-to-one. So as far as communication goes, we're doing alright because we're avoiding all-to-all communication.[2]

15.2.1 SGD

What about SGD? In stochastic gradient descent, we sample a training point from $i = 1, \ldots, n$ to get a point-estimate of the gradient $\nabla F_i(w)$, and then we update $x_{k+1} - = \alpha \nabla F_i(x_k)$. So instead of using the full summation, we only look at a single point. Now, realize that with SGD we need even more iterations. So the depth gets even worse (although we do decrease work). The solution was to do HogWild! I.e. don't wait between iterations for updates to be written to w.

15.3 Recapping differences between (Stochastic) Gradient Descent

The difference between SGD and GD: gradient descent is more expensive in work (T_1) but requires fewer iterations. On the other hand SGD requires less work per iteration, but instead requires more iterations. All of this is on a PRAM model. To get around synchronization barrier, we use HogWild! We've now written down how to do gradient descent with Spark. We have yet to distribute SGD. How on earth are we going to implement HogWild! on Spark? Spark has all these synchronization barriers: we have to wait for the broadcast to finish, then the map and reduce to finish, then another broadcast. We can't tell Spark "not to wait". The fate of Spark computations is that they must be fault-tolerant, and to this end we require synchronization barriers. So, SGD is not well suited for Spark, and neither is HogWild! Spark is not the right tool.

What is the right tool for SGD? Well, tools such as Tensor Flow allow us to set up clusters on our own that have fault tolerance but that don't have synchronization barriers. However, we have to program ourselves a parameter server that holds w. So, each worker asks the parameter server for a current copy of w. Then, the machine samples its own data, computes a gradient, and sends the resulting update back to the driver. If we wish to stay true to SGD, we would have to wait for each worker to finish their updates in sequence. However, we're going to go HogWild! and not wait. Using this setup, if a machine goes down, it's OK. We won't get updates from that part of the training data for some time, but we can detect that the machine goes down and restart computation. With the parameter server, we need to checkpoint our results once in a while to ensure that if it dies, we don't have to re-do too much work. Optimization lends itself well to warm-starts.

[2]Technically, when we call `points.map()`, we serialize code and send it to each machine. One reason we chose Scala to implement Spark is that it's easy to serialize code. We remark that there is a function called `allReduce` which merges the concept of a reduce and broadcast. After a reduce, the result of the reduce ends up on the driver machine. After an `allReduce` the result of the `reduce` gets sent to all machines via a broadcast; the broadcast happens simultaneously with the regular reduce.

References

[1] F. McSherry, M. Isard and D. Murray. *Scalability! but at what COST?*. 15th Workshop on Hot Topics in Operating System (HotOS XV).