NetID : iv447
University ID: N17385760

Stochastic Gradient Descent :

During last decades, data size have grown faster than the processing speed of the cpu's. Because of this problem optimizing algorithms such as Stochastic Gradient Descent are used. SGD shows amazing performance while running with very huge data sets.

Gradient Descent : Using the gradient descent algorithm the weights are updated incrementally for each epoch i.e each iteration of training data set. The magnitude and direction of the weight update is computed by taking a step in the opposite direction of the cost gradient. The weights can be updated after each epoch by the following rule :

W = W + delta(W), where delta(W) is a vector that contains weight updates of each weight coefficient W. Finally we can picture gradient descent as a hiker (weight coefficient) who wants to climb down the mountain (cost function) in to a valley (cost minimum), and each step is determined by the steepness of the slope (gradient) and the leg length of the hiker (learning rate).

Stochastic Gradient Descent : In GD optimization, we compute the cost gradient based on the complete training set, we therefore can call it as batch GD. In case very huge dataset using GD can be quite costly since we are taking single step for one pass over the training set. Thus larger the training set, slower will be the algorithm. The algorithm has to update the weights and the longer it may take take until it converges to the global cost minimum. In Stochastic Gradient Descent (sometime also known as SGD , sometimes also referred as iterative or on-line GD), we don't accumulate the weight updates as in case of gradient descent. Instead we update the weight after each training sample. The term “stochastic” comes from fact that the gradient based on a single training sample is a “stochastic approximation” of the “true” cost gradient. Due to it's stochastic nature, the path towards the global cost minimum is not “direct” as in GD, but may go zig-zag if we are visualizing the cost surface in a 2D space. However, the SGD always almost surely converges to the global global cost minimum if the cost function is convex.

How can we leverage SGD for huge data calculation :

1. In many big data setting (say several million data points), calculating cost or gradient takes very long time, because we need to sum over all data points.

2. We do NOT need to have exact gradient to reduce the cost in a given iteration. Some approximation of gradient would work OK.

3. Stochastic gradient descent (SGD) approximate the gradient using only one data point. So, evaluating gradient saves a lot of time compared to summing over all data.

4. With "reasonable" number of iterations (this number could be couple of thousands, and much less than the number of data points, which may be millions), stochastic gradient descent may get a reasonable good solution.

Let us assume regression on squared loss, the cost function can be :

J(θ) = 1 / 2m * (summation (i = 1 to m)[(hθ(x (i)) − y (i)) ** 2)]

And the gradient descent is :

∂ J(θ) /∂(θ) = 1/m * (summation (i = 1 to m)[(hθ(x (i)) − y (i)) ** x(i)]

For stochastic gradient descent we get rid of the sum and 1/m constant, but get the gradient for current data point $x_{(i)}$,$y_{(i)}$x(i),y(i), where comes time saving. And the parameter as updated by :

θ(new) := θ(old) - α *[(hθ(x (i)) − y (i)) ** x(i)]

Suppose we have 1 billion data points.

In GD, in order to update the parameters once, we need to have the (exact) gradient. This requires to sum up these 1 billion data points to perform 1 update.

In SGD, we can think of it as trying to get **an approximated gradient instead of exact gradient**. The approximation is coming from one data point (or several data points called mini batch). Therefore, in SGD, we can update the parameters very quickly. In addition, if we "loop" over all data (called one epoch), we actually have 1 billion updates.

The trick is that, in SGD you do not need to have 1 billion iterations/updates, but much less iterations/updates, say 1 million, and you will have "good enough" model to use.

How can we implement SGD in spark :

We have an RDD and its associated operations. First off, we need to find out how data is stored on the distributed cluster. Then we have to 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,

```
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 xi vector and a single label yi .

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 :

The 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
```

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

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

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. points.map(p => $\nabla F_p(w)$).reduce(+)

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
```