Machine Learning - Home | Coursera

We mainly benefit from a very large dataset when our algorithm has high variance when m is small. Recall that if our algorithm has high bias, more data will not have any benefit.

Datasets can often approach such sizes as m = 100,000,000. In this case, our gradient descent step will have to make a summation over all one hundred million examples. We will want to try to avoid this -- the approaches for doing so are described below.

Stochastic gradient descent is an alternative to classic (or batch) gradient descent and is more efficient and scalable to large data sets.

Stochastic gradient descent is written out in a different but similar way:

$$cost\left(\theta,\left(x^{(i)},y^{(i)}\right)\right)=\frac{1}{2}\left(h_\theta\left(x^{(i)}\right)-y^{(i)}\right)^2$$

The only difference in the above cost function is the elimination of the m constant within $\frac{1}{2}$.

$$J_{train}\left(\theta\right)=\frac{1}{m}\sum_{i\ =\ 1}^{m}cost\left(\theta,\left(x^{(i)},y^{(i)}\right)\right)$$

J_{train} is now just the average of the cost applied to all of our training examples.

The algorithm is as follows

1. Randomly 'shuffle' the dataset
2. For $i=1...m$

$$\Theta_j:=\Theta_j-\alpha(h_\Theta(x^{(i)})-y^{(i)})\cdot x_j^{(i)}$$

This algorithm will only try to fit one training example at a time. This way we can make progress in gradient descent without having to scan all m training examples first. Stochastic gradient descent will be unlikely to converge at the global minimum and will instead wander around it randomly, but usually yields a result that is close enough. Stochastic gradient descent will usually take 1-10 passes through your data set to get near the global minimum.

Mini-Batch Gradient Descent

Mini-batch gradient descent can sometimes be even faster than stochastic gradient descent. Instead of using all m examples as in batch gradient descent, and instead of using only 1 example as in stochastic gradient descent, we will use some in-between number of examples b.

Typical values for b range from 2-100 or so.

For example, with b=10 and m=1000:

Repeat:

For $i = 1, 11, 21, 31, \ldots, 991$

$$\theta_j := \theta_j - \alpha \frac{1}{10} \sum_{k=i}^{i+9} \left(h_\theta\left(x^{(k)}\right) - y^{(k)} \right) x_j^{(k)}$$

We're simply summing over ten examples at a time. The advantage of computing more than one example at a time is that we can use vectorized implementations over the b examples.

How do we choose the learning rate α for stochastic gradient descent? Also, how do we debug stochastic gradient descent to make sure it is getting as close as possible to the global optimum?

One strategy is to plot the average cost of the hypothesis applied to every 1000 or so training examples. We can compute and save these costs during the gradient descent iterations.

With a smaller learning rate, it is **possible** that you may get a slightly better solution with stochastic gradient descent. That is because stochastic gradient descent will oscillate and jump around the global minimum, and it will make smaller random jumps with a smaller learning rate.

If you increase the number of examples you average over to plot the performance of your algorithm, the plot's line will become smoother.

With a very small number of examples for the average, the line will be too noisy and it will be difficult to find the trend.

One strategy for trying to actually converge at the global minimum is to **slowly decrease α over time**. For example $\alpha = \frac{const1}{iterationNumber + const2}$

However, this is not often done because people don't want to have to fiddle with even more parameters.

With a continuous stream of users to a website, we can run an endless loop that gets (x,y), where we collect some user actions for the features in x to predict some behavior y.

You can update θ for each individual (x,y) pair as you collect them. This way, you can adapt to new pools of users, since you are continuously updating theta.

We can divide up batch gradient descent and dispatch the cost function for a subset of the data to many different machines so that we can train our algorithm in parallel.

You can split your training set into z subsets corresponding to the number of machines you have. On each of those machines calculate $\sum_{i=p}^{q} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) \cdot x_j^{(i)}$, where we've split the data starting at p and ending at q.

MapReduce will take all these dispatched (or 'mapped') jobs and 'reduce' them by calculating:

$$\Theta_j := \Theta_j - \alpha \frac{1}{z}(temp_j^{(1)} + temp_j^{(2)} + \cdots + temp_j^{(z)})$$

For all $j = 0, \ldots, n$.

This is simply taking the computed cost from all the machines, calculating their average, multiplying by the learning rate, and updating theta.

Your learning algorithm is MapReduceable if it can be *expressed as computing sums of functions over the training set*. Linear regression and logistic regression are easily parallelizable.

For neural networks, you can compute forward propagation and back propagation on subsets of your data on many machines. Those machines can report their derivatives back to a 'master' server that will combine them.