Stanford Machine Learning - Week X

Eric N Johnson

September 13, 2016

Part I
Large-Scale Machine Learning

This set of lectures will deal with algorithms for large datasets. Why large datasets?

1. Having a lot of data to train with makes algorithms work better.
2. There is simply a lot of data available...

But the sheer magnitude of available data also presents some challenges.

"It's not who has the best algorithm that wins. It's who has the most data."

1 Gradient Descent with Large Datasets

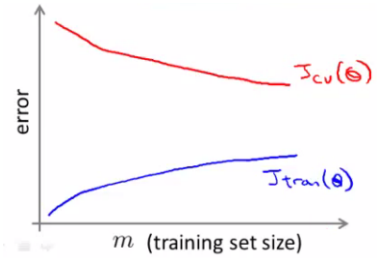
1.1 Learning with Large Datasets

If you have a size $m = 100,000,000$ training set, for instance, some obvious computational problems arise. A dataset this size is not unusual for large scale ML. Census data, internet traffic data, etc. are common ML domains with massive amounts of data. If we were to train a linear regression model using Gradient Descent, for example,

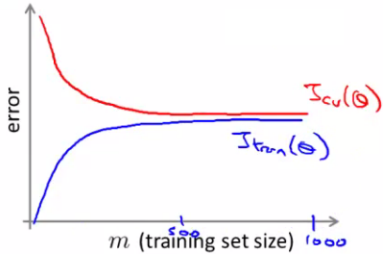
$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)}$$

we have to carry out a sum over a hundred million terms...

Can we train our algorithm on a very small subset of these? To test we could plot a learning curve for a range of values of m and verify that the algorithm has high variance when m is small. This is a worthwhile sanity check.



Recall that a high variance case will look like the above plot. If those training curves were much closer together, it may not be worth the effort to use many many more training examples.



1.2 Stochastic Gradient Descent

Recall that in most of the learning algorithms we have worked on so far the main mode is to determine a cost function then minimize that cost to parameterize a model. But with a huge number of training examples, performing that step may be cost-prohibitive. A stochastic gradient descent algorithm addresses this. Time for a quick review:

Linear Regression with Batch Gradient Descent (Review)

- Hypothesis:

$$h_\theta(x) = \sum_{j=0}^{n} \theta_j x_j$$

- Cost Function:

$$J_{\text{train}}(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

- Algorithm:

Repeat {

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)}$$
$$\forall j = 0, 1, ..., n$$

}

Stochastic Gradient Descent

This is a modified Gradient Descent algorithm.

- Hypothesis:

$$h_\theta(x) = \sum_{j=0}^{n} \theta_j x_j$$

- Cost Function:

$$\text{Cost}(\theta, (x^{(i)}, y^{(i)}) = \frac{1}{2} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$
$$J_{\text{train}}(\theta) = \frac{1}{m} \sum_{i=1}^{m} \text{Cost}(\theta, (x^{(i)}, y^{(i)})$$

  Which is just the mean of the training costs.

- Algorithm:

1. Randomly shuffle dataset
2. Repeat {

for $i = 1, 2, .., m$ {

$\theta_j := \theta_j - \alpha(h_\theta(x^{(i)} - y^{(i)}) x_j^{(i)}$

$\forall j = 1, 2, ..., n$

}

}

There are a few differences between the Stochastic and Batch versions. First, the randomization allows the algorithm to take very small steps by introducing only one training example at a time rather than waiting to run through the entire sets worth of examples and taking the sum which is much faster. Also the random shuffling tends to help the algorithm converge more rapidly. Second, there is no summation which is computationally less expensive.

Stochastic Gradient Descent meanders more than Batch Gradient Descent. Each step of the algorithm may or may not move closer to the global maximum. In other words, the algorithm doesn't converge in the same way that Batch Gradient Descent does. But in practice it tends to work pretty well. That outer loop can be calculated as few as 1 time.

1.3 Mini-Batch Gradient Descent

We have discussed:

- Batch Gradient Descent: We use all m examples in each iteration.
- Stochastic Gradient Descent: Use 1 example in each iteration.

  And now for...

- Mini-batch Gradient Descent: Use b examples in each iteration. b is a 'mini-batch size' parameter. We may pick b to be somewhere between 2 and 100 and pick b training examples randomly.

For $b = 10$, we pick 10 examples

$$(x^{(i)}, y^{(i)}), ..., (x^{(i+a)}, y^{(i+a)})$$

and perform a Gradient Descent step for each:

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

In other words:

Say $b = 10$, $m = 1000$
Repeat {

For $i = 1, 11, 21, 31, ..., 991${

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

$\forall j = 0, 1, ..., n$

}

}

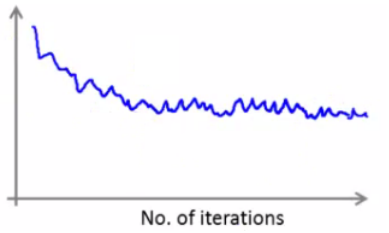
I would rather notate the sums with some index set $i \in \mathscr{I}$, but it really doesn't matter. There are some advantages to performing Mini-Batch Gradient Descent as well as some disadvantages. With Mini-Batch Gradient Descent we can take advantage of all of the linear algebra libraries out there for performing fast vectorized optimization. However, there is also this extra parameter b which we may have to play with.

1.4 Stochastic Gradient Descent Convergence

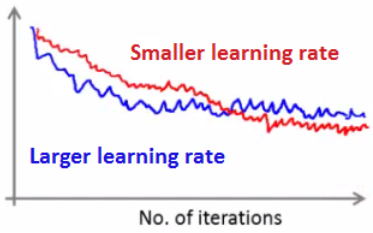
How do we know that we have a good stopping point - and how to we pick α?

1. Take $\text{Cost}(\theta, (x^{(i)}, y^{(i)}) = \frac{1}{2} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$ and while learning, compute $\text{Cost}(\theta, (x^{(i)}, y^{(i)})$ before updating θ using $(x^{(i)}, y^{(i)})$.
2. Pick some interval (every 1000 iterations, say) and plot $\text{Cost}(\theta, (x^{(i)}, y^{(i)})$ averaged over the last 1000 examples processed by the algorithm.

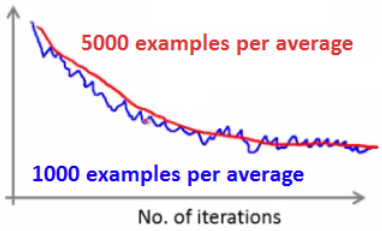
Here are some useful notes of what the cost plots may look like:



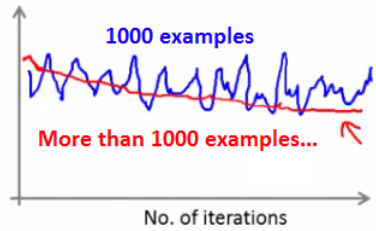
Remember that the cost won't decrease monotonically. There will be a little noise.



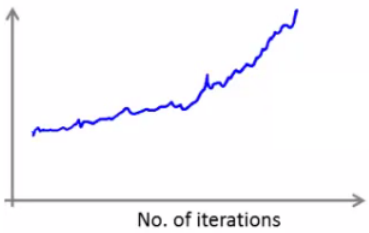
A smaller learning rate will have smaller oscillations.



How often we plot the cost will effect how smooth the curve is. If the number of iterations is large the curve will be more smooth - but we will get less points.



If we plot over too few examples, the plots will be very noisy and it may not see how the cost is changing.



If the cost is consistently starting to increase, we should decrease the learning rate α. We may also decide to decrease the learning rate as we iterate. We can set α to decrease:

$$\alpha = \frac{c_1}{N(\text{iteration} + c_2}$$

for some constants c_1 and $c_2 \in \mathbb{R}$. But again, doing this adds more parameters to our algorithm that has to be tweaked.

Part II

2 Advanced Topics

2.1 Online Learning

Online Learning allows us to continuously learn from data as it comes in. Several large website companies do this all of the time to learn from users that are going to their site and thus optimize their site. Some examples:

- Shipping websites where user specifies origin and destination. Companies offer to ship this package for some fixed price. Sometimes users choose your service ($y = 1$), and sometimes not ($y = 0$). Our features x capture properties of user, the shipping origin and destination, as well as asking price. We then learn $p(y = 1|x; \theta)$ to optimize price, using, for example, logistic regression.

  Repeat {

  Get (x, y) corresponding to user
  Update θ using (x, y) :
  $\theta_j := \theta_j - \alpha(h_\theta(x) - y)x_j$
  $\forall j = 0, 1, ..., n$

  }

  Instead of having a fixed training set, we just take each example one at a time and optimize our price on that. This allows us to adapt to user preferences. They may become more or less price sensitive (price elasticity), more or less willing to pay for addon features... All of this data is free.

- "Predicted Click-Through Rate" (CTR): We can learn to help users search. A user searches for a query "Android 1080p". We wish to return 10 of the best results first. Our feature x captures features for the phone, words in the query match, the name of the phone, how many words in query match the description, etc. We learn $p(y = 1|x; \theta)$ given that $y = 1$ corresponds to the user clicking through to the phone. Use to show user the 10 best phones. Hopefully sell some of these.
- Decide which offer to show users...
- Decide which news article to show...
- Perform product recommendation in real time...

2.2 Map Reduce and Data Parallelism

Map Reduce allows us to run problems on multiple machines in parallel over a network. Here's a Map Reduce algorithm for Batch Gradient Descent. For these let $m = 400,000,000$. Suppose we have 4 machines.

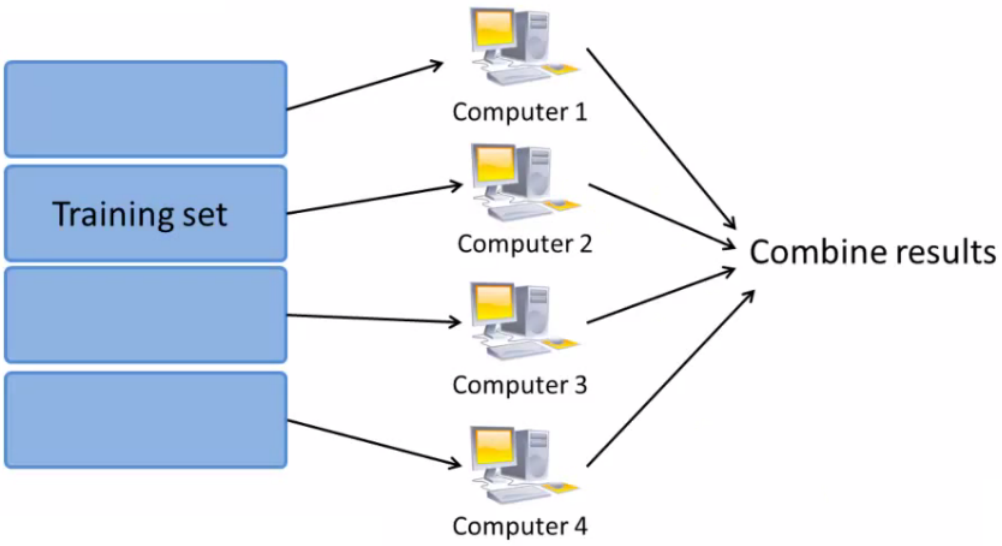
- We use Batch Gradient Descent:

$$\theta_j := \theta_j - \alpha \frac{1}{400} \sum_{i=1}^{4} 00(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$

1. Split data
   - Machine I: Perform work for $\{(x^{(1)}, y^{(1)}), ..., (x^{(100)}, y^{(100)})\}$ Send temp_1.
   - Machine II: Perform work for $\{(x^{(101)}, y^{(101)}), ..., (x^{(200)}, y^{(200)})\}$ Send temp_2.
   - Machine III: Perform work for $\{(x^{(201)}, y^{(201)}), ..., (x^{(300)}, y^{(300)})\}$ Send temp_3.
   - Machine IV: Perform work for $\{(x^{(301)}, y^{(301)}), ..., (x^{(400)}, y^{(400)})\}$ Send temp_4.
2. Receive and combine results:

$$\theta_j := \theta_j - \alpha \frac{1}{400} (\text{temp}_1 + \text{temp}_2 + \text{temp}_3 + \text{temp}_4)$$

Here's a picture:



Not all problems can be split up in this manner. Yadda yadda.