ML

Andrew Ng

Coursera

Week 10

→ large data sets have computational problems

Date:

Learning with large Data Sets

ML & data:

To get high performance ML algo:
Take low bias algo & train on lots of data

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

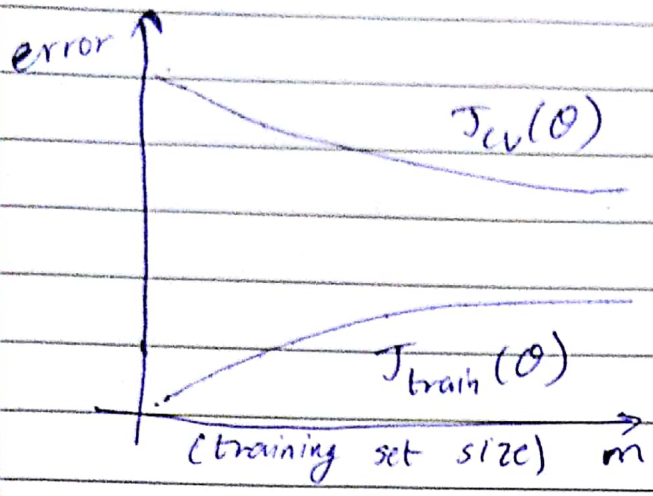
Suppose $m = 100,000,000$ & you are applying linear regression via ~~the~~ gradient descent:

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

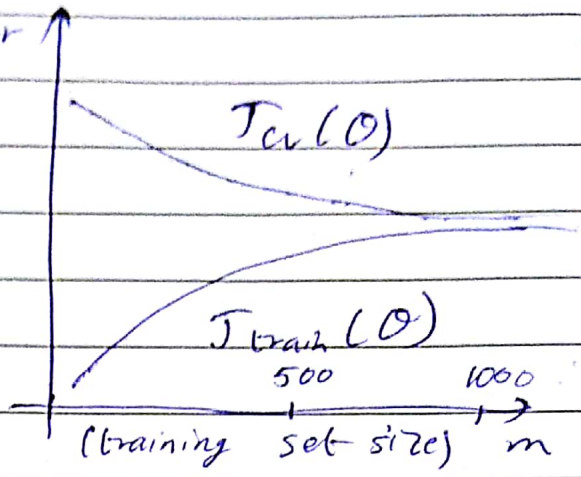
↳ to take 1 step of gradient descent you need to sum over 100,000,000

Why not just randomly pick 1,000 examples from data set & use those for ~~linear~~ training linear regression?

Date:



High variance issue



High bias issue

If the problem is high variance then stick to $m = 100,000,000$ as the use of more examples will decrease $J_{cv}(\theta)$

If the problem is high bias then it is possible to keep $m = 1000$ as there is no point in using more data.

Naturally if we have a scenario that resembles the situation to the right we will try to change the scenario as such to make it look like the situation on the left (high variance)

When dealing with large data sets we need to find computationally efficient ways of training ML algos

Victory

Page No.

modification to gradient descent that scales to bigger training sets

applies to NN, logistic regression & other functions just as gradient descent

Date:

Stochastic Gradient Descent

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

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

Repeat {

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

(for every $j = 0, \ldots, n$)

}

Normal gradient descent is also called batch gradient descent because all training examples are considered in a gradient descent step

Suppose $m = 300,000,000$ then you can't bring all records into memory but instead you will have to read from disk

↳ slows down process even further

Stochastic Gradient Descent

Date:

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

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

Random shuffling ensures that each record scanned is visited in randomly sorted order
↳ speeds up stochastic gradient descent by a bit

sees how well hypothesis is doing on single example $x^{(i)}, y^{(i)}$

Sequence: ✓

1) Randomly shuffle dataset

2) Repeat {

external loop causes program to run over entire training set multiple times

for $i = 1, \ldots, m$ {

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

(for $j = 0, \ldots, n$)

}

}

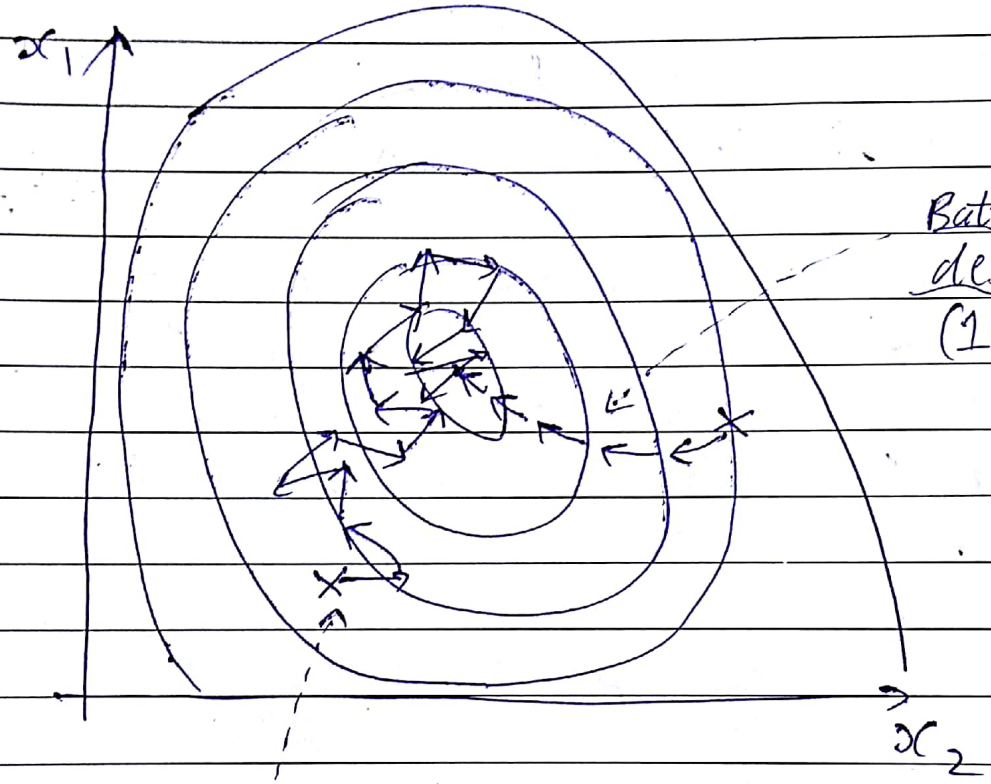
$$\frac{\partial}{\partial \theta_j} cost(\theta, (x_j^{(i)}, y^{(i)}))$$

Stochastic gradient descent will look at the first example only & take a step based on only that first example. Then modify $\theta_1, \ldots, \theta_n$ based on only 2nd example & etc until it goes through whole data set

Victory

Date:

Stochastic gradient descent - instead of waiting for all values to be summed in data set we start improving the parameters early after iterating over a single data point



Batch gradient descent steps (1 per training data set iteration)

Stochastic gradient descent steps

Stochastic gradient descent will generally move towards global minimum but not always, move specifically in that direction at a point in time. You can get unlucky based on the randomly chosen record that is used to retrieve cost.

Date:

Stochastic gradient descent does not converge as is the case with batch gradient descent, instead it wanders around in ~~the~~ some region that is close to global minimum - does not get to global minimum & stay there

↳ gives θ values close to global minimum & that is enough

In stochastic gradient descent, it depends on ~~how~~ m, how many times to go through the outer loop i.e. how many times to go through entire training set.

m ↑ then less outer loop iteration

1-10 passes is fairly common

Stochastic gradient descent is faster because outer loop is repeated less times as parameter update happens m times when going through entire training set while batch gradient descent updates only once

Date:

Mini-batch gradient Descent

↳ can sometimes work even faster than stochastic gradient descent

Batch gradient descent: use all m examples in each iteration

Stochastic gradient descent: use 1 example in each iteration

Mini-batch gradient descent: Use b examples in each iteration

↳ mini-batch is in between batch & stochastic. Batch ~~use~~ (b) usually ranges from 2-100

$b = 10$

Get $b = 10$ examples $(x^{(i)}, y^{(i)}), \ldots, (x^{(i+9)}, y^{(i+9)})$

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

$i := i + 10$

→ update parameters after b examples

Victory

Page No.

Date:

Mini-batch gradient descent

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

Repeat {

for $i = 1, 11, 21, 31, \ldots, 991$ {

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

(for every $j = 0, \ldots, n$)

}

}

Minibatch can outperform stochastic gradient descent only if implementation of minibatch is well vectorized

↳ the sum over b examples is calculated in a vectorised manner to parallelize gradient computations

Stochastic Gradient Descent

Convergence

↳ how do you know that stochastic gradient descent is completely debugged & is converging?

↳ how do you tune α in stochastic gradient descent

Checking for convergence

Batch gradient descent:

- Plot $J_{train}(\theta)$ as a function of # of iterations of gradient descent
- $J_{train}(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$

Suppose $m = 300,000,000$

When updating parameters while stochastic grad. descent you don't want to pause periodically to calculate cost because calculating cost requires summation over all training examples hence becomes costly when m is large

Stochastic gradient descent:

- $cost(\theta, (x^{(i)}, y^{(i)})) = \frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2$
- during learning, compute ~~to~~ cost $(\theta, (x^{(i)}, y^{(i)}))$ before updating θ using $(x^{(i)}, y^{(i)})$
- every 1000 iterations, plot $cost(\theta, (x^{(i)}, y^{(i)}))$ averaged over the last 1000 examples processed by algo

cost computed before θ update so that cost is computed for data set examples that it has not seen yet

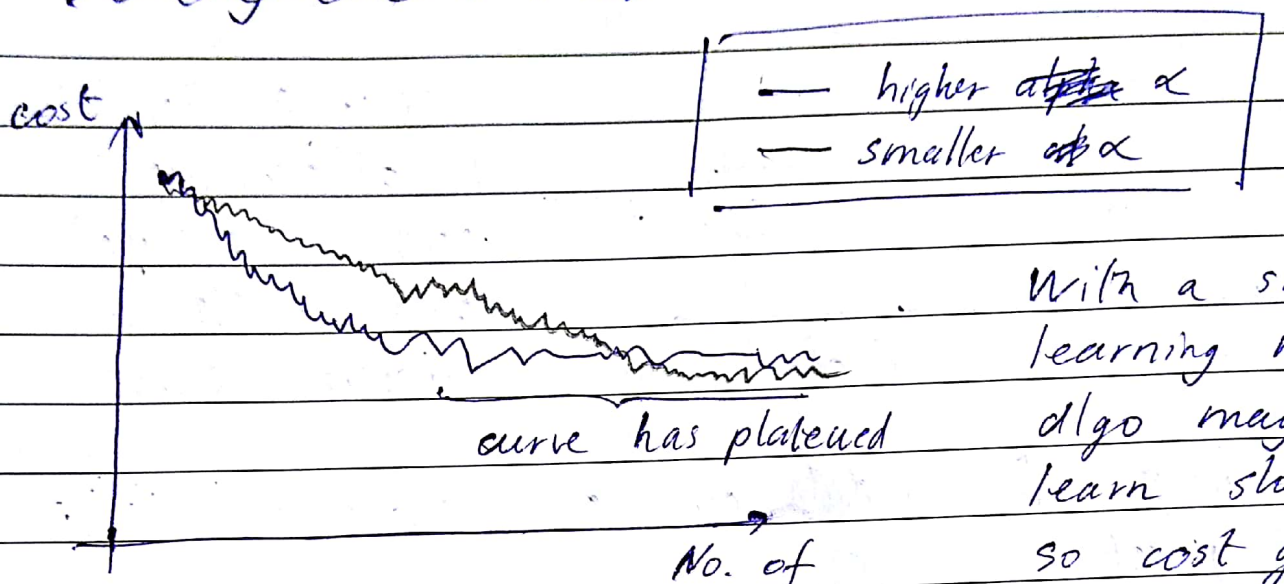
estimates how well algo is doing

you don't need to iterate over entire data set now

→ noise is introduced, will not necessarily decrease on every iteration

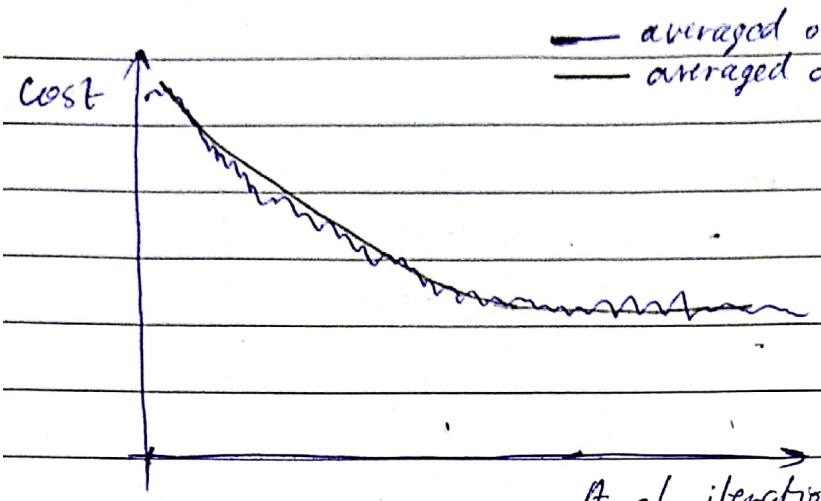
Date:

Checking for convergence



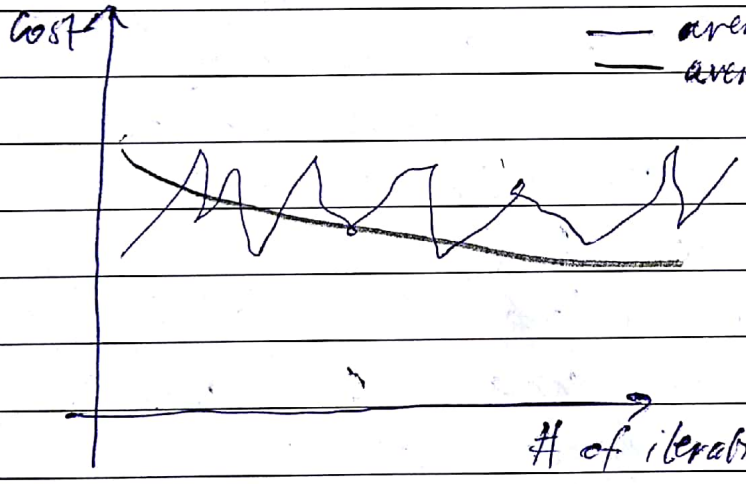
With a smaller learning rate, algo may learn slower so cost goes down slowly but learning rate may end up at better solution because stochastic gradient descent does not converge but rather oscillates in a region close to global minimum so when α is small the area of oscillation is smaller hence cost is lower.

Date:

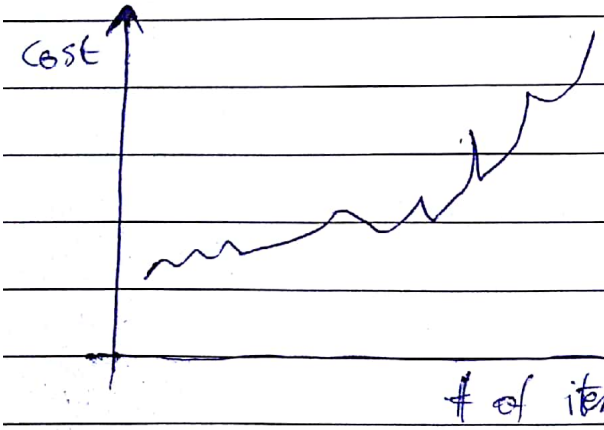


Increasing the # of examples cost is averaged over creates a smoother curve

Disadvantage - fewer data points over entire training set



The noise with small value averages may prohibit us from seeing the trend



Shows that algorithm is diveraging - use smaller value of α

→ Stochastic gradient descent does not converge but rather wanders around the global minimum, we can make it converge by decreasing α & get a better value

Date:

Learning rate α is typically held constant. Can slowly decrease α over time if we want θ to converge $\boxed{\alpha \to 0}$

↳ e.g $\alpha = \frac{\text{constant 1}}{\text{iterationNumber} + \text{constant 2}}$

↳ The reason why people don't do this sometimes is because then you have additional parameters to the algo that you need to tinker with

Online Learning

↳ large-scale ML setting that allows to model problems where algorithm must learn from a continuous stream of data

In online ~~Learning~~ learning, we discard the notion of there being a fixed training set. You get an example, learn using that example & then discard it.

↳ this is only practically implemented when you have a high-width, high velocity data stream available

Date:

Scenario

Shipping service website where user comes, specifies ~~a~~ origin & destination, you offer to ship their package for some asking price & users sometimes choose to use your shipping service ($y=1$), sometimes not ($y=0$) based on the price provided.

Features x capture properties of user, of origin/destination & asking price. We want to learn $p(y=1 \mid x; \theta)$ to optimize price.

Use logistic regression.

Repeat forever {
  Get (x, y) corresponding to user
  Update θ using (x, y):
    $\theta_j := \theta_j - \alpha (h_\theta(x) - y) x_j$
    $(j = 0, \ldots, n)$
}

Online learning platforms can adapt to changing user preferences

↳ θ will reflect the change in preference

Date:

Online learning is very similar to stochastic gradient descent, only instead of scanning a fixed training set, we're getting one example from a user

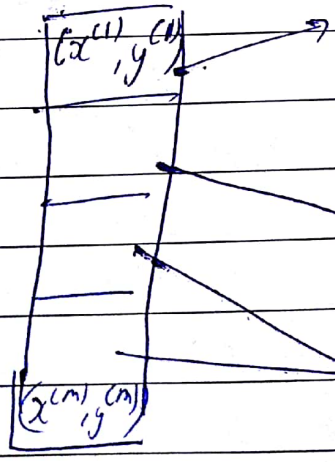
Map-reduce & Data Parallelism

↳ run ML algorithms on multiple machines

Batch gradient descent: ($m = 400$)

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

↳ if m is large then this becomes computationally expensive



Machine 1: Use $(x^{(1)}, y^{(1)}), \ldots, (x^{(100)}, y^{(100)})$

$$temp_j^{(1)} = \sum_{i=1}^{100} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

Machine 2: Use $(x^{(101)}, y^{(101)}), \ldots, (x^{(200)}, y^{(200)})$

$$temp_j^{(2)} = \sum_{i=101}^{200} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

Machine 3: $temp_j^{(3)}$

Machine 4: $temp_j^{(4)}$

Split training set into subsets

Date:

The architecture used works 4 times as fast now considering no network latencies

Master Server will combine all the $temp_j^{(i)}$ values :

$$\theta_j := \theta_j - \alpha \frac{1}{400} \left(temp_j^{(1)} + temp_j^{(2)} + temp_j^{(3)} + temp_j^{(4)} \right)$$

$$j = 0, \ldots, n$$

Map-reduce & summation over the training set

If you wish to apply Map reduce to some learning algo, you must find out if learning algo can be expressed as summation over training set

↳ if yes then Map reduce can be applied to scale learning algo

Many learning algos can be expressed as computing sums of functions over the training set

Date:

E.g:

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

↳ The training set can be divided amongst PCs & computation executed then recombined on central server

Uptil now map reduce has been discussed as a way of parallelizing over multiple computers in a computer cluster but it is also applicable on CPUs with multiple cores

↳ each cores works on a share of the training set

↳ problem of network latency goes away when working with cores

At times some really good linear algebra libraries automatically parallelize the work over multiple cores so all you need to do is implement the vectorized code & the libraries will automatically parallelize it for you