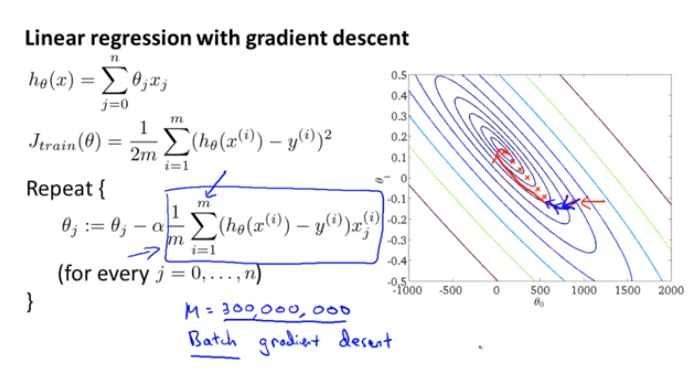
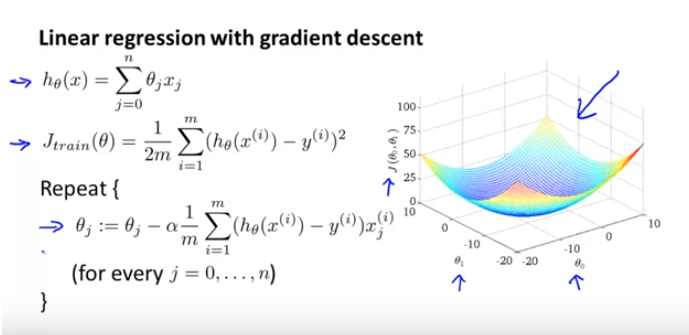
**Stochastic gradient descent** (often shortened to **SGD**)

Also known as **incremental** gradient descent, is an [iterative method](https://en.wikipedia.org/wiki/Iterative_method) for [optimizing](https://en.wikipedia.org/wiki/Mathematical_optimization) a [differentiable](https://en.wikipedia.org/wiki/Differentiable_function) [objective function](https://en.wikipedia.org/wiki/Objective_function), a [stochastic approximation](https://en.wikipedia.org/wiki/Stochastic_approximation) of [gradient descent](https://en.wikipedia.org/wiki/Gradient_descent) optimization. It is called **stochastic** because samples are selected randomly (or shuffled) instead of as a single group (as in standard [gradient descent](https://en.wikipedia.org/wiki/Gradient_descent)) or in the order they appear in the training set.

In gradient descent, a **batch** is the total number of examples you use to calculate the gradient in a single iteration. So far, we've assumed that the batch has been the entire data set. When working at Google scale, data sets often contain billions or even hundreds of billions of examples. Furthermore, Google data sets often contain huge numbers of features. Consequently, a batch can be enormous. A very large batch may cause even a single iteration to take a very long time to compute.

A large data set with randomly sampled examples probably contains redundant data. In fact, redundancy becomes more likely as the batch size grows. Some redundancy can be useful to smooth out noisy gradients, but enormous batches tend not to carry much more predictive value than large batches.

What if we could get the right gradient on average for much less computation? By choosing examples at random from our data set, we could estimate (albeit, noisily) a big average from a much smaller one. **Stochastic gradient descent** (**SGD**) takes this idea to the extreme--it uses only a single example (a batch size of 1) per iteration. Given enough iterations, SGD works but is very noisy. The term "stochastic" indicates that the one example comprising each batch is chosen at random.



Above algorithm says, **to perform the GD**, we need to calculate the gradient of the cost function J. And **to calculate the gradient of the cost function**, we need to sum the cost of each sample. If we have 3 million samples, we have to loop through 3 million times or use the dot product.

The cost function can also be written in the following vectorized form,

|  |  |
| --- | --- |
| \begin{displaymath} J(\theta) &=& \frac{1}{2m}\left(X\theta-\vec{y}\right)^{T}(X\theta-\vec{y}) \nonumber \end{displaymath} |  |

where

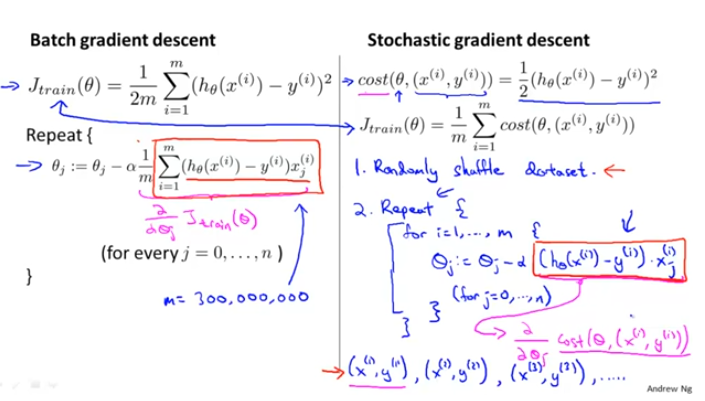
|  |
| --- |
| \begin{displaymath} \begin{array}{cc} \par \vec{y} = \left[ \begin{array}{c} y... ...m)})^T\mbox{-} \end{array}\right] \nonumber \par \end{array}\end{displaymath} |

Do you see np.dot(X.T, y\_hat-y) above? That’s the vectorized version of “looping through (summing) 3 million samples”.

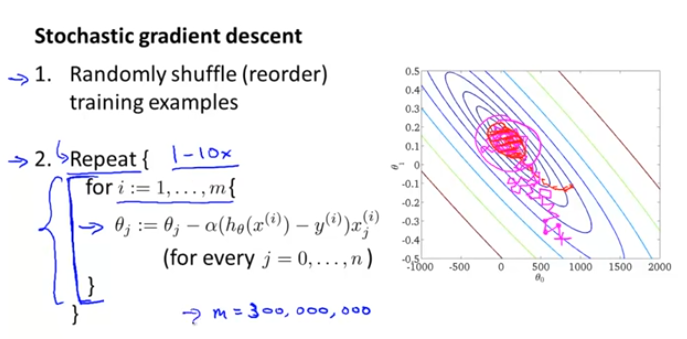
Wait.. just to move a single step towards the minimum, do we really have to calculate each cost 3 million times?

Yes. If you insist to use GD.

But if you use Stochastic GD, you don’t have to!



Basically, in SGD, we are using the cost gradient of **1** **example** at each iteration, instead of using the sum of the cost gradient of **ALL** examples.



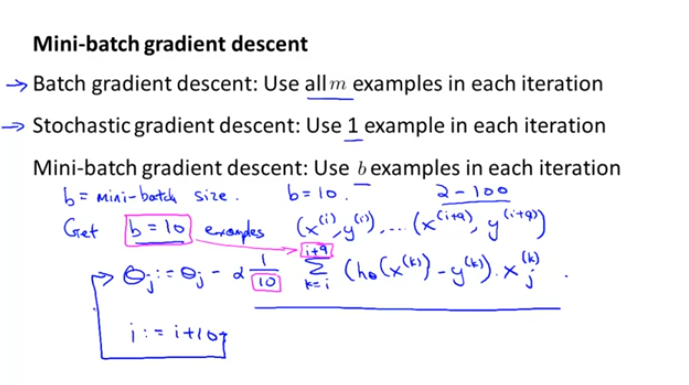
Few things to note:

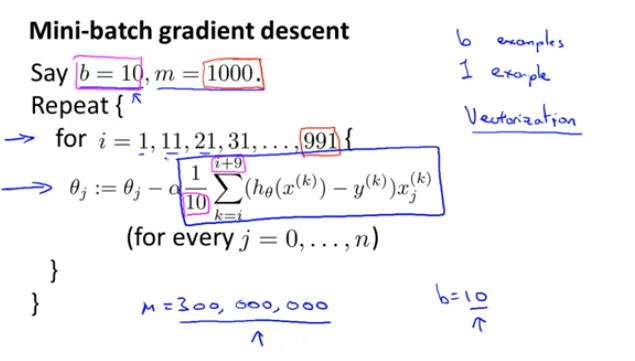
a) In SGD, before for-looping, you need to randomly shuffle the training examples.

b) In SGD, because it’s using only one example at a time, its path to the minima is noisier (more random) than that of the batch gradient. But it’s ok as we are indifferent to the path, as long as it gives us the minimum AND the shorter training time.

c) Mini-batch gradient descent uses **n** data points (instead of **1** sample in SGD) at each iteration.

**Mini-Batch Gradient Descent**

****

****

**AdaGrad**

AdaGrad (for **Adaptive** [**Gradient**](https://en.wikipedia.org/wiki/Gradient_descent) **Algorithm**) is a modified stochastic gradient descent with per-parameter [learning rate](https://en.wikipedia.org/wiki/Learning_rate), first published in 2011.Informally, this **increases the learning rate for more sparse parameters and decreases the learning rate for less sparse ones**. This strategy often improves convergence performance over standard stochastic gradient descent in settings where data is sparse and sparse parameters are more informative. Examples of such applications include natural language processing and image recognition.

**Adagrad** is more **preferable for a sparse data** set as it makes big updates for infrequent parameters and small updates for frequent parameters. It uses a different learning Rate for every parameter θ at a time step based on the past gradients which were computed for that parameter. Thus we do not need to manually tune the learning rate.

**Adam**

Adam[ (short for **Adaptive Moment Estimation**) is an update to the **RMSProp** optimizer. In this optimization algorithm, running averages of both the gradients and the second moments of the gradients are used.

It also calculates different learning rate. Adam works well in practice, is faster, and outperforms other techniques.