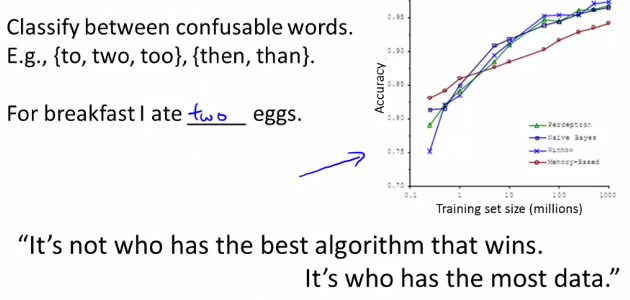
***Gradient Descent W/ Large Datasets***

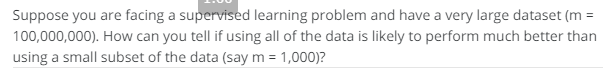
**I. Learning W/ Large Datasets**

* **Large scale ML** = algorithms dealing w/ big data sets
* 1 reason learning algorithms work so much better now than say 5 years ago is just the sheer amount of data we now have + can train algorithms on.
* 1 of the best ways to get a high-performance ML system = take a *low-bias* learning algorithm + train it on a lot of data.
* Ex: Classifying between confusable words



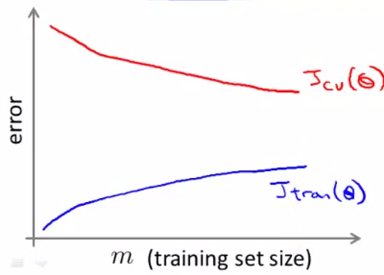
* As long as you feed the algorithm a lot of data, it seems to do very well.
* Results like these has led to the saying in ML “It's not who has the best algorithm that wins. It's who has the most data”
* So you want to learn from large data sets when you can get large data sets.
* But learning w/ large data sets comes w/ its own unique problems, specifically, computational ones.
* Say training set size m = 100M (realistic for many modern data sets) + you want to train a linear or logistic regression model
* To compute the gradient when m = 100M, you need to carry out a summation over a 100M terms in order to compute the derivatives terms/to perform a *single* step of decent.



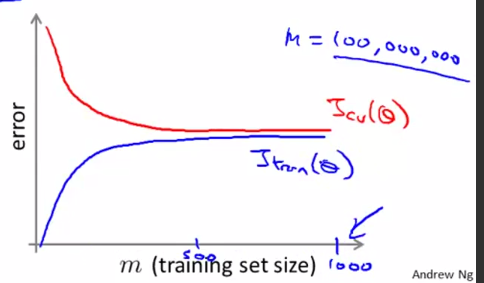
* B/c of the computational expense of summing over a 100M entries in order to compute just 1 step of gradient descent, there are techniques for either replacing this w/ something else or to find more efficient ways to compute this derivative.
* Of course, before you put in the effort into training a model w/ a 100M examples, ask yourself “why not use just a 1K examples?”.
* Maybe randomly picking subsets of 1k from 100M examples + training an algorithm on those
* So before investing effort into actually developing software needed to train massive models, it’s often good to sanity check if training on just 1k examples might do just as well.
* 



* The way to sanity check if using a much smaller training set might do just as well, the usual method is plotting the **learning curves**
* If your training objective Jtrain(Ө) + J\_cv(Ө) were to look like this



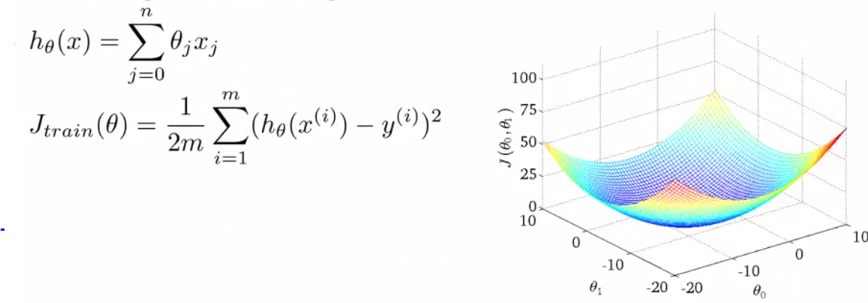
* This looks like a *high-variance* learning algorithm, + we are more confident that adding extra training examples would improve performance.
* Whereas in contrast if you were to plot the learning curves + the objectives were to look like this:



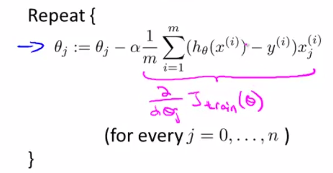
* This looks like a *high-bias* learning algorithm, so it seems unlikely that increasing m to a 100M will do much better + you'd be just fine sticking w/ m = 1K rather than investing a lot of effort to figure out how the scale up the algorithm.
* If in this 2nd situation, 1 natural thing to do would be to add extra features (or extra hidden units to your NN) so that you end up w/ a situation closer to that on the 1st scenario
* This would give more confidence that trying to add infrastructure to change the algorithm to use much more than a 1k examples might actually be a good use of time.
* So, in large-scale ML, come up w/ *computationally-reasonable/efficient* ways to deal w/ very big datasets.

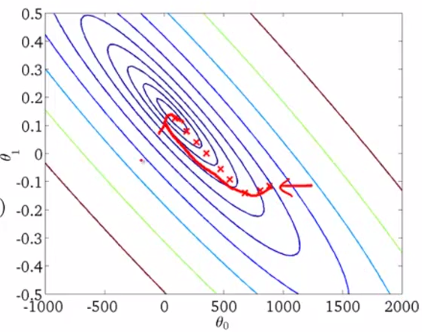
**II. Stochastic Gradient Descent**

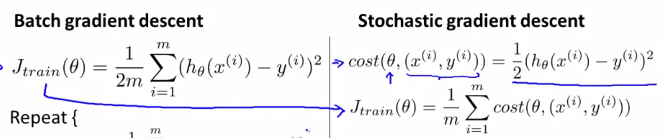
* For many learning algorithms, among them linear + logistic regression, NN, etc., the way we derive them was by coming up w/ a cost function/optimization objective + then using an algorithm like gradient descent to *minimize* that cost function.
* W/ a very large training set, gradient descent becomes very computationally-expensive
* A modification to the basic gradient descent algorithm called **Stochastic gradient descent** allows us to scale these algorithms to much bigger training sets.



* Cost function Jtrain = 1/2 the sum of the average square error of the hypothesis on the m training examples + looks like a bow-shaped function plotted as function of the parameters Ө0 + Ө1
* Then gradient descent repeatedly updates the parameters Ө:



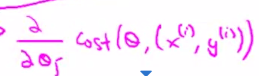
* Remember the partial derivative term of Jtrain w/ respect to Ө
* We’ll keep using linear regression as the running example, but the idea of Stochastic gradient descent is fully general + also applies to other learning algorithms like logistic regression, NN, + others that are based on training gradient descent on a specific training set.
* 
* Remember gradient descent above: if parameters are initialized to the point all the way to the right, as you run more iterations, gradient descent takes the parameters to the global minimum (middle)
* The problem w/ gradient descent is that if m is large, computing this derivative term can be very expensive, b/c summing over all m examples (summing over 300M people in the US census data is very expensive)
* This particular version of gradient descent = **Batch gradient descent** + the term **Batch** refers to the fact that we're looking at ALL training examples at a time
* The way this algorithm works is you need to read into CPU memory *ALL 300M records* in order to compute the derivative term
* Or we need to **stream** all records through a CPU b/c we can't store all these records in memory.
* i.e. read through them + slowly accumulate the sum in order to compute the derivative
* Then having done all that work, that allows you to take ONE step of gradient descent, + then you need to do the whole thing again
* It's will take a long time in order to get the algorithm to **converge**.
* In contrast to batch gradient descent, a different algorithm *doesn't* need to look at *all* training examples in every single iteration, but only needs to look at only a *single* training example in *1* iteration
* **Stochastic gradient descent** has the cost function written a slightly different way



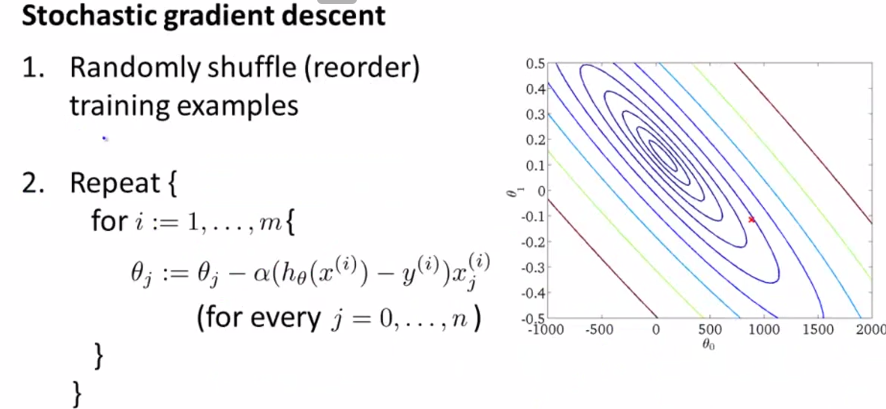
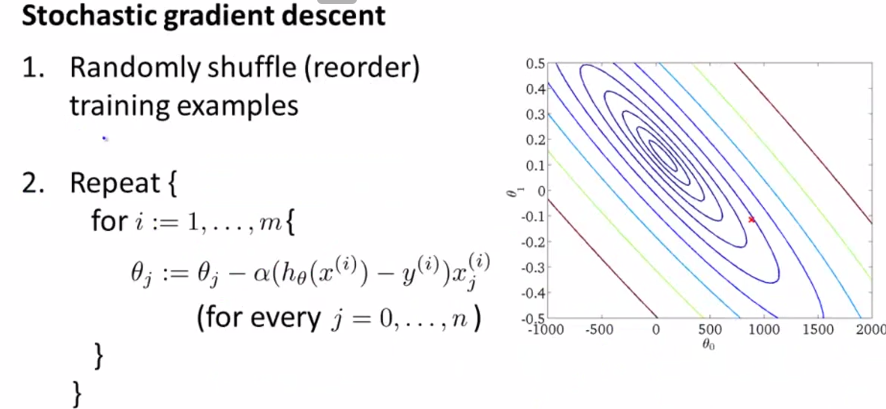
* Cost of a parameter Ө w/ respect to a training example {x(i), y(i)} = 1/2 times the squared error my hypothesis incurs on that example, {x(i), y(i)}.
* So, *this* cost function term measures how well my hypothesis does on a SINGLE example {x(i), y(i)}
* Notice the overall cost function Jtrain can now be written in an equivalent form 🡪 the average over the m training examples of the cost of my hypothesis on *that one example* {x(i), y(i)}.
* Steps of Stochastic gradient descent
* 1) Randomly shuffle/reorder the data set. 🡺
* 2) Repeat, for i = 1-m (scan through training examples),+ perform following:
* Update, for all values of j (j = 0-n):



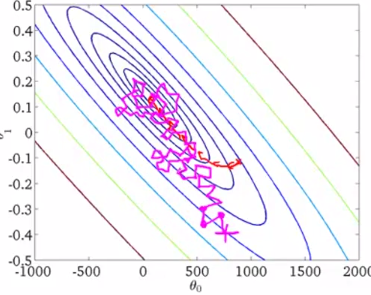
* Main work of Stochastic gradient descent
* Notice this term alpha is multiplied by in stochastic GD is exactly what we had inside the summation for Batch GD.
* It’s possible to show that this term is equal to the partial derivative w/ respect to my parameter Өj of the cost of the parameters Ө on {x(i), y(i)}



* What Stochastic GD is doing is scan through the training examples, + starting w/ the 1st training example {x(1), y(1)} + looking *only* at this 1st example, take a small gradient descent step w/ respect to the cost of *just this 1st training example*.
* In other words, it looks at the 1st example + modifies the parameters a bit just to fit this 1st training example a bit better.
* Having done this, it then goes on to the 2nd training example + takes another little step in parameter space + modify the parameters just a bit to try to fit the 2nd training example a little bit better.
* It continues until you get through the entire training set
* This repeat loop may cause the algorithm to take multiple passes over the entire training set.
* This view of Stochastic GD also motivates why we wanted to start by randomly shuffling the dataset.
* In practice, this just speeds up the convergence of Stochastic GD just a little bit.
* In the interest of safety, it's usually better to randomly shuffle the dataset if you aren't sure it came to you in randomly-sorted order
* More importantly, another view of Stochastic GD is that it's a lot like batch GD
* But rather than waiting to sum up the gradient terms over all m training examples, we find the gradient term using just 1 single training example + are starting to make progress in improving the parameters right from the start
* Rather than waiting to take a pass through all of 300M records before we can modify the parameters a little + make progress towards a global minimum, Stochastic GD instead looks at a single training example + starts to make progress right away



* Batch GD will tend to take a reasonably straight-line trajectory to get to the global minimum
* In contrast, in Stochastic GD, every iteration is going to be much faster b/c we don't need to sum up over all training examples b/c every iteration is just trying to fit single training example better.
* But as you run Stochastic GD, what you find is it will *generally* move the parameters in the direction of the global minimum, but not always (some steps may be in the “wrong” direction)
* It can take some more random-looking, circuitous path to the global minimum (pink0



* Stochastic GD doesn't actually converge in the same sense as Batch GD
* It actually ends up wandering around continuously in some region *close to* the global minimum, but it doesn't just *get* to the global minimum + stay there.
* In practice, this isn't a problem b/c so long as the parameters end up in some region close to the global minimum, we get a hypothesis that’s good enough for most practical purposes.
* 1 final detail: Stochastic GD has an outer loop which says to do the inner loop multiple times.
* Depending on the size of the training set, doing this outer loop just a single time may be enough, + maybe 10 times may be typical
* If we have a truly massive data set, it’s possible that by the time you've taken just a single pass through your training set, you might already have a perfectly good hypothesis + you might need to do the inner loop only once even if m is very, very large.
* But, in general, taking anywhere from 1-10 passes through a data set is fairly common, but really, it depends on the size of your training set.
* Meanwhile w/ Batch GD, after taking a pass through your entire training set, you would’ve taken just a *single* gradient descent steps
* This is why Stochastic gradient descent can be much faster.

**III. Mini-batch Gradient Descent**

**IV. Stochastic Gradient Descent Convergence**