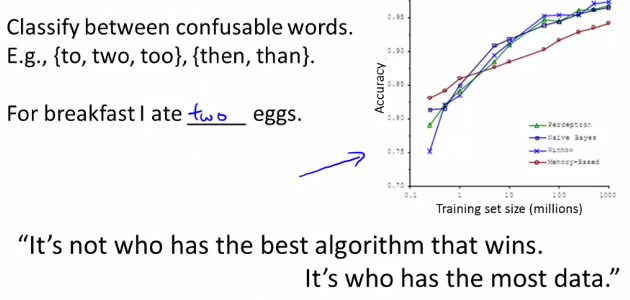
***Descent W/ Large Datasets***

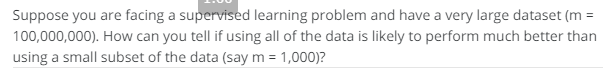
**I. Learning W/ Large Datasets**

* **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



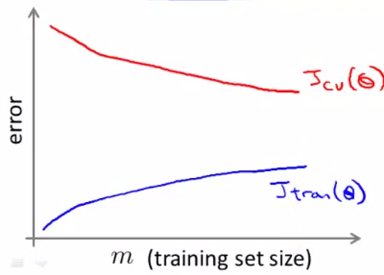
* As long as you feed the algorithm a lot of data, it seems to do very well.
* Results like these has led to the saying in ML “It's not who has the best algorithm that wins. It's who has the most data”
* So you want to learn from large data sets when you can get large data sets.
* But learning w/ large 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.



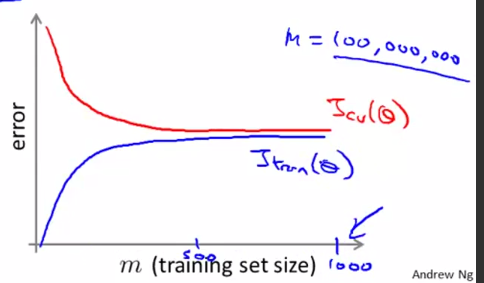
* B/c of the computational expense of summing over a 100M entries in order to compute just 1 step of gradient descent, there are techniques for either replacing this w/ something else or to find more efficient ways to compute this derivative.
* Of course, before you put in the effort into training a 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



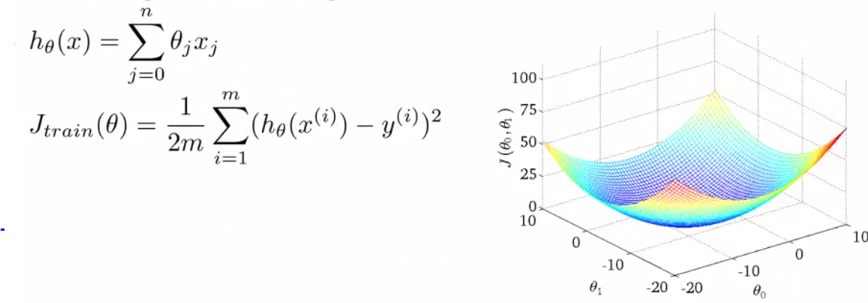
* This looks like a *high-variance* learning algorithm, + we are more confident that adding extra training examples would improve performance.
* Whereas in contrast if you were to plot the learning curves + the objectives were to look like this:



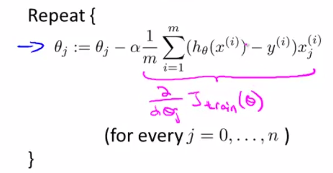
* This looks like a *high-bias* learning algorithm, so it seems unlikely that increasing m to a 100M will do much better + you'd be just fine sticking w/ m = 1K rather than investing a lot of effort to figure out how the scale up the algorithm.
* If in this 2nd situation, 1 natural thing to do would 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.

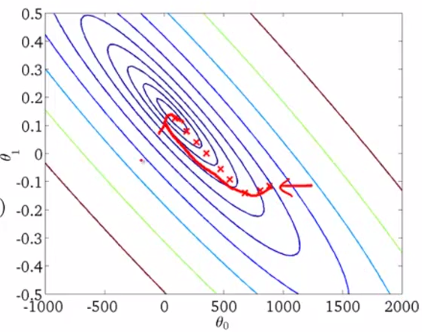
**II. Stochastic Gradient Descent**

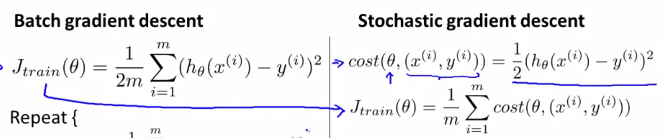
* For many learning algorithms, among them linear + logistic regression, NN, etc., the way we derive them was by coming up w/ a cost function/optimization objective + then using an algorithm like gradient descent to *minimize* that cost function.
* W/ a very large training set, gradient descent becomes 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:



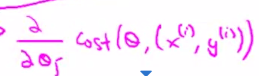
* Remember the partial derivative term of Jtrain w/ respect to Ө
* We’ll keep using linear regression as the running example, but the idea of Stochastic gradient descent is fully general + also applies to other learning algorithms like logistic regression, NN, + others that are based on training gradient 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



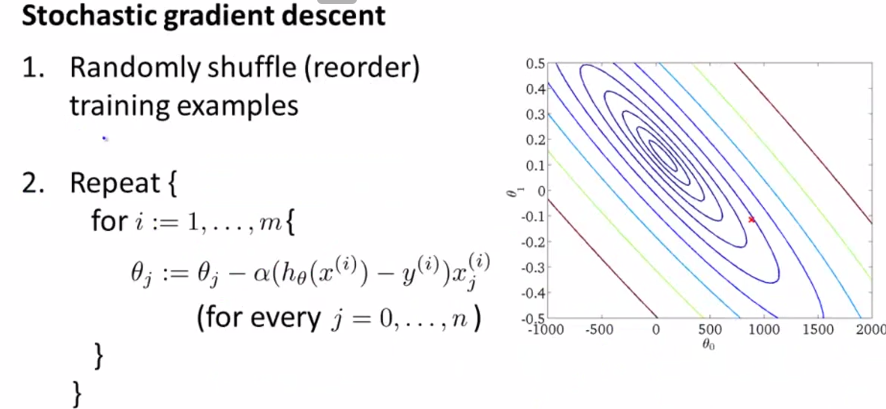
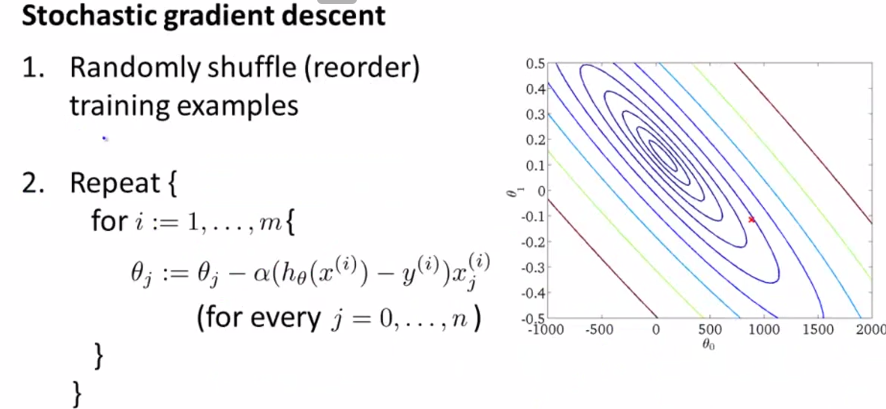
* Cost of a parameter Ө w/ respect to a training example {x(i), y(i)} = 1/2 times the squared error my hypothesis incurs on that example, {x(i), y(i)}.
* So, *this* cost function term measures how well my hypothesis does on a SINGLE example {x(i), y(i)}
* Notice the overall cost function Jtrain can now 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):



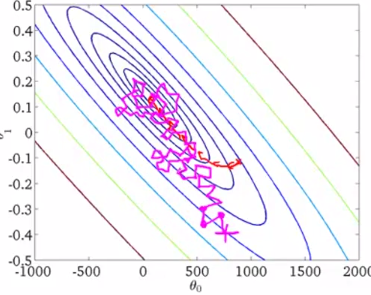
* Main work of Stochastic gradient descent
* Notice this term alpha is multiplied by in Stochastic GD is exactly what we had inside the summation for Batch GD.
* It’s possible to show that this term is equal to the partial derivative w/ respect to my parameter Өj of the cost of the parameters Ө on {x(i),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

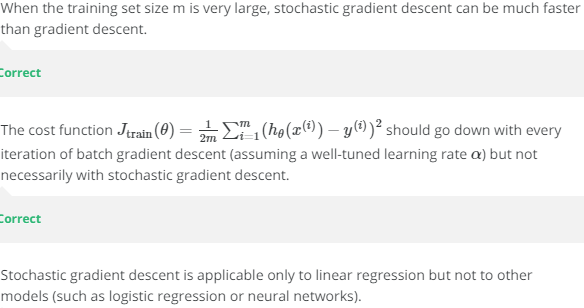
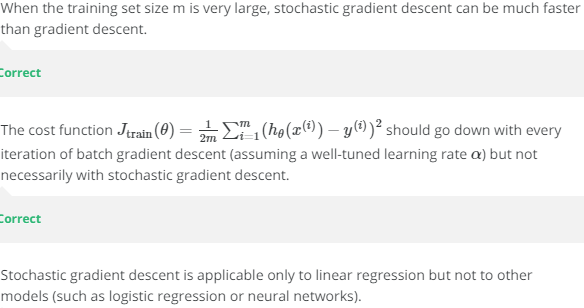


* Stochastic GD = scan through the training examples, + starting w/ the 1st training {x(1), y(1)} + looking *only* at this 1st example, take a *small* gradient descent step w/ respect to the cost of *just this 1st training example*.
* In other words: look at the 1st example + modifies the parameters 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
* The outer 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



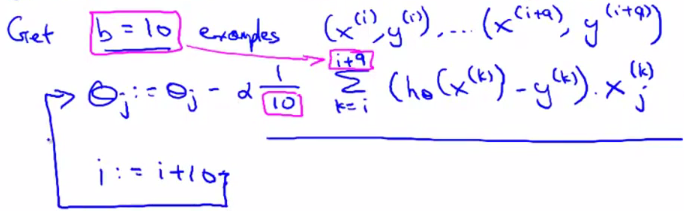
* Batch GD will tend to take a reasonably straight-line trajectory to get to the global minimum
* In contrast, in Stochastic GD, every iteration is going to be much faster b/c we don't need to sum up over all training examples b/c every iteration is just trying to fit single training example better.
* 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



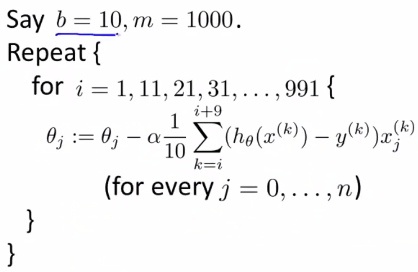
* Stochastic GD doesn't actually converge in the same sense as Batch GD
* It actually ends up wandering around continuously in some region *close to* the global minimum, but it doesn't just *get* to the global minimum + stay there.
* In practice, this isn't a problem b/c so long as the parameters end 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**

* So Stochastic GD can be much faster than Batch GD.
* Another variation = **Mini-batch GD**, which can sometimes work even faster than stochastic GD
* In Batch GD, we use ALL m examples in each generation, whereas in Stochastic GD, we use a single example in each generation, + what Mini-batch GD does is somewhere in between.
* W/ this algorithm we use **b** examples in each iteration where **b** = a parameter, the **mini batch size**
* It’s like Batch GD, except w/ a much smaller batch size 🡪 typical choice for b might be from 2-100
* The idea is that rather than using just 1 or using m examples at a time, use b examples at a time.



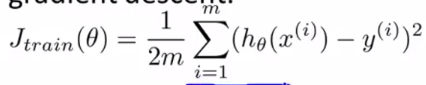
* For this example, say b = 10 examples from my training set, {x(i), y(i)} – {x(i+9), y(i+9)} 🡪 essentially perform a GD update using these 10 examples.
* In this expression, we’re summing the gradient terms over the 10 examples, then go on to the next 10 examples, + so on.



* Our loop has steps of 10 b/c we look at 10 examples at a time + we perform a sort of GD update using 10 examples at a time
* Compared to Batch GD, we make progress much faster.
* U.S. Census data w/ 300 million training examples 🡪 after looking at just the 1st 10 examples, we can start to make progress in improving the parameters ϴ
* Don't need to scan through the entire training set 🡪 just need to look at the 1st 10 examples + this will start letting us make progress +
* Then we can look at the next 10 examples + modify the parameters a little bit again + so on.
* Why Mini-batch GD can be faster than Batch GD = can start making progress in modifying parameters after looking at b examples rather than needing to wait to scan through every example
* Why do we want to look at b examples at a time rather than look at just a single example at a time like in Stochastic GD?
* The answer is in **vectorization**.
* Mini-batch GD = likely to outperform Stochastic GD *only w/ a good vectorized implementation*
* In that case, the sum over 10 examples can be performed in a *more vectorized way* which allows you to partially **parallelize** your computation over the 10 examples.
* In other words, by using appropriate vectorization to compute the derivative terms, you can sometimes partially use good numerical algebra libraries to parallelize gradient computations over the b examples
* Whereas if you were looking at just a single example at a time w/ Stochastic GD, there isn't much to parallelize over (At least there is less to parallelize over)
* 1 *disadvantage* of Mini-batch GD = there is now this *extra* parameter b, the Mini-batch size, which you may have to fiddle w/ + may therefore take time.
* But if you have a good vectorized implementation, MBGD can sometimes run even faster that Stochastic GD + IT does something somewhat in between what Stochastic GD Batch GD do.
* W/ a reasonable value of b + a good vectorized implementation, sometimes it can be faster than both Stochastic GD + Batch GD.

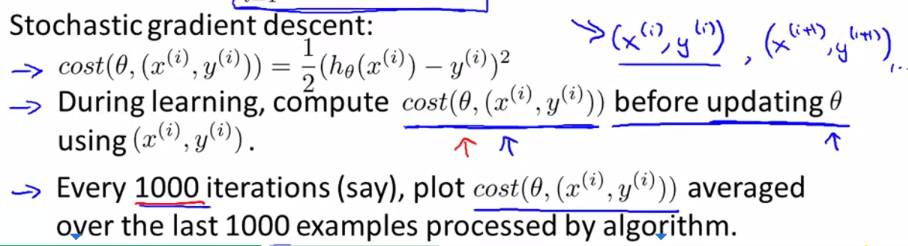
**IV. Stochastic GD Convergence**

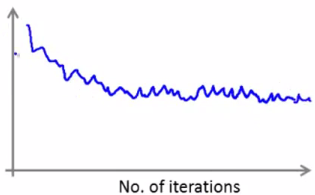
* When you're running a stochastic GD algorithm, how do you make sure it's completely debugged + is converging okay? How do you tune the learning rate **α** w/ Stochastic GD?
* When using batch GD, standard way for making sure GD was converging = plot the optimization cost function as a function of the number of iterations + make sure it is decreasing on every iteration.



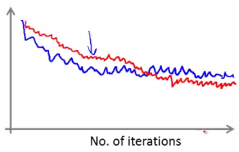
* When the training set sizes were small, we could do that b/c we could compute the sum of squared errors efficiently.
* But w/ a massive training set size, you don't want to have to pause the stochastic GD algorithm periodically in order to compute the cost function, since it requires a sum of over the *entire* training set
* Want Stochastic GD to start to make progress after looking at just a single example, w/out needing to scan through an entire training set in the middle of the algorithm just to compute the cost function over the entire training set.
* So for stochastic GD, in order to check the algorithm is converging, take the cost of the parameters ϴ w/ respect to a single training example, ({x(i), y(i})



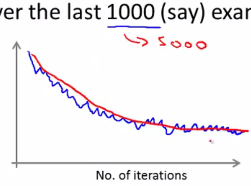
* Then, while stochastic GD is learning, compute this *before* we train on a specific example 🡪 *before we update ϴ using ({x(i), y(i})*
* In stochastic GD, we look at the examples ({x(i), y(i}) in order, make an update w/ respect to this example, then go on to the next example ({x(I + 1), y(I + 1}), + so on
* So, while the algorithm is looking at the example ({x(i), y(i}) but *compute the cost of that example before* it updates the parameters ϴ using that example
* As stochastic GD is scanning through a training set, right before updating ϴ using a specific training example ({x(i), y(i}), let's the cost on that ({x(i), y(i}) = how well our hypothesis h(ϴ)x is doing on that training example.
* Do this *before* updating ϴ b/c if we've just updated ϴ using example ({x(i), y(i}), than the algorithm might be doing better on that example than what would be representative of the whole training set
* Finally, in order to check for the convergence of stochastic GD, every, say, 1K iterations, plot average of the costs we've been computing over last 1K iterations/examples processed by the algorithm
* This gives a running estimate of how well the algorithm is doing on the last 1K training examples the algorithm has seen.
* In contrast to computing Jtrain periodically, which needed to scan through the entire training set, w/ this procedure, being part of stochastic GD, doesn't cost much to compute costs, + it’s before updating parameters ϴ
* Then all we're doing is every 1K integrations, average the last 1K costs computed + plot that
* By looking at those plots, we check if stochastic GD is converging.
* 
* Suppose you plot the cost average over last 1K examples (noisy b/c it’s an average + may not decrease on every single iteration)



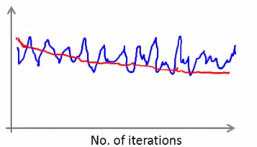
* This is a pretty decent run w/ the algorithm 🡪 cost has gone down + plateaued = learning algorithm has converged.
* If you want to try a smaller learning rate, the algorithm may initially learn more slowly (cost goes down more slowly), but it’s possible the algorithm ends at a slightly better solution.



* This is b/c stochastic GD doesn't just converge to the global minimum, but the parameters will oscillate a bit around the global minimum
* So, by using a smaller learning rate, you end up w/ smaller oscillations
* Sometimes this little difference will be negligible, + sometimes you get slightly better values for the parameters.



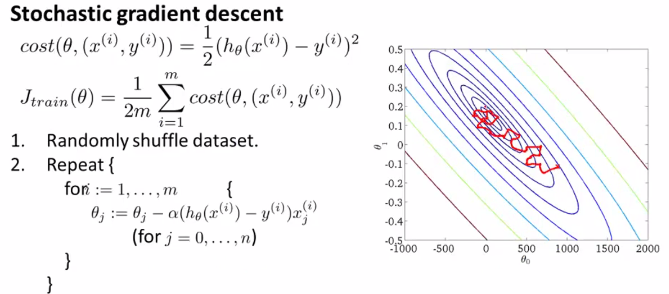
* This looks like its converged w/ 1000 iterations, but by averaging over 5K examples, you might be able to get a smoother curve = an effect of increasing the # of examples you average over.
* *Disadvantage* of making this *too* big = now you get 1 data point every 5K examples 🡪 feedback you get on how well your learning algorithm is doing is more delayed

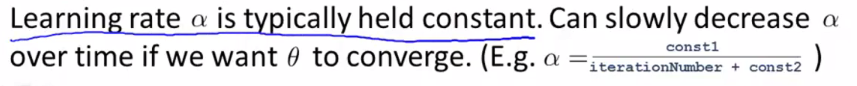


* Here, cost just is not decreasing at all when averaging over 1K examples 🡪 looks like the algorithm is just not learning.
* But if we increase this to averaging over a larger # of examples, it‘s possible you see cost is actually decreasing + the blue line was just too noisy 🡪 couldn't see the actual trend in cost decreasing
* It is also possible a learning curve still ends up flat even averaged over a larger number of examples.
* That's maybe just a more firm verification that the algorithm unfortunately just isn't learning much for whatever reason + you need to either change the learning rate, the features, or something else about the algorithm.



* 1 last thing that you might see is a curve that actually looks like it's increasing = a sign the algorithm is **diverging**. 🡺 Must use a smaller value of the learning rate alpha.
* *So if the plot looks too noisy, try increasing the # of examples you're averaging over*
* *If you see errors/costs increasing, try using a smaller value of alpha.*
* It's worth examining the issue of the learning rate a bit more



* When we run stochastic GD, the algorithm will meander towards the minimum + won't really converge but will wander around the minimum forever
* You end up w/ a parameter value that is *hopefully* close to the global minimum but won't be *exactly* at the global minimum.
* In most typical implementations of stochastic GD, the learning rate **α** is typically held constant
* To get stochastic GD to *actually* converge to the global minimum, slowly decrease the **α** over time
* A pretty typical way of doing 🡪 set **α** equals some constant, const1, 1 divided by the iteration # plus some constant, const2. 
* # of iterations you've run of stochastic GD = # of training examples seen
* const1 + const2 = additional parameters of the algorithm you might have to play w/ a bit in order to get good performance
* 1 reason people tend not to do this = end up needing to spend time playing w/ these 2 extra parameters, which makes the algorithm more finicky
* It’s just more parameters to fiddle w/ in order to make an algorithm work well.
* But if you manage to tune the parameters well, the algorithm meanderings will get smaller + smaller until it pretty much just to the global minimum.
* As the algorithm runs, the iteration # becomes large, **α** will slowly become small, + so you take smaller + smaller steps until it hopefully converges to the global minimum.
* So, if you slowly decrease **α** to 0, you can end up w/ a slightly better hypothesis
* But b/c of the extra work needed to fiddle w/ constants (+ b/c frankly usually we're pretty happy w/ any parameter value close to the global minimum), typically this process of decreasing **α** slowly is usually not done + keeping **α** constant is the more common application of stochastic GD





