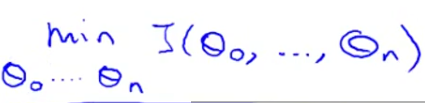
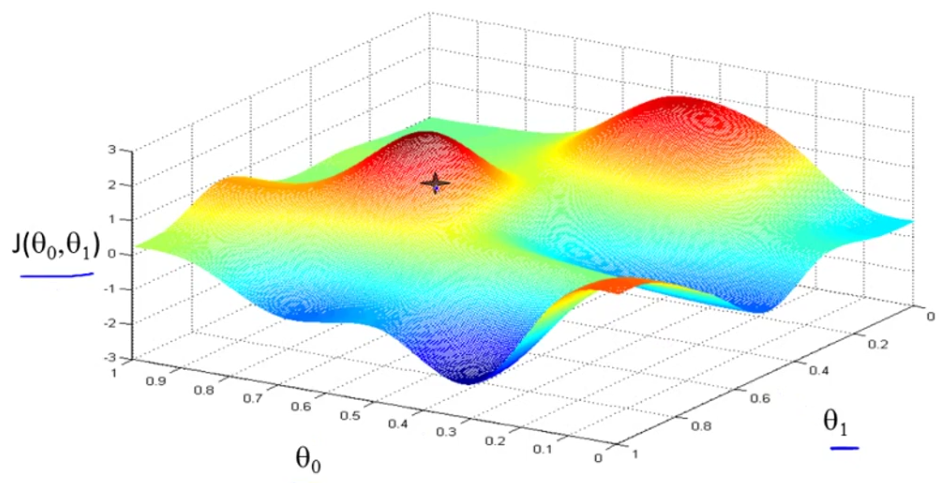
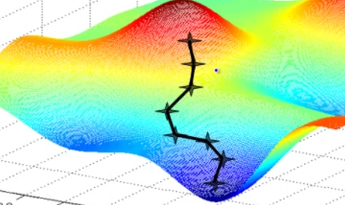
***PARAMETER LEARNING***

**I. GRADIENT DESCENT**

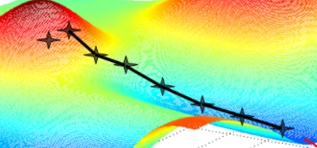
* **Gradient Descent** = algorithm for minimizing the cost function J(θ0,θ1)
* It’s a more general algorithm used not only in linear regression, but all over in ML to minimize other functions besides J(θ0,θ1)
* Assume we have some function J(θ0,θ1) + we want an algorithm for minimizing it
* **NOTE:** It turns out that gradient descent actually applies to more general functions.
* i.e. minimize the value of J(θ0…..θ(n)) over θ0…..θ(n) 
* Idea for gradient descent = to start w/ some initial guesses for θ0 + θ1 (common choice 🡪θ0 = 0, θ1 = 0)
* Then we keep changing these values little by little bit to try to reduce J(θ0,θ1), until hopefully, we wind up at a minimum (or a local minimum)
* θ0 + θ1 = horizontal axes, J(θ0,θ1) = height of the surface shows J(θ0,θ1) + we want to minimize it.



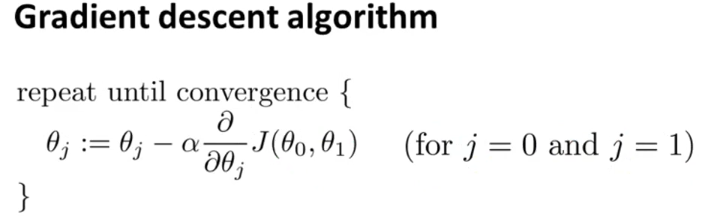
* Imagine that this figure shows 2 hills + you ‘re physically standing at the point on the hill
* Start off w/ {θ0, θ} at some point on the surface (black cross above)
* In gradient descent, we spin 360 degrees, look around, + ask “if I were to take a little baby step in *some* direction, + I want to go downhill as quickly as possible, in what direction do go?”
* To go down, want to physically walk down this hill as rapidly as possible.
* Then you take that “best direction” + end up new point on the hill
* Then, again, look all around + see what direction to step in order to take a little baby step downhill + take another step in that direction + then keep going until you converge to a **local minimum**.



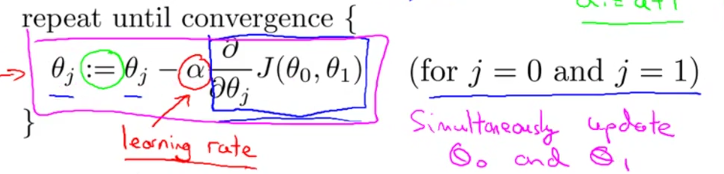
* Now imagine we initialized gradient descent just a couple steps to the right instead, + then repeated the process from that point
* Now gradient descent would've taken you to a 2nd local minimum, more to the right.



* This is an interesting property of gradient descent
* This is the definition of the gradient descent algorithm.



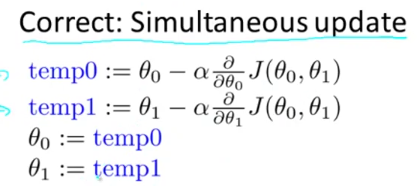
* Repeatedly do the formula above until **convergence**
* update parameter θj by taking θj + subtracting from it **α**\*a derivative term
* **:=** 🡪 denotes assignment
* a **:=** b 🡪 in a CPU, this means take the value in b + use it to overwrite whatever value is a.
* “set a to be equal to the value of b”
* Whereas in contrast, w/ an equal 🡪 a = b, this is a **truth assertion.**
* **Alpha (α)** = the **learning rate**.
* This controls *how big a step we take* downhill w/ gradient descent.
* Very large α = very aggressive gradient descent procedure (huge steps downhill)
* Very small α = taking baby steps downhill.
* Then we have a **derivative term** in blue

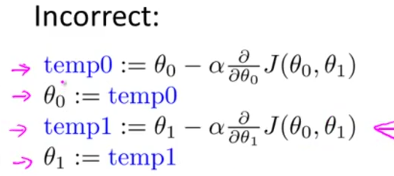


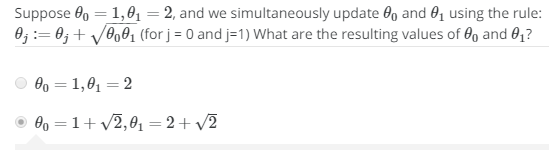
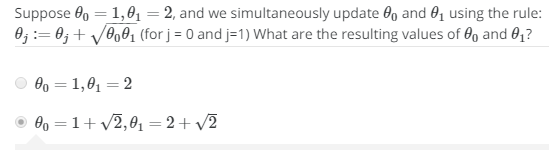
* There's 1 more subtlety about gradient descent 🡪 in gradient descent we're going to update θ0 + θ1,
* “the update takes place for j = 0 and j = 1”
* So when updating θ0 + θ1, we ***simultaneously update*** θ0 + θ1.
* Meaning in this update equation, we're going update θ0 := …. + update θ1 := …..
* The way to implement = compute the right-hand for θ0 + θ1 + then simultaneously update θ0 + θ1

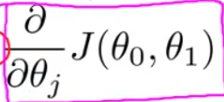


* Ex: Set *temp0* + *temp1* to compute the right-hand sides + store them into variables temp0 + temp1 + then update θ0 + θ1 simultaneously = CORRECT

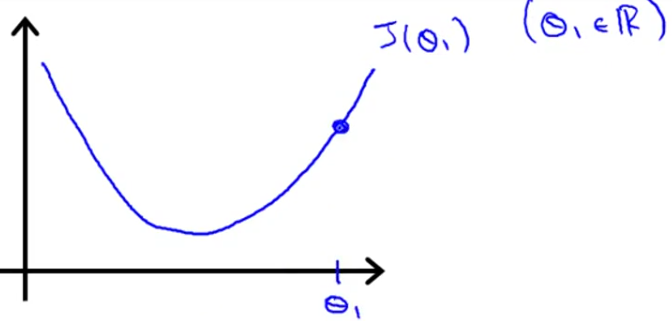


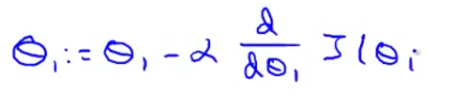


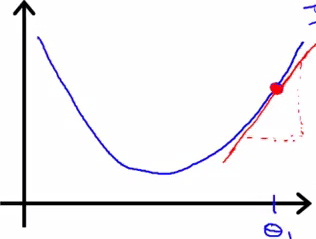
* In this 2nd attempt, by the time we tried to update θ1 (3rd step), we’d be using a new value of θ0 (from 2nd step), which gives a different value of temp1 than what we *should* have
* The way gradient descent is usually naturally implemented is w/ simultaneous updates.
* When people talk about gradient descent, they always mean simultaneous update
* If you implement the NON-simultaneous update, it will probably work anyway, but the algorithm won’t be right + is actually some other algorithm w/ different properties that can, for various reasons, behave in slightly stranger ways
* 
* The **derivative term:**



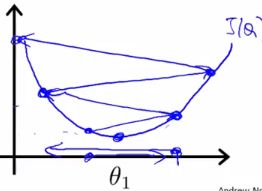
* **Derivative =** measures the sensitivity to change of a function (output) value w/ respect to a change in its argument (input value)
* **Partial derivative** = derivative of a function w/ respect to ONE of its variables, with the others held constant (as opposed to the **total derivative**, in which all variables are allowed to vary)
* Think of it as how much we expect the value of the function to change is we make a small change to ONE of its arguments, leaving the others unchanged
* Say we have a cost function of just 1 parameter, j(θ1), where θ1 = a real number
* Let's try to understand what gradient decent would do on this function.



* So we initialize gradient descent w/ θ1 to the right of the x-axis
* Gradient descent will update θ1 as 
* We compute the derivative of (take the tangent to) the J(θ1) + look at the slope of this line



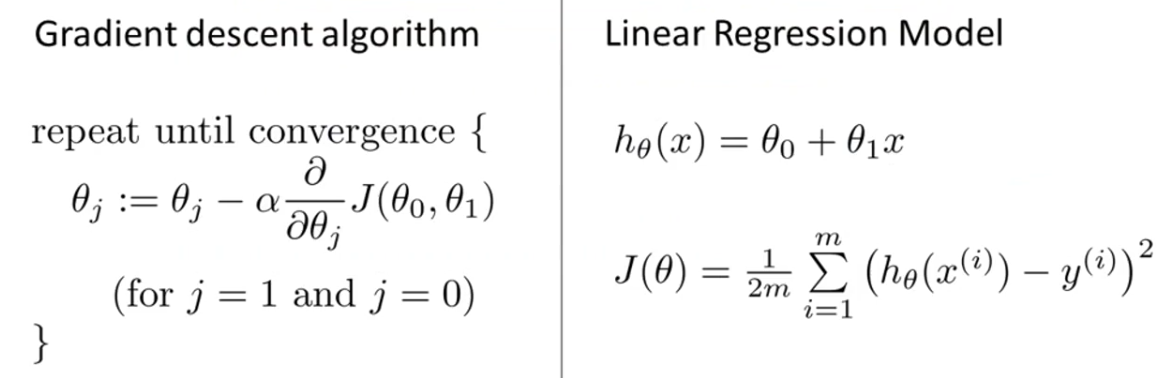
* If slope = positive = positive derivative 🡺 so my update ( **:=** ) to θ1 is going to be θ1 – **α**\*some positive number (**α** *is always a positive number*)
* So we end up w/ θ1 minus a positive 🡪 we move to a new θ value to the left (decrease θ1)
* We *want* to head in this direction 🡪 towards minimum J(θ1) 🡪 dip of the curve
* If the initial θ1 was to the left of the minimum, we’d have a negative tangent slope = negative derivative = θ1 minus a negative # 🡺 increase θ1 🡺 update θ1 to the right towards the minimum J(θ1)
* If **α** is too small, we multiply updated θj by some small number = taking a small step = we’ll need a lot of steps to get to the minimum = gradient descent is *very slow*
* If **α** is too large= gradient descent can *overshoot* the minimum + may even *fail* to converge or even actually *diverge*



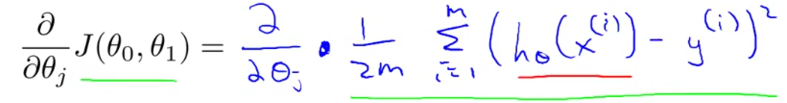
* See we end up w/ a worse value of θ1
* Can continue to overshoot actually get further + further away from the minimum.
* What if parameter θ1 is *already at a local minimum*?
* Initialize θ1 at a local minimum/optimum 🡪 the derivative = 0 so the tangent slope = 0 🡪 derivative term = 0 🡪 gradient descent update = θ1 – 0
* *If you're already at the local optimum, gradient descent does absolutely nothing*
* This also explains why gradient descent can converge to the local minimum even w/ a fixed learning rate/**α**.
* Imagine cost function J(θ) + we start gradient descent w/ a steep derivative
* Next step = derivative = less steep b/c as we approach the minimum, the derivative gets closer + closer to 0
* Now we take that smaller step + are now even closer to global minimum (derivative is even smaller)
* As gradient descent runs, you will automatically take smaller + smaller steps until eventually you're taking very small steps + finally converge to the local minimum.
* So as we approach local minimum, the *derivative term* will automatically *get smaller*, so gradient descent will automatically take smaller steps = NO *need to decrease* **α**.
* This gradient descent algorithm can be used to try to minimize any cost function J

**I. GRADIENT DESCENT FOR LINEAR REGRESSION**

* We get an algorithm for linear regression by putting together gradient descent w/ a cost function



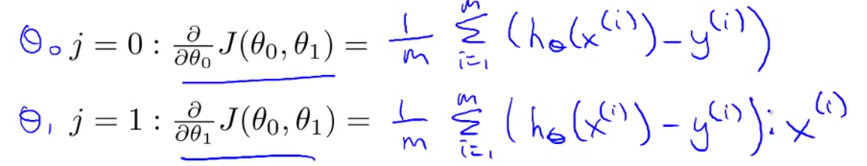
* We want to apply gradient descent *to minimize squared error/cost function*.
* In order to apply gradient descent, the key term needed is the *partial* *derivative term,* into which we plug in the definition of J(θ0,θ1)



* Then we plug in the definition for hypothesis(x(i))

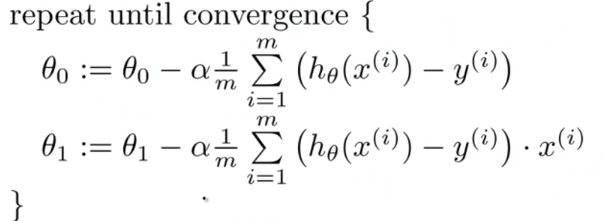


* Turns out we need to figure out the **partial derivative** for TWO cases 🡪 for J = θ0 and J = θ1.

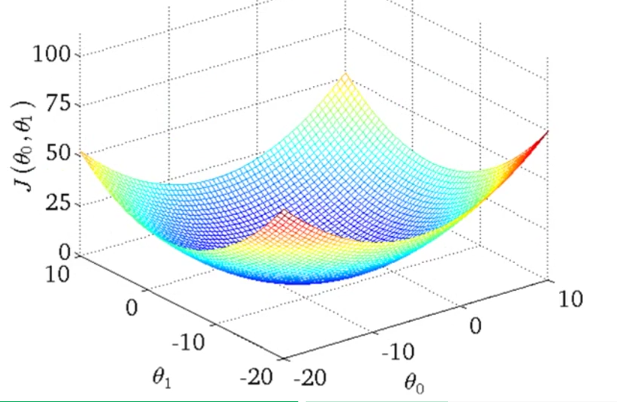


\*\*\* Computing these partial derivative terms requires some multivariate calculus

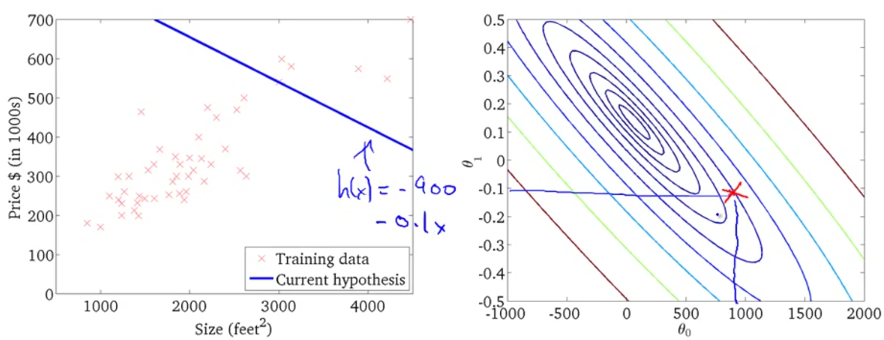
* So out of these definitions, we worked out the partial derivatives, which end up being the slope of the cost function J(θ0,θ1), which we can plug back in to the gradient descent algorithm:
* **Gradient Descent Update For Linear Regression**



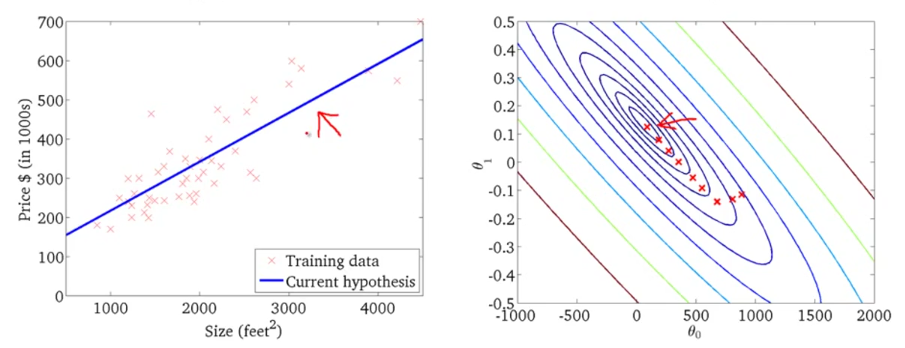
* For θ0, we have θ0 minus **α** multiplied by the partial derivative w/ respect to θ0, and the same thing happens for θ1 with the partial derivative w/ respect to θ1
* REMINDER: Should be implementing this so the updates to θ0 + θ1 are simultaneous
* 1 of the issues we saw w/ gradient descent is it can be *susceptible to local optima*
* i.e. depending on where you initialize gradient descent, you can end up at different local optima.
* The cost function for linear regression is *always going to be a bow shaped function =* a **convex function**.



* This function only has ONE global optimum and NO local optima
* Gradient descent on *this* type of cost function (which you will get whenever using linear regression) will *always* converge to the **global optimum** (b/c there are *no other local optimum*)

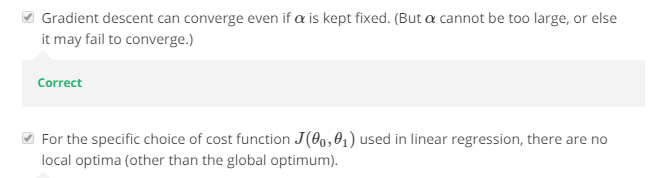
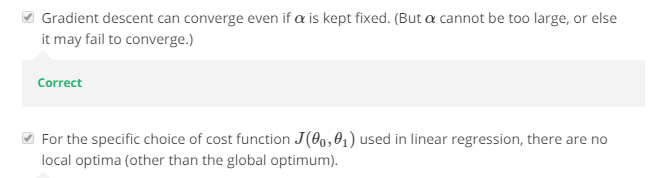


* We usually initialize parameters at (0,0), but for this demonstration, do θ0 = 900, θ1 = ~-0.1 🡪 this corresponds to hθ(x) = 900 - 0.1x
* Now, if we take 1 step in gradient descent, we move down + left + the hθ(x) lines changes a bit
* As we take more steps of gradient descent, we go *down in cost* (move parameter data point closer to middle/minimum) + hθ(x) line better fits to the data
* Eventually we wind up at the global minimum = gets a good fit to the data set of housing prices



* We can now use our linear regression line to predict the price to sell a 1250 square ft. house = ~$250k
* The algorithm just used is sometimes called **batch gradient descent**
* **Batch Gradient Descent**refers to the fact that, in *every* step of gradient descent, we're looking at/using ALL training examples
* When computing derivatives, we're computing the sums over our “m” training examples



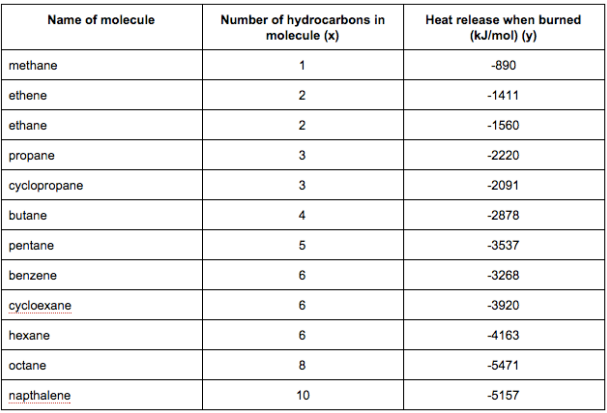
* This refers to the fact that we're looking at the *entire batch* of training examples
* There are sometimes other versions of gradient descent that are *not* batch versions + do *not* look at the *entire* training set, but instead look at small subsets of the training set at a time.
* 
* If you've seen advanced linear algebra, you might know there exists a solution for numerically solving for the minimum of the cost function w/out needing to use an iterative algorithm like gradient descent =
* This would be **the normal equations method**, but gradient descent scales better to larger data sets than that this method, + we can use gradient descent in lots of different contexts + different ML problems.

Quiz

* Consider the problem of predicting how well a student does in her 2nd year of college, given how well she did in her 1st year. Specifically, let x = # of "A" grades (A-, A, A+) a student receives in their 1st year of college. We’d like to predict the value of y = the # of "A" grades they get in their 2nd year. Here, each row is 1 training example. Recall that in linear regression, our hypothesis is hθ(x) = θ0 + θ1x + we use “m” to denote the # of training examples.



* For the training set given above, what is the value of m?
* **4**
* Suppose we set θ0=−1, θ1=0.5. What is hθ(4)?
* **1**
* Many substances that can burn (such as gasoline + alcohol) have a chemical structure based on carbon atoms, + for this reason they’re called hydrocarbons. A chemist wants to understand how the # of carbon atoms in a molecule affects how much energy is released when that molecule combusts (is burned). The chemist obtains the dataset below. In the column on the right, “kJ/mol” is the unit measuring the amount of energy released.



* You’d like to use linear regression (hθ(x) = θ0 + θ1x) to estimate the amount of energy released (y) as a function of the # of carbon atoms (x). Which of the following do you think will be the values you obtain for θ0 + θ1?
* **θ0=−569.6, θ1=−530.9**
* Let f be some function so that f(θ0,θ1) outputs a #. For this problem, f is some arbitrary/unknown smooth function (not necessarily the cost function of linear regression, so f *may* have local optima). Suppose we use gradient descent to try to minimize f(θ0,θ1) as a function of θ0 and θ1. Which of the following statements are true?
* **If θ0 + θ1 are initialized at a local minimum, then 1 iteration will not change their values.**
* **If α is too small, gradient descent may take a very long time to converge.**
* **If the first few iterations of gradient descent cause f(θ0,θ1) to *increase* rather than decrease, the most likely cause is that we have set the learning rate α to too large a value.**
* Suppose that for some linear regression problem, we have some training set, and for our training set we managed to find some θ0, θ1 such that J(θ0,θ1) = 0. Which of the statements below must then be true?
* **Our training set can be fit perfectly by a straight line**
* **We can perfectly predict the value of y for new examples we have not yet seen**