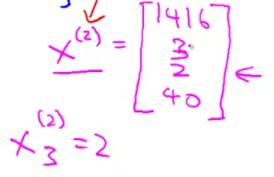
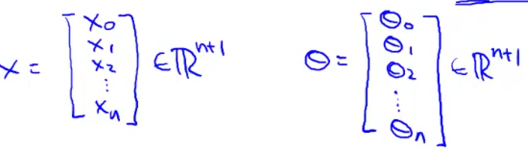
***Linear Regression with Multiple Variables***

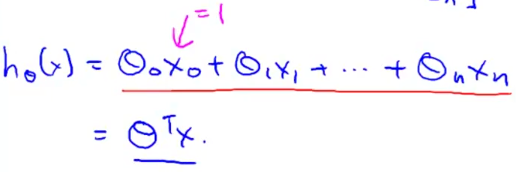
* Imagine if we had not only the size of a house as a **feature**/variable of which to try to predict the price, but also knew # of bedrooms +age of home in years 🡪 seems like this would give a lot more info w/ which to predict selling price.

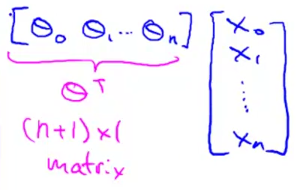


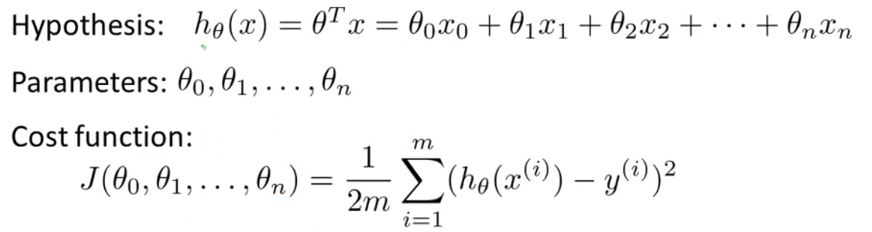


* For the convenience of notation, define x0 = 1 🡪 means the 1st item in each feature’s (j’s) vector of values = 1 🡪 so theta0 is always = theta0 \* 1
* Think of each set of x values as a vector and each set of coefficients/theta’s as another vector







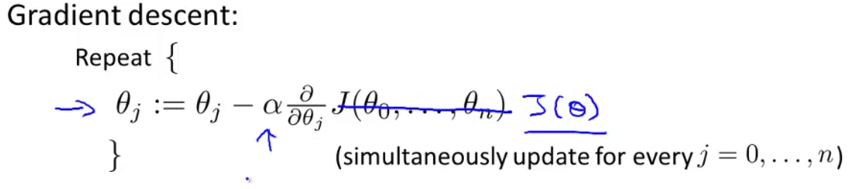


* New parameters of the model = theta0 through theta n, but instead of thinking of this as n separate parameters, which is valid, instead think of the parameters as *an n+1-dimensional vector.*
* And again, instead of thinking of J as a function of these n+1 numbers, think of J as a function of the parameter *vector*, called theta 🡪 J(theta)

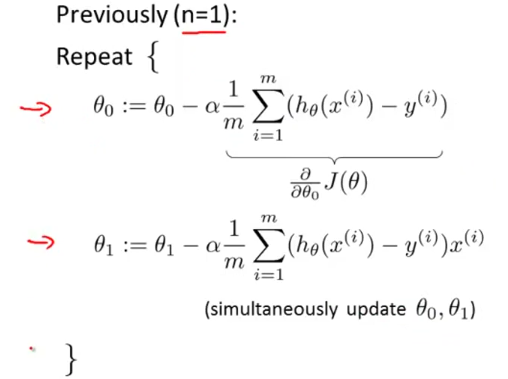




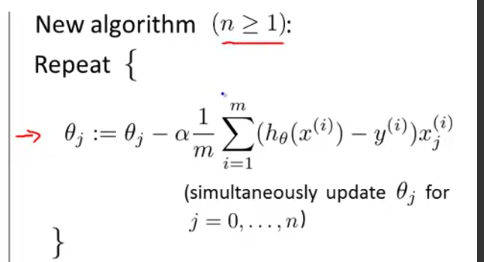
* *All of the above 3 are the same*



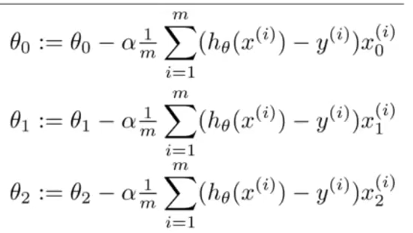
* So for gradient descent, we're going to repeatedly update each parameter thetaj according to thetaj – alpha\*the derivative term\*J(theta)
* So thetaj is updated as thetaj minus the **learning rate** times a partial derivative of the cost function w/ respect to the parameter theta j.

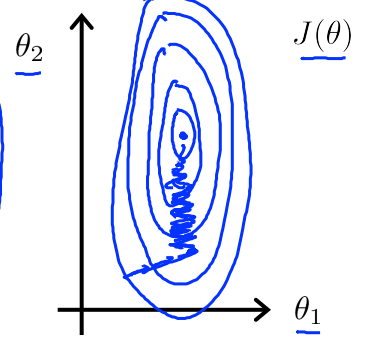


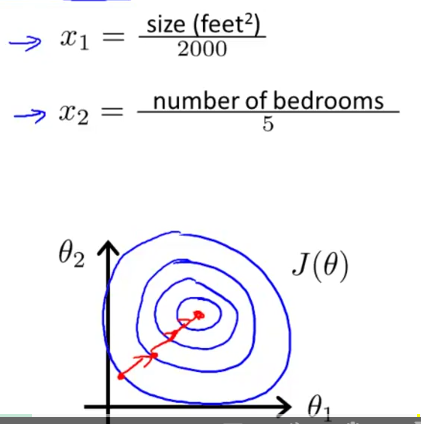
* Here's what we have for gradient descent for 1 feature = 2 separate update rules for parameters theta0 + theta1
* Now, where we previously had only 1 feature, x(i), we now have x(1)(i) to denote our 1st feature



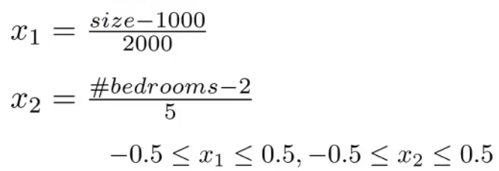
* Why these new and old algorithms sort of the same thing/are both similar algorithms/are both gradient descent algorithms?
* Consider a case where we have 2+ features 🡪 have 3 update rules for parameters theta0, theta1, theta2, maybe other values of theta as well.



* The update rules for theta0 and theta1 are the same as the single variable linear regression version.
* And now that we have more than 1 feature, we have similar update rules for other parameters
* **Feature Scaling** 🡪 If you have a problem w/ multiple features, if you make sure they are all on a similar scale (take on similar ranges of values), gradient descents can converge more quickly.
* Concretely let's say you have a problem w/ 2 features, X1 = size of house w/ values between 0-2k, X2 = # of bedrooms w/ values between 1-5.
* If you plot the contours of the cost function J, then the contours may look like this:
* 
* theta1 takes on a much larger range of values than theta2 🡪 contours of cost function J can take on a very skewed elliptical shape = very tall and skinny ellipses
* If you run gradient descents on this cost function, they may end up oscillating back and forth and taking a long time before it can finally find its way to the global minimum.
* In these settings, a useful thing to do is to **scale** the features.
* If you instead define the size of the house to be divided by 2k + define # of bedrooms to be divided by 5, the counters of the cost function J can become much less skewed + look more like circles.
* If you run gradient descent on a cost function like this, then w/ gradient descent, you can find a much more direct path to the global minimum rather than taking a convoluted path



* By scaling the features we end up w/ both features being between 0-1.
* More generally, when performing feature scaling, what we often want to do is get every feature into approximately a -1 - +1 range (concretely, feature x0 is always = 1, so, that's already in that range)
* If you end up w/ a feature between -2-0.5, this is close enough to -1 and +1 that its fine
* Too large of a range (-100 to +100) or too small of a range then (-0.0001 - 0.0001) = poorly scaled.
* Take-home message = don't worry if features are not *exactly* on the same scale, so long as they're all close enough, gradient descent should work okay.
* In addition to dividing by a #, sometimes people will also do **mean normalization**
* Take a feature Xi + replace it w/ Xi minus Mu(i) 🡪 observation – mean to make your features have approximately mean = 0
* Obviously we don’t want to apply this to x(0), b/c its always = 1, so it cannot have an average value of 0.



* A more general rule: take a feature X1 + replace it w/ (X1 - mu1) / S1 where S1 = range of values of that feature
* setting S1 = standard deviation of the variable would be fine, too
* Feature scaling = makes gradient descent run much faster + converge in lot fewer iterations.