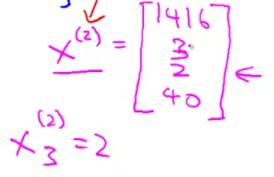
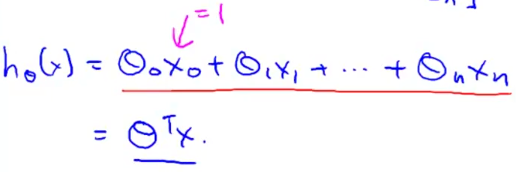
***Linear Regression with Multiple Variables***

**I. MULTIPLE FEATURES**

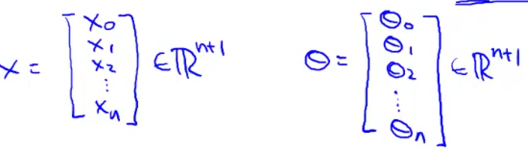
* Imagine if we had not only house size as a feature to use to try to predict the price, but also knew other features like # of bedrooms + age of home in years
* This would give a lot more info w/ which to predict selling price.
* n = number of features
* x(i) = features of the ith example/record (matrix of 4 feature values for record 2)
* x(i)(j) = value of feature j for the ith record (value of feature 3 for 2nd record)



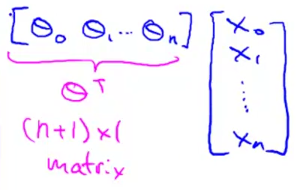
* New formula:



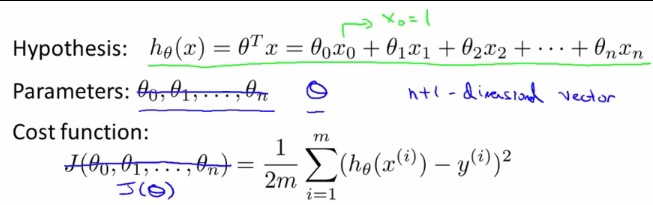
* For the convenience of notation, define x0 = 1 so the 1st feature value (j) for each record is 1 to make sure the intercept is not changed (*θ0 is always = θ0 \* 1*)
* x(i)(0) = 1 🡪 defining an additional **zero feature** vector that is always the value 1
* Think of each set of x values as a vector and each set of coefficients/θ’s as another vector
* *With the zero feature, we get an n+1 dimensional vector indexed at 0, and our parameters is also an n+1 dimensional vector indexed at 0*



* Can now write the hypothesis as the transposed parameter vector multiplied by out feature value vector

 =  = 

**II. GRADIENT DESCENT FOR MULTIPLE VARIABLES**

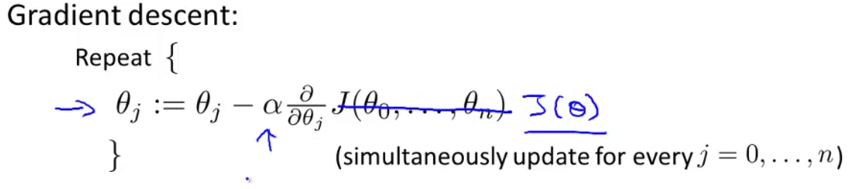


* New parameters of the model = θ0 through θ n, but instead of thinking of this as n separate parameters, which is valid, instead think of the parameters as *an n+1-dimensional vector.*
* And again, instead of thinking of J as a function of these n+1 numbers, think of J as a function of the parameter *vector*, called θ 🡪 J(θ)
* Can write this function in multiple ways:

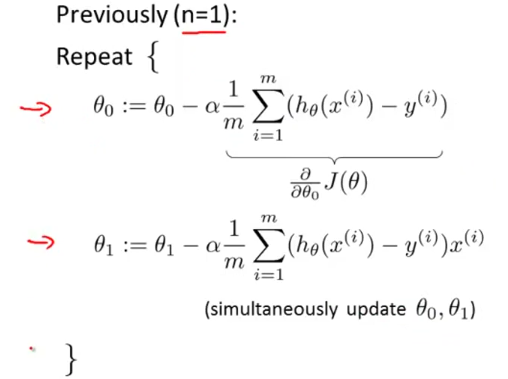


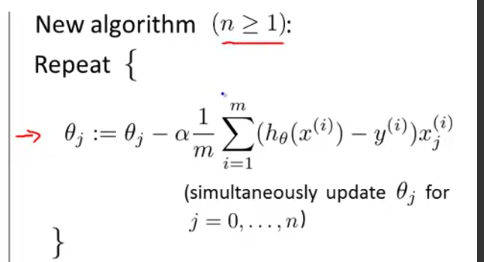


* *All of the above 3 are the same function*

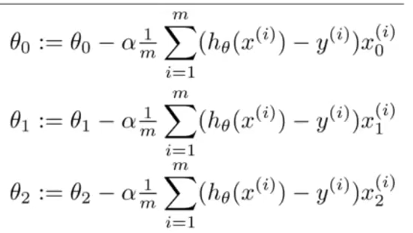


* So for gradient descent, we're going to repeatedly update *each* parameter θj (from θ0 to θn), according to θj – **α**\*partial derivative term\*J(θ) w/ respect to the parameter θj
* i.e. partial derivative of the cost function w/ respect to the current parameter
* Ex: of gradient descent for 2 parameters (1 feature): *2 separate update rules for parameters θ0 + θ1*

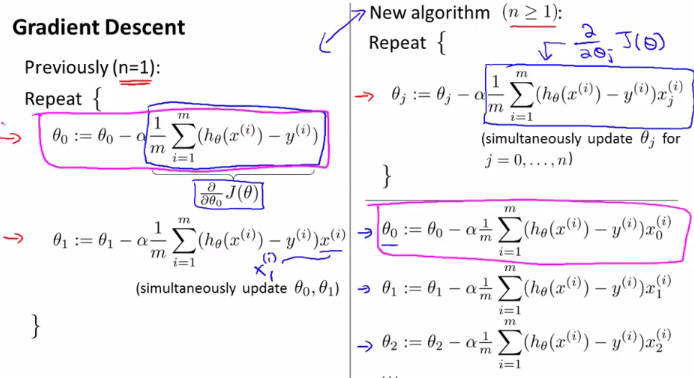




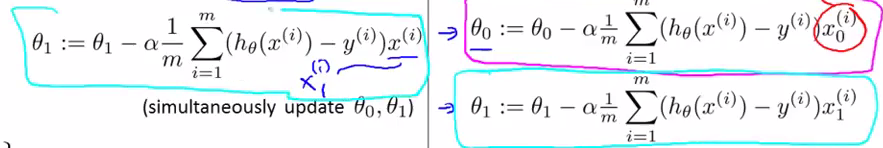
* So we’re taking the cost function multiplied by **α** and then *multiplied by x(i)(j)* to get the implementation of gradient descent for multiple linear regression
* Why are these new + old algorithms sort of the same thing/similar algorithms/both gradient descent algorithms?
* Consider a case where we have 2+ features 🡪 so we have 3 update rules for parameters θ0-θ2



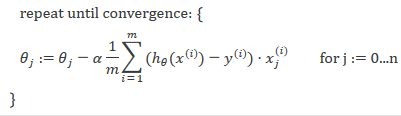
* The update rules for θ0 for multiple linear regression is the same as the update rule for θ0 for univariate linear regression *because x(i)(0) = 1*



* And now that we have more than 1 feature, we have similar update rules for other parameters

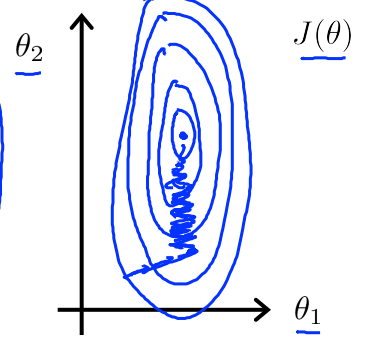


* Now we just have a subscript 1 on the right to denote the 1st feature for the record x
* i.e. x(i) = x(i)(1) 🡪 same record, but now for feature 1 instead of just “feature”
* So the gradient descent equation is the same, we just repeat it for n features

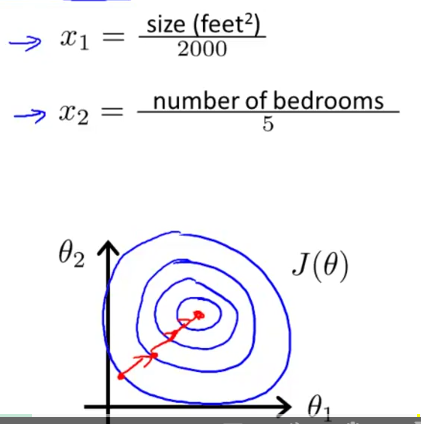
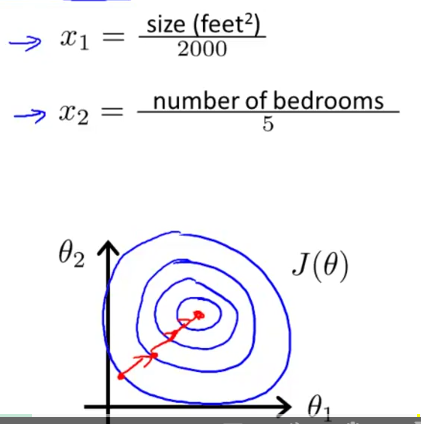


**III. GRADIENT DESCENT IN PRACTICE: FEATURE SCALING**

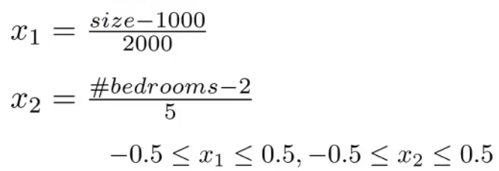
* **Feature Scaling** 🡪 For a problem w/ multiple features, make sure they’re all on a *similar scale* (take on similar ranges of values), so *gradient descent can converge more quickly* in a lot fewer iterations
* Ex: 2 features, X1 = size of house w/ values between 0-2k ft^2, X2 = # of bedrooms (between 1-5)
* If you plot the contours of a cost function J(θ0,θ1,θ2) for this problem, they may look like this:



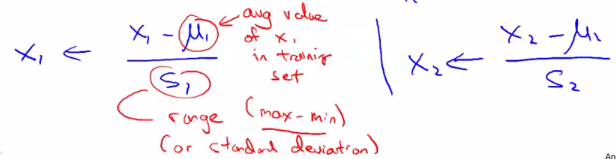
* X1 takes on a much larger range of values than X2, so the contours of J(θ) take on a very skewed elliptical shape (very tall and skinny ellipses in above)
* If you run gradient descent on *this* cost function, our gradient may end up oscillating back + forth + taking a long time before it can finally find its way to the global minimum.
* See the blue squiggles 🡪 very convoluted path w/ oscillations to the global min.
* If we exaggerate this even more (taller and skinner ellipses), gradient descent will take even longer
* 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 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



* See a much more direct path to a global min 🡪 gradient descent can mathematically find this path
* By scaling the features in this example, we end up w/ both features being between 0-1
* More generally, when performing feature scaling, we often want to get every feature into approximately a -1<x<1 range (concretely, feature x0 is always = 1, so, that's already in that range)
* If you end up w/ a feature between -2<x<0.5 or -2<x<3, this is close enough to -1<x<1 + is fine
* Too large (-100<x<100) or too small of a range then (-0.0001<x<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.
* Rule of thumb 🡪 -3<x<3 is a good range
* In addition to dividing by a #, sometimes people will also do **mean normalization**
* Take a feature X(i) + replace it w/ X(i) minus Mu(i)
* This is equivalently (observation – mean) + makes your features have approximately mean = 0
* Don’t want to apply this to X0 (b/c its always = 1), so it cannot have an average value of 0.

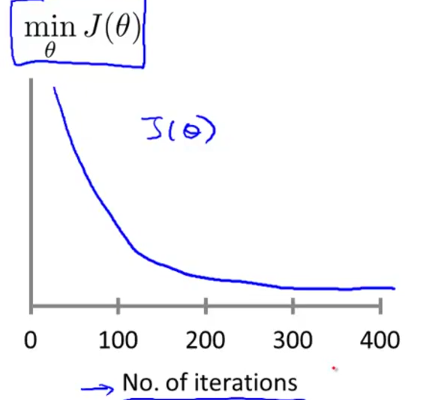


* More general rule: take feature X1 + replace w/ **(X1 - mu1) / S1** where S1 = feature’s range of values
* setting *S1 = standard deviation* of the variable would be fine, too

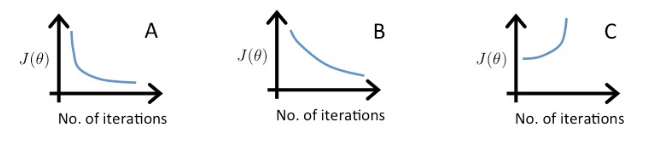


**IV. GRADIENT DESCENT IN PRACTICE: LEARNING RATE**

* Now to look into **debugging Gradient Descent** + choose the learning rate/**α**
* Remember, the *job of gradient descent is to minimize the cost function J(θ)*
* To make sure gradient descent is working, we can plot the value of the J(θ) over a # of iterations of gradient descent to make sure its decreasing



* If gradient descent it working properly, J(θ) should decrease after every iteration
* The # of iterations gradient descent takes to converge for an application can vary a lot
* For 1 application, it may converge after just 30, for a different one, it may take 3k, + 3M for another
* It is very difficult to tell in advance how many iterations gradient descent needs to converge.
* It's also possible to come up w/ an **automatic convergence test** = *have an algorithm try to tell you if gradient descent has converged*.
* Ex: if cost function J(θ) decreases by less than some small value *epsilon* (say 10^-3), then we declare convergence
* *It’s easier to look at plots rather than rely on an automatic convergence test.*
* Looking at plots can also give a warning if gradient descent is not working correctly (J(θ) = increases)
* If J(θ) is actually increasing, the most common cause for that is **α** is too big, so gradient descent may overshoot the minimum + overshoot again + so on = end up getting worse + worse
* Make sure the code doesn't have a bug of it, but usually a too-large value of **α** is the problem.
* Similarly sometimes you may also see J(θ) go down for a while then go up, then go down, + so on.
* A fix for something like this is *also to use a smaller value of alpha.*
* Under other assumptions about J(θ) that *do* hold true for linear regression, mathematicians have shown that if **α** is small enough, J(θ) should decrease on *every* iteration.
* If this *doesn't* happen, alpha's probably too big
* Also don't want your **α** to be *too small* 🡪 gradient descent can be too slow to convergence
* To summarize, if **α** = too small = slow convergence problem, + if **α** = too large, J(θ) may not decrease on every iteration + may not even converge.
* In order to debug this, often plotting J(θ) as a function of the # of iterations can help figure out what's going on.
* Try a range of values for **α** + for these different values, plot J(θ) as a function of # of iterations + pick the value of **α** that seems to be causing J(θ) to decrease rapidly.

Ex:

* Graph C = cost function increases 🡪 **α** rate too high (= 1), A + B converge, but B is much slower 🡪 **α** too small (= 0.01), so **α** A is best (= 0.1)

**V. FEATURES AND POLYNOMIAL REGRESSION**

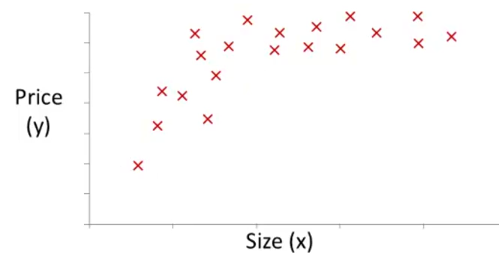
* Choices of **features** can get you different learning algorithms, some more powerful than others
* **Polynomial Regression** allows you to use the machinery of linear regression to fit very complicated, even very *non-linear* functions (how to fit a polynomial, like a quadratic or cubic function to data)
* Suppose you have 2 features, the frontage + depth of the property around your house



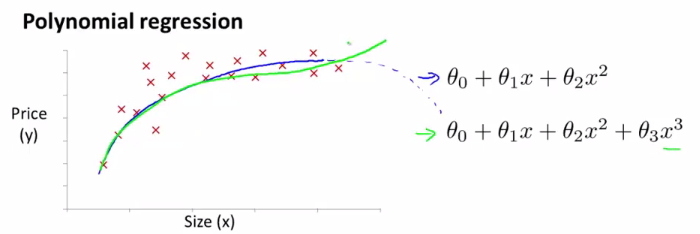
* You might build a linear regression model where frontage = feature x1 + depth = feature x2



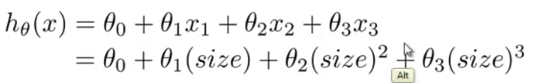
* When you're applying linear regression, you don't necessarily *have* to use *just* x1 + x2 (given)
* Can *create new features* = **feature engineering**
* Might decide that the *land area owned* is what really determines the price of a house 🡪 create new feature = *frontage \* depth =* x1\*x2
* Now one might change the hypothesis hθ(x) to just be a regression of just that new feature
* Depending on what insight you might have into a particular problem, rather than just taking given features, sometimes by defining *new* features, you might actually get a better model.
* Closely related to the idea of choosing your features is this idea **polynomial regression**.
* Let's say you have a housing price data set w/ a few different models you could fit to it.



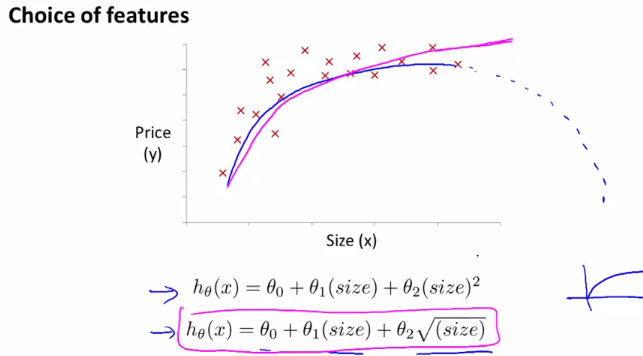
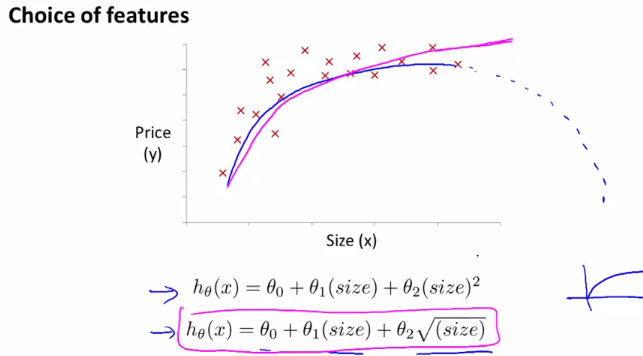
* 1: Fit a **quadratic model** 
* It doesn't look like a straight line fits this data very well
* We may decide a quadratic model *doesn't* make sense b/c quadratic functions eventually come back down + we don't think housing prices should go down when size goes up.
* 2: Fit a **cubic model** = w/ a third-order term
* Somewhat better fit to the data b/c it doesn't eventually come back down



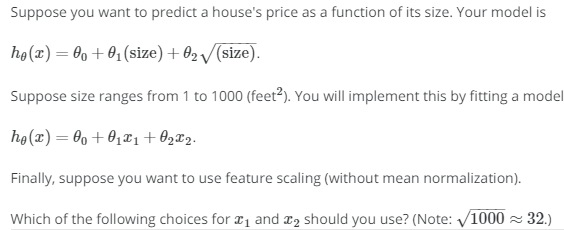
* So how do we actually fit a model like this to our data?
* Using **multivariate linear regression**, we can do this w/ a pretty simple modification to the linear regression algorithm.



* If you choose features like this, then ***feature scaling*** *becomes increasingly important*.
* If the size of the house ranges from 1 to 1k square feet, then the size of the house *squared* will range from 1 to *1M*, and the 3rd feature, size of the house cubed, will range from 1 to 10^9
* So these 3 features take on *very different* ranges of values + it's important to apply feature scaling if using gradient descent to get them into comparable ranges of values.
* We have broad choices in the features used.
* Like how a quadratic model might fit data okay, may go back down, but rather than going to a cubic model right away, there are many possible choices of features
* Another reasonable choice might be to say the price of a house = θ0 + θ1 + θ2\*sq root of the size
* Square root will let you take a quadratic model + flatten out a curve where it would decrease



* By choosing different features, you can sometimes get better models.



* *Size = 1000 🡪 x1 / 1000 = 1 sqrt(x2) = 32 🡪 32 / 32 = 1*