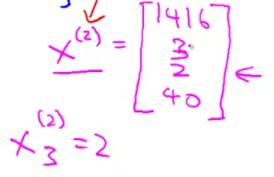
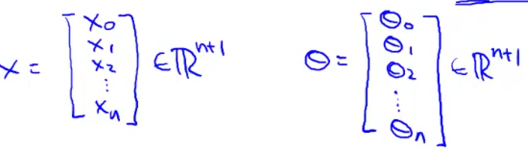
***Linear Regression with Multiple Variables***

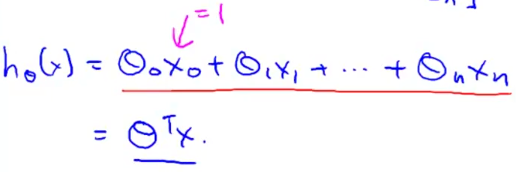
* Imagine if we had not only the size of a house as a **feature**/variable of which to try to predict the price, but also knew # of bedrooms +age of home in years 🡪 seems like this would give a lot more info w/ which to predict selling price.

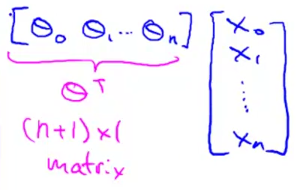


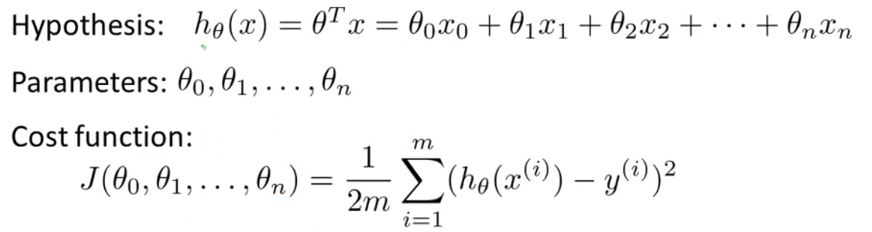


* For the convenience of notation, define x0 = 1 🡪 means the 1st item in each feature’s (j’s) vector of values = 1 🡪 so θ0 is always = θ0 \* 1
* Think of each set of x values as a vector and each set of coefficients/θ’s as another vector







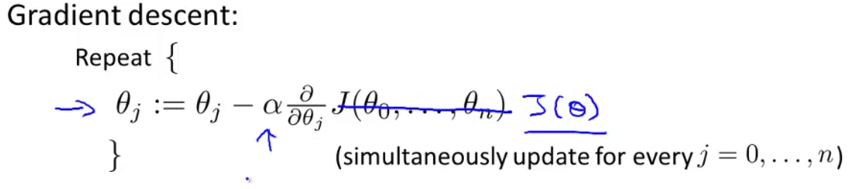


* New parameters of the model = θ0 through θ n, but instead of thinking of this as n separate parameters, which is valid, instead think of the parameters as *an n+1-dimensional vector.*
* And again, instead of thinking of J as a function of these n+1 numbers, think of J as a function of the parameter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

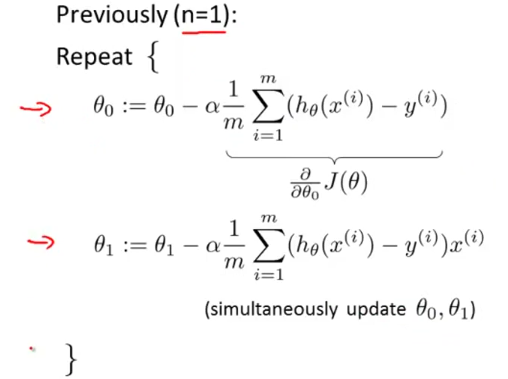




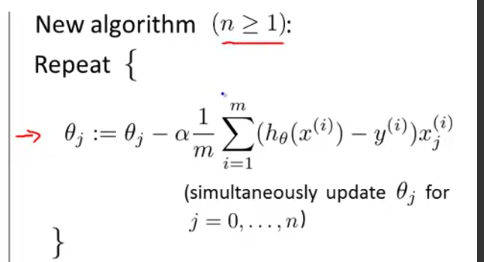
* *All of the above 3 are the same*



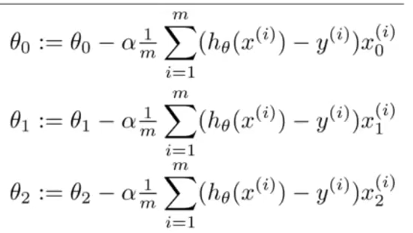
* So for gradient descent, we're going to repeatedly update each parameter θj according to θj – alpha\*the derivative term\*J(θ)
* So θj is updated as θj minus the **learning rate** times a partial derivative of the cost function w/ respect to the parameter θ j.

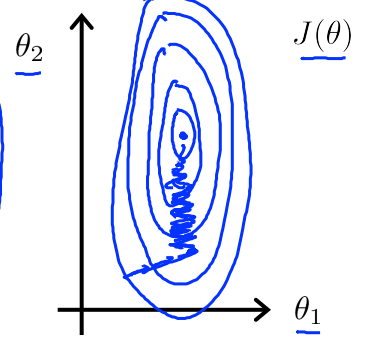


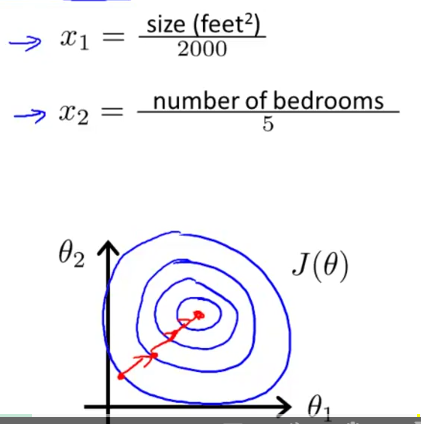
* Here's what we have for gradient descent for 1 feature = 2 separate update rules for parameters θ0 + θ1
* Now, where we previously had only 1 feature, x(i), we now have x(1)(i) to denote our 1st feature



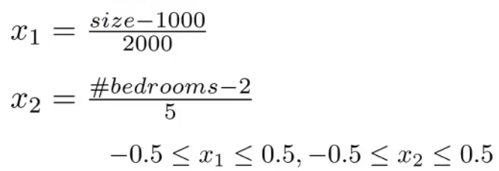
* Why these new and old algorithms sort of the same thing/are both similar algorithms/are both gradient descent algorithms?
* Consider a case where we have 2+ features 🡪 have 3 update rules for parameters θ0, θ1, θ2, maybe other values of θ as well.



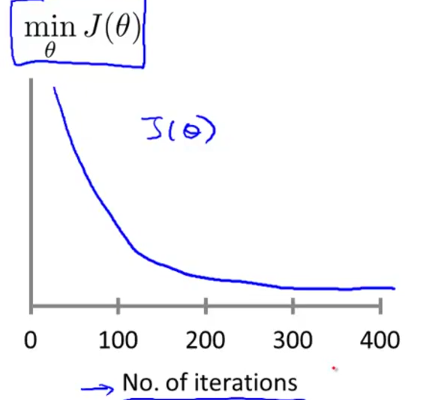
* The update rules for θ0 and θ1 are the same as the single variable linear regression version.
* And now that we have more than 1 feature, we have similar update rules for other parameters
* **Feature Scaling** 🡪 If you have a problem w/ multiple features, if you make sure they are all on a similar 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:
* 
* θ1 takes on a much larger range of values than θ2 🡪 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

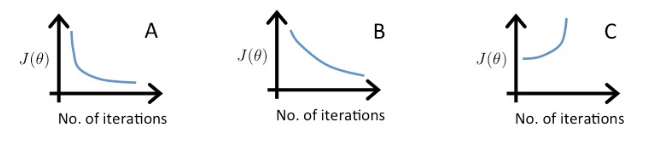


* By scaling the features we end up w/ both features being between 0-1.
* More generally, when performing feature scaling, what we often want to do is get every feature into approximately a -1 - +1 range (concretely, feature x0 is always = 1, so, that's already in that range)
* If you end up w/ a feature 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.



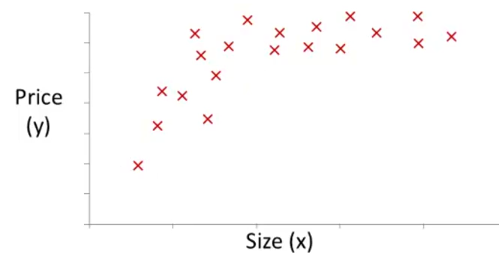
* More general rule: take feature X1 + replace w/ (X1 - mu1) / S1 where S1 = feature’s range of values
* setting S1 = standard deviation of the variable would be fine, too
* Feature scaling = makes gradient descent run much faster + converge in lot fewer iterations.
* Now to look into **debugging Gradient Descent** + choose the learning rate/alpha
* To make sure gradient descent is working, plot the value of the cost function J over the # of iterations to make sure its decreasing



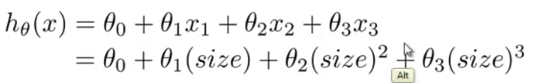
* The # of iterations gradient descent takes to converge for an application can vary a lot, so for 1, gradient descent may converge after just 30, for a different one, it may take 3k, and 3M for another
* It is very difficult to tell in advance how many iterations gradient descent needs to converge.
* It's also possible to come up w/ an **automatic convergence test** = *to 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 declare convergence
* 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) = increasing,
* If J(θ) is actually increasing, the most common cause for that is learning rate is too big = gradient descent may overshoot the minimum + overshoot again + so on = end up getting worse + worse
* Of course, make sure your code doesn't have a bug of it, but usually a too-large value of alpha could be the problem.
* Similarly sometimes you may also see J(θ) go down for a while then go up then go down for a while then go up, so on.
* A fix for something like this is *also to use a smaller value of alpha.*
* Under other assumptions about the cost function J, that *do* hold true for linear regression, mathematicians have shown that if learning rate alpha 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 learning rate to be *too small* 🡪 gradient descent can be slow to converge.
* To summarize, if learning rate = too small = slow convergence problem, + if learning rate = too large, J(θ) may not decrease on every iteration and it may not even converge.
* In order to debug these things, 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 alpha, like 0.001 or 0.003 and 0.01 or 0.03 (a factor of 10 difference) + for these different values, plot J(θ) as a function of # of iterations + pick the value of alpha that seems to be causing J(θ) to decrease rapidly.
* 
* Graph C = cost function increases 🡪 alpha rate too high (= 1), A + B converge, but B is much slower 🡪 alpha too small (= 0.01), so alpha A is best (= 0.1)
* *Choice* 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 (how wide your lot is) + depth of the house.
* You might build a linear regression model where frontage = feature x1 + depth = feature x2



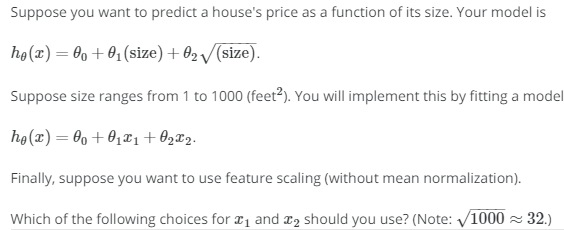
* When you're applying linear regression, you don't necessarily *have* to use *just* x1 + x2 (given)
* Can create new features = **feature engineering**
* Might decide that the land area owned really determines the size of the house 🡪 create new feature = frontage \* depth
* Now one might change hypothesis h(x) to just be a regression of that 1 new feature, land area
* 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** 
* b/c doesn't look like a straight line fits this data very well
* may decide quadratic model doesn't make sense b/c quadratic functions eventual come back down + we don't think housing prices should go down when size goes up too high.
* 2 = 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 the machinery of multivariant linear regression, we can do this w/ a pretty simple modification to our algorithm.



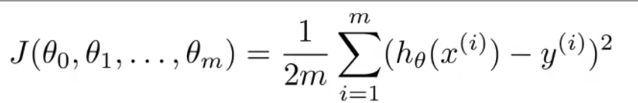
* If you choose your features like this, then *feature scaling becomes increasingly important*.
* If the size of the house ranges from 1 to 1k square feet, then the size of the house *squared* will range from 1 to *1M*, and your 3rd feature, size cubed of the house, will range from 1 to 10^9
* So these 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 \* the square 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.

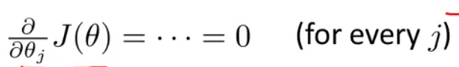




* *Size = 1000 🡪 x1 / 1000 = 1 sqrt(x2) = 32 🡪 32 / 32 = 1*
* **The normal equation 🡪** gives a much better way to solve for the optimal value of the parameters θ for some linear regression problems.
* So far, the algorithm we've been using for linear regression = gradient descent, where in order to minimize the cost function, we take an iterative algorithm that takes multiple iterations of gradient descent to converge to a global minimum.
* In contrast, the **normal equation** gives a method to solve for θ *analytically*, so rather than running an iterative algorithm, we just solve for the optimal value for θ all in one go
* Imagine a very simplified cost function J(θ) where θ is a real # 

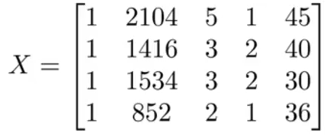
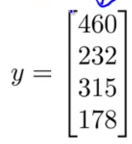


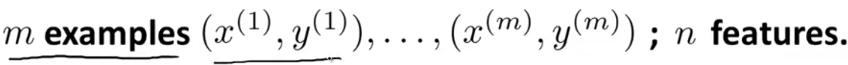
* For now, imagine θ is just a scalar value or a row value, just a #, rather than a vector.
* So we have a cost function = a quadratic function of a real-value parameter Θ
* How do you minimize a function? 🡪 = take derivatives + set them to 0
* Take derivative of J w/ respect to parameter Θ 🡪 set that derivative = 0 🡪 allows you to solve for the value of Theda that minimizes J(Θ).
* That was a simpler case of when θ was a real #.
* In the problem *we* are interested in, Θ is not a real #, but, instead, is an n+1-dimensional parameter vector 
* A cost function = a function of *this* vector = (Θ0,… Θ m)
* 
* How do we minimize *this* cost function?
* 1 way to do so is to take the **partial derivative** of J w/ respect to *every* parameter of Θ + to set all of these = 0.



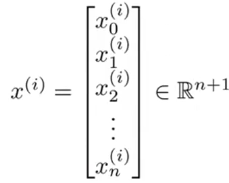
* Then solve for Θ 0 up to Θ m, this gives the values of Θ to minimize the cost function.
* If you actually work through the calculus + through the solution to the parameters Θ 0 through Θ N, the derivative ends up being somewhat long and involved.
* Only need to know a little in order to implement this algorithm and get it to work w/out heavy math
* Ex: m = 4 training examples.

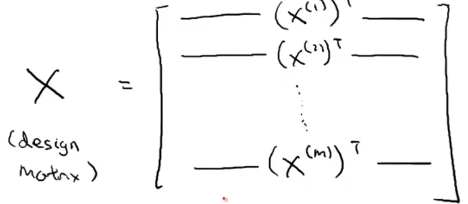


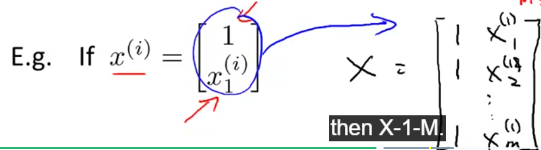
* In order to implement the normal equation method, take the data set + add an extra column that corresponds to an extra feature, x0, that always takes on value = 1.
* Then construct a matrix, X , which contains all features from the training data
* 
* Then we do the same for y's (values trying to predict)
* 
* X = an *m \* (n+1)*-*dimensional* matrix, Y = a *m-dimensional* vector, where m = # of training examples and n = # of original features, so we have n+1 b/c of the extra feature X0
* Calculate the inverse of (X(transpose)\*X) \* X(transpose)\*Y, this gives the value of θ that minimizes the cost function.
* 
* Θ ends up being a *n-dimensional vector*
* General case:



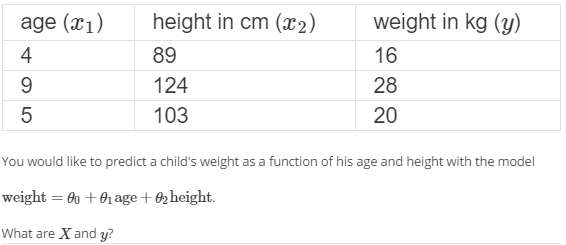
* M training examples + n features + each training example x(i) looks like an n+1 dimensional feature vector.



* Matrix "X" = **the design matrix** 🡪 take the transpose of the 1st training example vector (x(1)) + make it 1st first row of the design matrix and repeat until the last training example.
* 
* That makes an M by N+1 dimensional matrix.
* As a concrete example, let's say I have only 1 feature other than X zero (always = ).



* End up w/ an M by 2-dimensional matrix.





* Concretely, what is this X(T)\*X inverse = inverse of the matrix X(T)\*X.
* If using this normal equation method, then *feature scaling isn't actually necessary*
* If using gradient descent, *features scaling is still important*.
* Advantages of gradient descent
* Works pretty well even when n (# of features) is large
* Disadvantage of gradient descent
* need to choose the learning rate Alpha.
* Often, this means running GD a few times w/ different alphas + seeing what works best
* it needs many more iterations.
* Advantages of normal equation:
* don't need to choose learning rate alpha = simple to implement = just run it + it usually works
* don't need to iterate (don't need to plot J(Θ), check convergence, or take those extra steps)
* Disadvantages of the normal equation:
* In order to solve for the parameter θ, we need to solve for X(transpose)\*X inverse (n \* n matrix
* For most computed implementations, the cost of inverting a matrix grows roughly as much as the cube of the dimension of the matrix 🡪 
* So, computing the inverse costs roughly order \* n^cubed time
* Sometimes, it's slightly faster than n^cubed
* If # of features n is very large, computing that matrix can be slow + the normal equation method will be much slower.
* What does small and large mean?
* If n is on the order of a hundred, then inverting a 100x100 matrix is no problem by computing standards.
* If n = 1k, still use normal equation method b/c Inverting a 1000-by-1000 matrix is actually really fast on a modern CPU.
* If n = 10k, inverting a 10k x 10k matrix starts to get kind of slow 🡪 might start to lean in the direction of gradient descent, but maybe not quite.
* But if it gets much bigger than that, probably use gradient descent.
* So, if n = 1-^6 (1M features), then inverting a 1M x 1M matrix = very expensive = favor GD
* It’s hard to give a strict # of how large set of features has to be before converting to GD
* So long as n is not too large, the normal equation gives us a great alternative method to solve for the parameter θ.
* There's a phenomenon, **non-invertibility**, that you may run into that may be somewhat useful to understand
* 
* **Issue:** What if the matrix X(t)\*X is **non-invertible?**
* Some matrices do not have an inverse 🡪 **Singular** or **degenerate matrices**.
* The issue of X(t)\*X being non-invertible should only rarely happen rarely
* If X(t)&X is non-invertible, there are usually 2 common causes for this.
* 1: Somehow in your learning problem, you have *redundant features*
* If you're trying to predict housing prices + if x1 = size of the house in square feet + x2 = size of the house in square *meters*
* 1 meter = 3.28 feet, so your 2 features will always satisfy the constraint x1 = 3.28\*x2.
* You can actually show that if your 2 features are related via the above linear equation
* 2: If in training + trying to run the learning algorithm w/ too many features (*if m <= n*)
* Imagine m = 10 training example +n = 100 features
* Trying to fit a parameter back to θ, which is n+1-dimensional = 101-dimensional, you're trying to fit 101 parameters from just 10 training examples.
* This turns out to work *sometimes*, but its not always be a good idea.
* Might not have enough data if you only have 10 examples to fit 101 parameters to
* Commonly, if m <= n, we see if we can either delete some features or we use a technique called **regularization** that lets you fit a lot of parameters/use a lot features, even if you have a relatively small training set.
* To summarize, if ever you find that x(t)\*x is singular/find it non-invertible, 1st look at your features + see if you have redundant features = You're being **linearly dependent** 🡪 just delete 1 of these features
* If so, keep deleting redundant features until they're no longer redundant
* If your features are not redundant, check if we may have too many features + if so, either delete some features (if we can bear to use fewer features) or else consider using regularization. Which is this topic that we'll talk about later.
* X(t)\*X being non-invertible should happen pretty rarely + should not be a problem for most implementations of linear regression.

1. Suppose m=4 students have taken some class, and the class had a midterm exam and a final exam. You have collected a dataset of their scores on the two exams, which is as follows:

midterm exam (midterm exam)2 final exam

89 7921 96

72 5184 74

94 8836 87

69 4761 78

Use polynomial regression to predict a student's final score from their midterm score. Suppose you want to fit a model of form hθ(x)=θ0+θ1x1+θ2x2, where x1 = midterm score + x2 = (midterm score)2^. Further, we use *both* feature scaling (dividing by range of a feature) *and* mean normalization. What is the normalized feature x(1)1? (Hint: midterm = 89, final = 96 is training example 1.) Please round off your answer to two decimal places and enter in the text box below.

* **0.32**

1. You run gradient descent for 15 iterations w/ α=0.3 + compute J(θ) after each + find that the value of J(θ) increases over time. Based on this, which of the following conclusions seems most plausible?

* **Rather than use current value of α, it'd be more promising to try a smaller value (say α=0.1)**

1. You run gradient descent for 15 iterations w/ α=0.3 and compute J(θ) after each iteration. You find the value of J(θ) decreases quickly then levels off. Based on this, which of the following conclusions seems most plausible?

* **α=0.3 is an effective choice of learning rate.**

1. Suppose you have m=23 training examples w/ n=5 features (excluding the additional all-ones feature for the intercept term, which you should add). The normal equation is θ=(XTX)−1XTy. For the given values of m and n, what are the dimensions of θ, X, and y in this equation?

* **X is 23×6, y is 23×1, θ is 6×1 🡪** X = m \* n+1, y = m, θ = n

1. Suppose you have m=50 examples + n=15 features for each example. You want to use multivariate linear regression to fit the parameters θ to our data. Should you prefer gradient descent or the normal equation?

* **The normal equation, since it provides an efficient way to directly find the solution.**

1. Suppose you have m=1000000 examples + n=200000 features for each example. You want to use multivariate linear regression to fit the parameters θ to our data. Should you prefer gradient descent or the normal equation?

* **Gradient descent, since (XTX)−1 will be very slow to compute in the normal equation 🡪** *n too high*

1. Which of the following are reasons for using feature scaling?

* **It speeds up gradient descent by making it require fewer iterations to get to a good solution.**