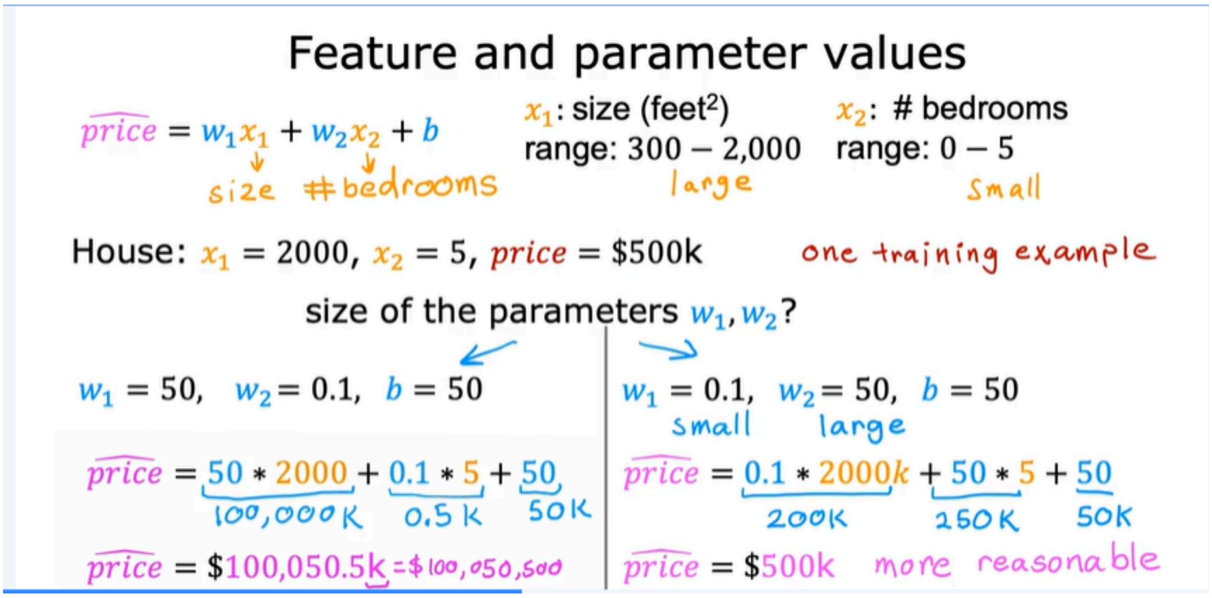
Gradient Descent in practice

Feature scaling

- In this video you see a technique called feature scaling that will enable gradient descent to run much faster.



Feature and parameter values

$\widehat{price} = \underset{\text{size}}{w_1 x_1} + \underset{\text{\#bedrooms}}{w_2 x_2} + b$

x_1: size (feet^2)
range: 300 – 2,000
large

x_2: # bedrooms
range: 0 – 5
small

House: $x_1 = 2000$, $x_2 = 5$, $\widehat{price} = \$500k$ one training example

size of the parameters w_1, w_2?

$w_1 = 50, \quad w_2 = 0.1, \quad b = 50$

$\widehat{price} = \underbrace{50 * 2000}_{100{,}000K} + \underbrace{0.1 * 5}_{0.5K} + \underbrace{50}_{50K}$

$\widehat{price} = \$100{,}050.5k = \$100{,}050{,}500$

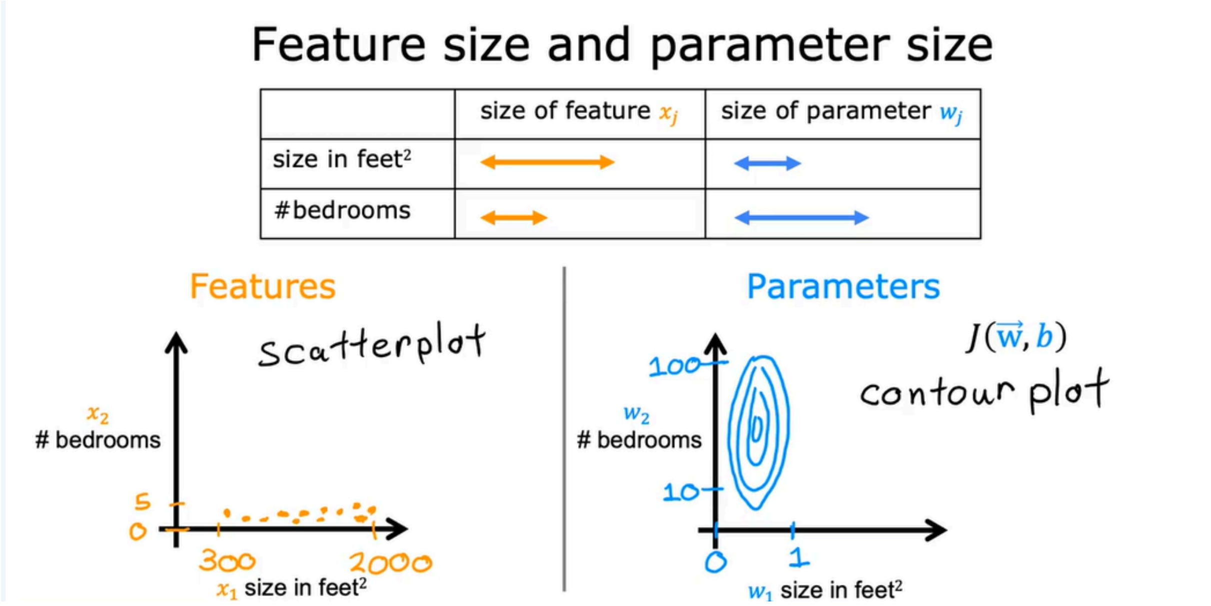
$w_1 = 0.1, \quad w_2 = 50, \quad b = 50$
small large

$\widehat{price} = \underbrace{0.1 * 2000k}_{200K} + \underbrace{50 * 5}_{250K} + \underbrace{50}_{50K}$

$\widehat{price} = \$500k$ more reasonable

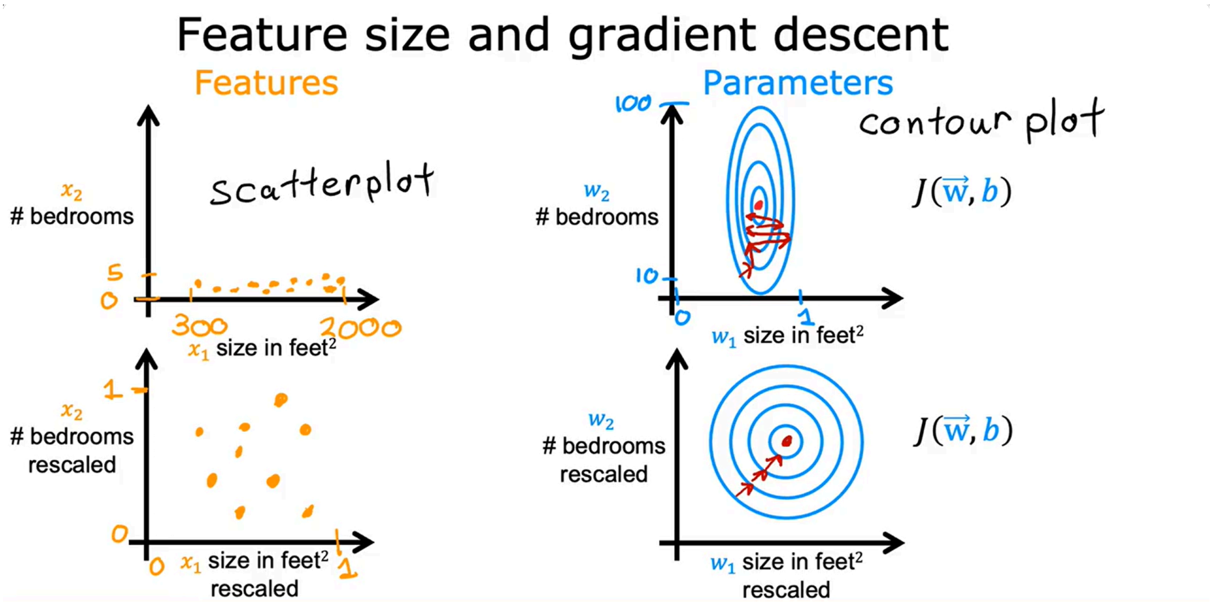
- Now let's take an example of a house that has a size of 2000 square feet has five bedrooms and a price of 500k or $500,000. For this one training example, what do you think are reasonable values for the size of parameters w1 and w2?
- Say w1 is 50 and w2 is 0.1 and b is 50 for the purposes of discussion. So in this case the estimated price in thousands of dollars is 100,000k here plus 0.5 k plus 50 k. Which is slightly over 100 million dollars. So that's clearly very far from the actual price of $500,000. And so this is not a very good set of parameter choices for w1 and w2.
- Now let's take a look at another possibility. Say w1 and w2 were the other way around. W1 is 0.1 and w2 is 50 and b is still also 50. In this choice of w1 and w2, w1 is relatively small and w2 is relatively large, 50 is much bigger than 0.1. So here the predicted price is 0.1 times 2000 plus 50 times five plus 50. The first term becomes 200k, the second term becomes 250k, and the plus 50. So this version of the model predicts a price of $500,000

which is a much more reasonable estimate and happens to be the same price as the true price of the house.



Feature size and parameter size

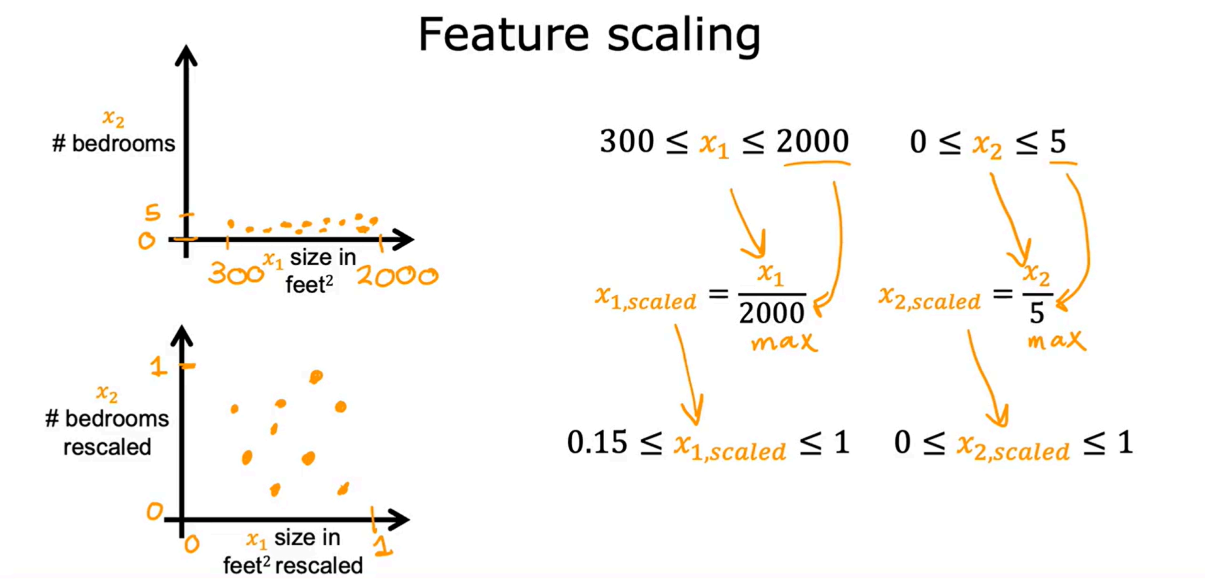
	size of feature x_j	size of parameter w_j
size in feet^2	⟷	↔
#bedrooms	↔	⟷



Feature size and gradient descent

when you have different features that take on very different ranges of values, it can cause gradient descent to run slowly but re scaling the different features so they all take on comparable range of values. because speed, upgrade and dissent significantly.

How to implement feature scaling



Feature scaling

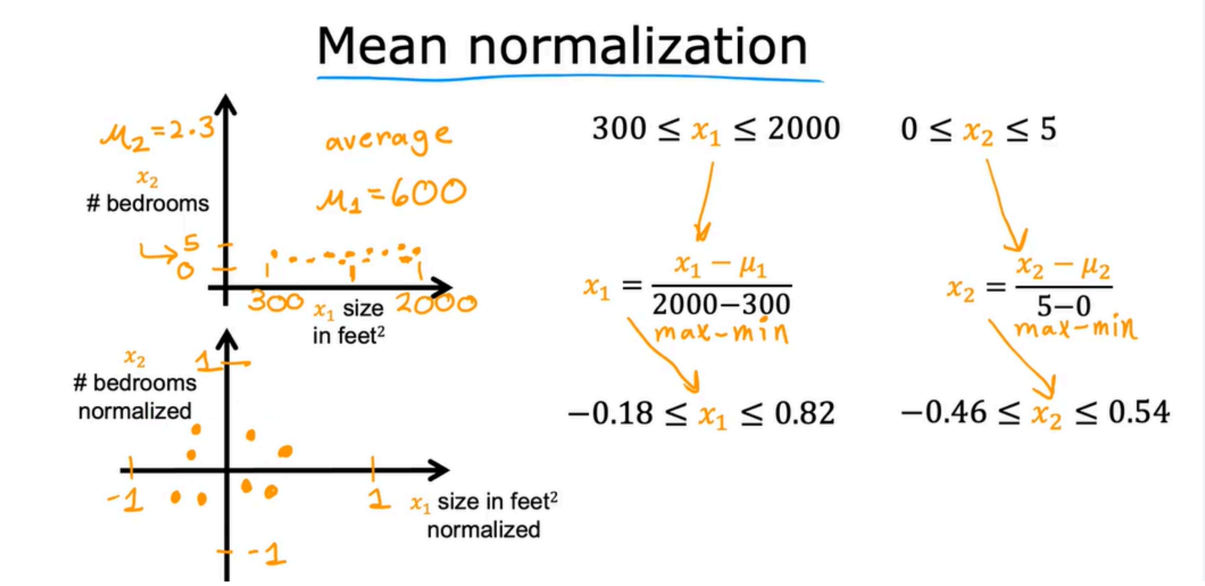
$300 \leq x_1 \leq 2000$ $\quad 0 \leq x_2 \leq 5$

$x_{1,scaled} = \frac{x_1}{2000}$ max $\quad x_{2,scaled} = \frac{x_2}{5}$ max

$0.15 \leq x_{1,scaled} \leq 1$ $\quad 0 \leq x_{2,scaled} \leq 1$

- Divide the feature value by “the maximum of the range”

Mean normalization



Mean normalization

$300 \leq x_1 \leq 2000$ $\quad 0 \leq x_2 \leq 5$

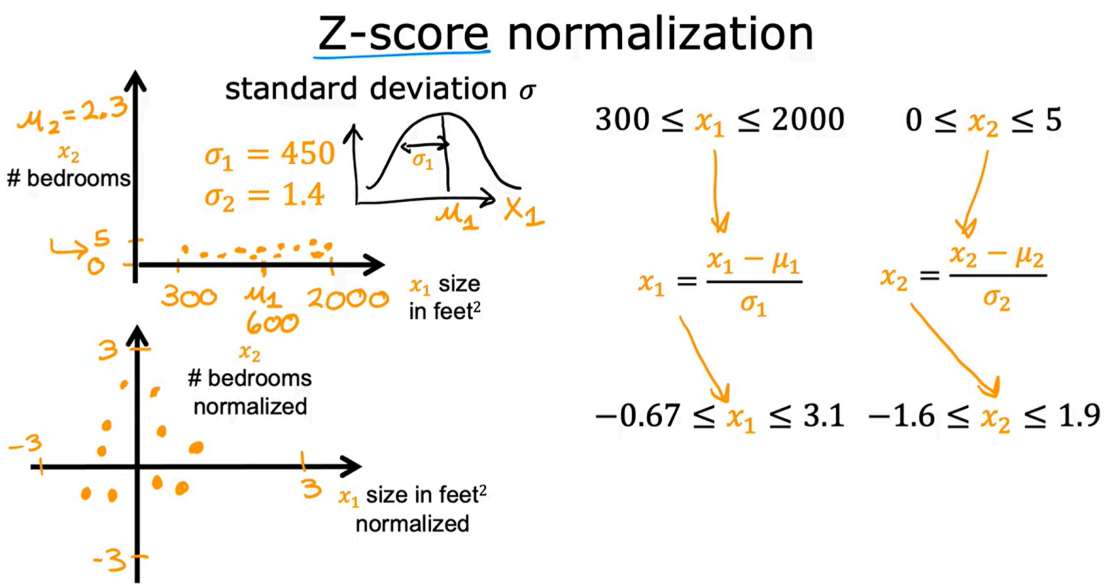
$x_1 = \frac{x_1 - \mu_1}{2000-300}$ max-min $\quad x_2 = \frac{x_2 - \mu_2}{5-0}$ max-min

$-0.18 \leq x_1 \leq 0.82$ $\quad -0.46 \leq x_2 \leq 0.54$

- Whereas before they only had values greater than zero, now they have both negative and positive values that may be usually between negative one and plus one.
- To calculate the mean normalization of x_1, first find the average, also called the mean of x_1 on your training set, and let's call this mean Mu_1, with this being the Greek alphabets Mu.

- For example, you may find that the average of feature 1, Mu_1 is 600 square feet. Let's take each x_1, subtract the mean Mu_1, and then let's divide by the difference 2,000 minus 300, where 2,000 is the maximum and 300 the minimum, and if you do this, you get the normalized x_1 to range from negative 0.18-0.82.
- Similarly, to mean normalized x_2, you can calculate the average of feature 2. For instance, Mu_2 may be 2.3. Then you can take each x_2, subtract Mu_2 and divide by 5 minus 0. Again, the max 5 minus the mean, which is 0. The mean normalized x_2 now ranges from negative 0.46-0 54. If you plot the training data using the mean normalized x_1 and x_2, it might look like this.

Z-score normalization

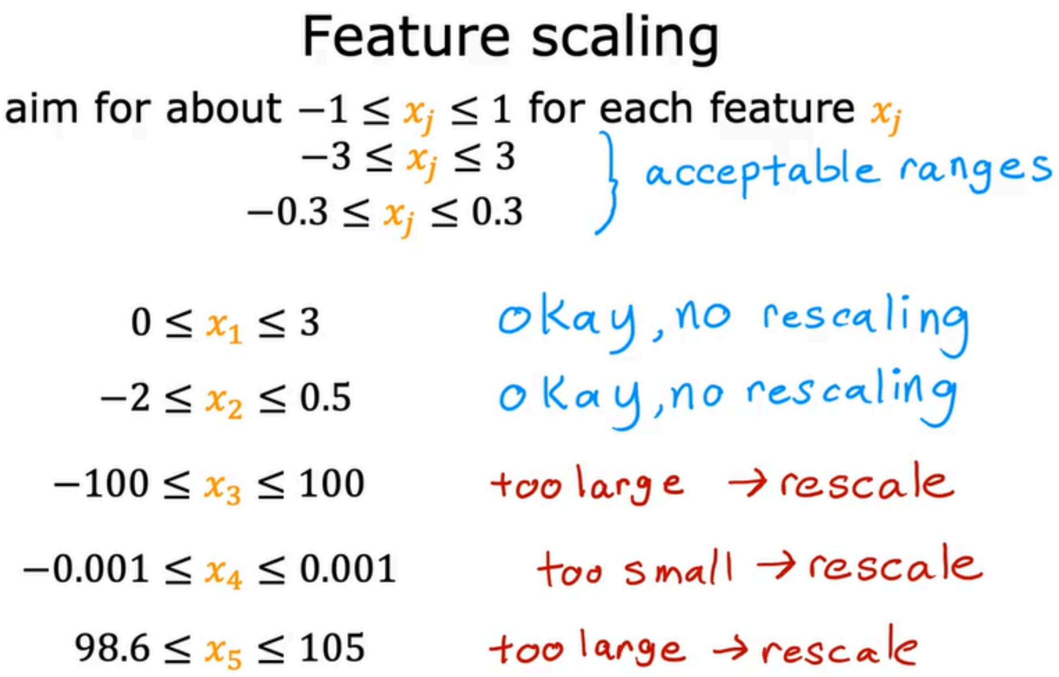


$300 \leq x_1 \leq 2000$ $\quad$ $0 \leq x_2 \leq 5$

$$x_1 = \frac{x_1 - \mu_1}{\sigma_1} \qquad x_2 = \frac{x_2 - \mu_2}{\sigma_2}$$

$-0.67 \leq x_1 \leq 3.1$ $\quad$ $-1.6 \leq x_2 \leq 1.9$

- Find mean
- Find standard deviation
- Then rescale



Feature scaling

aim for about $-1 \le x_j \le 1$ for each feature x_j

$-3 \le x_j \le 3$

$-0.3 \le x_j \le 0.3$ } acceptable ranges

$0 \le x_1 \le 3$ okay, no rescaling

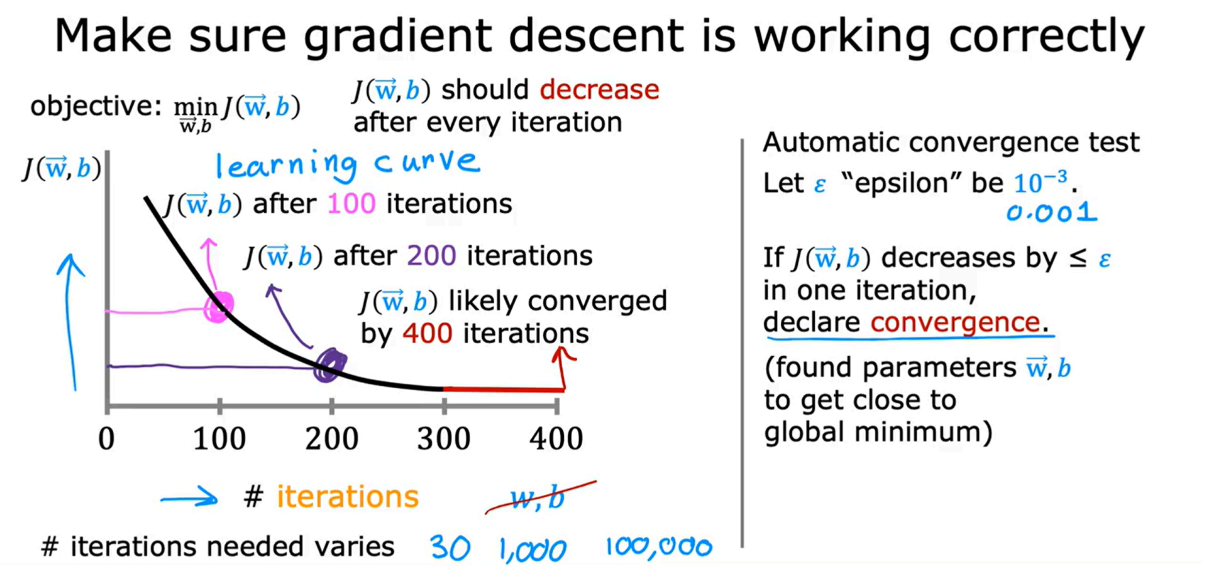
$-2 \le x_2 \le 0.5$ okay, no rescaling

$-100 \le x_3 \le 100$ too large → rescale

$-0.001 \le x_4 \le 0.001$ too small → rescale

$98.6 \le x_5 \le 105$ too large → rescale

Checking gradient descent for convergence



Make sure gradient descent is working correctly

objective: $\min_{\vec{w},b} J(\vec{w},b)$

$J(\vec{w},b)$ should decrease after every iteration

learning curve

$J(\vec{w},b)$ after 100 iterations

$J(\vec{w},b)$ after 200 iterations

$J(\vec{w},b)$ likely converged by 400 iterations

iterations needed varies 30 1,000 100,000

Automatic convergence test

Let ε "epsilon" be 10^{-3}. 0.001

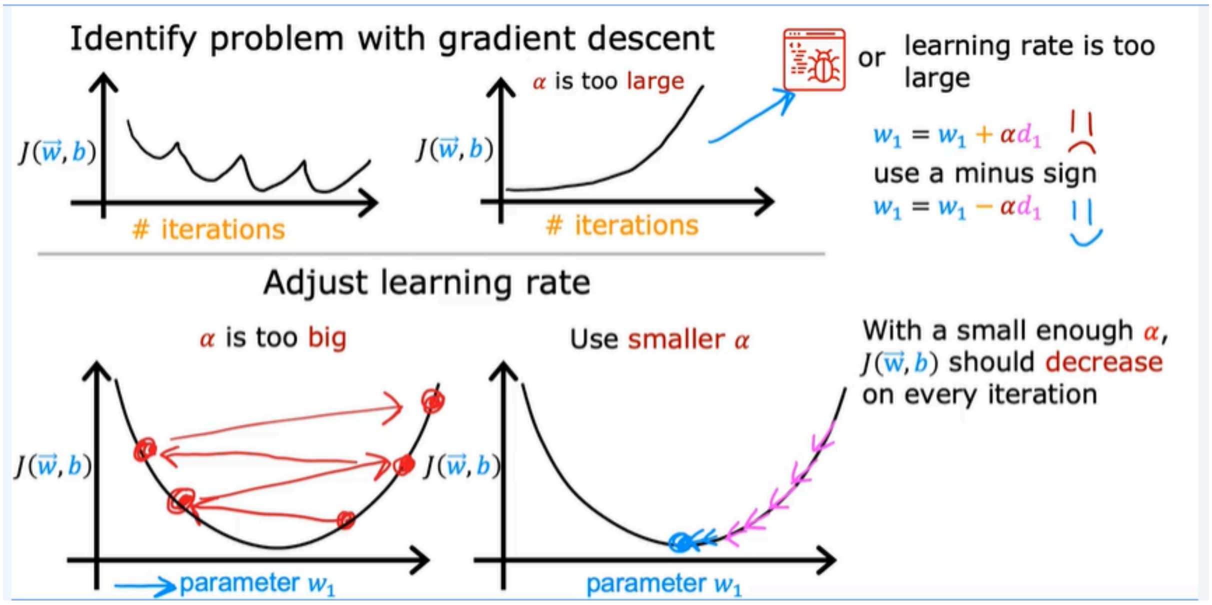
If $J(\vec{w},b)$ decreases by $\le \varepsilon$ in one iteration, declare convergence.

(found parameters $\vec{w}, b$ to get close to global minimum)

- If gradient descent is working properly, then the cost J should decrease after every single iteration.
- If J ever increases after one iteration, that means either Alpha is chosen poorly, and it usually means Alpha is too large, or there could be a bug in the code.

- Another useful thing that this part can tell you is that if you look at this curve, by the time you reach maybe 300 iterations also, the cost J is leveling off and is no longer decreasing much. By 400 iterations, it looks like the curve has flattened out. This means that gradient descent has more or less converged because the curve is no longer decreasing.
- Looking at this learning curve, you can try to spot whether or not gradient descent is converging. By the way, the number of iterations that gradient descent takes a conversion can vary a lot between different applications.
- For a different application, it could take 1,000 or 100,000 iterations. It turns out to be very difficult to tell in advance how many iterations gradient descent needs to converge, which is why you can create a graph like this, a learning curve. Try to find out when you can start training your particular model. Another way to decide when your model is done training is with an automatic convergence test.

Choosing the learning rate



Identify problem with gradient descent

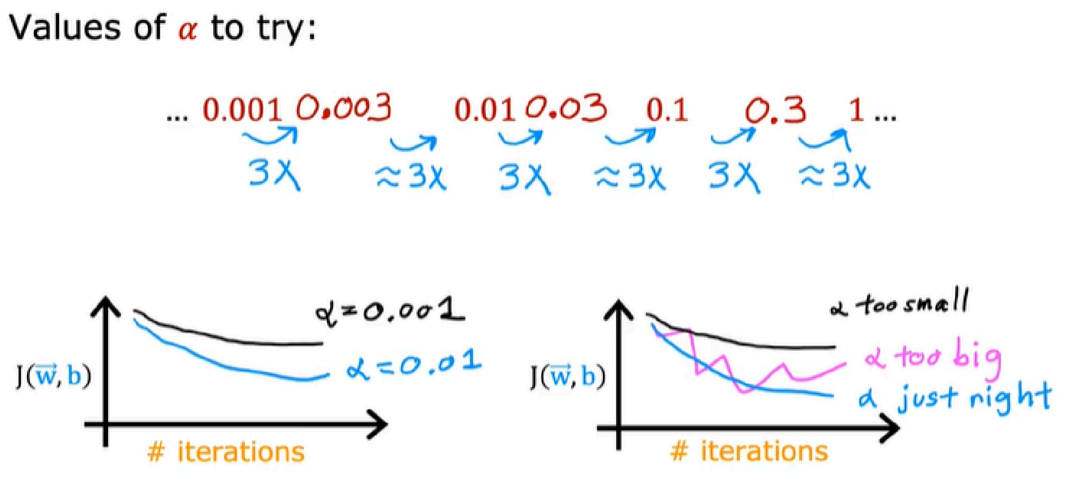
Adjust learning rate

With a small enough α, $J(\vec{w},b)$ should decrease on every iteration

> If α is too small, gradient descent takes a lot more iterations to converge

- One debugging tip for a correct implementation of gradient descent is that with a small enough learning rate, the cost function should decrease on every single iteration

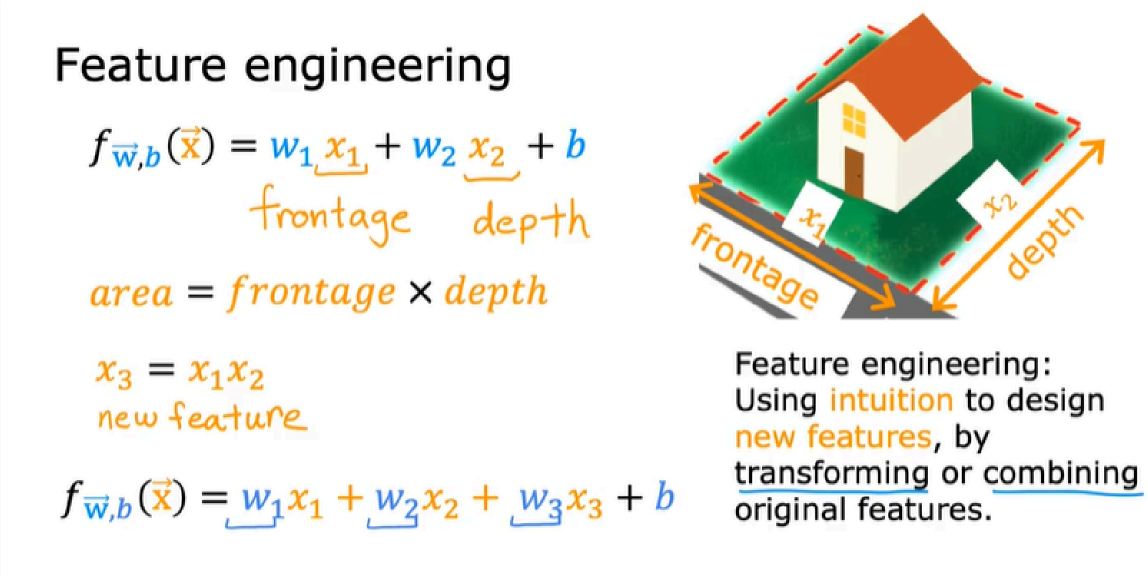
- So if gradient descent isn't working, one thing I often do and I hope you find this tip useful too, one thing I'll often do is just set Alpha to be a very small number and see if that causes the cost to decrease on every iteration.
- If even with Alpha set to a very small number, J doesn't decrease on every single iteration, but instead sometimes increases, then that usually means there's a bug somewhere in the code.
- Note that setting Alpha to be really small is meant here as a debugging step and a very small value of Alpha is not going to be the most efficient choice for actually training your learning algorithm.



- So when I am running gradient descent, I will usually try a range of values for the learning rate Alpha. I may start by trying a learning rate of 0.001 and I may also try learning rate as 10 times as large say 0.01 and 0.1 and so on. For each choice of Alpha, you might run gradient descent just for a handful of iterations and plot the cost function J as a function of the number of iterations and after trying a few different values, you might then pick the value of Alpha that seems to decrease the learning rate rapidly, but also consistently.
- In fact, what I actually do is try a range of values like this. After trying 0.001, I'll then increase the learning rate threefold to 0.003. After that, I'll try 0.01, which is again about three times as large as 0.003. So these are roughly trying out gradient descents with each value of Alpha being roughly three times bigger than the previous value.

- What I'll do is try a range of values until I found the value of that's too small and then also make sure I've found a value that's too large. I'll slowly try to pick the largest possible learning rate, or just something slightly smaller than the largest reasonable value that I found. When I do that, it usually gives me a good learning rate for my model. I hope this technique too will be useful for you to choose a good learning rate for your implementation of gradient descent.

Feature Engineering



Feature engineering

$$f_{\vec{w},b}(\vec{x}) = w_1 \underbrace{x_1}_{\text{frontage}} + w_2 \underbrace{x_2}_{\text{depth}} + b$$

$$area = frontage \times depth$$

$$\underbrace{x_3 = x_1 x_2}_{\text{new feature}}$$

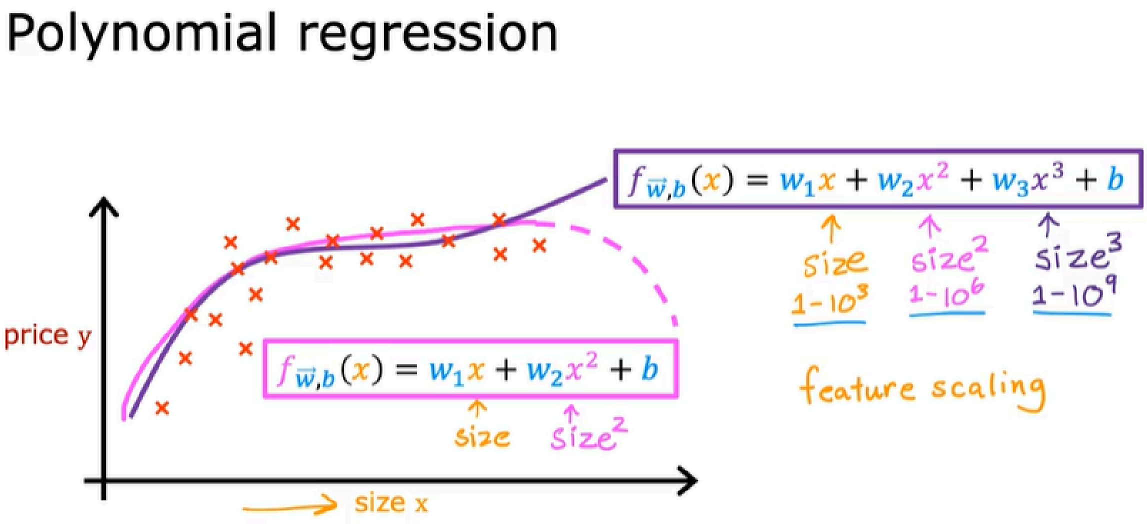
$$f_{\vec{w},b}(\vec{x}) = \underbrace{w_1}x_1 + \underbrace{w_2}x_2 + \underbrace{w_3}x_3 + b$$

Feature engineering:
Using intuition to design new features, by transforming or combining original features.

- Creating a new feature is an example of what's called feature engineering, in which you might use your knowledge or intuition about the problem to design new features usually by transforming or combining the original features of the problem in order to make it easier for the learning algorithm to make accurate predictions.
- Depending on what insights you may have into the application, rather than just taking the features that you happen to have started off with sometimes by defining new features, you might be able to get a much better model. That's feature engineering.
- It turns out that this one flavor of feature engineering, that allow you to fit not just straight lines, but curves, non-linear functions to your data.

Polynomial regression

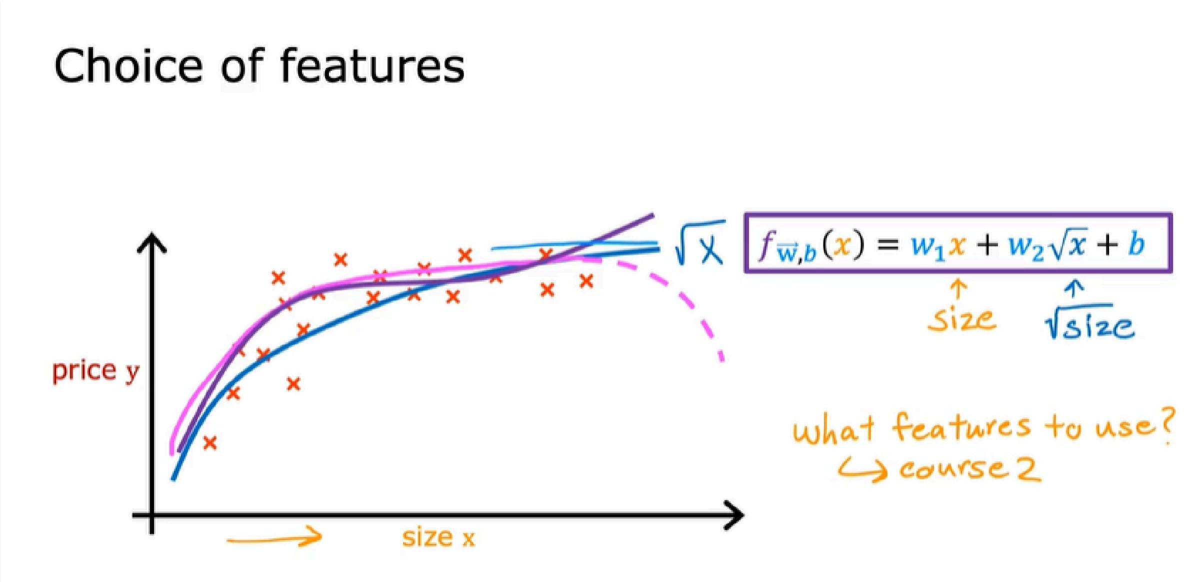
So far we've just been fitting straight lines to our data. Let's take the ideas of multiple linear regression and feature engineering to come up with a new algorithm called polynomial regression, which will let you fit curves, non-linear functions, to your data.



- Let's say you have a housing data-set that looks like this, where feature x is the size in square feet. It doesn't look like a straight line fits this data-set very well.
- Maybe you want to fit a curve, maybe a quadratic function to the data like this which includes a size x and also x^2, which is the size raised to the power of two. Maybe that will give you a better fit to the data. (the pink curve)
- But then you may decide that your quadratic model doesn't really make sense because a quadratic function eventually comes back down. Well, we wouldn't really expect housing prices to go down when the size increases. Big houses seem like they should usually cost more.
- Then you may choose a cubic function where we now have not only x squared, but x cubed. Maybe this model produces this curve here (the purple curve), which is a somewhat better fit to the data because the size does eventually come back up as the size increases.
- These are both examples of polynomial regression, because you took your optional feature x, and raised it to the power of two or three or any other

power. In the case of the cubic function, the first feature is the size, the second feature is the size squared, and the third feature is the size cubed.

- I just want to point out one more thing, which is that if you create features that are these powers like the square of the original features like this, then feature scaling becomes increasingly important.
    - If the size of the house ranges from say, 1-1,000 square feet,
    - then the second feature, which is a size squared, will range from one to a million,
    - and the third feature, which is size cubed, ranges from one to a billion.
- These two features, x squared and x cubed, take on very different ranges of values compared to the original feature x. If you're using gradient descent, it's important to apply feature scaling to get your features into comparable ranges of values.



Choice of features

$f_{\vec{w},b}(x) = w_1x + w_2\sqrt{x} + b$

Another reasonable alternative to taking the size squared and size cubed is to say use the square root of x. Your model may look like w_1 times x plus w_2 times the square root of x plus b. The square root function looks like this, and it becomes a bit less steep as x increases, but it doesn't ever completely flatten out, and it certainly never ever comes back down. This would be another choice of features that might work well for this data-set as well.