Week 2: Regression with Multiple Input Variables

Multiple Linear Regression

- ➔ Multiple Features

Size in feet^2	Number of bedrooms	Number of floors	Age of home in years	Price ($) in $1000's
x_1	x_2	x_3	x_4	
2104	5	1	45	460
1416	3	2	40	232
1534	3	2	30	315
852	2	1	36	178
...	...	...	...	...

  - ◆ You can have a data set with multiple features
    - ● Notation
      - ○ x_j = j^{th} feature
      - ○ n = total number of features
      - ○ $\overrightarrow{x^{(i)}}$= features of i^{th} training example
        - ◆ In this case it would be a list of 4 numbers across a row (x_1 to x_4)
        - ◆ →$x^{(2)}$ is the row of i=2

i=2	1416	3	2	40

      - ○ $x_j^{(i)}$ = value of feature j in the i^{th} training example
  - ◆ →With multiple features you will have a new model
    - ● In the example above, it would be $f_{w,b}(x) = w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b$
      - ○ An example of this may be:

$$f_{w,b}(x) = 0.1\underset{\text{size}}{\underset{\uparrow}{x_1}} + 4\underset{\text{\#bedrooms}}{\underset{\uparrow}{x_2}} + 10\underset{\text{\#floors}}{\underset{\uparrow}{x_3}} + -2\underset{\text{years}}{\underset{\uparrow}{x_4}} + \underset{\text{base price}}{\underset{\uparrow}{80}}$$

        - ◆ Interpretation Example: For each additional bedroom, the house price increases by 4K
    - ● For *n* features $f_{w,b}(x) = w_1x_1 + w_2x_2 + w_nx_n + b$
      - ○ If you have multiple features, it is called multiple linear regression
      - ○ To simplify:
        - ◆ $\vec{w}$ = [w_1, w_2, w_3, ...w_n]
        - ◆ b is a single number
          - ● *w and b are the parameters*
        - ◆ $\vec{x} = [x_1, x_2, x_3, \ldots x_n]$
        - ◆ $\rightarrow \boldsymbol{f_{\vec{w},b} = \vec{w} \cdot \vec{x} + b}$
          - ● This is a dot product of two vectors

- Means taking corresponding pairs of numbers → $w_1x_1 + w_2x_2$, etc and adding them
- Same expression as above

- Vectorization
  - Parameters and features (Part 1)
    - Example
      - $\vec{w} = [w_1, w_2, w_3]$
        - n = 3
      - b is a number
      - $\vec{x} = [x_1, x_2, x_3]$
    - In linear algebra the index starts from 1 (means to start from 1)
    - In python, the code would look like using **NumPy**

      w = np.array ([1, 2.5, -3.3])

      b = 4

      x = np.array ([10, 20, 30])
      - **In python though, counting starts from 0, therefore to access the first number in the w array (1), you would use w[0], etc.**
    - Without vectorization
      - $f_{w,b}(x) = w_1x_1 + w_2x_2 + w_3x_3 + b$
        - Code would be:
          - *f = w[0] * x[0] +*
            *w[1] * x[1] +*
            *w[2] * x[2] + b*
            - This is great in terms of coding but can get very tedious in the case where there is someehting line n = 100,000
      - Could use a summation operator to create **for loop**
        - $f_{\vec{w},b}(\vec{x}) = (\sum_{j=1}^{n} w_jx_j) + b$
          - Code would be:

            f = 0

            for j in range (0,n)

            *f = f + w[j] *x[j]*

            f = f + b (Please note this is outsite the for loop)
            - Code is still not super efficient
    - With Vectorization
      - $\boldsymbol{f_{\vec{w},b}(\vec{x}) = \vec{w} \cdot \vec{x} + b}$
        - Code would be:

          f = np.dot(w,x) + b
          - This is an efficient code and will run a lot faster than the without vectorization code
  - What happens to the computer during vectorization vs without vectorization

- Without Vectorization for loop

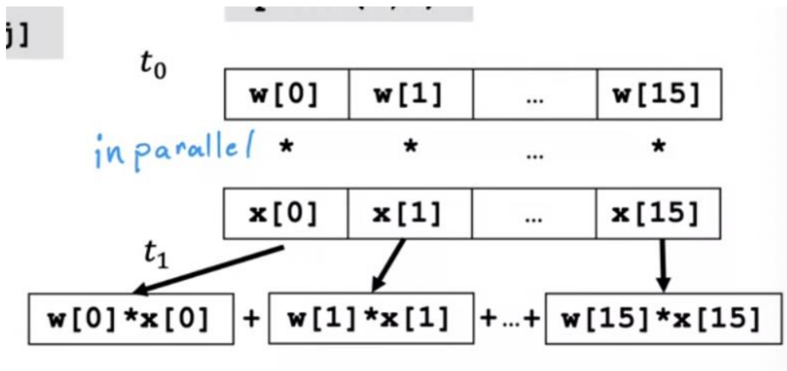
  for j in range(0, 16) :

  *f = f + w[j] * x[j]*

  - At each time point in the code the algorithm operates as so:
    - At t_0
      - *f + w[0] * x[0]*
    - At t_1
      - *f + w[1] * x[1]*
    - Etc. until the 15th step
- With Vectorization

  np.dot(w, x)

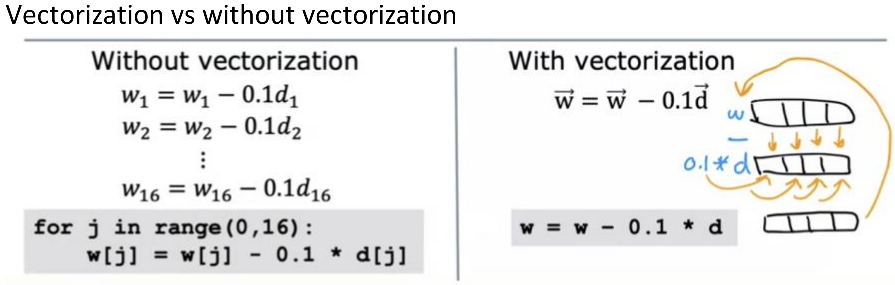
  - The computer gets all values of the vector w and x and in a single step multiplies them in parallel (t_0)
    - → The computer then adds them all together (t_1)



- Gradient Descent
  - Parameters
    - $\vec{w} = (w_1, w_2, w_3, \ldots w_{16})$
    - B
  - Derivative terms
    - $\frac{d}{dx}(\vec{w}) = \vec{d} = d_1, d_2, d_3, \ldots d_{16})$
  - → stored terms of w and d

    w = np.array ([0.5, 1.3, … 3.4])

    d = np.array ([0.3, 0.2, … 0.4])
  - → Compute
    - $w_j = w_j - 0.1\ (learning\ rate) d_j\ for\ j = 1 \ldots 16$
    - Vectorization vs without vectorization



- With vectorization there is parallel processing and the values of the new *w* will be implemented back automatically

➔ Gradient Descent for Multiple Linear Regression

	Previous Notation	**Vector Notation**
Parameters	$w_1, \dots, w_n$ b	$\vec{w}$ b
Model	$f_{\vec{w},b}(\vec{x}) = w_1x_1 + \dots + w_nx_n + b$	$f_{\vec{w},b}(\vec{x}) = \vec{w} \cdot \vec{x} + b$
Cost Function	$J(w_1, \dots, w_n, b)$	$J(\vec{w}, b)$

Gradient Descent	repeat{
	$w_j = w_j - a\frac{\partial}{\partial w_j}J(w_1, \dots, w_n, b)$
	$b = b - a\frac{\partial}{\partial b}J(w_1, \dots, w_n, b)$
	}
		repeat{
		$w_j = w_j - a\frac{\partial}{\partial w_j}J(\vec{w}, b)$
		$b = b - a\frac{\partial}{\partial b}J(\vec{w}, b)$
		}

- Gradient Descent with multiple features
    - One feature

      repeat{

      $w = w - a\frac{1}{m}\sum_{i=1}^{m} (f_{w,b}(x^{(i)}) - y^{(i)})x^{(i)}$

      $b = b - a\frac{1}{m}\sum_{i=1}^{m} (f_{w,b}(x^{(i)}) - y^{(i)})$

      simaltaneous update w, b

      }
    - *n* features ($n \geqq 2$)

      repeat{

      $w_n = w_n - a\frac{1}{m}\sum_{i=1}^{m} (f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)})x_n^{(i)}$ # $\frac{\partial}{\partial w_1}J(\vec{w}, b)$;

      $$b = b - a\frac{1}{m}\sum_{i=1}^{m} (f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)})$$

      simultaneously update w_j (for *j* =1, …, *n*) and *b*)

      }
- An alternative to gradient descent
    - A normal equation
        - Works for **only** linear regression
        - Solves for w, b without iterations
        - Disadvatages
            - Doesn’t generalize to toher learning algorithms
            - Slow when number of features is large (>10,000)

Practice Quiz: Multiple Linear Regression

1. In the training set below, what is $x_4^{(3)}$? Please type in the number below (this is an integer such as 123, no decimal points).

1 point

Size in feet^2 x_1	Number of bedrooms x_2	Number of floors x_3	Age of home in years x_4	Price ($) in $1000's
2104	5	1	45	460
1416	3	2	40	232
1534	3	2	30	315
852	2	1	36	178
...	...	...	...	...

30

2.

1 point

Which of the following are potential benefits of vectorization? Please choose the best option.

- ○ It makes your code run faster
- ○ It can make your code shorter
- ○ It allows your code to run more easily on parallel compute hardware
- ◉ All of the above

3. True/False? To make gradient descent converge about twice as fast, a technique that almost always works is to double the learning rate $alpha$.

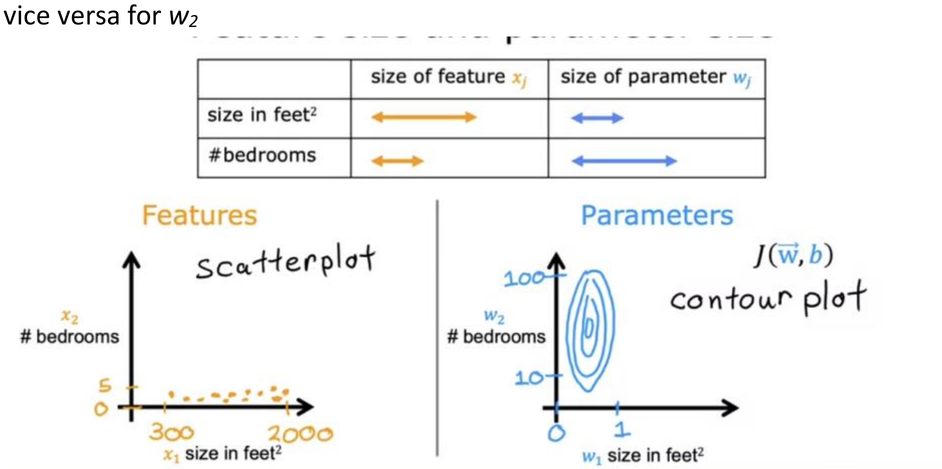
1 point

- ◉ False
- ○ True

Gradient Descent in Practice

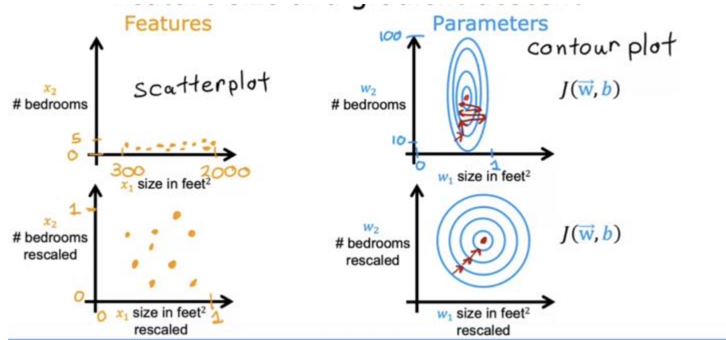
- ➔ Feature Scaling
  - Feature size (how big the number is) and the size of the associated parameter value
    - Ex: Size of house prediction using $price = w_1x_1 + w_2x_2 + b$
      - x_1 = size in ft^2

- Range is typically from 300 - 2000
  - Large range of values
- x_2 = # of bedrooms
  - range from 0 - 5
    - Small range of values
- House: x_1 = 2000, x_2 = 5, price = 500k
  - Size w_1 and w_2 ?
    - One example where w_1 = 50, w_2 = 0.1, b = 50:
      - $price = 50 * 2000 * 0.1 + 5 + 50$
        - $\rightarrow 100{,}000k + 0.5k + 50k = \$100{,}050{,}500$
          - Very far from the actual price of 500k – not good parameter choices
    - Another example where w_1 = 0.1, w_2 = 50, b = 50
      - **In this case, the values of w_1 and w_2 are switched**
      - $price = 0.1 * 2000 * 50 + 5 + 50$
        - $\rightarrow 200k + 250k + 50k = \$500{,}000$
          - More reasonable and matches our price
      - Hopefully a learning algorithm can decipher to attach a low x_2 value to a high w_2 parameter value and vice versa → will lead to the more accurate measurement
- Gradient Descent
  - Takes a small change in w_1 to make a big change in the cost function, vice versa for w_2



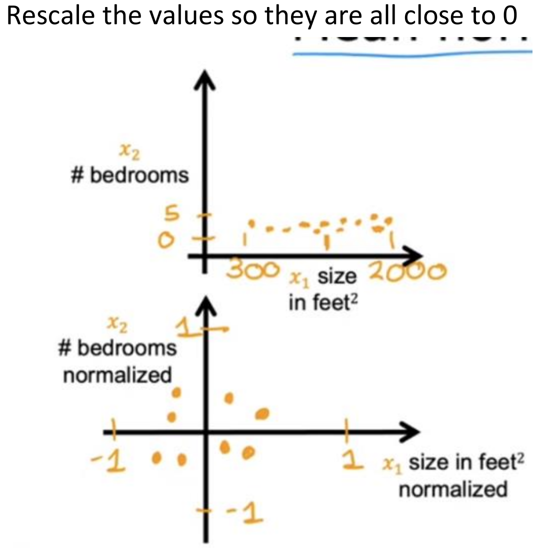
  - Gradient descent might take time bounding back around till it finds its global minimum since the contour plot is skinny
    - → could scale the feature which could make it better

- To do this scale the features in such a way that they are taking comparable ranges to one another



  - As you can see the contour plot looks much better and easier for a gradient descent algorithm

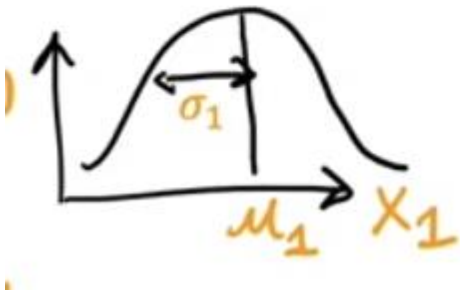
- Feature Scaling
  - One way – dividing by the maximum
    - If $300 \le x_1 \le 2000$ → $x_{1,scaled} = \frac{x_1}{2000}$ → will then range from $0.15 \le x_{1,\,scaled} \le 1$
      - Similarly for x_2, $0 \le x_2 \le 5$ → $x_{2,scaled} = \frac{x_2}{5}$ → $0 \le x_{2,\,scalled} \le 1$
  - Another way – Mean normalization
    - Rescale the values so they are all close to 0



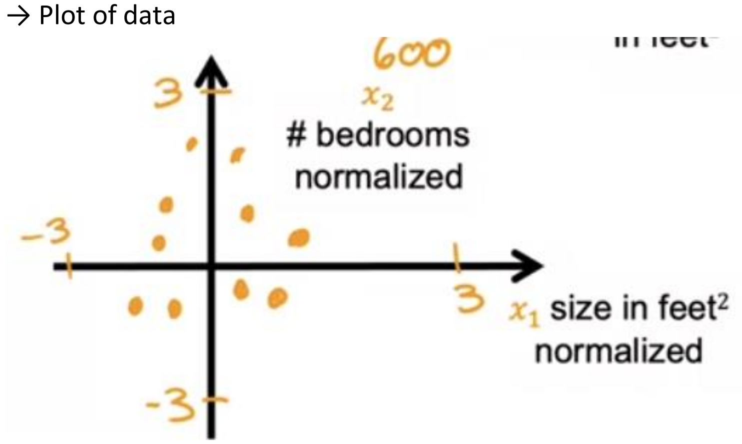
    - Calculation
      - Find mean of x_1 on the training set = μ_1(for this example set to 600)
        - $x_1 = \frac{x_1 - \mu_1}{2000 - 300}$ → $-0.18 \le x_1 \le 0.82$
      - Find mean of x_2 on the training set = μ_2 (for this example set to 2.3)

- $x_2 = \frac{x_2 - \mu_2}{5-0}$ → -0.46 ≤ x_2 ≤ 0.54

- Another way – Z-score normalization
    - To do this you need to calculate the standard deviation of each feature

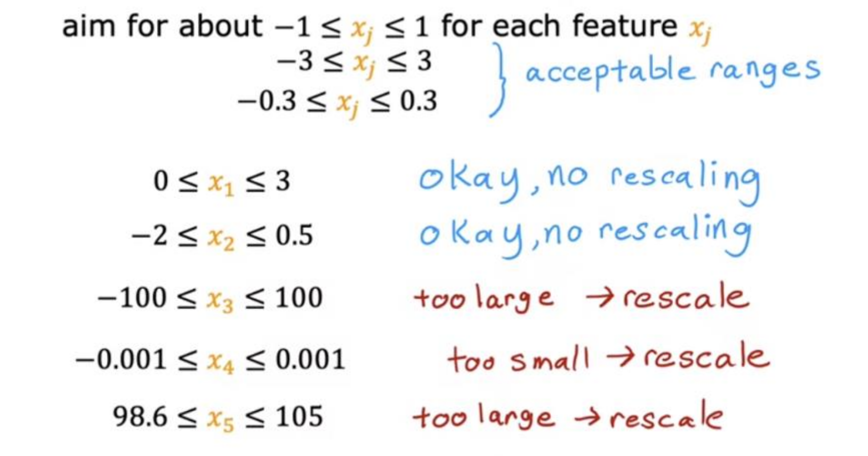


    - Calculation (for this example set $\boldsymbol{\sigma}_1$ to 450 and μ_1 to 200; and $\boldsymbol{\sigma}_2$ to 1.4 and μ_2 to 2.3)
        - $x_1 = \frac{x_1 - \mu_1}{\sigma_1}$
            - → -0.68 ≤ x_1 ≤ 3.1
        - $x_2 = \frac{x_2 - \mu_2}{\sigma_2}$
            - → -1.6 ≤ x_2 ≤ 1.6
    - → Plot of data



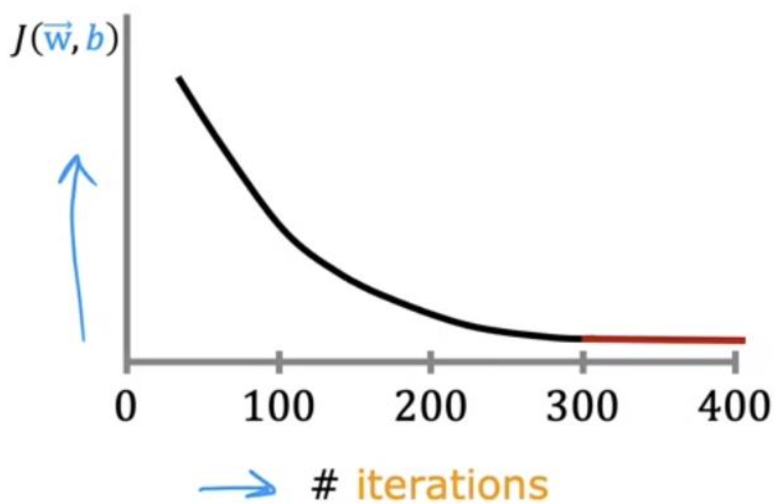
- General guidelines
    - Aim for -1 ≤ x_j ≤ 1 for feature x_j

- Or a scaled value



- Usually there is no hard to carrying out feature rescaling

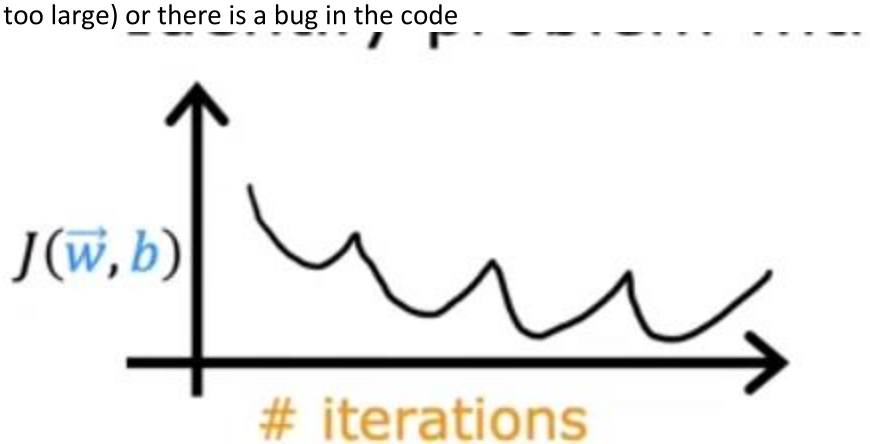
➔ Checking Gradient Descent for Convergence
- Equations:
  - $w_j = w_j - a\frac{\partial}{\partial w_j}J(\vec{w}, b)$
  - $b = b - a\frac{\partial}{\partial b}J(\vec{w}, b)$
- How to make sure gradient descent is working correctly
  - Objective: $min\ for\ J(\vec{w}, b)$
  - Could plot cost function ($J(\vec{w}, b)$) vs iterations of gradient descent



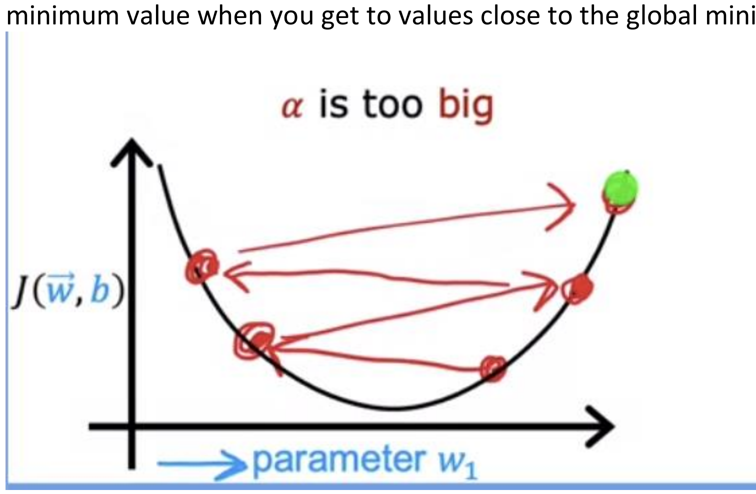
  - Also known as a **learning curve**
    - Shows how cost function changes after every iteration
      - If working correctly, $J(\vec{w}, b)$ should decrease after every iteration
      - If increases after any interaction could mean that α is chosen incorrectly (usually too large) or there is a bug in the code
  - # of iterations needed varies greatly depending on the application

- Could also do an automatic convergence test
    - Ex: $\varepsilon = 10^{-3}$ *or* 0.001
        - If $J(\vec{w}, b)$ decreases by ≤ ε in one iteration, declare convergence
            - Likely to be on the flattened part of the curve and the values of $\vec{w}$, *and* b are close to the global minimum
    - Choosing the threshold ε is somewhat difficult

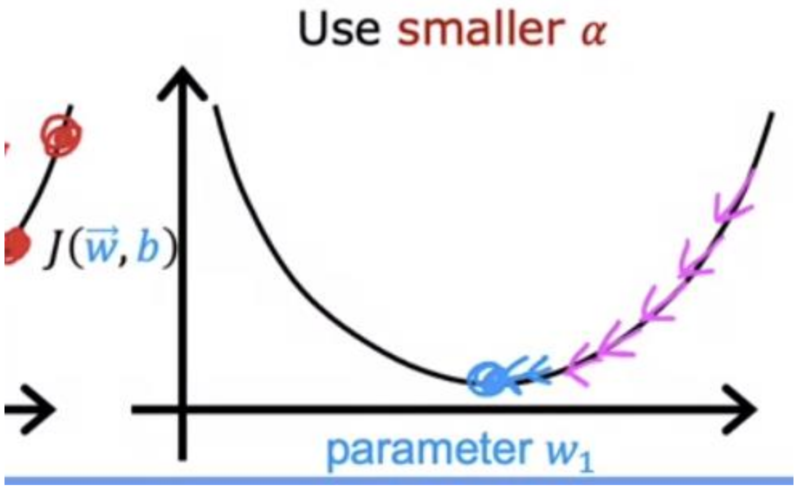
➔ Choosing the Learning Rate
- Identifying problem with gradient descent
    - If increases after any interaction could mean that α is chosen incorrectly (usually too large) or there is a bug in the code



    - If the learning rate is too big, the update step may overshoot the global minimum value when you get to values close to the global minimum



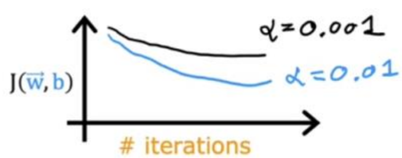
- To fix this, use a smaller learning rate



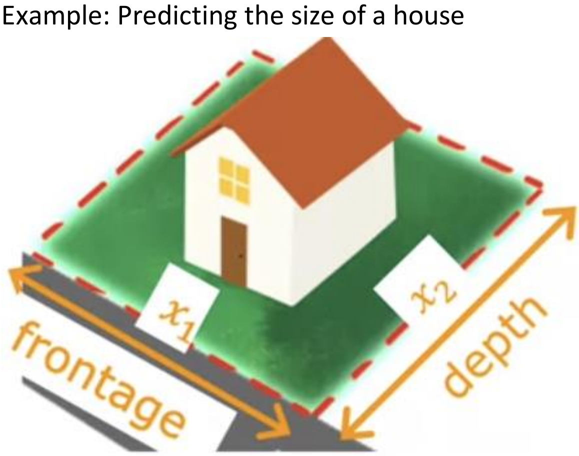
- With a small enough smaller learning rate, $J(\vec{w}, b)$ should decrease with every iteration
- Could try a couple of different learning rate values to see how they affect the cost function

Values of α to try:

... 0.001 0.01 0.1 1 ...



➔ Feature Engineering
- Example: Predicting the size of a house



- Features
  - x_1 is the frontage of the lot
  - x_2 is the depth of the house

- Model

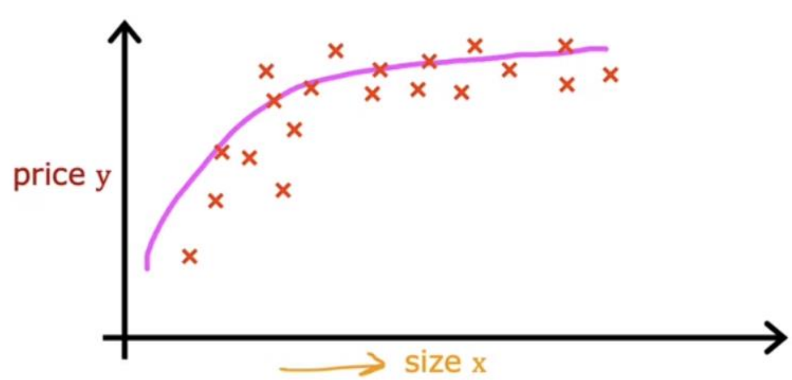
$$f_{\vec{w},b}(\vec{x}) = w_1 \underbrace{x_1}_{\text{frontage}} + w_2 \underbrace{x_2}_{\text{depth}} + b$$

  - Could be a good way to predict the cost of a house but you can also use area if you want
    - $area = frontage \cdot depth$
      - Might be more indicative of the price than just the front and depth as separate features
        - → $x_3 = x_1 x_2$ *where* x_3= area
          - Creating this new feature is the process of **feature engineering**
            - Transforming or combining original features to define new features
    - → New model

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

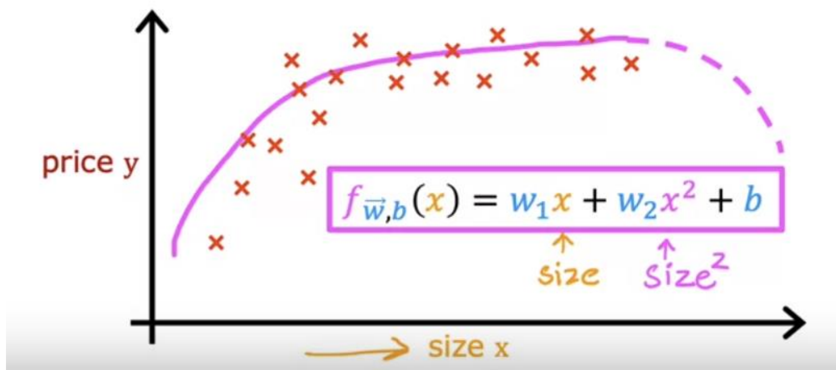
      - New model now uses all 3 features, and therefore the *w* value of each can be assigned a different weight depending on what the algorithm deems as being the most important for predicting the price of the house

➔ Polynomial Regression
  - Type of feature engineering
  - Ex: Housing data set

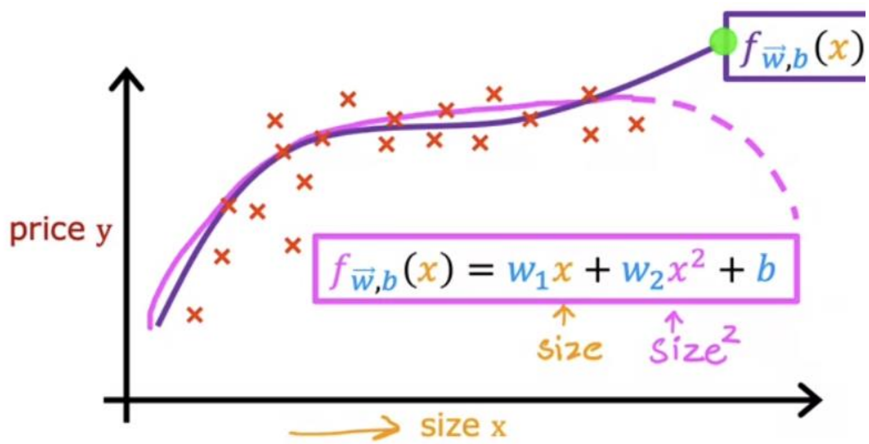


- Might be easier to fit a different to the data set than a linear regression

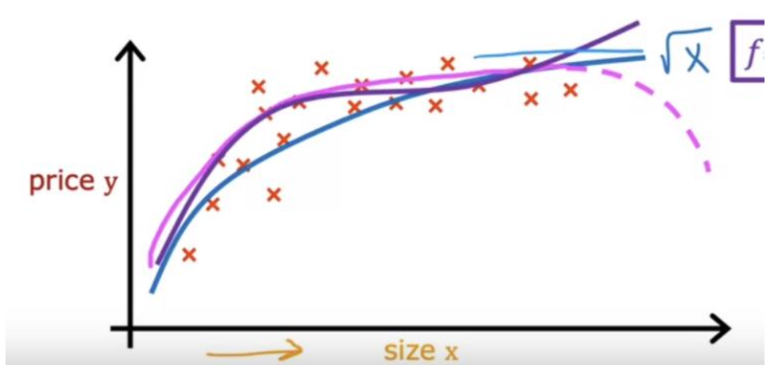
- Could you use this quadratic equation to predict? $f_{\vec{w},b}(x) = w_1x + w_2x^2 + b$
  - While this could be good, it doesn't work cause quadratic functions eventually go down



- How about a cubic function? $f_{\vec{w},b}(x) = w_1x + w_2x^2 + w_3x^3 + b$



  - Somewhat better than the quadratic model
- How about a $\sqrt{x}$ model? $f_{\vec{w},b}(x) = w_1x + w_2\sqrt{x} + b$

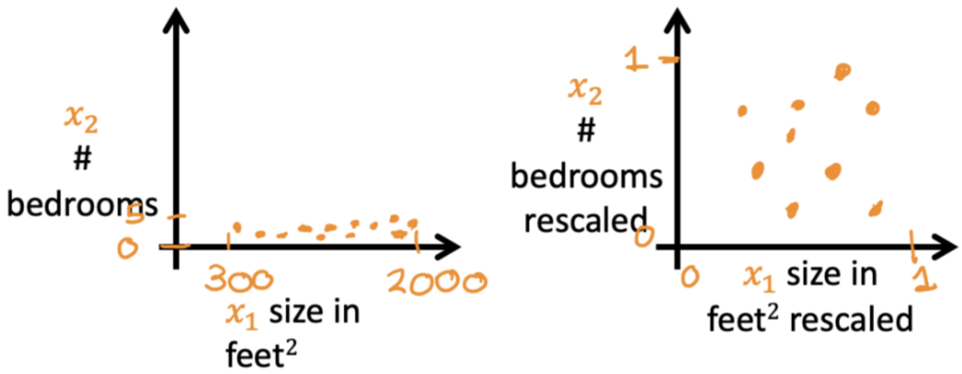


- When using polynomial regression feature scaling becomes more important than ever

Practice Quiz: Gradient Descent in Practice

1.

1 point

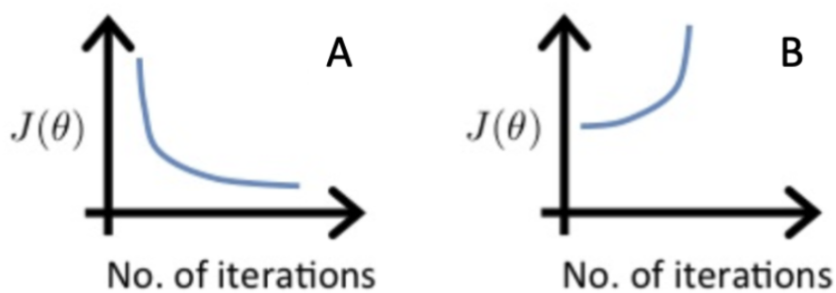


Which of the following is a valid step used during feature scaling?

- [ ] Add the mean (average) from each value and and then divide by the (max - min).
- [x] Subtract the mean (average) from each value and then divide by the (max - min).

2. Suppose a friend ran gradient descent three separate times with three choices of the learning rate α and plotted the learning curves for each (cost J for each iteration).

1 point



For which case, A or B, was the learning rate α likely too large?

- [ ] case A only
- [ ] Both Cases A and B
- [ ] Neither Case A nor B
- [x] case B only

3. Of the circumstances below, for which one is feature scaling particularly helpful?

1 point

- [x] Feature scaling is helpful when one feature is much larger (or smaller) than another feature.
- [ ] Feature scaling is helpful when all the features in the original data (before scaling is applied) range from 0 to 1.

4.

1 point

You are helping a grocery store predict its revenue, and have data on its items sold per week, and price per item. What could be a useful engineered feature?

- [x] For each product, calculate the number of items sold times price per item.
- [ ] For each product, calculate the number of items sold divided by the price per item.

5. True/False? With polynomial regression, the predicted values f_w,b(x) does not necessarily have to be a straight line (or linear) function of the input feature x.

1 point

- [x] True
- [ ] False