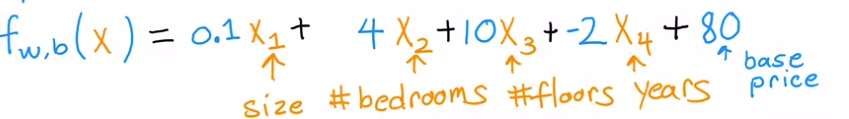
# **Week 2: Regression with Multiple Input Variables**

## Multiple Linear Regression

### Multiple Features

* + You can have a data set with multiple features
    - Notation
      * xj = jth feature
      * *n* = total number of features
      * = features of ith training example
        + In this case it would be a list of 4 numbers across a row (x1 to x4)
        + →x(2) is the row of i=2 
      * xj(i) = value of feature j in the ith training example
  + →With multiple features you will have a new model
    - In the example above, it would be
      * An example of this may be:
        + Interpretation Example: For each additional bedroom, the house price increases by 4K
    - For *n* features
      * If you have multiple features, it is called multiple linear regression
      * To simplify:
        + = [w1, w2, w3, …wn]
        + b is a single number

*w and b are the parameters*

* + - * + →

This is a dot product of two vectors

Means taking corresponding pairs of numbers → , etc and adding them

Same expression as above

### Vectorization

* + Parameters and features (Part 1)
    - Example
      * + n = 3
      * b is a number
    - 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
      * + 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**

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
      * + 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 *t0*

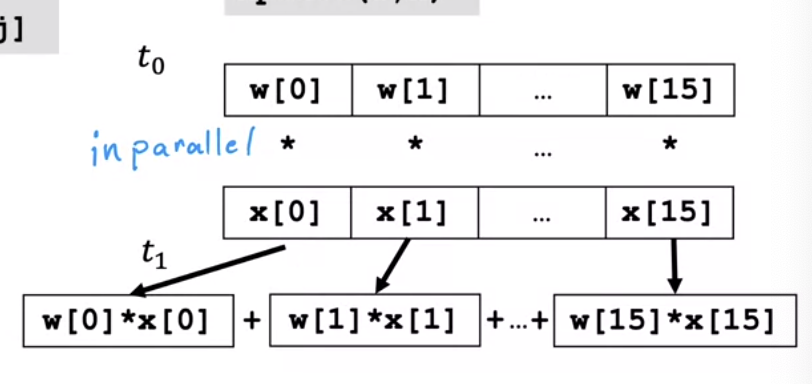
*f + w[0] \* x[0]*

* + - * + At t1

*f + w[1] \* x[1]*

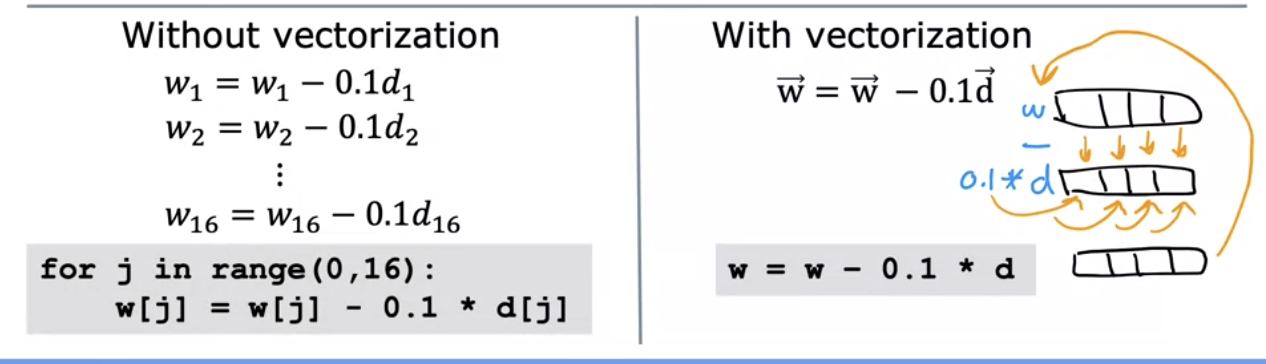
* + - * + Etc. until the 15th step
    - With Vectorization

*np.dot(w, x)*

* + - * The computer gets all values of the vector w and x and in a single step multiplies them in parallel (*t0*)
        + → The computer then adds them all together (*t1*)
  + Gradient Descent
    - Parameters
      * B
    - Derivative terms
    - → 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
      * 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** | *b* | *b* |
| **Model** |  |  |
| **Cost Function** |  |  |
| **Gradient Descent** | repeat{    } | repeat{  )  } |

* + Gradient Descent with multiple features
    - One feature

*repeat{*

*simaltaneous update w, b*

}

* + - *n* features (*n* ≧ 2)

repeat{

# ;

simultaneously update *wj* (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

## 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
      * *x1* = size in ft2
        + Range is typically from 300 - 2000

Large range of values

* + - * *x2* = # of bedrooms
        + range from 0 - 5

Small range of values

* + - * House: *x1* = 2000, *x2* = 5, price = 500k
        + Size *w1* and *w2* ?

One example where *w1* = 50, *w2* = 0.1, *b* = 50:

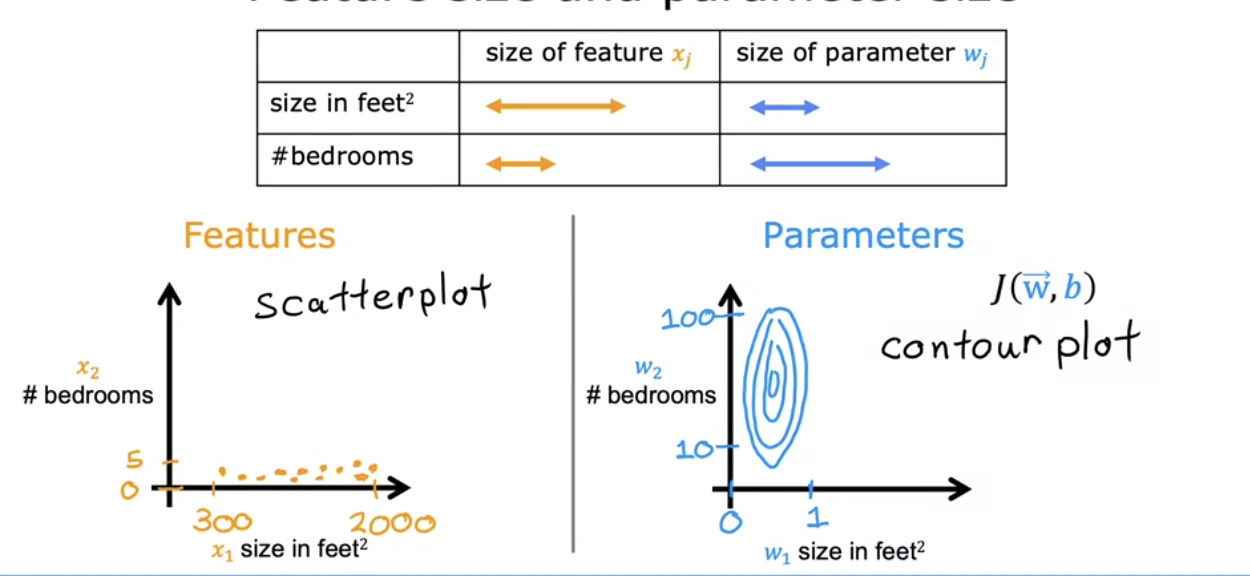
Very far from the actual price of 500k – not good parameter choices

Another example where *w1* = 0.1, *w2* = 50, *b* = 50

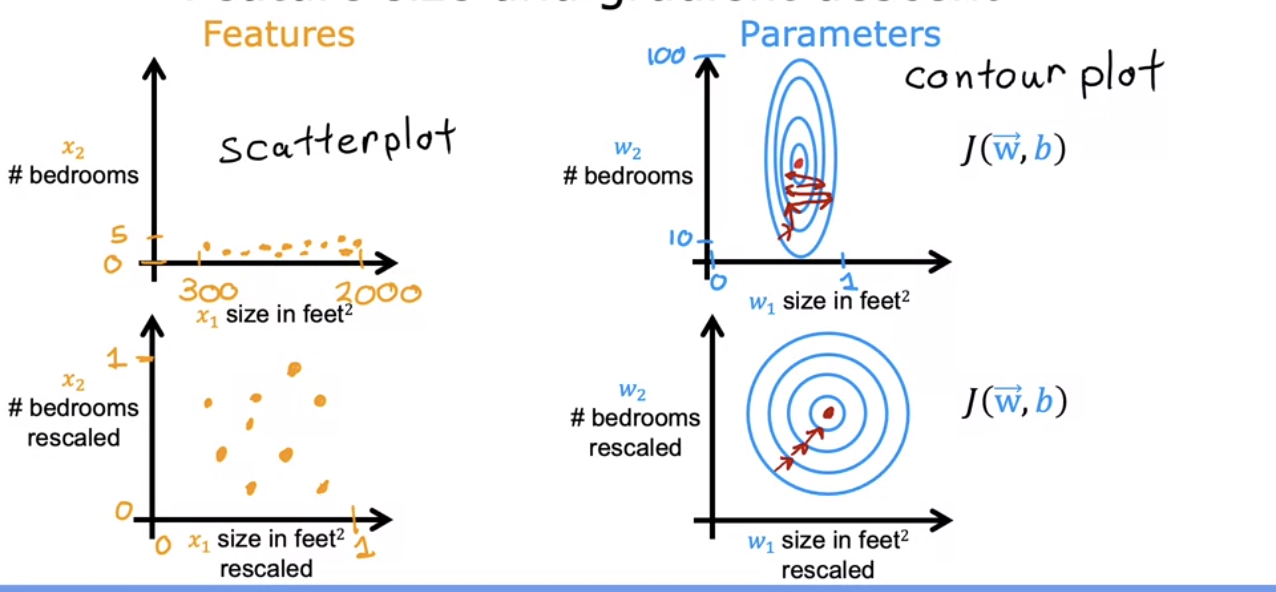
**In this case, the values of *w1* and *w2* are switched**

More reasonable and matches our price

Hopefully a learning algorithm can decipher to attach a low *x2* 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 *w1* to make a big change in the cost function, vice versa for *w2*
        + Gradient descent might take time bounding back around till it finds its global minimum since the contour plot is skinny

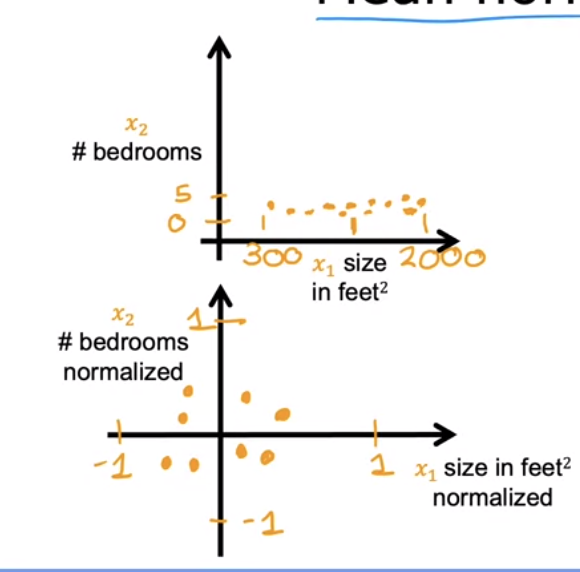
→ could scale the feature which could make it better

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 ≤ *x1* ≤ 2000 → → will then range from

0.15 ≤ *x1, scaled* ≤ 1

* + - * + Similarly for *x2,*, 0 ≤ x2 ≤ 5 → → 0 ≤ *x2, scalled* ≤ 1
    - Another way – Mean normalization
      * Rescale the values so they are all close to 0
      * Calculation
        + Find mean of *x1* on the training set = (for this example set to 600)

→ -0.18 ≤ *x1* ≤ 0.82

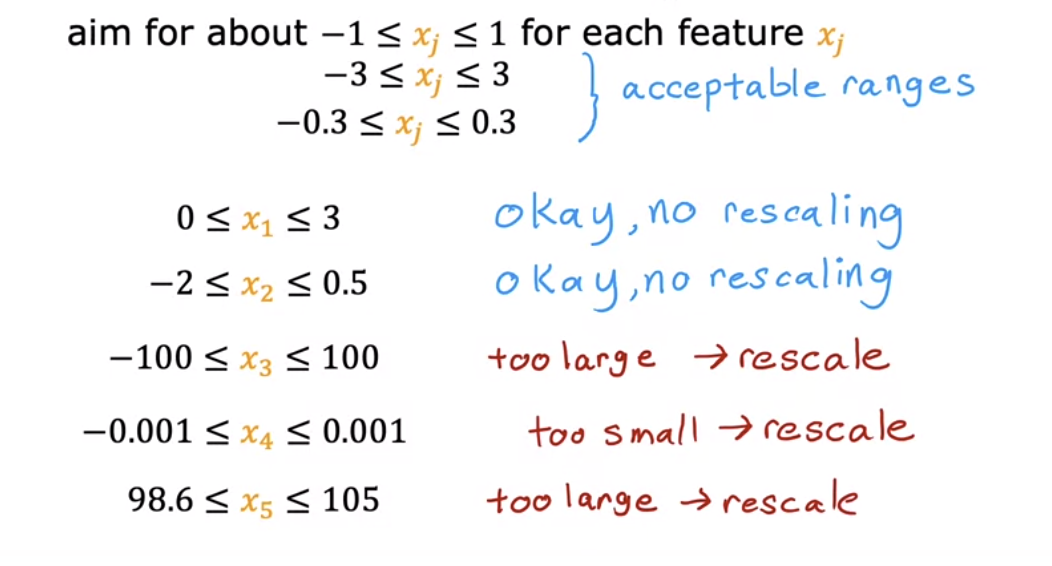
* + - * + Find mean of *x2* on the training set = (for this example set to 2.3)

→ -0.46 ≤ *x2* ≤ 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 𝝈1 to 450 and to 200; and 𝝈2 to 1.4 and to 2.3)

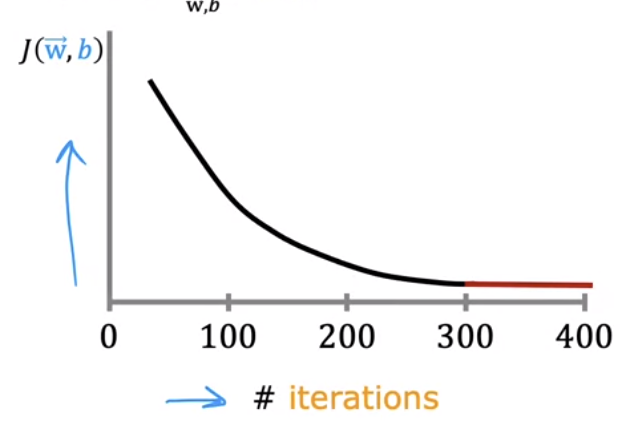
→ -0.68 ≤ *x1* ≤ 3.1

→ -1.6 ≤ *x2* ≤ 1.6

* + - * → Plot of data
    - General guidelines
      * Aim for -1 ≤ *xj* ≤ 1 for feature *xj*
        + Or a scaled value 

Usually there is no hard to carrying out feature rescaling

### Checking Gradient Descent for Convergence

* + Equations:
    - )
  + How to make sure gradient descent is working correctly
    - Objective:
    - Could plot cost function () vs iterations of gradient descent 
      * Also known as a **learning curve**
        + Shows how cost function changes after every iteration

If working correctly, should decrease after every iteration

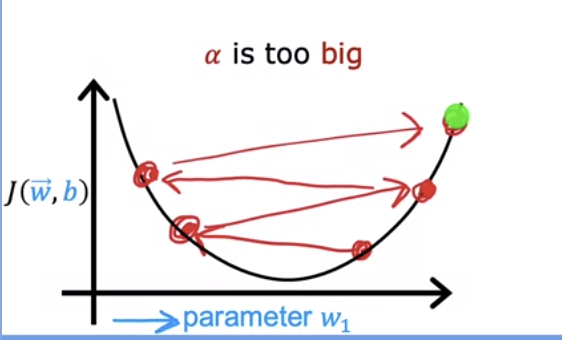
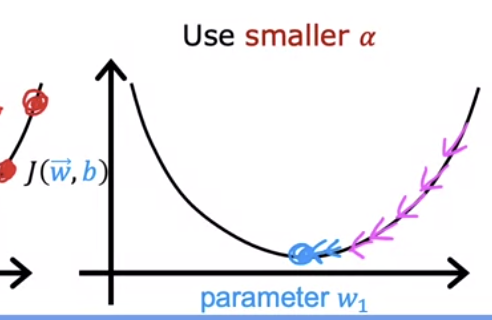
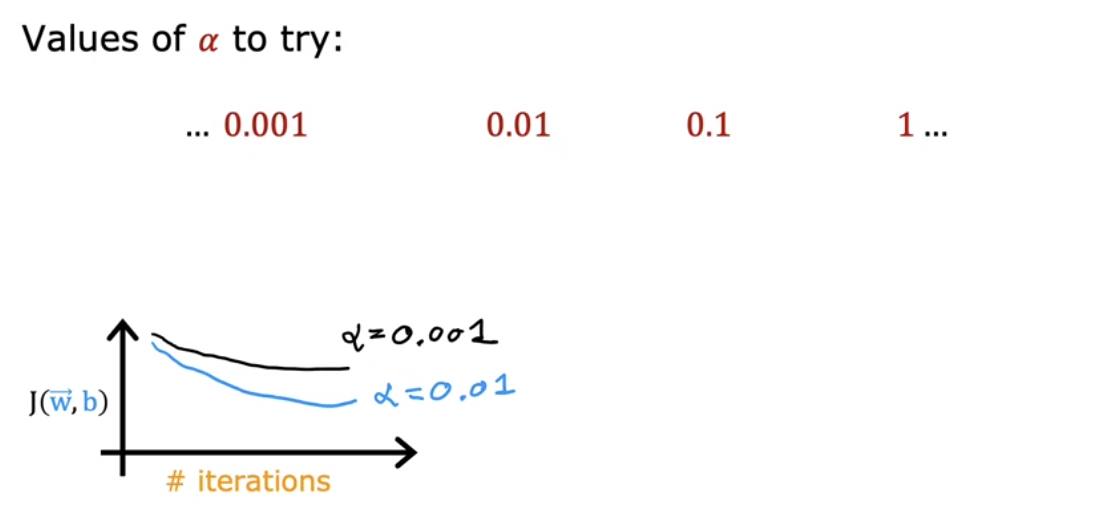
If increases after any interaction could mean that ɑ is chosen incorrectly (usually too large) or there is a bug in the code

* + - * # of iterations needed varies greatly depending on the application
    - Could also do an automatic convergence test
      * Ex:
        + If decreases by ≤ ε in one iteration, declare convergence

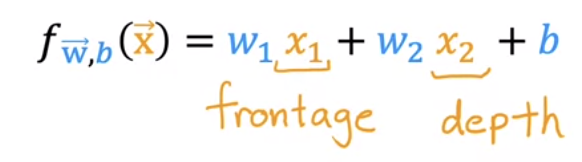
Likely to be on the flattened part of the curve and the values of are close to the global minimum

* + - * Choosing the threshold ε is somewhat difficult

### Choosing the Learning Rate

* + Identifying problem with gradient descent
    - If increases after any interaction could mean that ɑ 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, should decrease with every iteration
      * Could try a couple of different learning rate values to see how they affect the cost function

### Feature Engineering

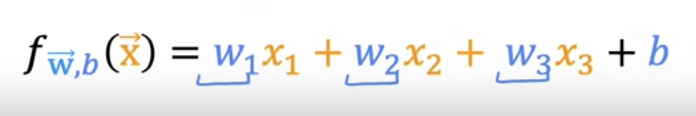
* + Example: Predicting the size of a house
    - Features
      * *x1*is the frontage of the lot
      * *x2* is the depth of the house
    - Model
      * Could be a good way to predict the cost of a house but you can also use area if you want

Might be more indicative of the price than just the front and depth as separate features

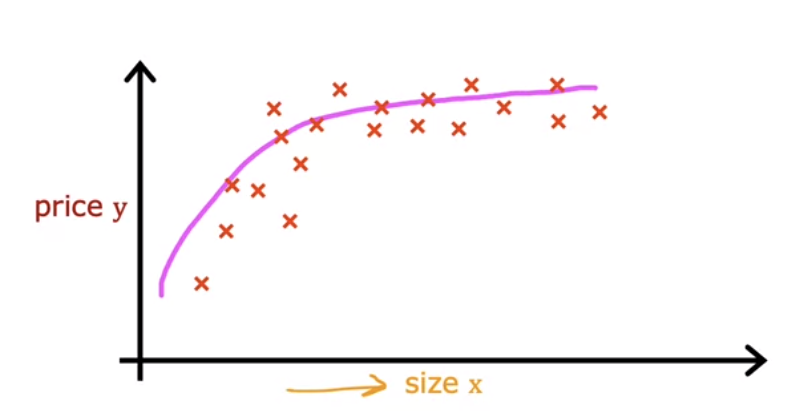
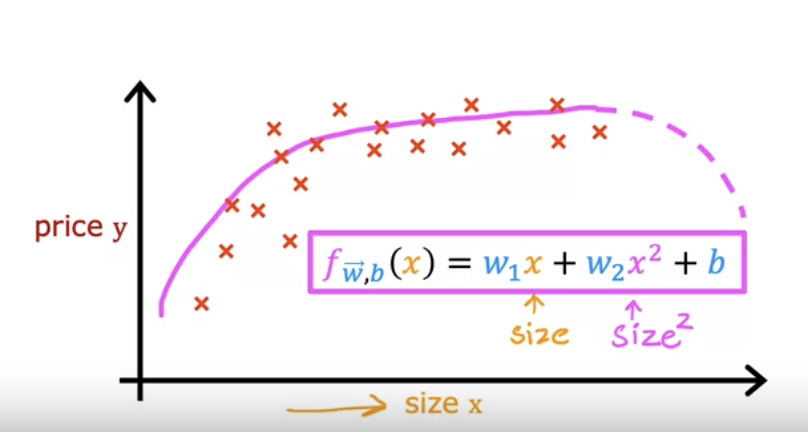
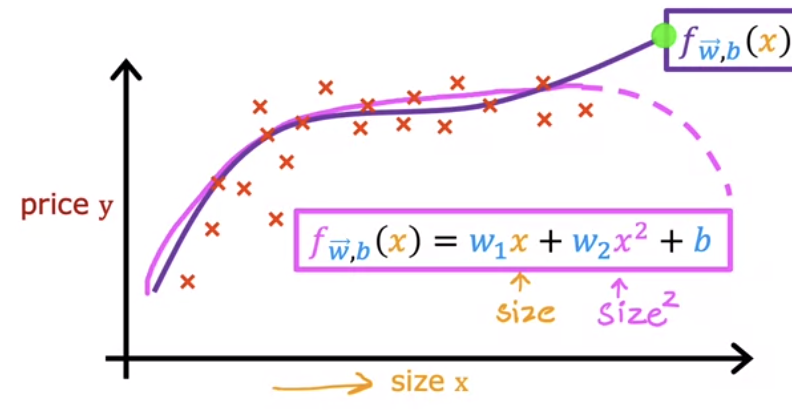
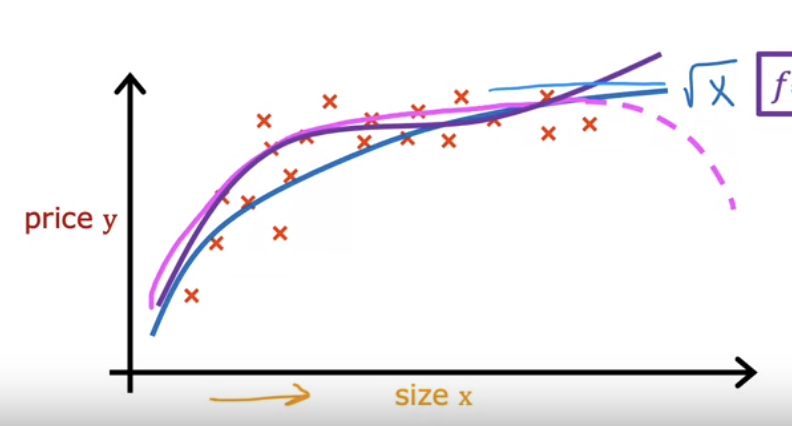
→ = area

Creating this new feature is the process of **feature engineering**

Transforming or combining original features to define new features

* + - * + → New model

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?
        + While this could be good, it doesn’t work cause quadratic functions eventually go down
      * How about a cubic function? 
        + Somewhat better than the quadratic model
      * How about a model? 
    - When using polynomial regression feature scaling becomes more important than ever

## Practice Quiz: Gradient Descent in Practice

