Machine Learning

What is Machine Learning?

Machine learning: Field of study that gives computers the ability to learn without bening explicitly programmed

Machine learning algorithms

The two main types of machine learning are:

- Supervised learning - Used most in real-world applications
- Unsupervised learning

Supervised Learning

Classify or Predict target valiables that contain labels

- Data comes with input x, and output labels y.

Supervised is splitted into two kinds:

1. Regression - Predict a number from infinitely many possible numbers
2. Classification - predict categories (classes) all of a small number of possible output

Unsupervised learning

Clustring data that do not have labels

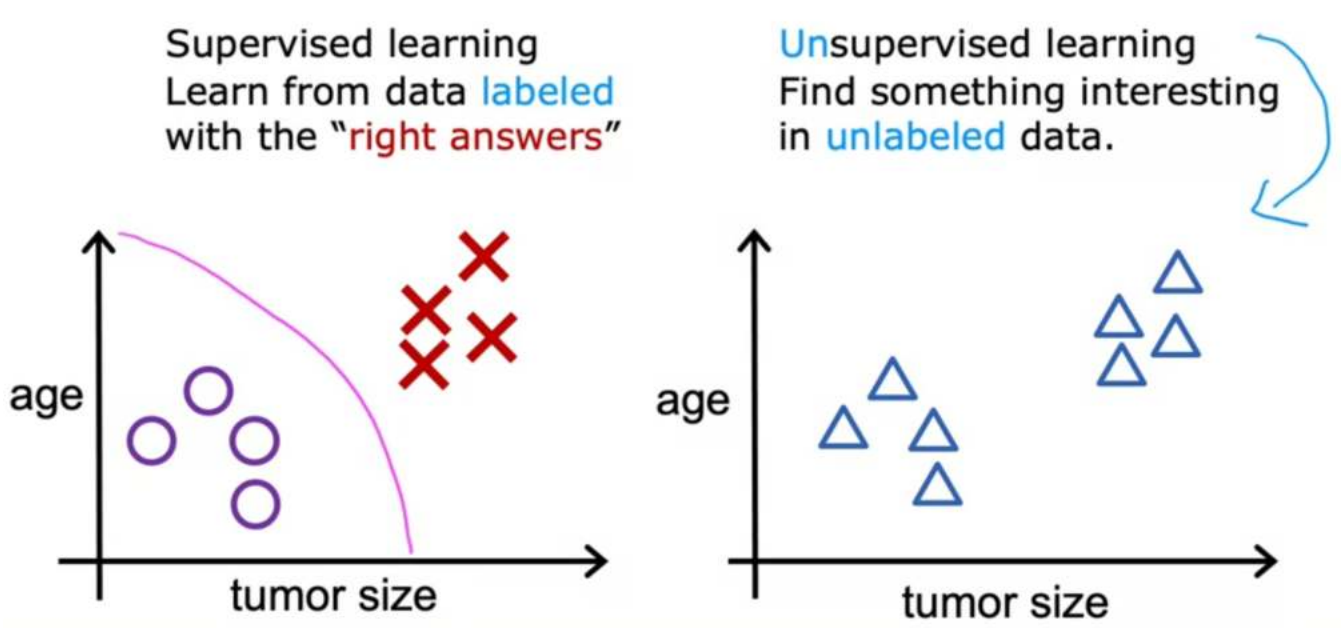
- Data only comes with input x, but not output labels y.

Unsupervised can be done by:

1. Clustring - Group similar data points together
2. Anomaly detection - Find unusual data points
3. Dimensionality reduction - Compress data using fewer numbers

---

Loading [MathJax]/extensions/Safe.js

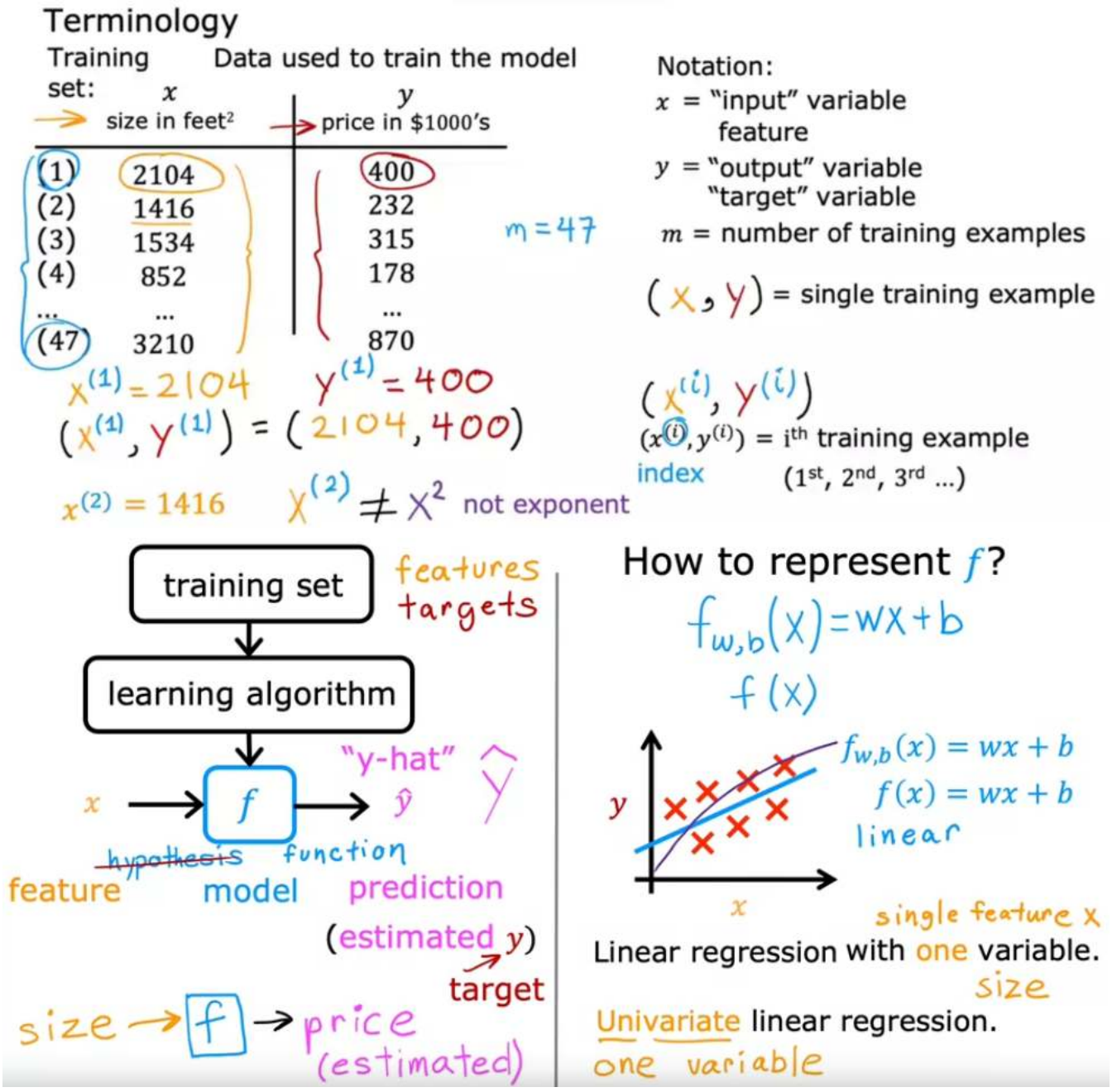


Regression Model

Linear regression model

The concept of supervised learning and linear regression model is introduced. The example of predicting house prices based on house size is used to explain the concepts. Supervised learning is defined as training a model with labeled data, where the model is provided with input features (x) and corresponding output labels (y). Linear regression is a type of supervised learning model that predicts numerical values (regression) such as house prices. It involves fitting a straight line to the data points in order to make predictions.

Loading [MathJax]/extensions/Safe.js



Notation

Here is a summary of some of the notation you will encounter.

General Notation	Description	Python (if applicable)
a	scalar, non bold	
$\mathbf{a}$	vector, bold	
Regression		
$\mathbf{x}$	Training Example feature values (in this lab - Size (1000 sqft))	`x_train`
$\mathbf{y}$	Training Example targets (in this lab Price (1000s of dollars))	`y_train`
$x^{(i)}$, $y^{(i)}$	i_{th}Training Example	`x_i`, `y_i`
m	Number of training examples	`m`

Loading [MathJax]/extensions/Safe.js

General Notation	Description	Python (if applicable)
w	parameter: weight	`w`
b	parameter: bias	`b`
$f_{w,b}(x^{(i)})$	The result of the model evaluation at $x^{(i)}$ parameterized by w,b: $f_{w,b}(x^{(i)}) = wx^{(i)}+b$	`f_wb`

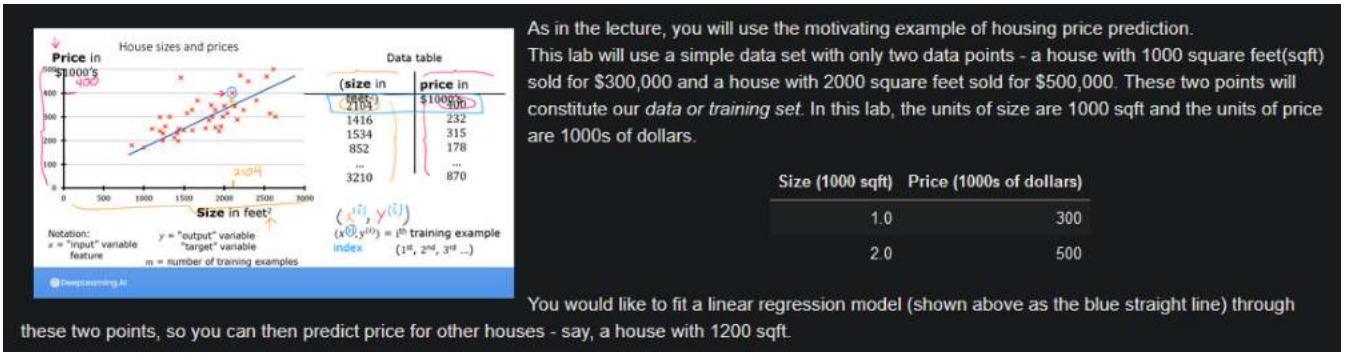
Lab: Model Representation

In this lab you will make use of:

- NumPy, a popular library for scientific computing
- Matplotlib, a popular library for plotting data

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('deeplearning.mplstyle')
```

Problem Statement



In [2]:
```
# x_train is the input variable (size in 1000 square feet)
# y_train is the target (price in 1000s of dollars)
x_train = np.array([1.0,2.0])
y_train = np.array([300.0,500.0])
print(f"x_train = {x_train}")
print(f"y_train = {y_train}")
```

```
x_train = [1. 2.]
y_train = [300. 500.]
```

Number of training examples `m`

You will use `m` to denote the number of training examples. Numpy arrays have a `.shape` parameter. `x_train.shape` returns a python tuple with an entry for each dimension. `x_train.shape[0]` is the length of the array and number of examples as shown below.

In [3]:
```
# m is the number of training examples
print(f"x_train.shape: {x_train.shape}")
m = x_train.shape[0]
print(f"Number of training examples is: {m}")
```

Loading [MathJax]/extensions/Safe.js

```
x_train.shape: (2,)
Number of training examples is: 2
```

One can also use the Python `len()` function as shown below.

In [4]:
```
# m is the number of training examples
m = len(x_train)
print(f"Number of training examples is: {m}")
```

```
Number of training examples is: 2
```

Training example `x_i, y_i`

You will use (x$^{(i)}$, y$^{(i)}$) to denote the i^{th} training example. Since Python is zero indexed, (x$^{(0)}$, y$^{(0)}$) is (1.0, 300.0) and (x$^{(1)}$, y$^{(1)}$) is (2.0, 500.0).

To access a value in a Numpy array, one indexes the array with the desired offset. For example the syntax to access location zero of `x_train` is `x_train[0]`. Run the next code block below to get the i^{th} training example.

In [5]:
```
i = 0 # Change this to 1 to see (x^1, y^1)

x_i = x_train[i]
y_i = y_train[i]
print(f"(x^({i}), y^({i})) = ({x_i}, {y_i})")
```

```
(x^(0), y^(0)) = (1.0, 300.0)
```

Plotting the data

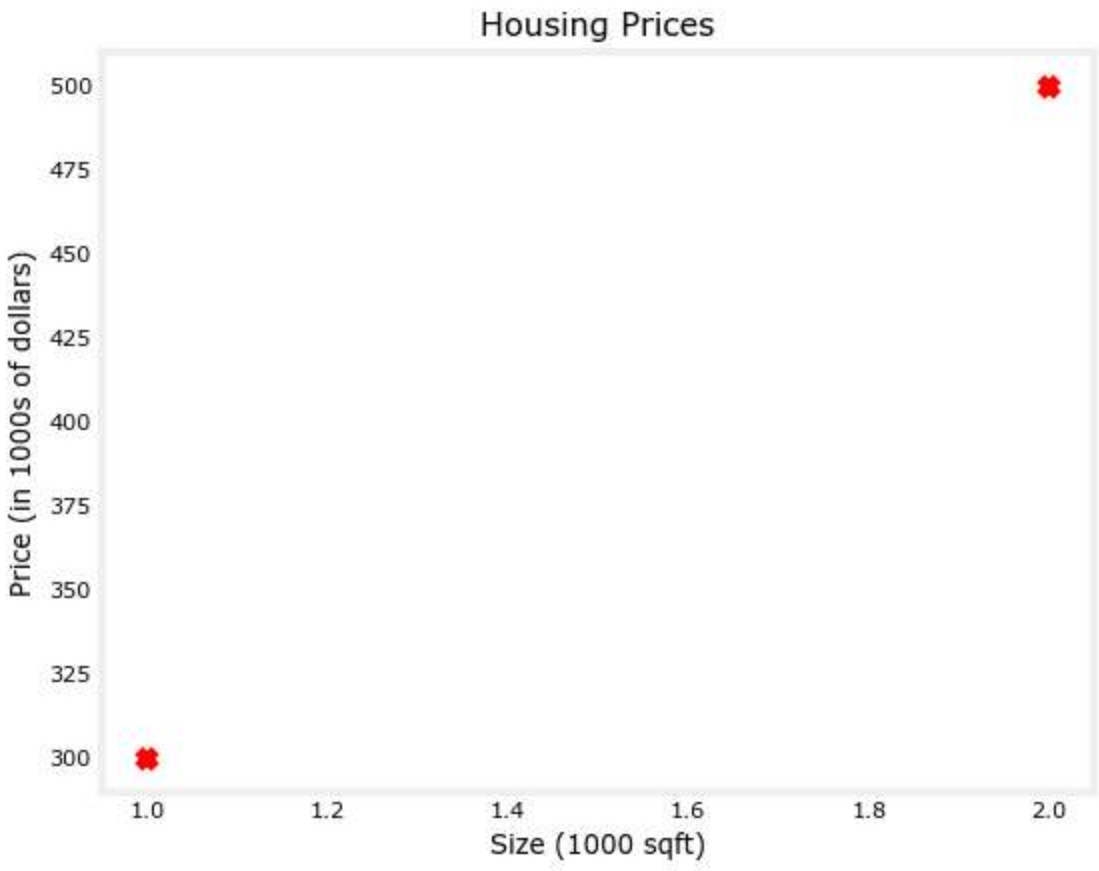
You can plot these two points using the `scatter()` function in the `matplotlib` library, as shown in the cell below.

- The function arguments `marker` and `c` show the points as red crosses (the default is blue dots).

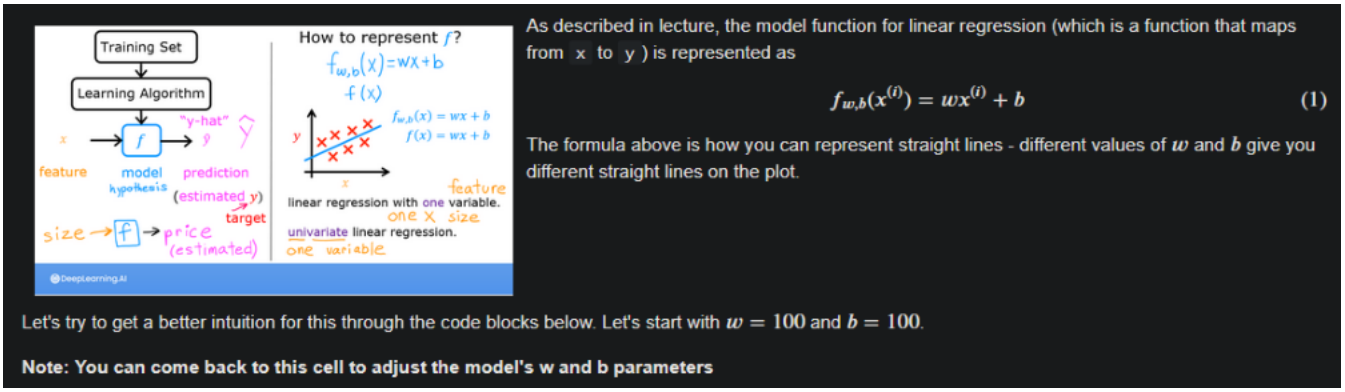
You can use other functions in the `matplotlib` library to set the title and labels to display

In [6]:
```
# Plot the data points
plt.scatter(x_train, y_train, marker='x', c='r')
# Set the title
plt.title("Housing Prices")
# Set the y-axis label
plt.ylabel('Price (in 1000s of dollars)')
# Set the x-axis label
plt.xlabel('Size (1000 sqft)')
plt.show()
```

Loading [MathJax]/extensions/Safe.js



Model function



In [21]:
```
w = 200
b = 100
print(f"w: {w}")
print(f"b: {b}")
```

```
w: 200
b: 100
```

Now, let's compute the value of $f_{w,b}(x^{(i)})$ for your two data points. You can explicitly write this out for each data point as -

for $x^{(0)}$, `f_wb = w * x[0] + b`

for $x^{(1)}$, `f_wb = w * x[1] + b`

For a large number of data points, this can get unwieldy and repetitive. So instead, you can calculate the function output in a `for` loop as shown in the `compute_model_output` function below.

Loading [MathJax]/extensions/Safe.js

> **Note**: The argument description `(ndarray (m,))` describes a Numpy n-dimensional array of shape (m,). `(scalar)` describes an argument without dimensions, just a magnitude.
> **Note**: `np.zero(n)` will return a one-dimensional numpy array with n entries

In [24]:
```
def compute_model_output(x_train, w, b):
    """
    Computes the prediction of a linear model
    Args:
      x (ndarray (m,)): Data, m examples
      w,b (scalar)    : model parameters
    Returns
      y (ndarray (m,)): target values
    """
    m = x_train.shape[0] #returns the size of the first dimension of x_train.
    # y must have same first dimension as x_train:
    f_wb = np.zeros(m) # Initialize f_wb as an array
    for i in range(m):
        f_wb[i] = w * x_train[i] + b

    return f_wb
```

In the compute_model_output() function, np.zeros(m) is used to initialize an array f_wb of zeros with the same shape as the x_train input.

The purpose of np.zeros(m) is to create an array of zeros with m elements, where m represents the number of examples in the x_train dataset. By initializing f_wb with zeros, we ensure that it has the same shape as x_train and can store the predicted output values for each example.

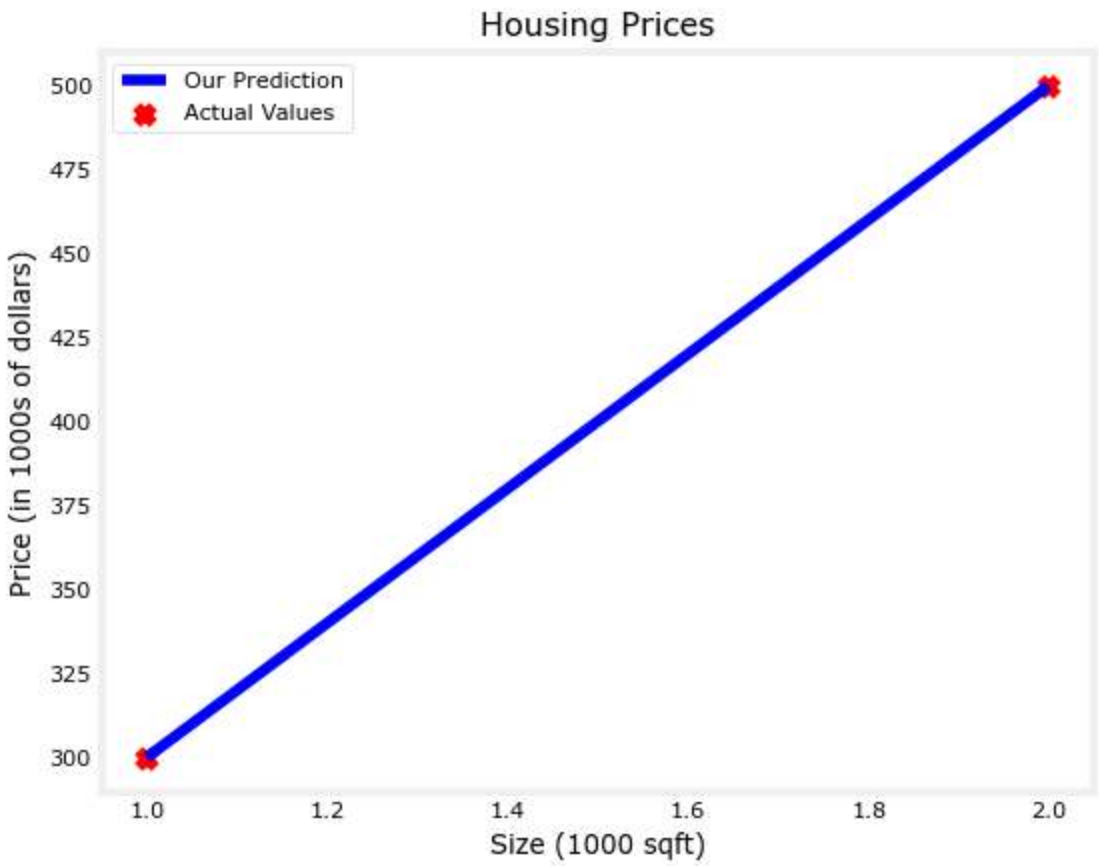
Later in the loop, each element of f_wb is updated with the corresponding predicted output value calculated using the linear equation w * x_train[i] + b. The i index is used to access the specific example in x_train and update the corresponding element in f_wb.

Overall, initializing f_wb with np.zeros(m) allows us to create an array of the same shape as x_train to store the predicted output values.

Now let's call the `compute_model_output` function and plot the output..

In [25]:
```
tmp_f_wb = compute_model_output(x_train, w, b,)

# Plot our model prediction
plt.plot(x_train, tmp_f_wb, c='b',label='Our Prediction')

# Plot the data points
plt.scatter(x_train, y_train, marker='x', c='r',label='Actual Values')

# Set the title
plt.title("Housing Prices")
# Set the y-axis label
plt.ylabel('Price (in 1000s of dollars)')
# Set the x-axis label
plt.xlabel('Size (1000 sqft)')
plt.legend()
plt.show()
```

Loading [MathJax]/extensions/Safe.js



Another way

Loading [MathJax]/extensions/Safe.js

In [86]:
```
m = x_train.shape[0]
tmp_f_wb = []
for _ in range(m):
    f_wb = w * x_train[_] + b
    tmp_f_wb.append(f_wb)
```

Now let's call the `compute_model_output` function and plot the output..

In [87]:
```
# Plot our model prediction
plt.plot(x_train, tmp_f_wb, c='b',label='Our Prediction')

# Plot the data points
plt.scatter(x_train, y_train, marker='x', c='r',label='Actual Values')

# Set the title
plt.title("Housing Prices")
# Set the y-axis label
plt.ylabel('Price (in 1000s of dollars)')
# Set the x-axis label
plt.xlabel('Size (1000 sqft)')
plt.legend()
plt.show()
```

As you can see, setting $w = 200$ and $b = 100$ result in a line that fits our data.

Prediction

Now that we have a model, we can use it to make our original prediction. Let's predict the price of a house with 1200 sqft. Since the units of x are in 1000's of sqft, x is 1.2.

In [20]:
```
w = 200
b = 100
x_i = 1.2
cost_1200sqft = w * x_i + b

print(f"${cost_1200sqft:.0f} thousand dollars")
```

```
$340 thousand dollars
```

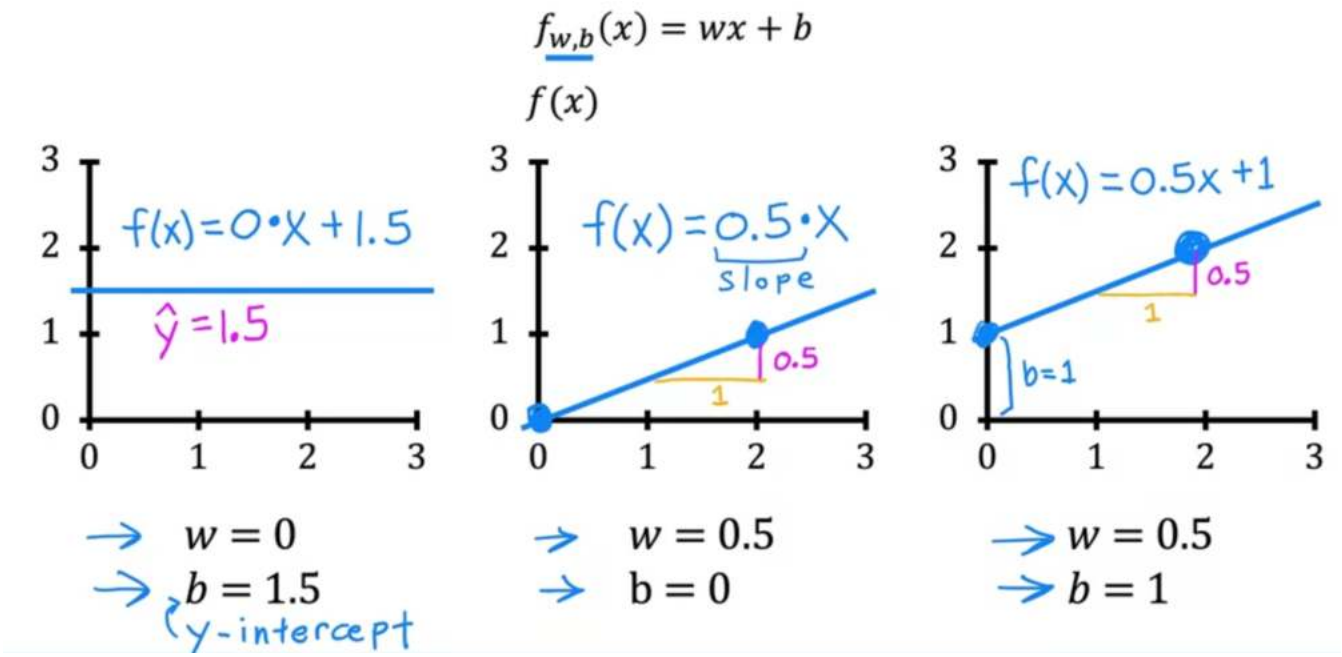
Cost function formula معادلة الخطأ

Loading [MathJax]/extensions/Safe.js

The cost function/ loss function (MSE) is primarily used in optimization problems, particularly in machine learning and statistical modeling. It measures the discrepancy or error between the predicted values and the actual values.

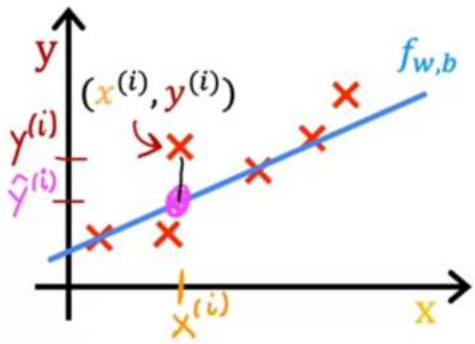
The specific number that determines whether a cost function is considered "bad" or "good" depends on the context and the specific problem being addressed. Generally, a lower value of the cost function is desirable, indicating a better fit of the model to the data.

IT COST W & B AUTOMATICALLY



In machine learning different people will use different cost functions for different applications, but the squared error cost function is by far the most commonly used one for linear regression and for that matter, for all regression problems where it seems to give good results for many applications.

Loading [MathJax]/extensions/Safe.js



$\hat{y}^{(i)} = f_{w,b}(x^{(i)})$ ←

$f_{w,b}(x^{(i)}) = wx^{(i)} + b$

Cost function: Squared error cost function

$$J(w,b) = \frac{1}{2m}\sum_{i=1}^{m}\left(\hat{y}^{(i)} - y^{(i)}\right)^2$$

$\hat{y}^{(i)}$: error

m = number of training examples

$$J(w,b) = \frac{1}{2m}\sum_{i=1}^{m}\left(f_{w,b}(x^{(i)}) - y^{(i)}\right)^2$$

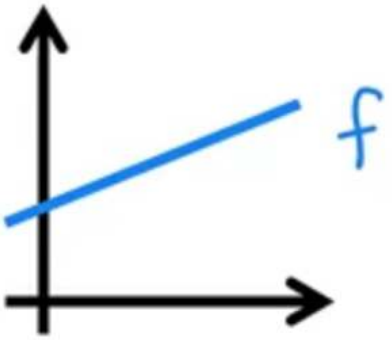
Find w, b:
$\hat{y}^{(i)}$ is close to $y^{(i)}$ for all $(x^{(i)}, y^{(i)})$.

model:

$$f_{w,b}(x) = wx + b$$

parameters:

w, b



cost function:

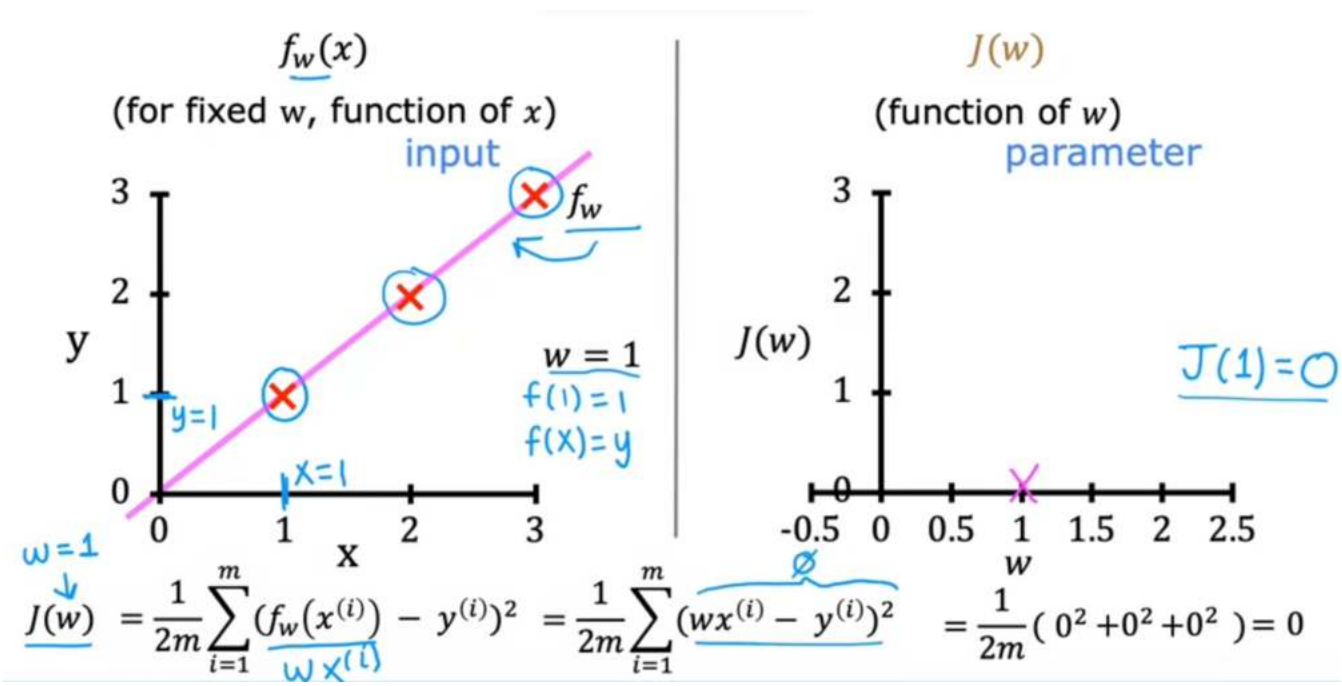
$$J(w,b) = \frac{1}{2m}\sum_{i=1}^{m}\left(f_{w,b}(x^{(i)}) - y^{(i)}\right)^2$$

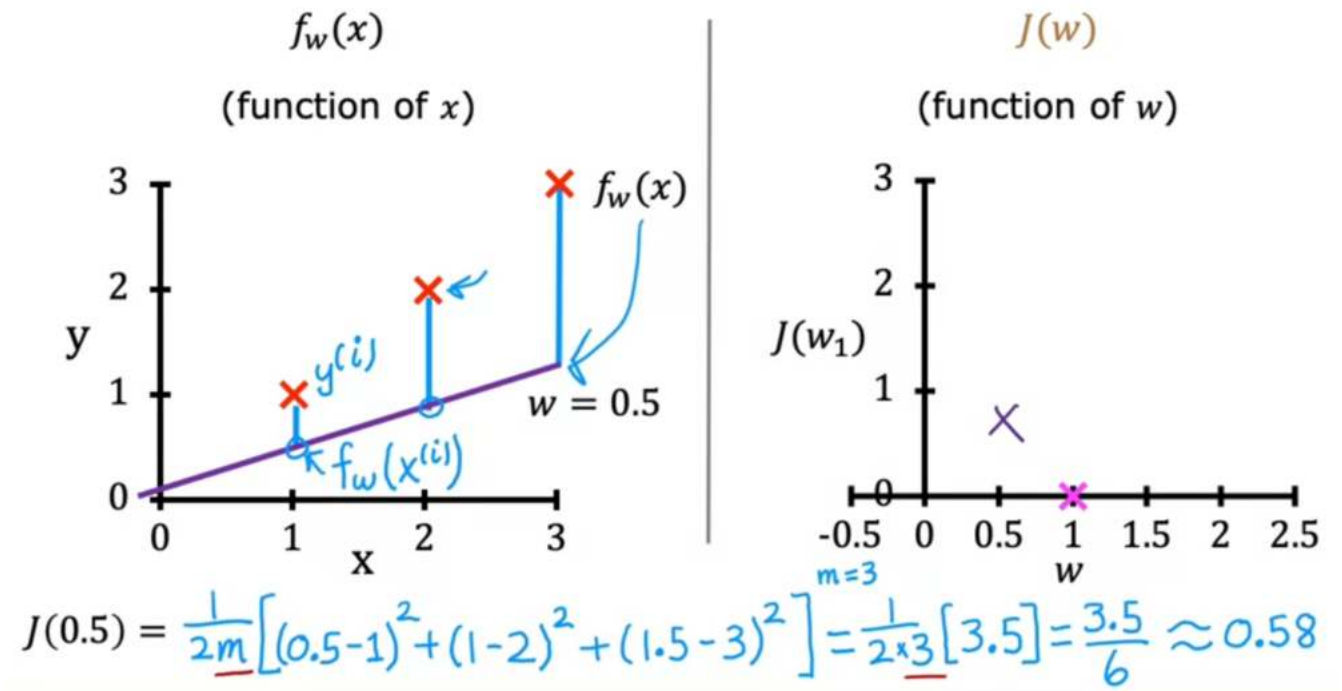
goal:

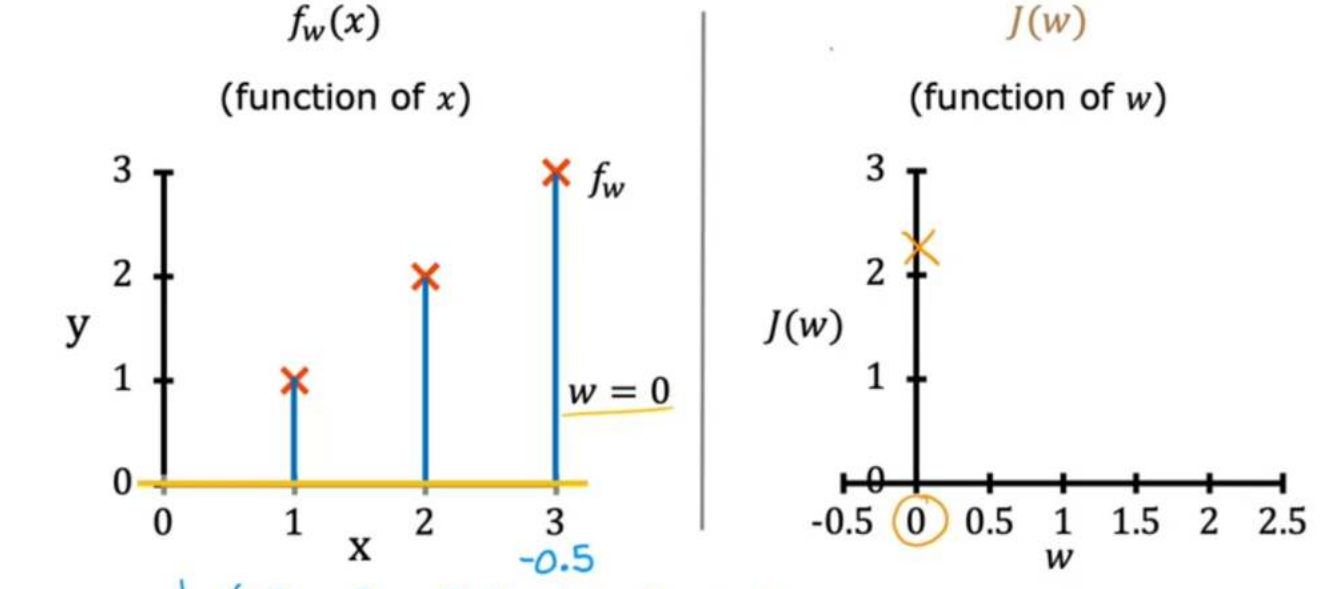
$$\underset{w,b}{\text{minimize}}\, J(w,b)$$

Cost function intuition

Loading [MathJax]/extensions/Safe.js

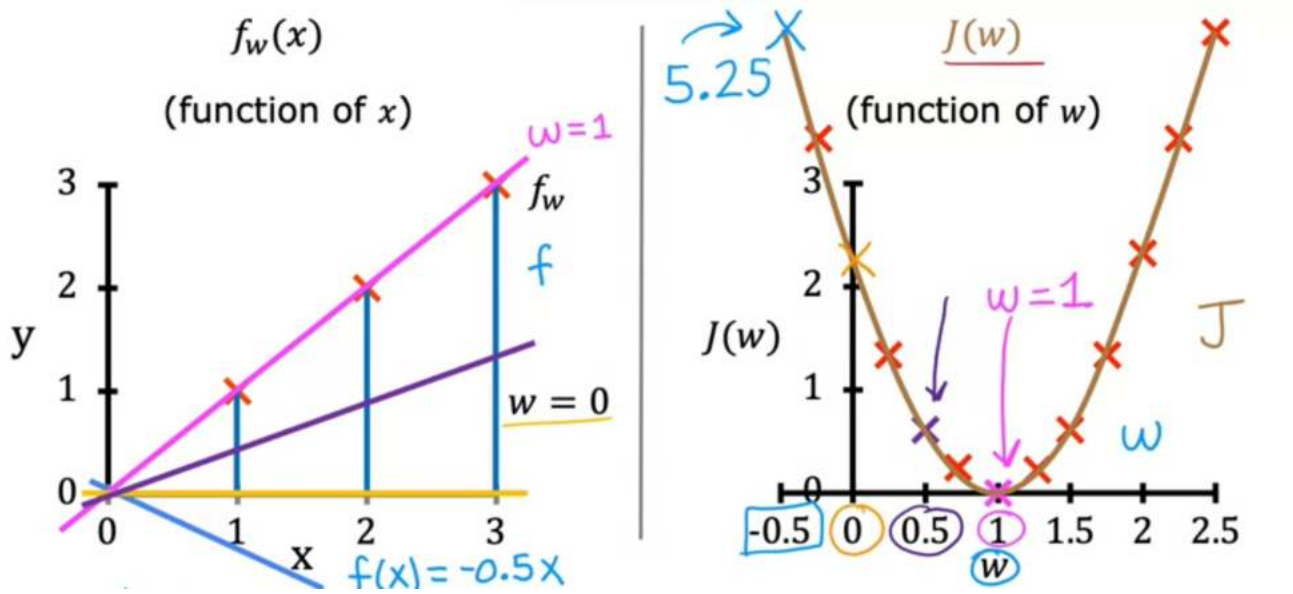






Loading [MathJax]/extensions/Safe.js

You can continue computing the cost function for different values of w and so on and plot these. It turns out that by computing a range of values, you can slowly trace out what the cost function J looks like and that's what J is.



How to choose w ?

well choosing w baised in the minimize J(w) which it **w = 1**

goal of linear regression:

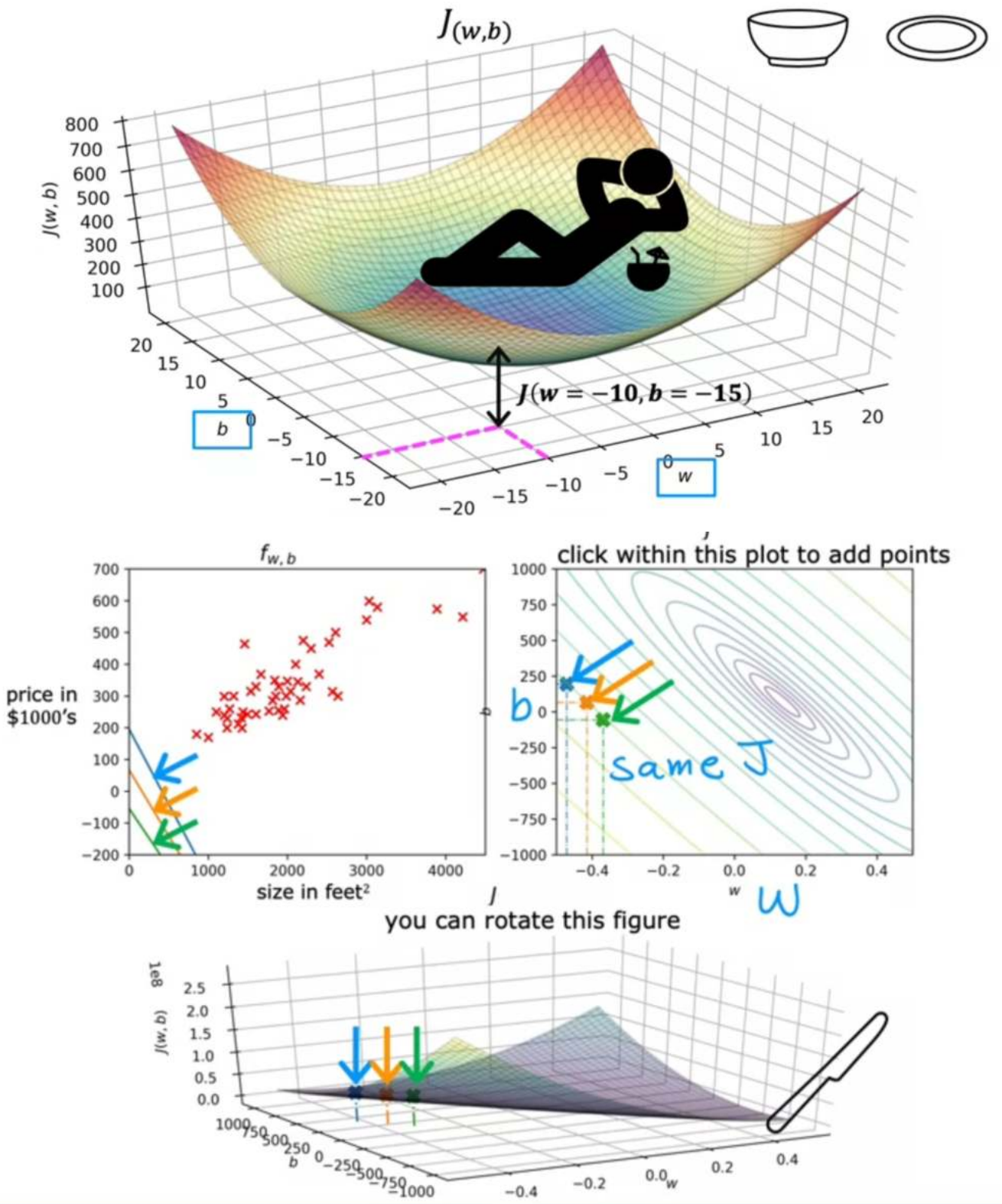
$$\underset{w}{\text{minimize}}\, J(w)$$

general case:

$$\underset{w,b}{\text{minimize}}\, J(w,b)$$

Visualizing the cost function

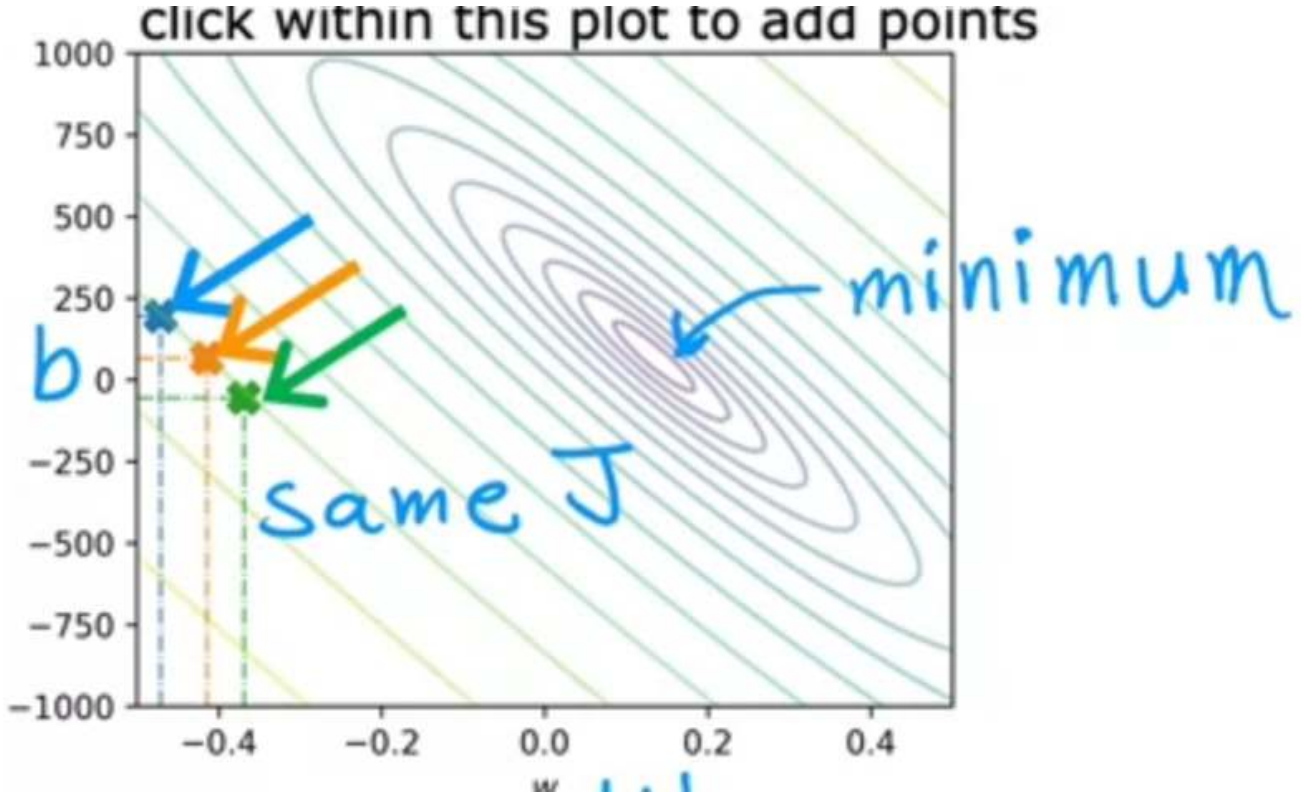
Loading [MathJax]/extensions/Safe.js



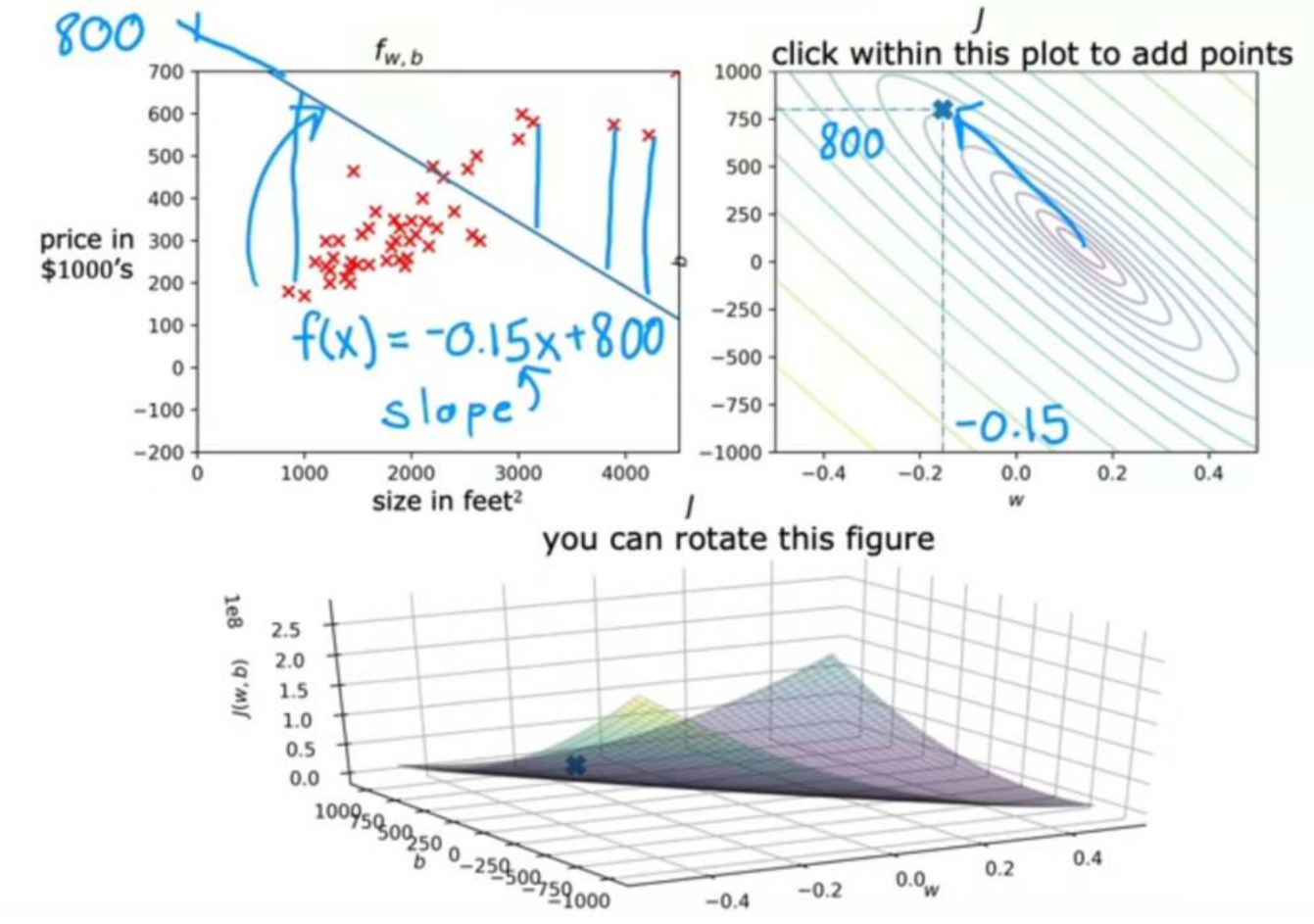
frfom the above visualization we can decide that the predecting of the three plot are very bad

Loading [MathJax]/extensions/Safe.js

The nearst plot to the bottom the best prediction

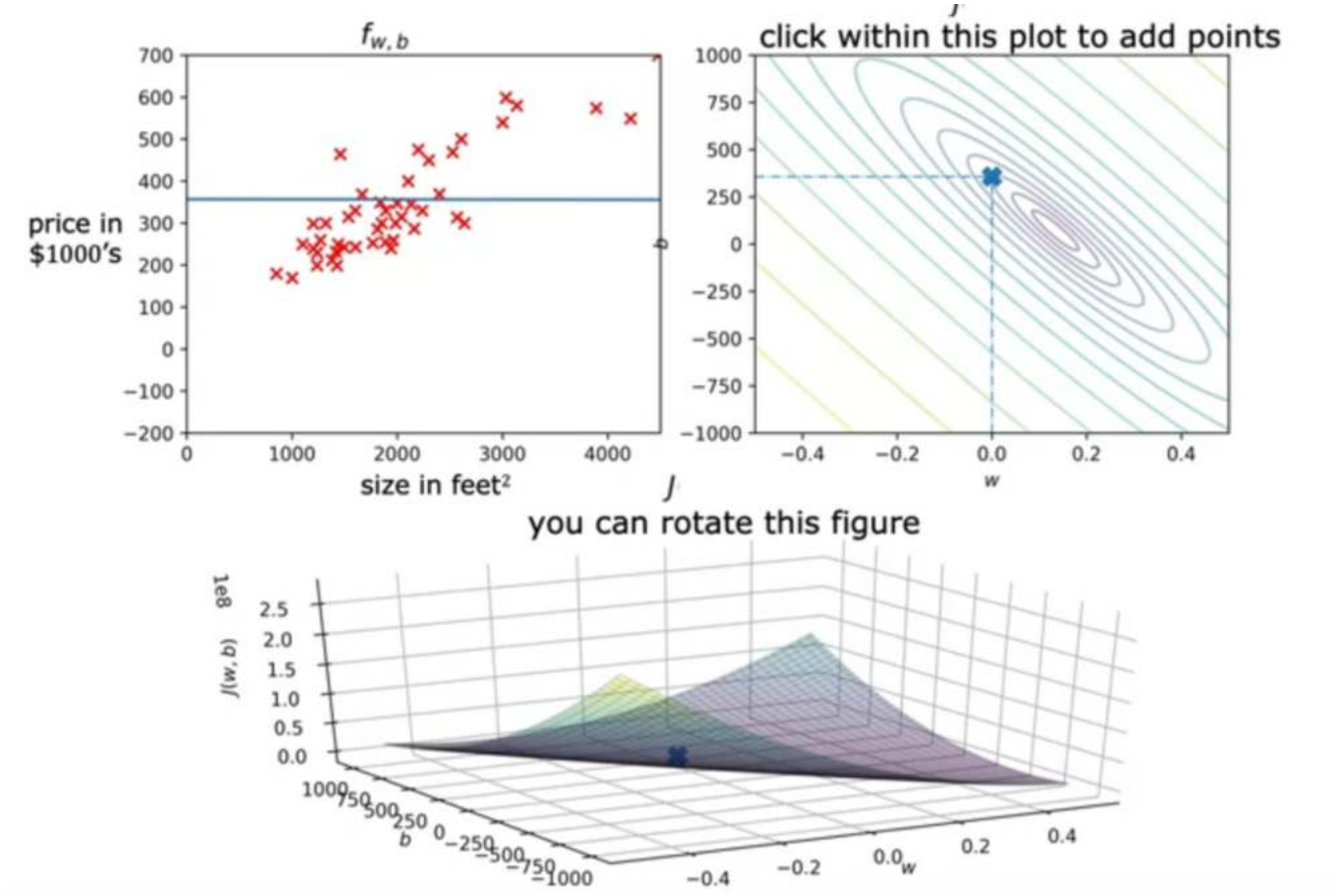


The below figure, it is pretty far from the minimum so it is pretty high cost and the predection is bad

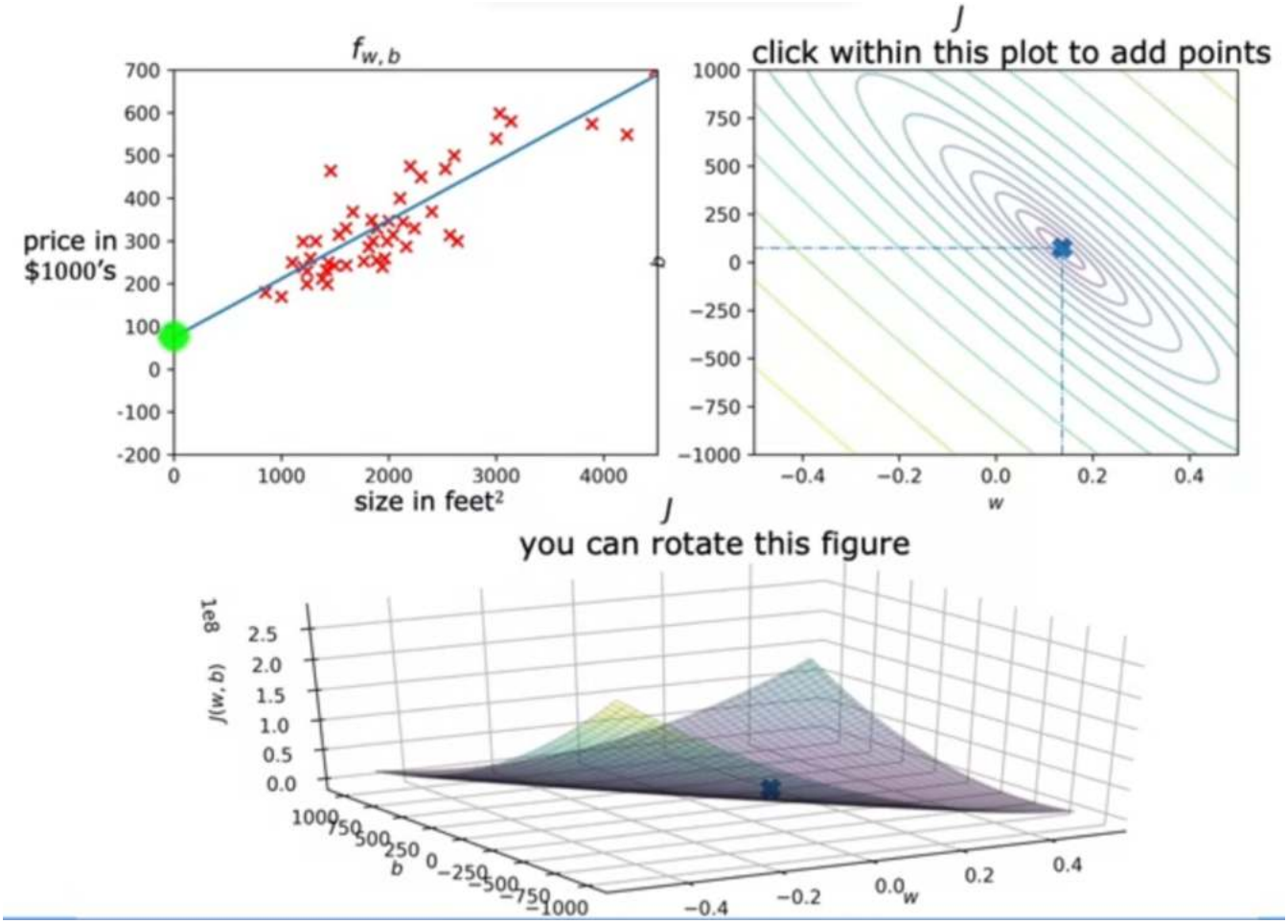


Loading [MathJax]/extensions/Safe.js

The below figure, is slightly less bad



**The below is very good predicton & low cost



Loading [MathJax]/extensions/Safe.js

Lab: Cost Function

In this lab you will:

- you will implement and explore the `cost` function for linear regression with one variable.

Tools

In this lab we will make use of:

- NumPy, a popular library for scientific computing
- Matplotlib, a popular library for plotting data
- local plotting routines in the lab_utils_uni.py file in the local directory

In [2]:
```
import numpy as np
%matplotlib widget
import matplotlib.pyplot as plt
from lab_utils_uni import plt_intuition, plt_stationary, plt_update_onclick, soup_bowl
plt.style.use('deeplearning.mplstyle')
```

Problem Statement

You would like a model which can predict housing prices given the size of the house.
Let's use the same two data points as before the previous lab- a house with 1000 square feet sold for \$300,000 and a house with 2000 square feet sold for \\$500,000.

Size (1000 sqft)	Price (1000s of dollars)
1	300
2	500

In [3]:
```
x_train = np.array([1.0, 2.0])   #(size in 1000 square feet)
y_train = np.array([300.0, 500.0])           #(price in 1000s of dollars)
```

Computing Cost

The term 'cost' in this assignment might be a little confusing since the data is housing cost. Here, cost is a measure how well our model is predicting the target price of the house. The term 'price' is used for housing data.

The equation for cost with one variable is: $$J(w,b) = \frac{1}{2m} \sum\limits_{i = 0}^{m-1} (f_{w,b}(x^{(i)}) - y^{(i)})^2 \tag{1}$$

where $$f_{w,b}(x^{(i)}) = wx^{(i)} + b \tag{2}$$

- $f_{w,b}(x^{(i)})$ is our prediction for example i using parameters w,b.
- $(f_{w,b}(x^{(i)}) -y^{(i)})^2$ is the squared difference between the target value and the prediction.
- These differences are summed over all the m examples and divided by `2m` to produce the cost, $J(w,b)$.

Loading [MathJax]/extensions/Safe.js

> Note, in lecture summation ranges are typically from 1 to m, while code will be from 0 to m-1.

The code below calculates cost by looping over each example. In each loop:

- `f_wb`, a prediction is calculated
- the difference between the target and the prediction is calculated and squared.
- this is added to the total cost.

In [4]:
```
def compute_cost(x, y, w, b):
    m = x.shape[0]

    cost_sum=0
    for _ in range(m):
        f_wb[_] = w * x[_] + b
        cost = (f_wb - y[_]) **2
        cost_sum = cost_sum + cost
    total_cost = (1 / (2 * m))* cost_sum

    return total_cost
```

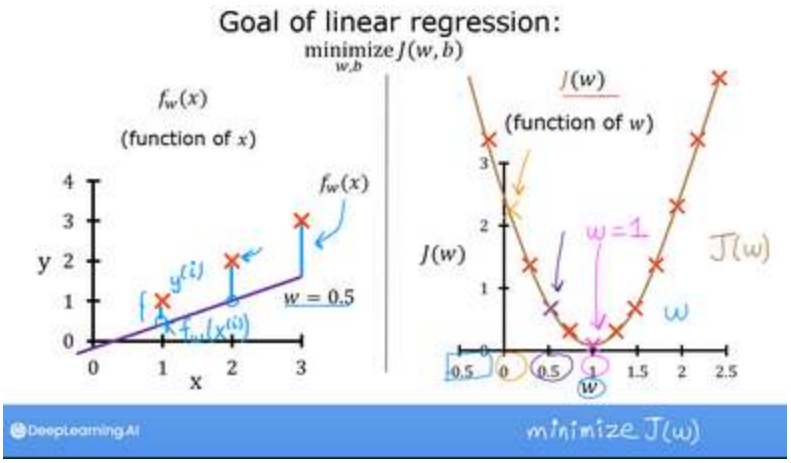
Cost Function Intuition

Your goal is to find a model $f_{w,b}(x) = wx + b$, with parameters w,b, which will accurately predict house values given an input x. The cost is a measure of how accurate the model is on the training data.

The cost equation (1) above shows that if w and b can be selected such that the predictions $f_{w,b}(x)$ match the target data y, the $(f_{w,b}(x^{(i)}) - y^{(i)})^2 $ term will be zero and the cost minimized. In this simple two point example, you can achieve this!

In the previous lab, you determined that $b=100$ provided an optimal solution so let's set b to 100 and focus on w.

Below, use the slider control to select the value of w that minimizes cost. It can take a few seconds for the plot to update.



In [6]:
```
plt_intuition(x_train,y_train)
```

```
interactive(children=(IntSlider(value=150, description='w', max=400, step=10), Output
sses=('widge…
```

Loading [MathJax]/extensions/Safe.js

The plot contains a few points that are worth mentioning.

- cost is minimized when $w = 200$, which matches results from the previous lab
- Because the difference between the target and pediction is squared in the cost equation, the cost increases rapidly when w is either too large or too small.
- Using the `w` and `b` selected by minimizing cost results in a line which is a perfect fit to the data.

Cost Function Visualization- 3D

You can see how cost varies with respect to *both* `w` and `b` by plotting in 3D or using a contour plot.
It is worth noting that some of the plotting in this course can become quite involved. The plotting routines are provided and while it can be instructive to read through the code to become familiar with the methods, it is not needed to complete the course successfully. The routines are in lab_utils_uni.py in the local directory.

Larger Data Set

It is instructive to view a scenario with a few more data points. This data set includes data points that do not fall on the same line. What does that mean for the cost equation? Can we find w, and b that will give us a cost of 0?

In [7]:
```
x_train = np.array([1.0, 1.7, 2.0, 2.5, 3.0, 3.2])
y_train = np.array([250, 300, 480,  430,   630, 730,])
```

In the contour plot, click on a point to select `w` and `b` to achieve the lowest cost. Use the contours to guide your selections. Note, it can take a few seconds to update the graph.

Loading [MathJax]/extensions/Safe.js

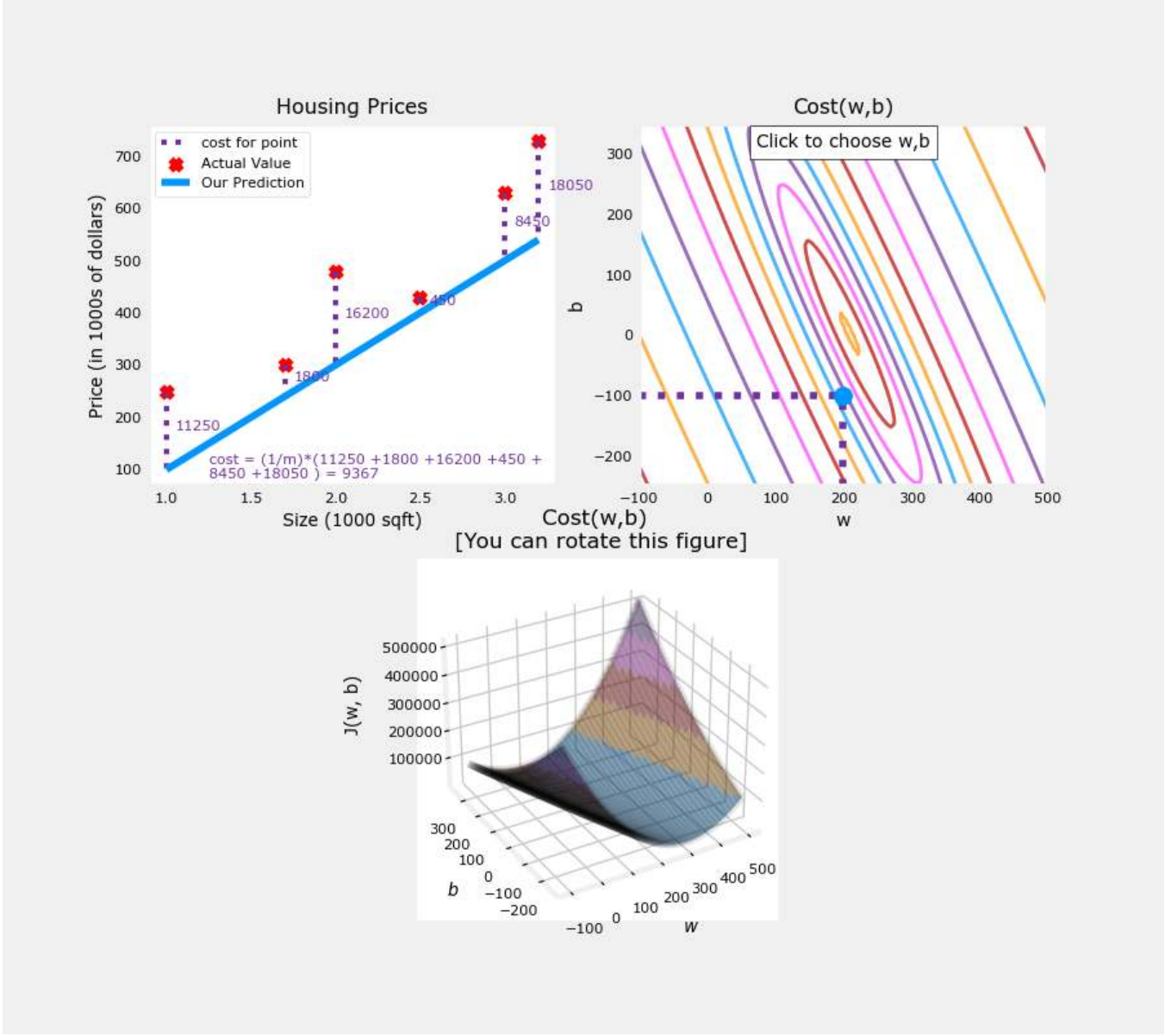
1. `plt.close('all')`: This line closes all currently open figures in Matplotlib. It ensures a clean slate before creating a new figure.
2. `fig, ax, dyn_items = plt_stationary(x_train, y_train)`: This line calls the function `plt_stationary` with the arguments `x_train` and `y_train`. The function `plt_stationary` returns three objects: `fig` (the Figure object), `ax` (the Axes object), and `dyn_items` (a collection of dynamic elements in the plot). These objects are assigned to the variables `fig`, `ax`, and `dyn_items`, respectively. This line essentially initializes the figure and axes for the plot and retrieves the dynamic elements associated with it.
3. `updater = plt_update_onclick(fig, ax, x_train, y_train, dyn_items)`: This line calls the function `plt_update_onclick` with the arguments `fig`, `ax`, `x_train`, `y_train`, and `dyn_items`. The function `plt_update_onclick` returns an updater object, which is assigned to the variable `updater`. This updater object enables interactivity in the plot by defining behavior upon mouse click events.

The purpose of these lines of code is to set up an interactive plot using Matplotlib. It first closes any existing figures, then creates a new figure and axes using `plt_stationary`. It also retrieves dynamic elements associated with the plot. Finally, it sets up an updater object using `plt_update_onclick` to enable interactivity in the plot.

In [11]:
```
fig, ax, dyn_items = plt_stationary(x_train, y_train);
updater = plt_update_onclick(fig, ax, x_train, y_train, dyn_items)
# benefit of update is update the three figs if you click in diff cost(w,b)

#try and click in cost fig to change w & b so you now the diff
# plt_update_onclick(fig, ax, x_train, y_train, dyn_items);
#it will not do anything
```

Loading [MathJax]/extensions/Safe.js

Figure

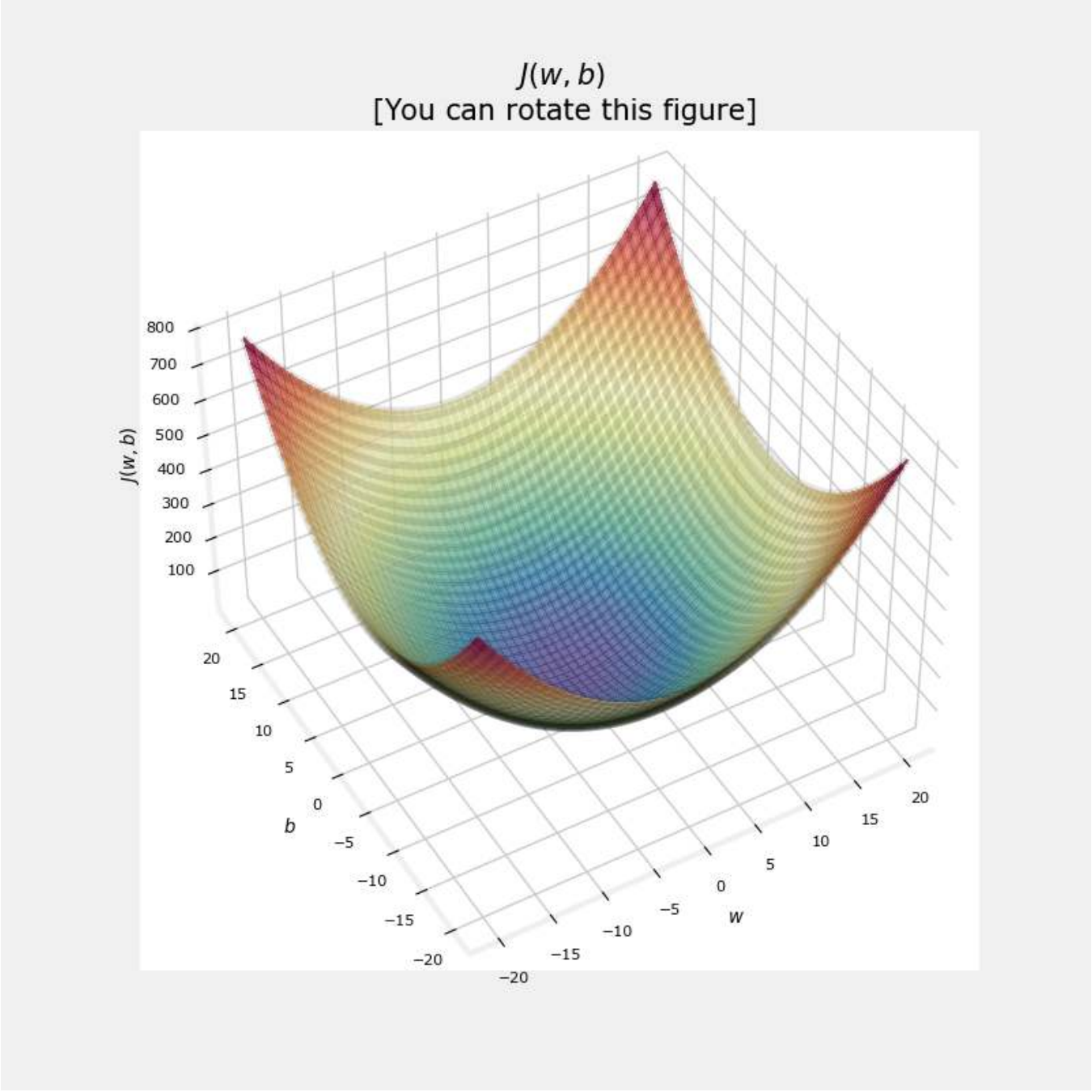


Above, note the dashed lines in the left plot. These represent the portion of the cost contributed by each example in your training set. In this case, values of approximately $w=209$ and $b=2.4$ provide low cost. Note that, because our training examples are not on a line, the minimum cost is not zero.

In [12]:
```
soup_bowl()
#Convex function:
```

Loading [MathJax]/extensions/Safe.js

Figure



!!!

> **Once you get to more complex machine learning models and working with large numbers , cost funtion will not be an efficient algorithm. So, there is an algorithm that do better efficient called Gradiant Descent,. This algorithm is one of the most important algorithns in machine learning, not just for linear regression but some of the biggest and most complex models in all of AI**
>
> !!!

Gradient Descent

Loading [MathJax]/extensions/Safe.js

Used for alot of complex models in all AI.

1. This algorithm is one of the most important algorithns in machine learning
2. Used for some of the most advanced neural network models (deep learning)

Have some function $J(w,b)$ for linear regression or any function

Want $\min_{w,b} J(w,b)$ $\min_{w_1,\dots,w_n,b} J(w_1,w_2,\dots,w_n,b)$

Outline:

Start with some w,b (set $w=0$, $b=0$)

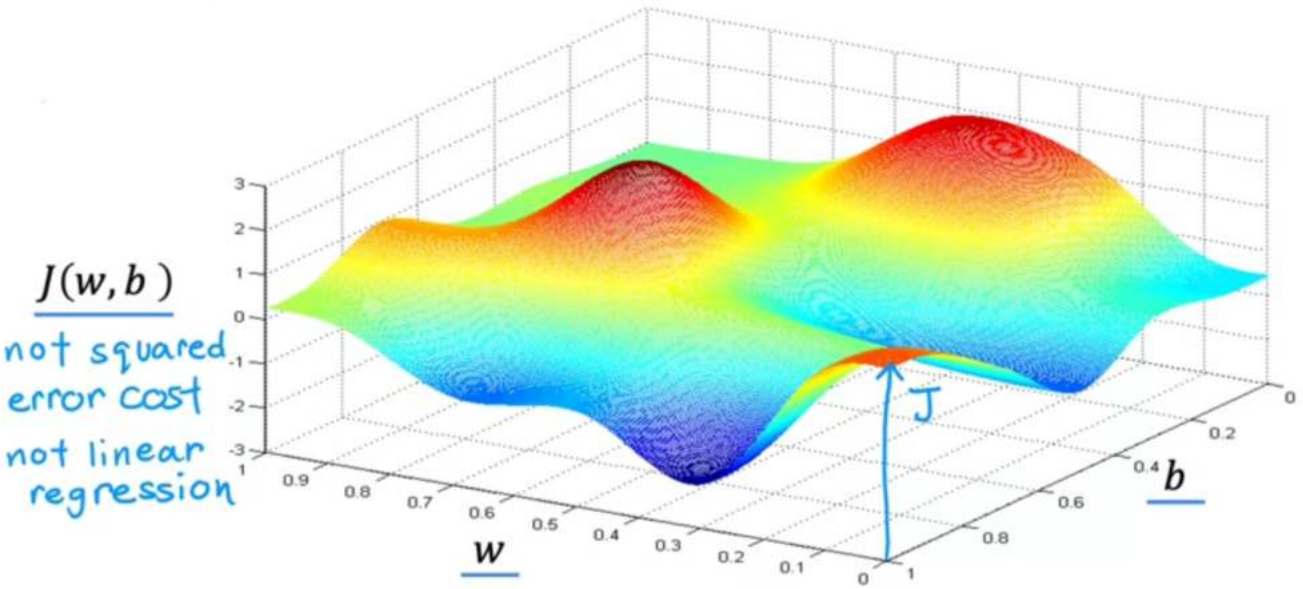
Keep changing w,b to reduce $J(w,b)$

Until we settle at or near a minimum

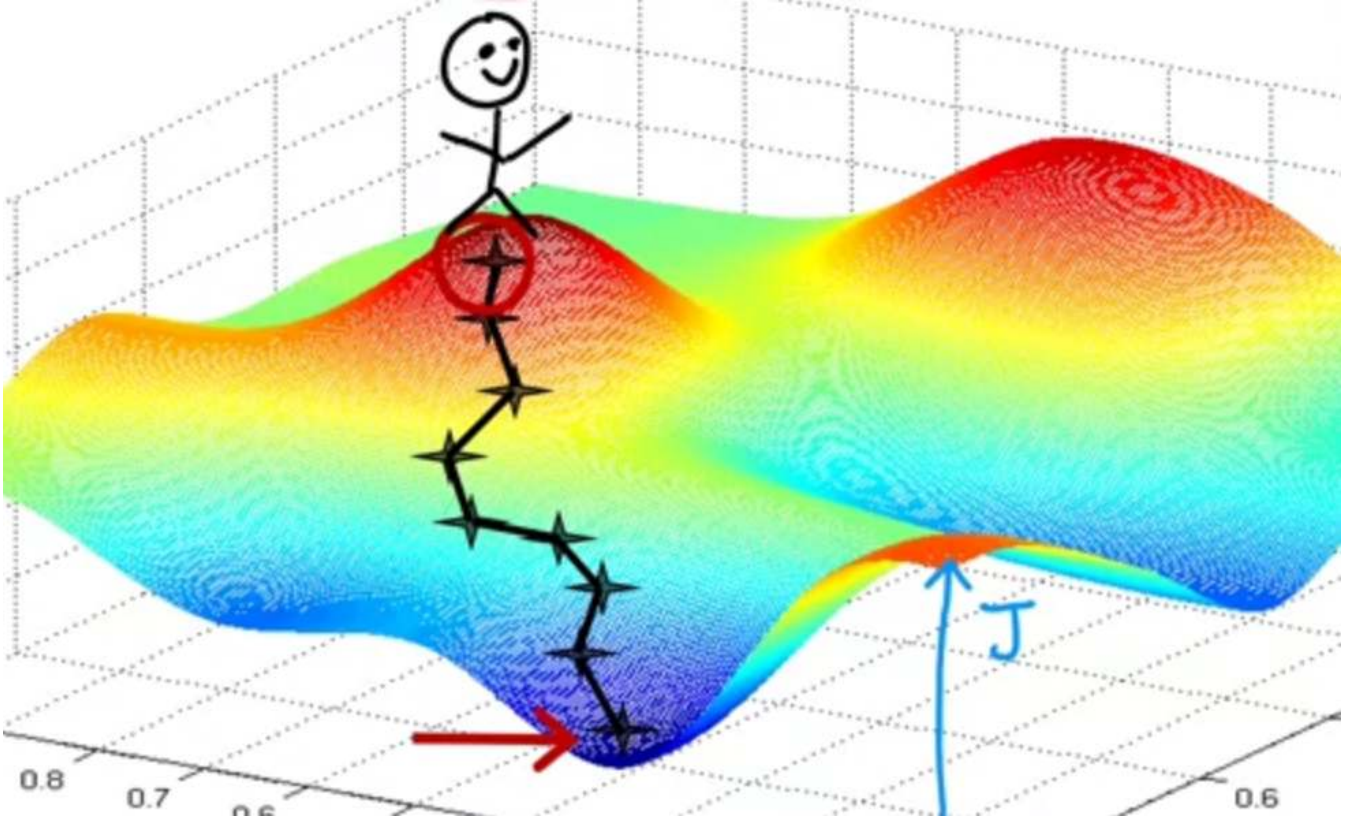
J not always

may have >1 minimum

gradient descent



Loading [MathJax]/extensions/Safe.js



Implementing gradient descent

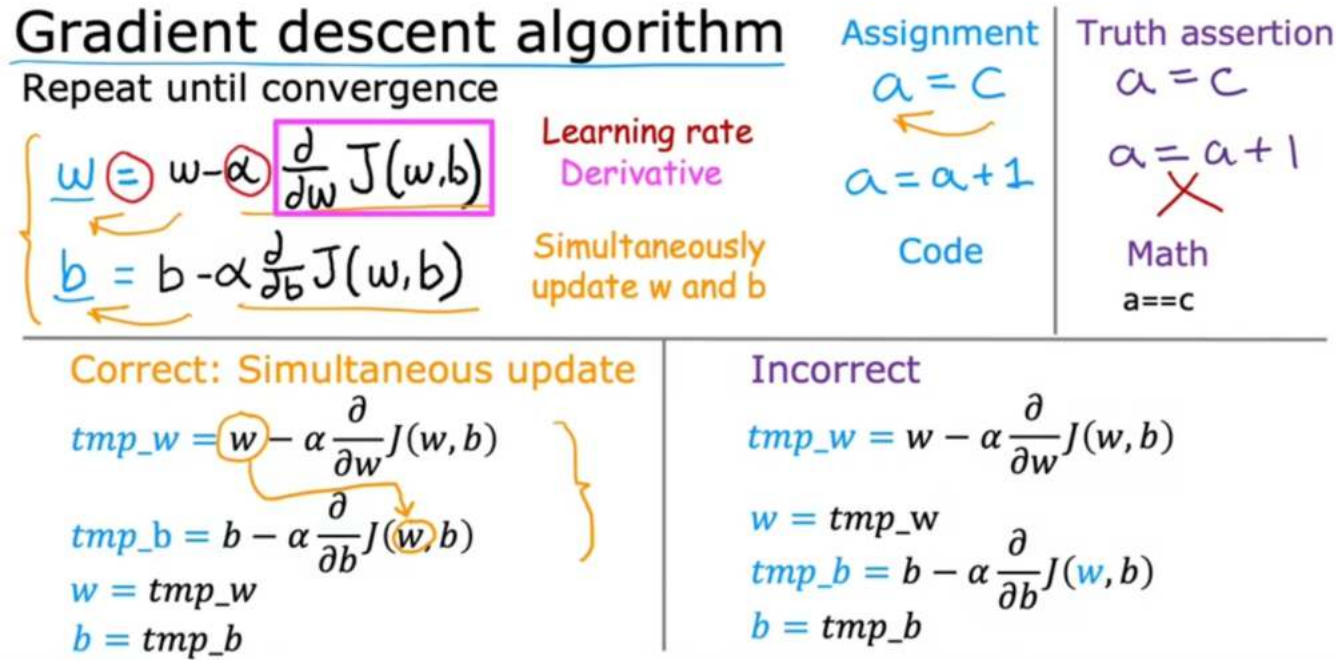
$$w = w - \alpha \frac{d}{dw} J(w,b)$$

Alpha is the learning rate So, if alpha is very large then that corresponds to a very aggressive gradient descent procedure where you are trying to take huge steps downhill. If Alpha is very small, then you'd be taking small baby steps downhill.

The derivative term of the cost function J- is telling us in which direction we want to take our baby step.

Loading [MathJax]/extensions/Safe.js

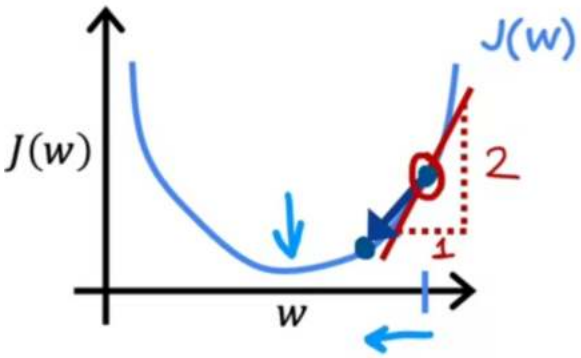
Here we have two parameters which are w & b NOT just w



This update takes place for both parameters, w and b. One important detail is that for gradient descent, you want to simultaneously update w and b, meaning you want to update both parameters at the same time.

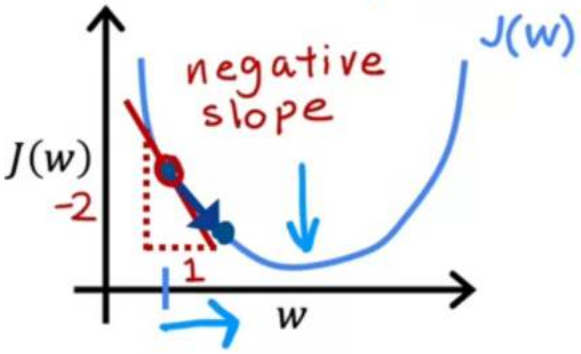
Gradient descent intuition

Loading [MathJax]/extensions/Safe.js



$$w = w - \alpha \underbrace{\frac{d}{dw} J(w)}_{>0}$$

$$w = w - \alpha \cdot (\text{positive number})$$



$$\underbrace{\frac{d}{dw} J(w)}_{<0}$$

$$w = w - \alpha \cdot (\text{negative number})$$

Assume the learning rate α is a small positive number. When $\frac{\partial J(w,b)}{\partial w}$ is a positive number (greater than zero) -- as in the example in the upper part of the slide shown above -- what happens to w after one update step?

- w increases
- It is not possible to tell if w will increase or decrease.
- w stays the same
- w decreases.

Correct

The learning rate α is always a positive number, so if you take W minus a positive number, you end up with a new value for W that is smaller

Learning rate (Alpha)

$$w = w - \alpha \frac{d}{dw} J(w)$$

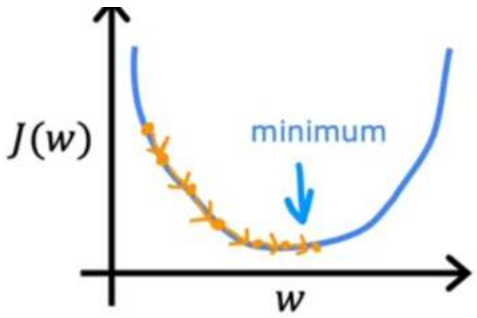
If α is too small...

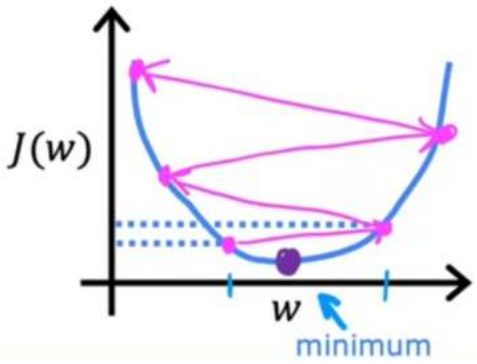
Gradient descent may be slow.

If α is too large...

Gradient descent may:

- Overshoot, never reach minimum
- Fail to converge, diverge





Gradient descent for linear regression

Loading [MathJax]/extensions/Safe.js

Linear regression model

$$f_{w,b}(x) = wx + b$$

Cost function

$$J(w,b) = \frac{1}{2m}\sum_{i=1}^{m}(f_{w,b}(x^{(i)}) - y^{(i)})^2$$

Gradient descent algorithm

repeat until convergence {

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

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

}

To calculate the derivative of cost function J with respect to w

(Optional)

$$\frac{\partial}{\partial w}J(w,b) = \frac{d}{dw}\frac{1}{2m}\sum_{i=1}^{m}(f_{w,b}(x^{(i)}) - y^{(i)})^2 = \frac{d}{dw}\frac{1}{2m}\sum_{i=1}^{m}(wx^{(i)} + b - y^{(i)})^2$$

$$= \frac{1}{2m}\sum_{i=1}^{m}(wx^{(i)} + b - y^{(i)})2x^{(i)} = \frac{1}{m}\sum_{i=1}^{m}(f_{w,b}(x^{(i)}) - y^{(i)})x^{(i)}$$

$$\frac{\partial}{\partial b}J(w,b) = \frac{d}{db}\frac{1}{2m}\sum_{i=1}^{m}(f_{w,b}(x^{(i)}) - y^{(i)})^2 = \frac{d}{db}\frac{1}{2m}\sum_{i=1}^{m}(wx^{(i)} + b - y^{(i)})^2$$

$$= \frac{1}{2m}\sum_{i=1}^{m}(wx^{(i)} + b - y^{(i)})2 = \frac{1}{m}\sum_{i=1}^{m}(f_{w,b}(x^{(i)}) - y^{(i)})$$

no $x^{(i)}$

Loading [MathJax]/extensions/Safe.js

Gradient descent algorithm

repeat until convergence {

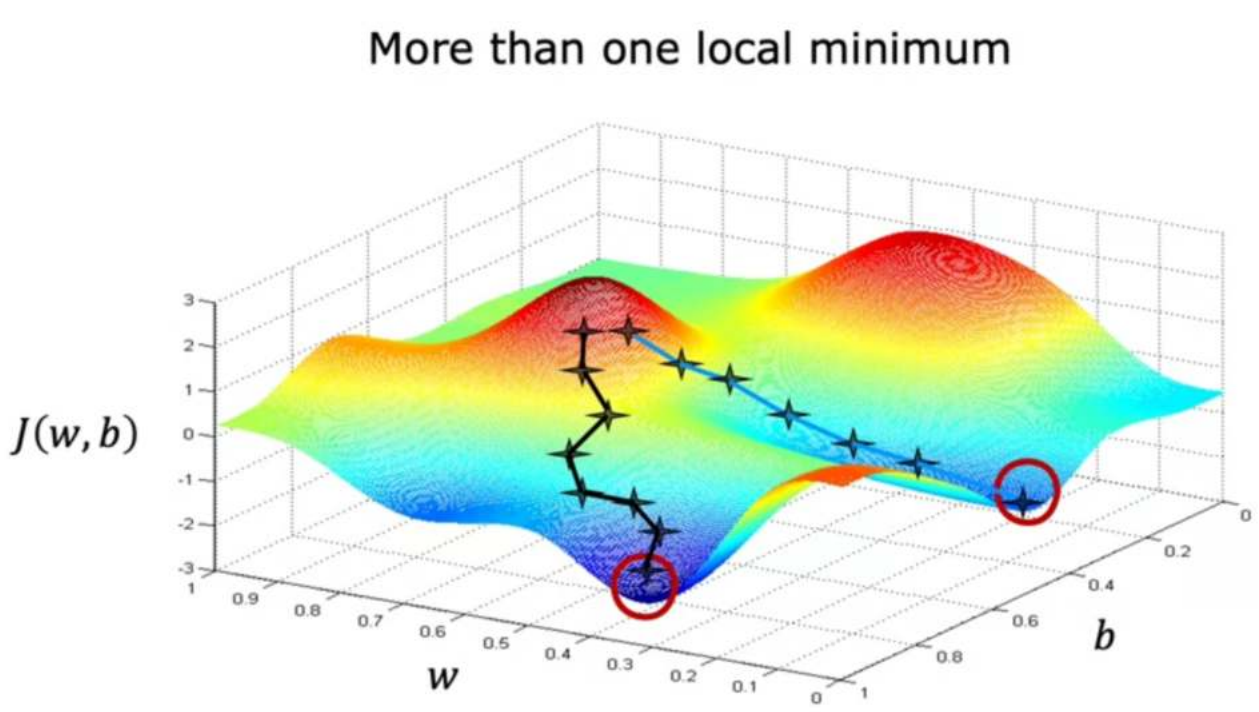
$$w = w - \alpha \frac{1}{m} \sum_{i=1}^{m} (f_{w,b}(x^{(i)}) - y^{(i)}) \quad x^{(i)}$$

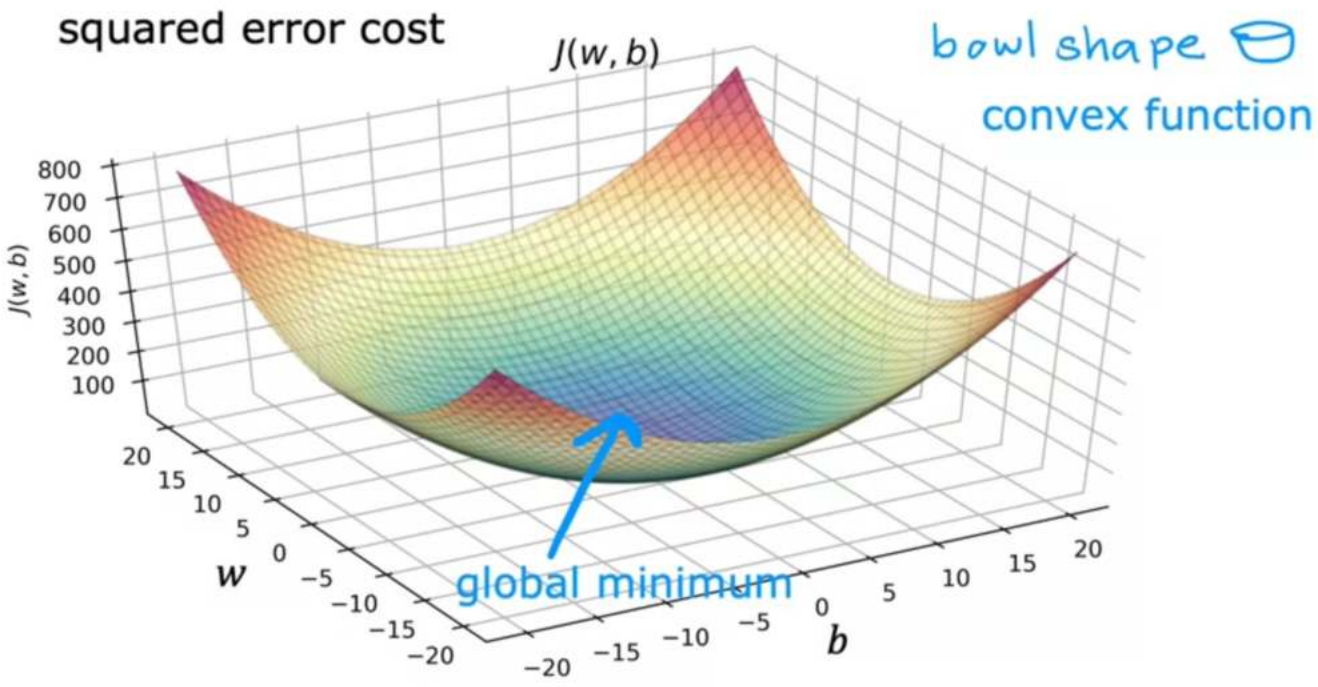
$$b = b - \alpha \frac{1}{m} \sum_{i=1}^{m} (f_{w,b}(x^{(i)}) - y^{(i)})$$

}

More different between the convex function and gradient descent is that gradient descent has more than onle local minimum. But convex function has only one

Loading [MathJax]/extensions/Safe.js



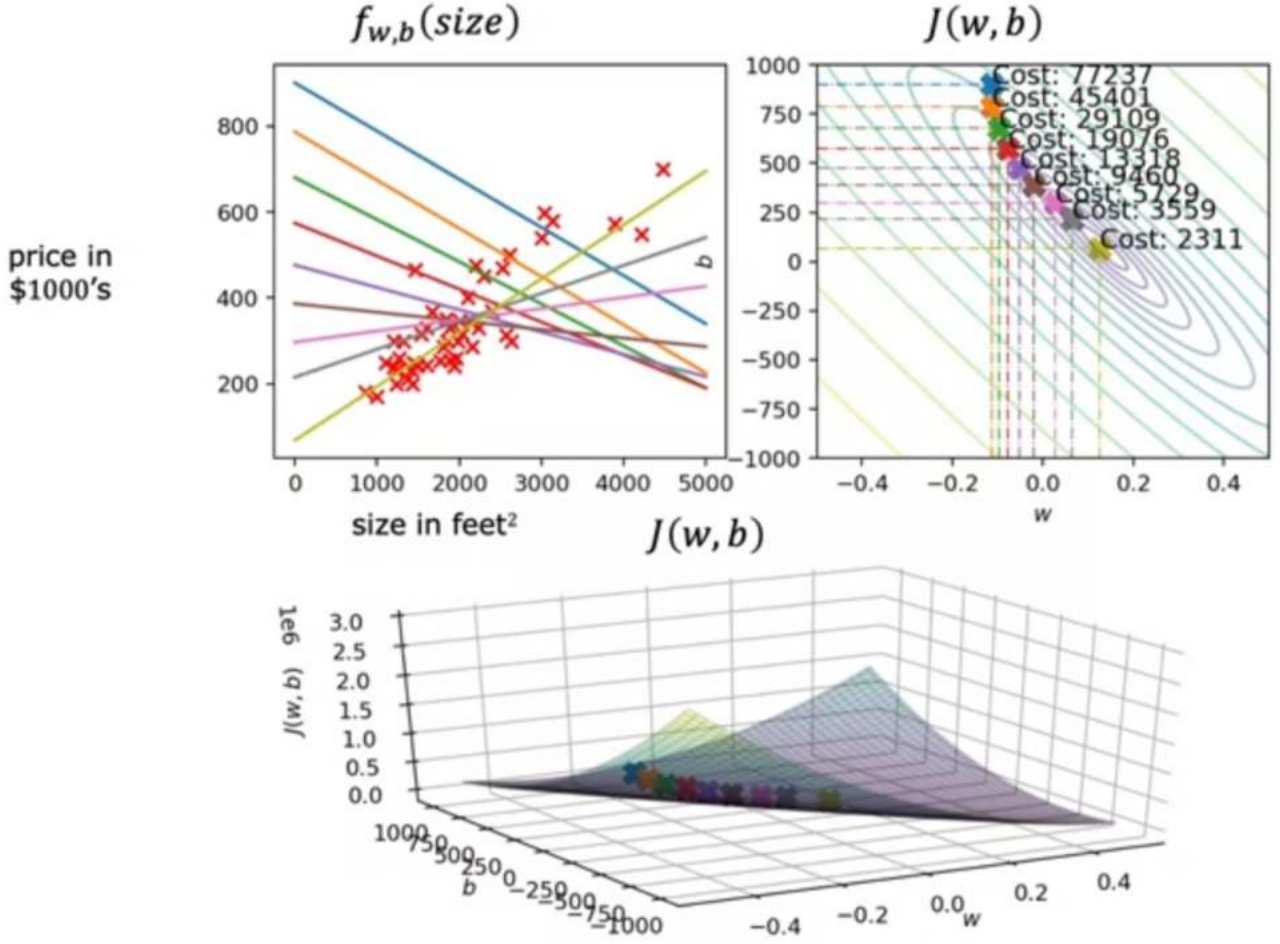


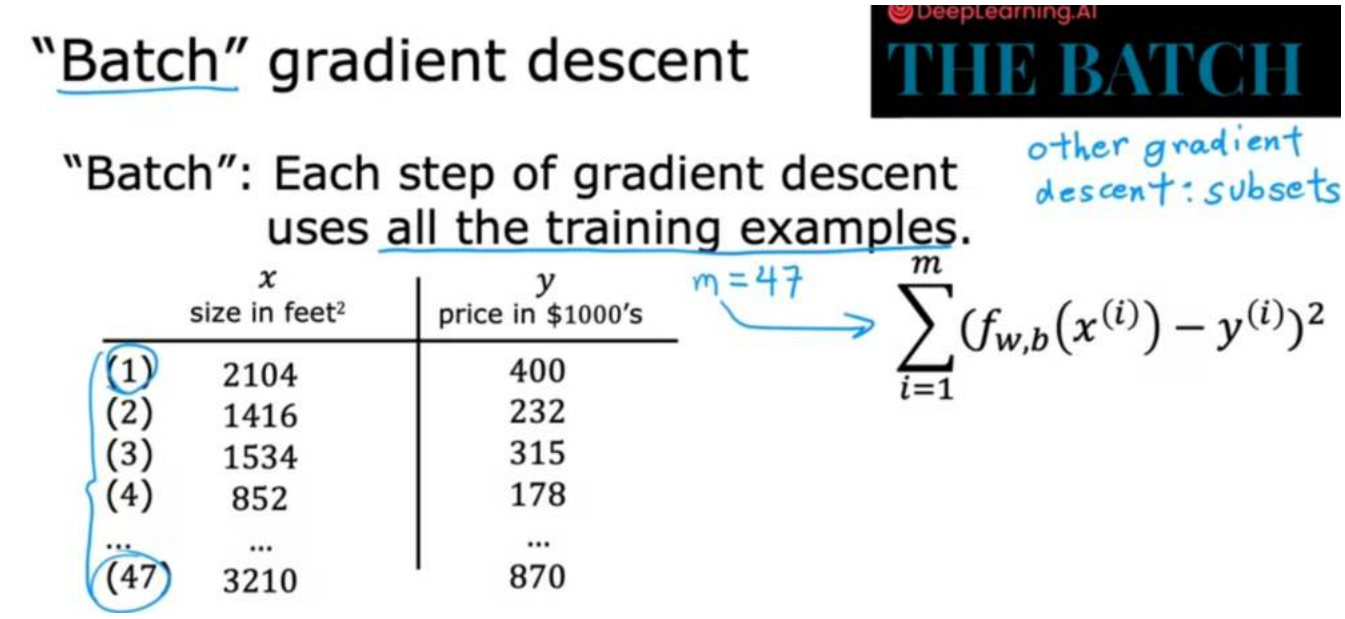
So, when you implement gradient descent on convex function, one nice property is that it will always converg to the global minimum

Running gradient descent

As you take more of these steps, the cost is decreasing at each update. So the parameters w and b are following this trajectory.

Loading [MathJax]/extensions/Safe.js





	x size in feet²	y price in $1000's
(1)	2104	400
(2)	1416	232
(3)	1534	315
(4)	852	178
...	...	...
(47)	3210	870

---

lab: Gradient descent for Linear Regression

In this lab, you will:

- automate the process of optimizing w and b using gradient descent.

Tools

Loading [MathJax]/extensions/Safe.js

In this lab, we will make use of:

- NumPy, a popular library for scientific computing
- Matplotlib, a popular library for plotting data
- plotting routines in the lab_utils.py file in the local directory

Problem Statement

Let's use the same two data points as before - a house with 1000 square feet sold for \$300,000 and a house with 2000 square feet sold for \\$500,000.

Size (1000 sqft)	Price (1000s of dollars)
1	300
2	500

In [34]:
```
# Load our data set
x_train = np.array([1.0,2.0])  #features
y_train = np.array([300.0, 500.0])   #target value
```

Compute_Cost

This was developed in the last lab. We'll need it again here.

In [36]:
```
def compute_cost(x, y, w, b):

    m = x_train.shape[0]

    cost = 0
    for i in range(m):
        f_wb = w*x[i] + b
        cost = cost + (f_wb - y[i])**2
    total_cost = 1 / (2 * m) * cost

    return total_cost
```

Gradient descent summary

So far in this course, you have developed a linear model that predicts $f_{w,b}(x^{(i)})$: $$f_{w,b}(x^{(i)}) = wx^{(i)} + b \tag{1}$$ In linear regression, you utilize input training data to fit the parameters w,b by minimizing a measure of the error between our predictions $f_{w,b}(x^{(i)})$ and the actual data $y^{(i)}$. The measure is called the $cost$, $J(w,b)$. In training you measure the cost over all of our training samples $x^{(i)},y^{(i)}$ $$J(w,b) = \frac{1}{2m} \sum\limits_{i = 0}^{m-1} (f_{w,b}(x^{(i)}) - y^{(i)})^2\tag{2}$$

In lecture, *gradient descent* was described as:

$$\begin{align*} \text{repeat}&\text{ until convergence:} \; \lbrace \newline \; w &= w - \alpha \frac{\partial J(w,b)}{\partial w} \tag{3} \; \newline b &= b - \alpha \frac{\partial J(w,b)}{\partial b} \newline \rbrace \end{align*}$$

Loading [MathJax]/extensions/Safe.js

where, parameters w, b are updated simultaneously.
The gradient is defined as: $$ \begin{align} \frac{\partial J(w,b)}{\partial w} &= \frac{1}{m} \sum\limits_{i = 0}^{m-1} (f_{w,b}(x^{(i)}) - y^{(i)})x^{(i)} \tag{4}\\ \frac{\partial J(w,b)}{\partial b} &= \frac{1}{m} \sum\limits_{i = 0}^{m-1} (f_{w,b}(x^{(i)}) - y^{(i)}) \tag{5}\\ \end{align} $$

Here *simultaniously* means that you calculate the partial derivatives for all the parameters before updating any of the parameters.

Implement Gradient Descent

You will implement gradient descent algorithm for one feature. You will need three functions.

- `compute_gradient` implementing equation (4) and (5) above
- `compute_cost` implementing equation (2) above (code from previous lab)
- `gradient_descent`, utilizing compute_gradient and compute_cost

Conventions:

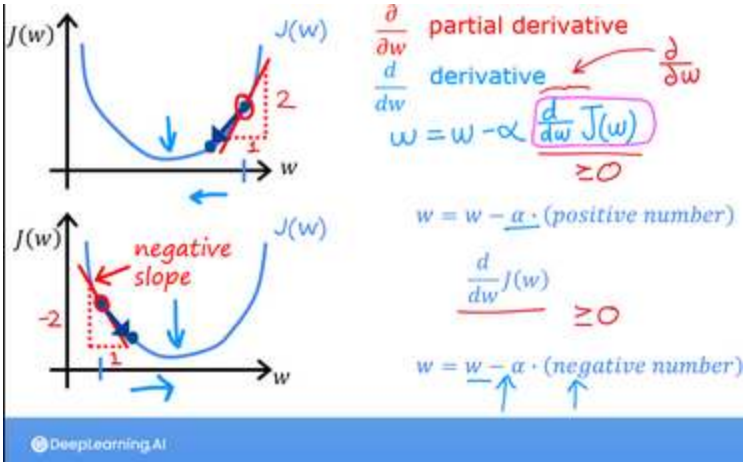
- The naming of python variables containing partial derivatives follows this pattern,$\frac{\partial J(w,b)}{\partial b}$ will be `dj_db`.
- w.r.t is With Respect To, as in partial derivative of $J(wb)$ With Respect To b.

compute_gradient

`compute_gradient` implements (4) and (5) above and returns $\frac{\partial J(w,b)}{\partial w}$,$\frac{\partial J(w,b)}{\partial b}$. The embedded comments describe the operations.

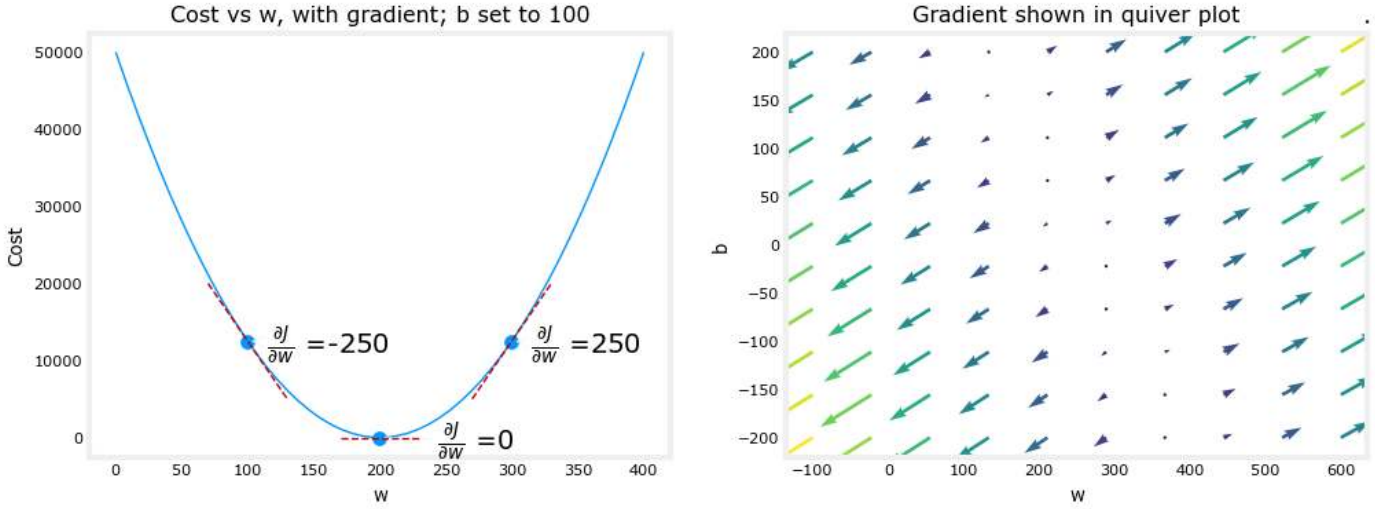
In [37]:
```
def compute_gradient(x, y, w, b):
    """
    Computes the gradient for linear regression
    Args:
      x (ndarray (m,)): Data, m examples
      y (ndarray (m,)): target values
      w,b (scalar)    : model parameters
    Returns
      dj_dw (scalar): The gradient of the cost w.r.t. the parameters w
      dj_db (scalar): The gradient of the cost w.r.t. the parameter b
     """

    # Number of training examples
    m = x.shape[0]
    dj_dw = 0
    dj_db = 0

    for i in range(m):
        f_wb = w * x[i] + b
        dj_dw_i = (f_wb - y[i]) * x[i]
        dj_db_i = f_wb - y[i]
        dj_db += dj_db_i
        dj_dw += dj_dw_i
    dj_dw = dj_dw / m
    dj_db = dj_db / m

    return dj_dw, dj_db
```

Loading [MathJax]/extensions/Safe.js



The lectures described how gradient descent utilizes the partial derivative of the cost with respect to a parameter at a point to update that parameter. Let's use our `compute_gradient` function to find and plot some partial derivatives of our cost function relative to one of the parameters, w_0.

In [38]:
```
plt_gradients(x_train,y_train, compute_cost, compute_gradient)
plt.show()
```



Above, the left plot shows $\frac{\partial J(w,b)}{\partial w}$ or the slope of the cost curve relative to w at three points. On the right side of the plot, the derivative is positive, while on the left it is negative. Due to the 'bowl shape', the derivatives will always lead gradient descent toward the bottom where the gradient is zero.

The left plot has fixed $b=100$. Gradient descent will utilize both $\frac{\partial J(w,b)}{\partial w}$ and $\frac{\partial J(w,b)}{\partial b}$ to update parameters. The 'quiver plot' on the right provides a means of viewing the gradient of both parameters. The arrow sizes reflect the magnitude of the gradient at that point. The direction and slope of the arrow reflects the ratio of $\frac{\partial J(w,b)}{\partial w}$ and $\frac{\partial J(w,b)}{\partial b}$ at that point. Note that the gradient points *away* from the minimum. Review equation (3) above. The scaled gradient is *subtracted* from the current value of w or b. This moves the parameter in a direction that will reduce cost.

Gradient Descent

Now that gradients can be computed, gradient descent, described in equation (3) above can be implemented below in `gradient_descent`. The details of the implementation are described in the comments. Below, you will utilize this function to find optimal values of w and b on the training data.

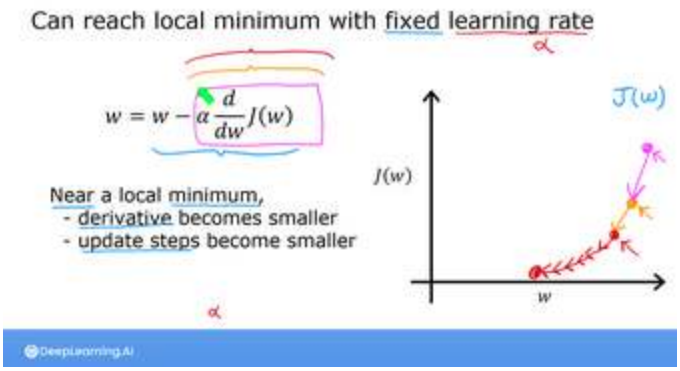
Loading [MathJax]/extensions/Safe.js

In [39]:
```
def gradient_descent(x, y, w_in, b_in, alpha, num_iters, cost_function, gradient_functio
    """
    Performs gradient descent to fit w,b. Updates w,b by taking
    num_iters gradient steps with learning rate alpha

    Args:
      x (ndarray (m,))  : Data, m examples
      y (ndarray (m,))  : target values
      w_in,b_in (scalar): initial values of model parameters
      alpha (float):     Learning rate
      num_iters (int):   number of iterations to run gradient descent
      cost_function:     function to call to produce cost
      gradient_function: function to call to produce gradient

    Returns:
      w (scalar): Updated value of parameter after running gradient descent
      b (scalar): Updated value of parameter after running gradient descent
      J_history (List): History of cost values
      p_history (list): History of parameters [w,b]
      """

    # An array to store cost J and w's at each iteration primarily for graphing later
    J_history = []
    p_history = []
    b = b_in
    w = w_in

    for i in range(num_iters):
        # Calculate the gradient and update the parameters using gradient_function
        dj_dw, dj_db = gradient_function(x, y, w , b)

        # Update Parameters using equation (3) above
        b = b - alpha * dj_db
        w = w - alpha * dj_dw

        # Save cost J at each iteration
        if i<100000:      # prevent resource exhaustion
            J_history.append( cost_function(x, y, w , b))
            p_history.append([w,b])
        # Print cost every at intervals 10 times or as many iterations if < 10
        if i% math.ceil(num_iters/10) == 0:
            print(f"Iteration {i:4}: Cost {J_history[-1]:0.2e} ",
                  f"dj_dw: {dj_dw: 0.3e}, dj_db: {dj_db: 0.3e}  ",
                  f"w: {w: 0.3e}, b:{b: 0.5e}")

    return w, b, J_history, p_history #return w and J,w history for graphing
```

In [44]:
```
# initialize parameters
w_init = 0
b_init = 0
# some gradient descent settings
iterations = 10000
tmp_alpha = 1.0e-1
# run gradient descent
w_final, b_final, J_hist, p_hist = gradient_descent(x_train ,y_train, w_init, b_init, tm
                                                    iterations, compute_cost, compute_gr
print(f"(w,b) found by gradient descent: ({w_final:8.4f},{b_final:8.4f})")
```

Loading [MathJax]/extensions/Safe.js

```
Iteration    0: Cost 3.67e+04  dj_dw: -6.500e+02, dj_db: -4.000e+02   w:  6.500e+01, b:
4.00000e+01
Iteration 1000: Cost 6.33e-06  dj_dw: -5.091e-04, dj_db:  8.238e-04   w:  2.000e+02, b:
1.00011e+02
Iteration 2000: Cost 2.77e-12  dj_dw: -3.366e-07, dj_db:  5.446e-07   w:  2.000e+02, b:
1.00000e+02
Iteration 3000: Cost 1.21e-18  dj_dw: -2.225e-10, dj_db:  3.600e-10   w:  2.000e+02, b:
1.00000e+02
Iteration 4000: Cost 5.49e-25  dj_dw: -1.705e-13, dj_db:  2.274e-13   w:  2.000e+02, b:
1.00000e+02
Iteration 5000: Cost 1.05e-25  dj_dw: -1.421e-13, dj_db:  5.684e-14   w:  2.000e+02, b:
1.00000e+02
Iteration 6000: Cost 1.05e-25  dj_dw: -1.421e-13, dj_db:  5.684e-14   w:  2.000e+02, b:
1.00000e+02
Iteration 7000: Cost 1.05e-25  dj_dw: -1.421e-13, dj_db:  5.684e-14   w:  2.000e+02, b:
1.00000e+02
Iteration 8000: Cost 1.05e-25  dj_dw: -1.421e-13, dj_db:  5.684e-14   w:  2.000e+02, b:
1.00000e+02
Iteration 9000: Cost 1.05e-25  dj_dw: -1.421e-13, dj_db:  5.684e-14   w:  2.000e+02, b:
1.00000e+02
(w,b) found by gradient descent: (200.0000,100.0000)
```



Take a moment and note some characteristics of the gradient descent process printed above.

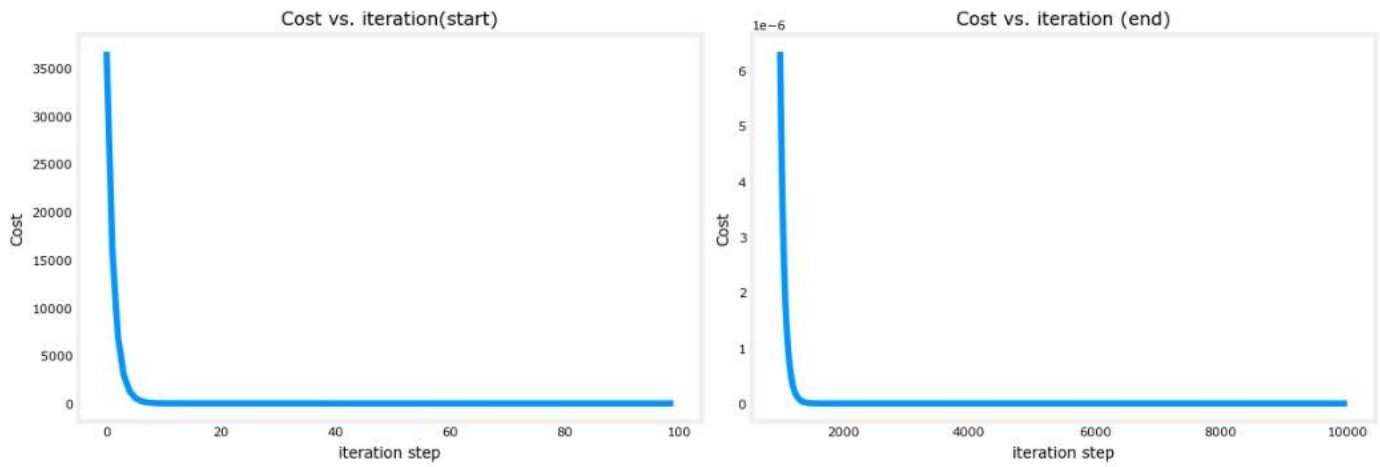
- The cost starts large and rapidly declines as described in the slide from the lecture.
- The partial derivatives, `dj_dw`, and `dj_db` also get smaller, rapidly at first and then more slowly. As shown in the diagram from the lecture, as the process nears the 'bottom of the bowl' progress is slower due to the smaller value of the derivative at that point.
- progress slows though the learning rate, alpha, remains fixed

Cost versus iterations of gradient descent

A plot of cost versus iterations is a useful measure of progress in gradient descent. Cost should always decrease in successful runs. The change in cost is so rapid initially, it is useful to plot the initial decent on a different scale than the final descent. In the plots below, note the scale of cost on the axes and the iteration step.

In [45]:
```
# plot cost versus iteration
fig, (ax1, ax2) = plt.subplots(1, 2, constrained_layout=True, figsize=(12,4))
ax1.plot(J_hist[:100])
ax2.plot(1000 + np.arange(len(J_hist[1000:])), J_hist[1000:])
ax1.set_title("Cost vs. iteration(start)");  ax2.set_title("Cost vs. iteration (end)")
ax1.set_ylabel('Cost')            ;  ax2.set_ylabel('Cost')
ax1.set_xlabel('iteration step')  ;  ax2.set_xlabel('iteration step')
plt.show()
```

Loading [MathJax]/extensions/Safe.js



Predictions

Now that you have discovered the optimal values for the parameters w and b, you can now use the model to predict housing values based on our learned parameters. As expected, the predicted values are nearly the same as the training values for the same housing. Further, the value not in the prediction is in line with the expected value.

In [47]:
```
print(f"1000 sqft house prediction {w_final*1.0 + b_final:0.1f} Thousand dollars")
print(f"1200 sqft house prediction {w_final*1.2 + b_final:0.1f} Thousand dollars")
print(f"2000 sqft house prediction {w_final*2.0 + b_final:0.1f} Thousand dollars")
```

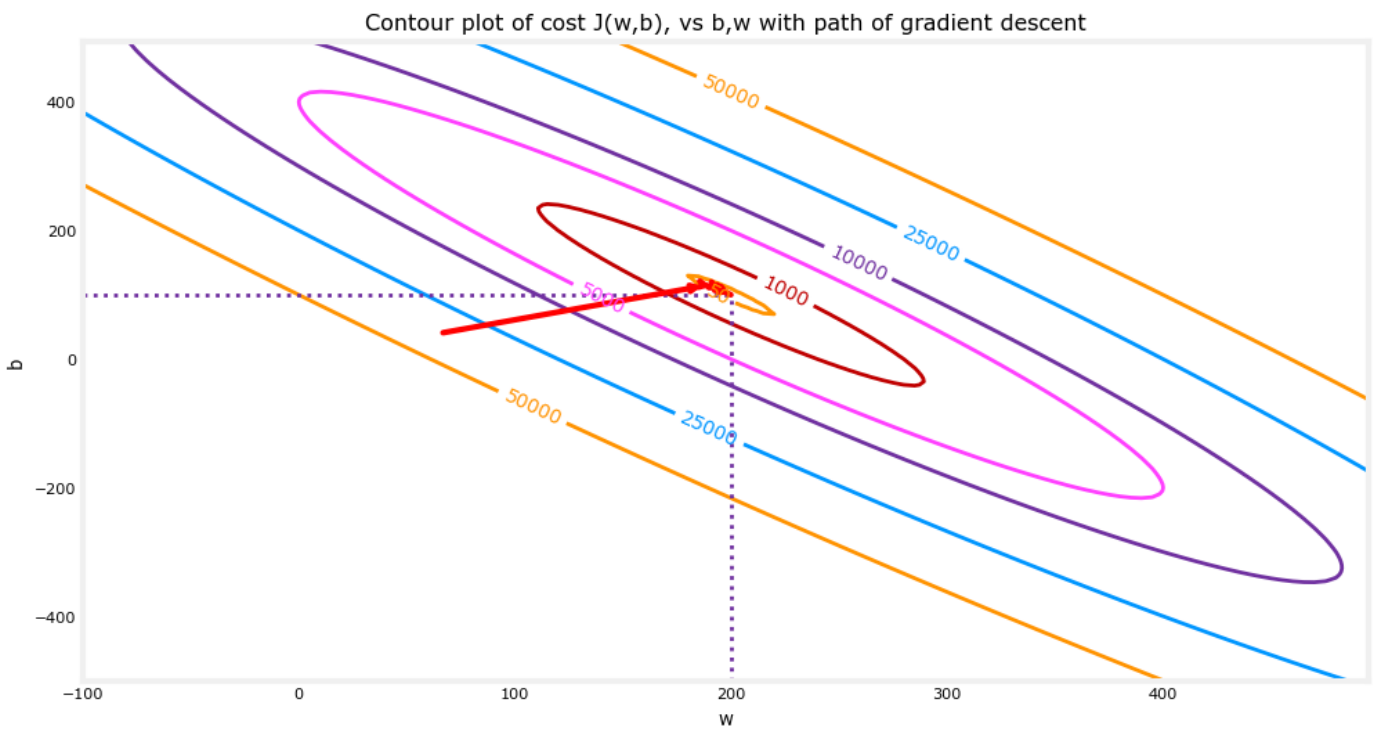
```
1000 sqft house prediction 300.0 Thousand dollars
1200 sqft house prediction 340.0 Thousand dollars
2000 sqft house prediction 500.0 Thousand dollars
```

Plotting

You can show the progress of gradient descent during its execution by plotting the cost over iterations on a contour plot of the cost(w,b).

In [48]:
```
fig, ax = plt.subplots(1,1, figsize=(12, 6))
plt_contour_wgrad(x_train, y_train, p_hist, ax)
```

Loading [MathJax]/extensions/Safe.js

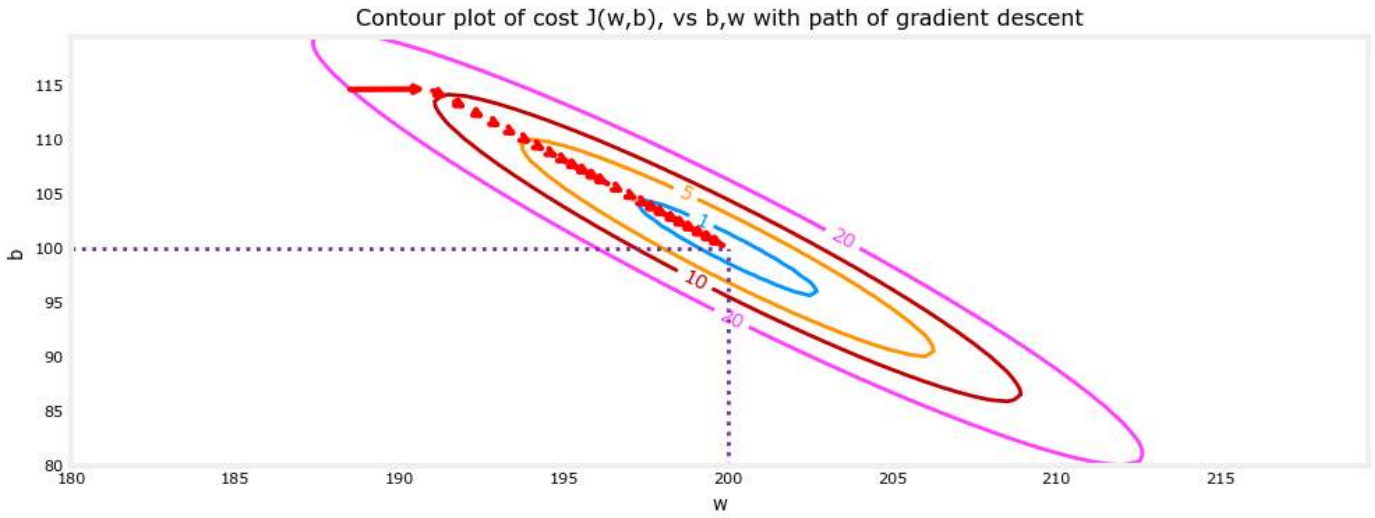


Above, the contour plot shows the $cost(w,b)$ over a range of w and b. Cost levels are represented by the rings. Overlayed, using red arrows, is the path of gradient descent. Here are some things to note:

- The path makes steady (monotonic) progress toward its goal.
- initial steps are much larger than the steps near the goal.

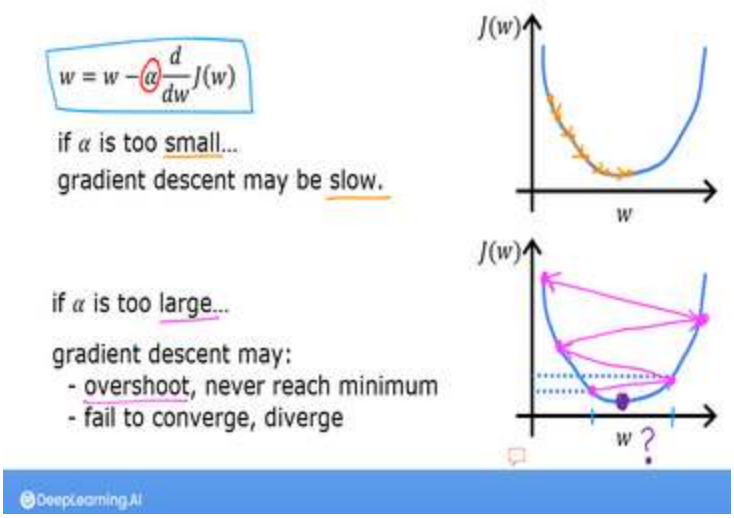
Zooming in, we can see that final steps of gradient descent. Note the distance between steps shrinks as the gradient approaches zero.

In [50]:
```
fig, ax = plt.subplots(1,1, figsize=(12, 4))
plt_contour_wgrad(x_train, y_train, p_hist, ax, w_range=[180, 220, 0.5], b_range=[80, 12
            contours=[1,5,10,20],resolution=0.5)
```



Increased Learning Rate

Loading [MathJax]/extensions/Safe.js



In the lecture, there was a discussion related to the proper value of the learning rate, α in equation(3). The larger α is, the faster gradient descent will converge to a solution. But, if it is too large, gradient descent will diverge. Above you have an example of a solution which converges nicely.

Let's try increasing the value of α and see what happens:

In [52]:
```
# initialize parameters
w_init = 0
b_init = 0
# set alpha to a large value
iterations = 10
tmp_alpha = 8.0e-1
# run gradient descent
w_final, b_final, J_hist, p_hist = gradient_descent(x_train ,y_train, w_init, b_init, tm
                                                    iterations, compute_cost, compute_gr
```

```
Iteration    0: Cost 2.58e+05  dj_dw: -6.500e+02, dj_db: -4.000e+02   w:  5.200e+02, b:
3.20000e+02
Iteration    1: Cost 7.82e+05  dj_dw:  1.130e+03, dj_db:  7.000e+02   w: -3.840e+02, b:-
2.40000e+02
Iteration    2: Cost 2.37e+06  dj_dw: -1.970e+03, dj_db: -1.216e+03   w:  1.192e+03, b:
7.32800e+02
Iteration    3: Cost 7.19e+06  dj_dw:  3.429e+03, dj_db:  2.121e+03   w: -1.551e+03, b:-
9.63840e+02
Iteration    4: Cost 2.18e+07  dj_dw: -5.974e+03, dj_db: -3.691e+03   w:  3.228e+03, b:
1.98886e+03
Iteration    5: Cost 6.62e+07  dj_dw:  1.040e+04, dj_db:  6.431e+03   w: -5.095e+03, b:-
3.15579e+03
Iteration    6: Cost 2.01e+08  dj_dw: -1.812e+04, dj_db: -1.120e+04   w:  9.402e+03, b:
5.80237e+03
Iteration    7: Cost 6.09e+08  dj_dw:  3.156e+04, dj_db:  1.950e+04   w: -1.584e+04, b:-
9.80139e+03
Iteration    8: Cost 1.85e+09  dj_dw: -5.496e+04, dj_db: -3.397e+04   w:  2.813e+04, b:
1.73730e+04
Iteration    9: Cost 5.60e+09  dj_dw:  9.572e+04, dj_db:  5.916e+04   w: -4.845e+04, b:-
2.99567e+04
```

Above, w and b are bouncing back and forth between positive and negative with the absolute value increasing with each iteration. Further, each iteration $\frac{\partial J(w,b)}{\partial w}$ changes sign and cost is increasing rather than decreasing. This is a clear sign that the *learning rate is too large* and the solution is diverging. Let's visualize this with a plot.

In [ ]:
```
plt_divergence(p_hist, J_hist,x_train, y_train)
plt.show()
```

Loading [MathJax]/extensions/Safe.js

Above, the left graph shows w's progression over the first few steps of gradient descent. w oscillates from positive to negative and cost grows rapidly. Gradient Descent is operating on both w and b simultaneously, so one needs the 3-D plot on the right for the complete picture.

In this lab you:

- delved into the details of gradient descent for a single variable.
- developed a routine to compute the gradient
- visualized what the gradient is
- completed a gradient descent routine
- utilized gradient descent to find parameters
- examined the impact of sizing the learning rate

In [31]:
```
df = pd.read_csv("clean_weather.csv", index_col = 0)
df = df.ffill()
df.head()
```

Out[31]:

	tmax	tmin	rain	tmax_tomorrow
1970-01-01	60.0	35.0	0.0	52.0
1970-01-02	52.0	39.0	0.0	52.0
1970-01-03	52.0	35.0	0.0	53.0
1970-01-04	53.0	36.0	0.0	52.0
1970-01-05	52.0	35.0	0.0	50.0

In [77]:
```
# Importing Libraries
import numpy as np
import matplotlib.pyplot as plt

def mean_squared_error(y_true, y_predicted):

        # Calculating the loss or cost
        cost = np.sum((y_true-y_predicted)**2) / len(y_true)
        return cost

# Gradient Descent Function
# Here iterations, learning_rate, stopping_threshold
```

Loading [MathJax]/extensions/Safe.js

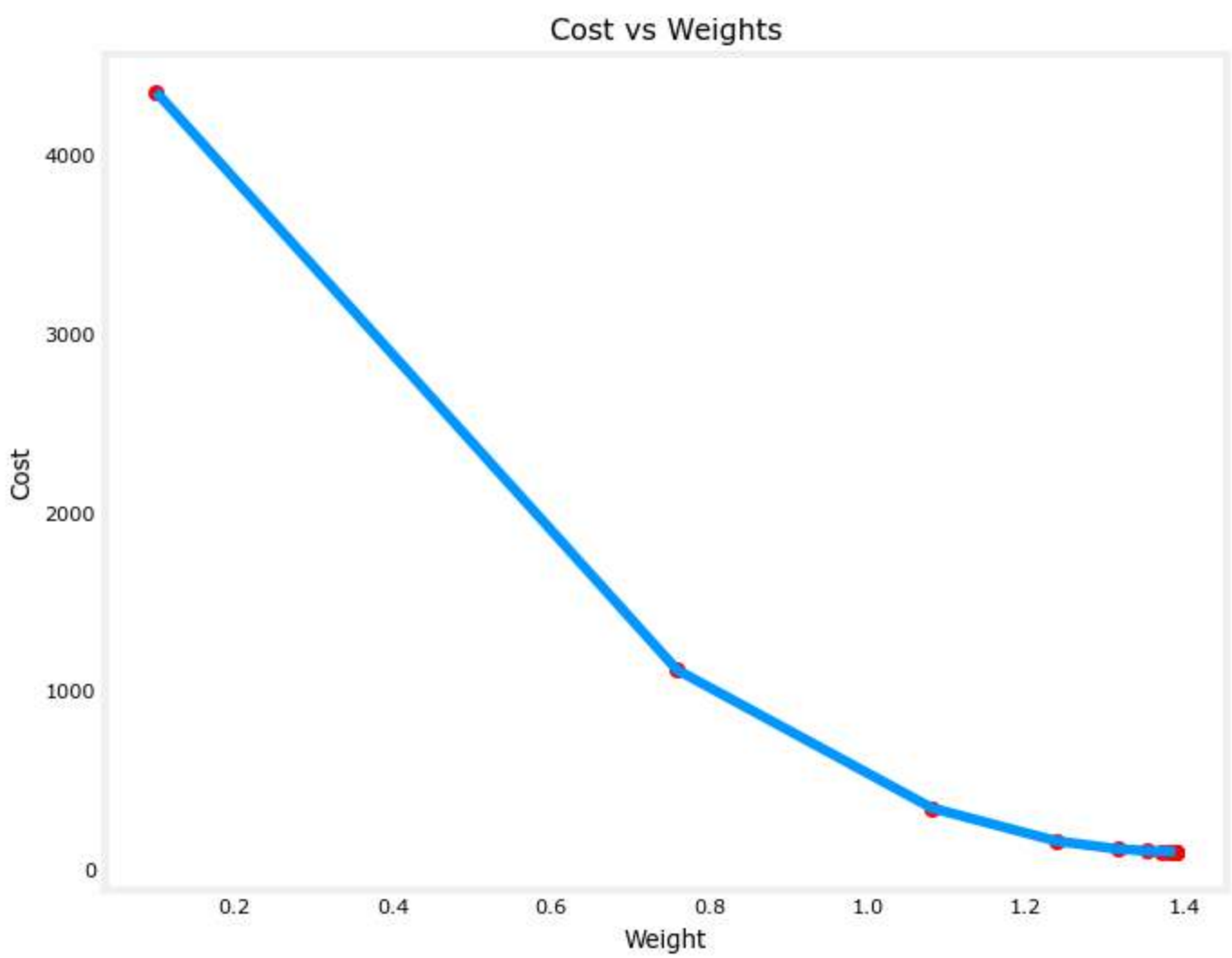
```
# are hyperparameters that can be tuned
def gradient_descent(x, y, iterations = 1000, learning_rate = 0.0001,
                                        stopping_threshold = 1e-6):

        # Initializing weight, bias, learning rate and iterations
        current_weight = 0.1
        current_bias = 0.01
        iterations = iterations
        learning_rate = learning_rate
        n = float(len(x))

        costs = []
        weights = []
        previous_cost = None

        # Estimation of optimal parameters
        for i in range(iterations):

                # Making predictions
                y_predicted = (current_weight * x) + current_bias

                # Calculating the current cost
                current_cost = mean_squared_error(y, y_predicted)

                # If the change in cost is less than or equal to
                # stopping_threshold we stop the gradient descent
                if previous_cost and abs(previous_cost-current_cost)<=stopping_threshold
                        break

                previous_cost = current_cost

                costs.append(current_cost)
                weights.append(current_weight)

                # Calculating the gradients
                weight_derivative = -(2/n) * sum(x * (y-y_predicted))
                bias_derivative = -(2/n) * sum(y-y_predicted)

                # Updating weights and bias
                current_weight = current_weight - (learning_rate * weight_derivative)
                current_bias = current_bias - (learning_rate * bias_derivative)

                # Printing the parameters for each 1000th iteration
                print(f"Iteration {i+1}: Cost {current_cost}, Weight \
                {current_weight}, Bias {current_bias}")


        # Visualizing the weights and cost at for all iterations
        plt.figure(figsize = (8,6))
        plt.plot(weights, costs)
        plt.scatter(weights, costs, marker='o', color='red')
        plt.title("Cost vs Weights")
        plt.ylabel("Cost")
        plt.xlabel("Weight")
        plt.show()

        return current_weight, current_bias


def main():

        # Data
        X = np.array([32.50234527, 53.42680403, 61.53035803, 47.47563963, 59.81320787,
                55.14218841, 52.21179669, 39.29956669, 48.10504169, 52.55001444,
```

Loading [MathJax]/extensions/Safe.js

```
                45.41973014, 54.35163488, 44.1640495 , 58.16847072, 56.72720806,
                48.95588857, 44.68719623, 60.29732685, 45.61864377, 38.81681754])
        Y = np.array([31.70700585, 68.77759598, 62.5623823 , 71.54663223, 87.23092513,
                78.21151827, 79.64197305, 59.17148932, 75.3312423 , 71.30087989,
                55.16567715, 82.47884676, 62.00892325, 75.39287043, 81.43619216,
                60.72360244, 82.89250373, 97.37989686, 48.84715332, 56.87721319])

        # Estimating weight and bias using gradient descent
        estimated_weight, estimated_bias = gradient_descent(X, Y, iterations=2000)
        print(f"Estimated Weight: {estimated_weight}\nEstimated Bias: {estimated_bias}")

        # Making predictions using estimated parameters
        Y_pred = estimated_weight*X + estimated_bias

        # Plotting the regression line
        plt.figure(figsize = (8,6))
        plt.scatter(X, Y, marker='o', color='red')
        plt.plot([min(X), max(X)], [min(Y_pred), max(Y_pred)], color='blue',markerfaceco
                        markersize=10,linestyle='dashed')
        plt.xlabel("X")
        plt.ylabel("Y")
        plt.show()



if __name__=="__main__":
        main()
```

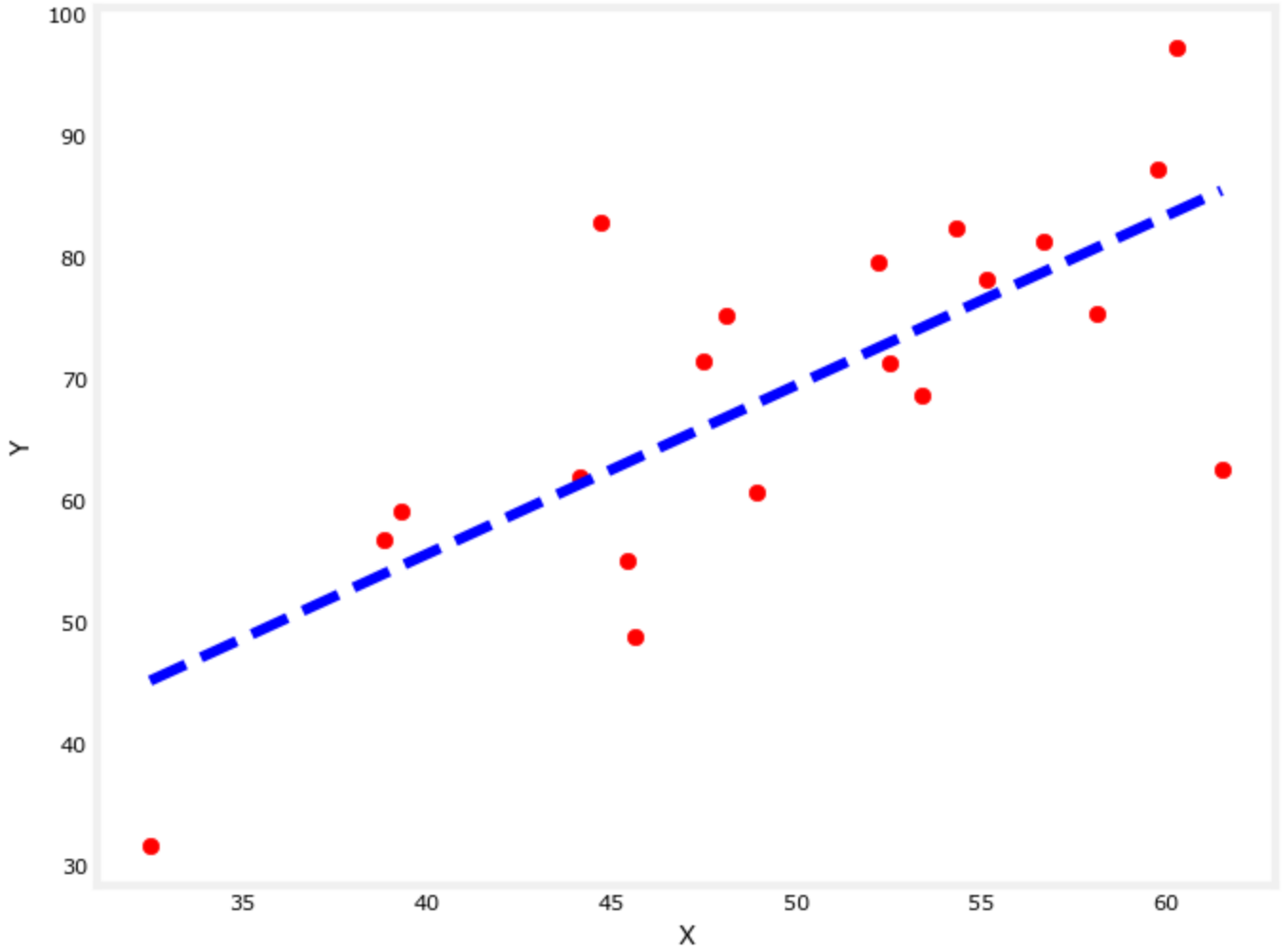
```
Iteration 1: Cost 4352.088931274409, Weight           0.7593291142562117, Bias 0.02288
558130709
Iteration 2: Cost 1114.8561474350017, Weight          1.081602958862324, Bias 0.029180
14748569513
Iteration 3: Cost 341.42912086804455, Weight          1.2391274084945083, Bias 0.03225
308846928192
Iteration 4: Cost 156.64495290904443, Weight          1.3161239281746984, Bias 0.03375
132986012604
Iteration 5: Cost 112.49704004742098, Weight          1.3537591652024805, Bias 0.03447
9873154934775
Iteration 6: Cost 101.9493925395456, Weight           1.3721549833978113, Bias 0.03483
2195392868505
Iteration 7: Cost 99.4293893333546, Weight            1.3811467575154601, Bias 0.03500
062439068245
Iteration 8: Cost 98.82731958262897, Weight           1.3855419247507244, Bias 0.03507
916814736111
Iteration 9: Cost 98.68347500997261, Weight           1.3876903144657764, Bias 0.03511
3776874486774
Iteration 10: Cost 98.64910780902792, Weight          1.3887405007983562, Bias 0.03512
6910596389935
Iteration 11: Cost 98.64089651459352, Weight          1.389253895811451, Bias 0.035129
54755833985
Iteration 12: Cost 98.63893428729509, Weight          1.38950491235671, Bias 0.0351270
53821718185
Iteration 13: Cost 98.63846506273883, Weight          1.3896276808137857, Bias 0.03512
2052266051224
Iteration 14: Cost 98.63835254057648, Weight          1.38968776283053, Bias 0.0351158
2492978764
Iteration 15: Cost 98.63832524036214, Weight          1.3897172043139192, Bias 0.03510
899846107016
Iteration 16: Cost 98.63831830104695, Weight          1.389731668997059, Bias 0.035101
879159522745
Iteration 17: Cost 98.63831622628217, Weight          1.389738813163012, Bias 0.035094
61674147458
```

Loading [MathJax]/extensions/Safe.js




```
Estimated Weight: 1.389738813163012
Estimated Bias: 0.03509461674147458
```

Loading [MathJax]/extensions/Safe.js



In [38]:

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Calculate loss/cost function
def loss_function(X, Y, w, b):
    m = X.shape[0]

    cost = 0
    for i in range(m):
        f_wb = w * X[i] + b
        cost += (f_wb - Y[i]) ** 2
    total_cost = 1 / (2 * m) * cost

    return total_cost

# Calculate partial derivative gradient
def derivative_gradient(X, Y, w, b):
    m = X.shape[0]
    dw = 0
    db = 0
    for i in range(m):
        f_wb = w * X[i] + b
        dw += (f_wb - Y[i]) * X[i]
        db += (f_wb - Y[i])
    dw /= m
    db /= m
    return dw, db

# Perform gradient descent
def gradient_descent(X, Y, learning_rate=0.0001, num_iterations=1000):
```

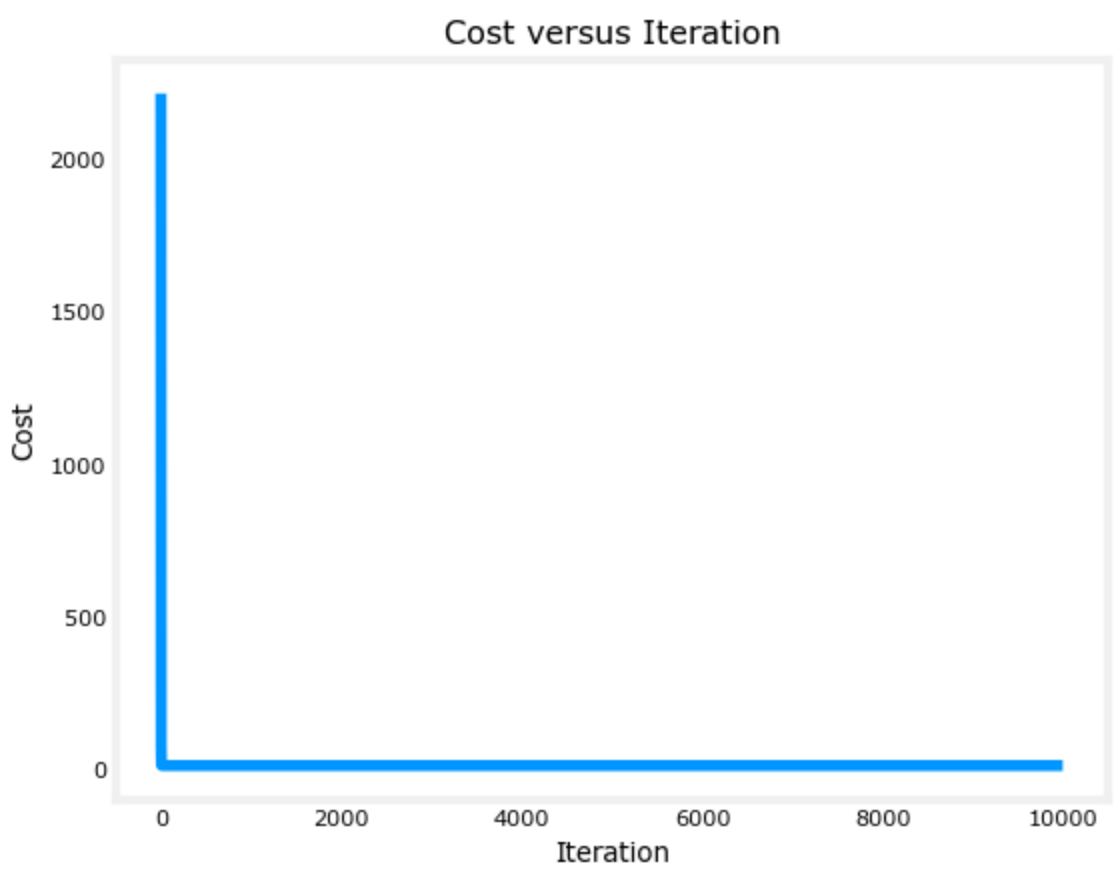
Loading [MathJax]/extensions/Safe.js

```
    w = 0  # Initial guess for weight
    b = 0  # Initial guess for bias
    m = X.shape[0]

    costs = []  # List to store the cost at each iteration
    for i in range(num_iterations):
        cost = loss_function(X, Y, w, b)
        dw, db = derivative_gradient(X, Y, w, b)

        # Update weights and bias using gradient descent
        w -= learning_rate * dw
        b -= learning_rate * db

        costs.append(cost)

    return w, b, costs

def plt_house_x(X, Y, w, b, ax=None):
    if not ax:
        fig, ax = plt.subplots(1, 1)

    # Plot scatter plot of actual values
    ax.scatter(X, Y, marker='x', c='r', label="Actual Value")

    # Generate x values for regression line
    x_line = np.linspace(np.min(X), np.max(X), 100)

    # Calculate predicted values for the regression line
    y_line = w * x_line + b

    # Plot regression line
    ax.plot(x_line, y_line, c='b', label="Regression Line")

    ax.set_xlabel("House Size (X)")
    ax.set_ylabel("House Price (Y)")
    ax.set_title("Housing Prices")
    ax.legend()

    return ax

def plt_contour_wgrad(X, Y, hist, w_range=(-2, 2), b_range=(-2, 2), contours=30, resolut
    if not ax:
        fig, ax = plt.subplots(1, 1)

    w = np.arange(w_range[0], w_range[1], resolution)
    b = np.arange(b_range[0], b_range[1], resolution)
    w, b = np.meshgrid(w, b)
    z = loss_function(X, Y, w.flatten(), b.flatten())
    z = z.reshape(w.shape)
    ax.contour(w, b, z, contours, linewidths=2, colors=['deepskyblue', 'orange', 'darkre

    if hist:
        w_hist = hist[0]
        b_hist = hist[1]
        ax.plot(w_hist, b_hist, 'wo', markersize=step)

    if w_final is not None and b_final is not None:
        ax.plot(w_final, b_final, 'wo', markersize=5)

    ax.set_xlabel('w')
    ax.set_ylabel('b')
    ax.set_title('Contour Plot with Gradient Descent')
    return ax
```

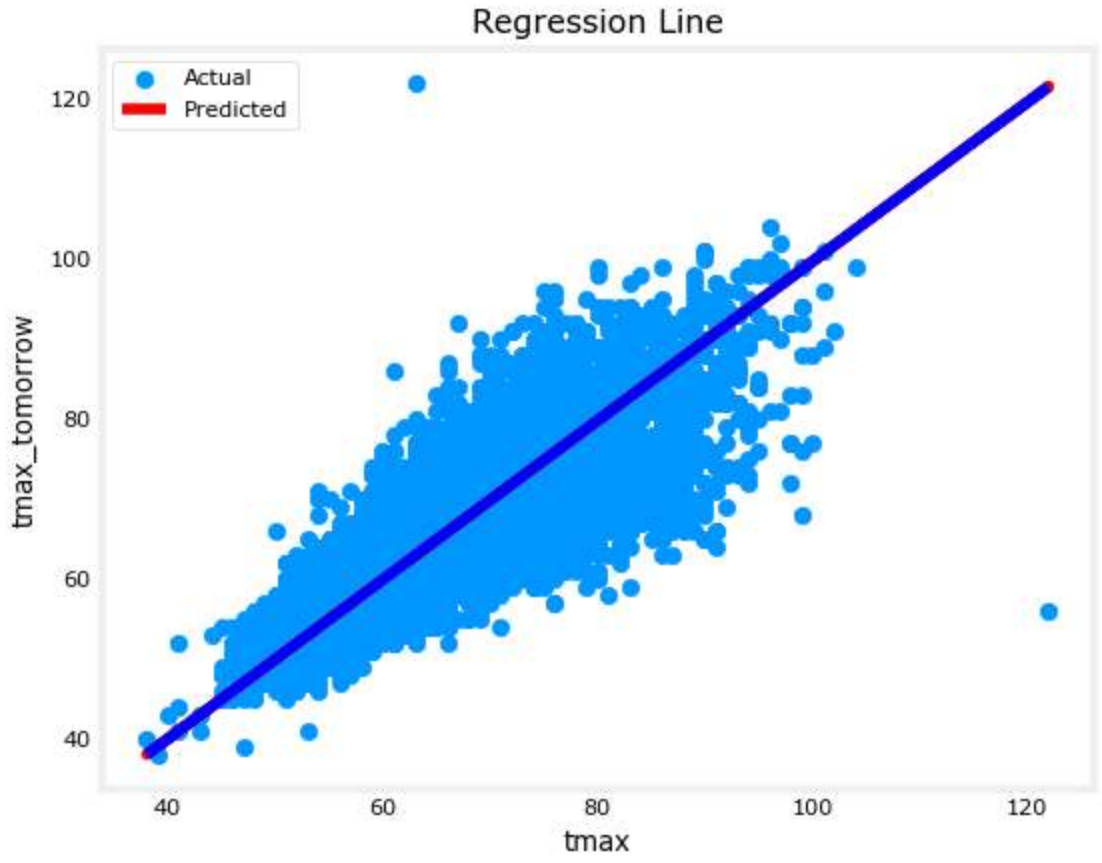
Loading [MathJax]/extensions/Safe.js

```
def main():
    # Read file
    df = pd.read_csv('clean_weather.csv', index_col=0)
    df = df.ffill()
    df.head()

    X = np.array(df['tmax'])
    Y = np.array(df['tmax_tomorrow'])


    # Run gradient descent with adjusted learning rate and iterations
    w, b, costs = gradient_descent(X, Y, learning_rate=0.0001, num_iterations=10000)


    # Plot cost versus iteration
    plt.plot(range(len(costs)), costs)
    plt.xlabel('Iteration')
    plt.ylabel('Cost')
    plt.title('Cost versus Iteration')
    plt.show()

    # Predict
    predicted_Y = w * X + b

    # Choose a specific tmax value for prediction
    tmax_value = 30  # Change this to the desired tmax value

    # Predict the corresponding tmax_tomorrow value
    predicted_tmax_tomorrow = w * tmax_value + b
    print("Predicted tmax_tomorrow for tmax =", tmax_value, "is", predicted_tmax_tomorro

    # Plot progress of gradient descent and regression line
    fig, ax = plt.subplots(1, 1)
    plt.scatter(X, Y, label='Actual')
    plt.plot(X, predicted_Y, color='red', label='Predicted')
    plt.xlabel('tmax')
    plt.ylabel('tmax_tomorrow')
    plt.title('Regression Line')
    plt.legend()

    # Plot the regression line
    x_line = np.linspace(np.min(X), np.max(X), 100)
    y_line = w * x_line + b
    plt.plot(x_line, y_line, color='blue', label='Regression Line')

    plt.show()

if __name__ == '__main__':
    main()
```

Loading [MathJax]/extensions/Safe.js



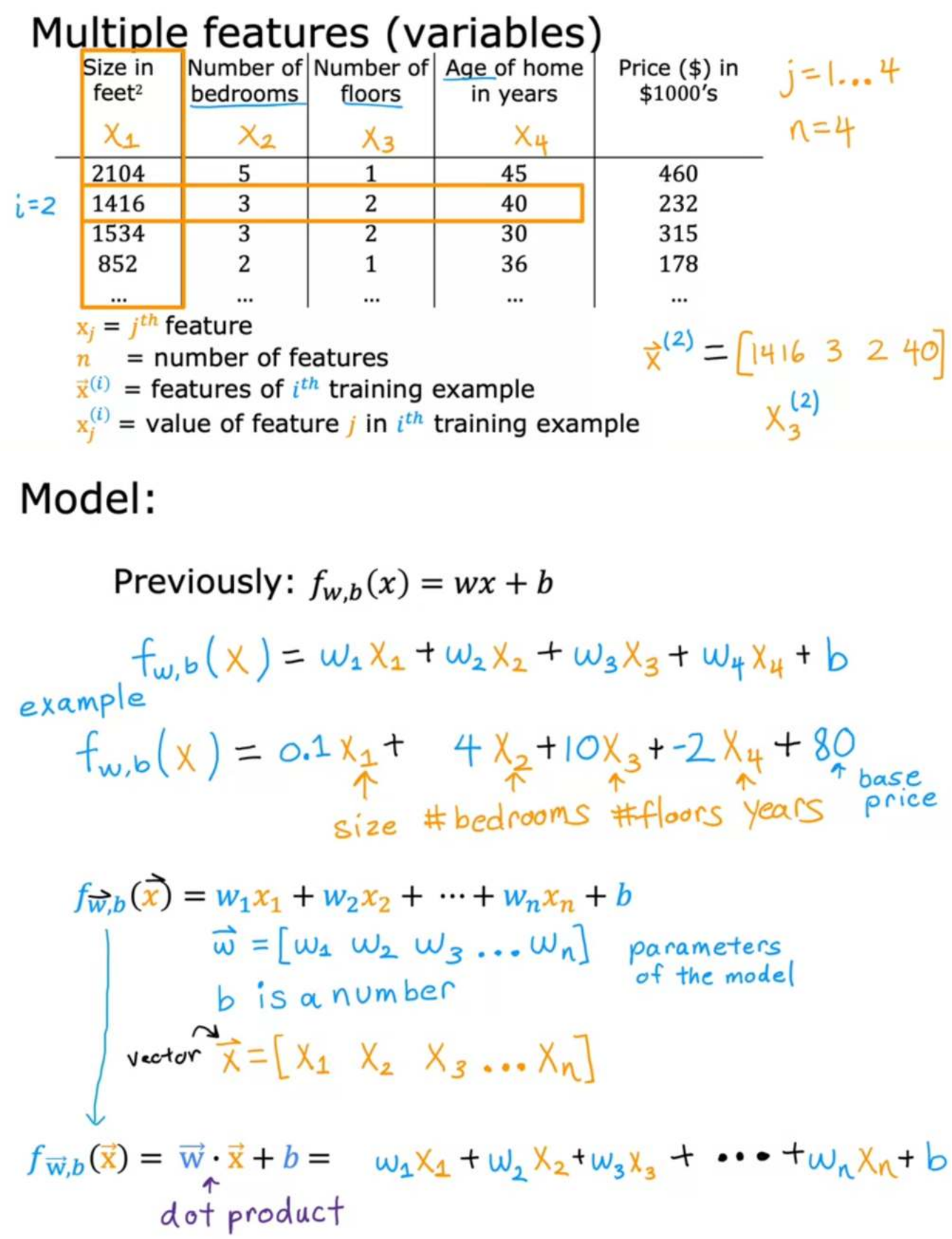

```
Predicted tmax_tomorrow for tmax = 30 is 30.025942297613845
```



Multiple linear regression

Multiple features

Loading [MathJax]/extensions/Safe.js

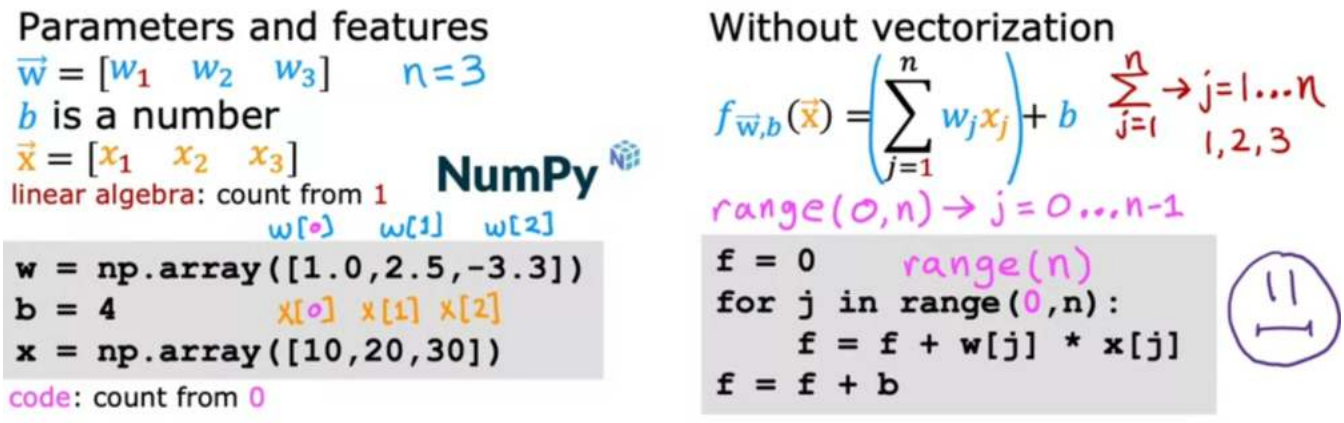


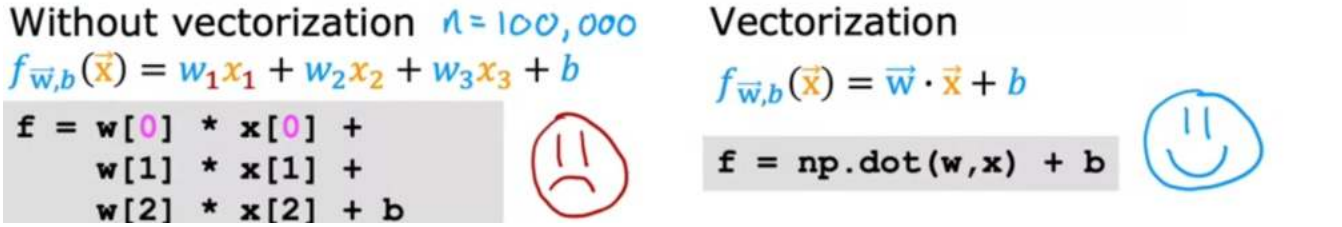
In order to implement multiple linear regression there is a neat trick called **vectorization** make the implementation much simpler

Vectorization

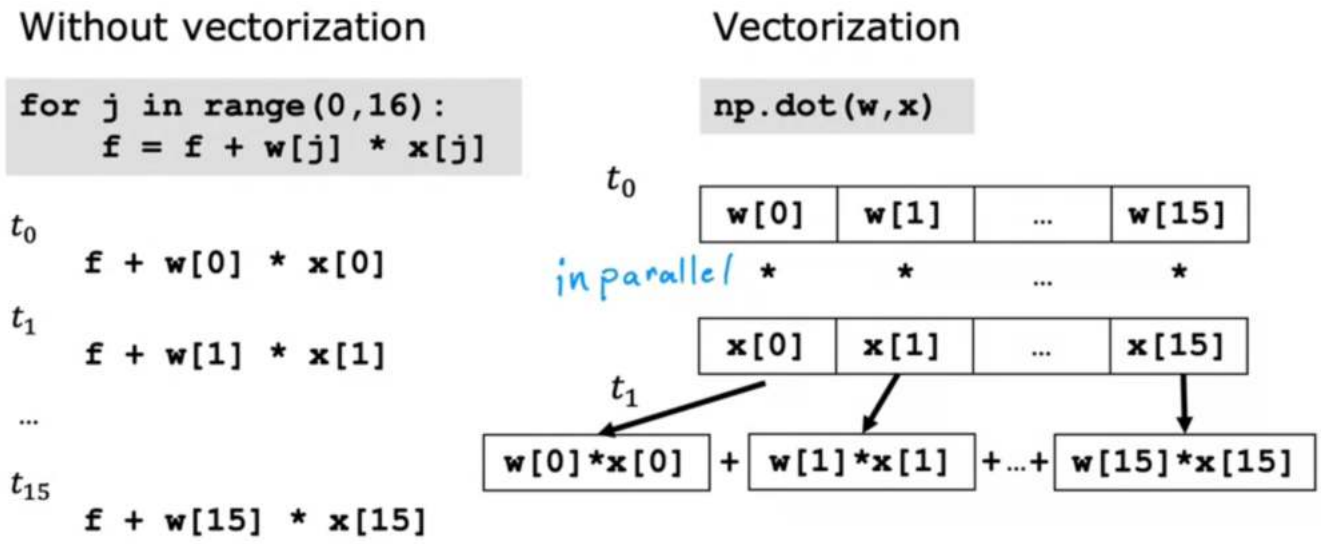
Loading [MathJax]/extensions/Safe.js

Using vectorization will both make your code shorter & also make it run much more efficiently





How Vectorization is work?



Gradient Descent for Multiple Regression

Loading [MathJax]/extensions/Safe.js

	Previous notation	Vector notation
Parameters	$w_1, \cdots, w_n$ b	$\vec{w} = [w_1 \quad \cdots \quad w_n]$ ← vector of length n b still a number
Model	$f_{\vec{w},b}(\vec{x}) = w_1x_1 + \cdots + w_nx_n + b$	$f_{\vec{w},b}(\vec{x}) = \vec{w} \cdot \vec{x} + b$ ↑ dot product
Cost function	$J(w_1, \cdots, w_n, b)$	$J(\vec{w}, b)$

Gradient descent

repeat {	repeat {
$w_j = w_j - \alpha \frac{\partial}{\partial w_j} J(w_1, \cdots, w_n, b)$	$w_j = w_j - \alpha \frac{\partial}{\partial w_j} J(\vec{w}, b)$
$b = b - \alpha \frac{\partial}{\partial b} J(w_1, \cdots, w_n, b)$	$b = b - \alpha \frac{\partial}{\partial b} J(\vec{w}, b)$
}	}

let's look at the derivative term

Gradient descent

One feature	n features ($n \geq 2$)
repeat {	repeat {
$w = w - \alpha \frac{1}{m} \sum_{i=1}^{m} (f_{w,b}(x^{(i)}) - y^{(i)}) x^{(i)}$	$w_1 = w_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}) x_1^{(i)}$ (j = 1)
→ $\frac{\partial}{\partial w} J(w, b)$	→ $\frac{\partial}{\partial w_1} J(\vec{w}, b)$
	⋮
	$w_n = w_n - \alpha \frac{1}{m} \sum_{i=1}^{m} (f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}) x_n^{(i)}$ (j = n)
$b = b - \alpha \frac{1}{m} \sum_{i=1}^{m} (f_{w,b}(x^{(i)}) - y^{(i)})$	$b = b - \alpha \frac{1}{m} \sum_{i=1}^{m} (f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)})$
simultaneously update w, b	simultaneously update w_j (for $j = 1, \cdots, n$) and b
}	}

An alternative way for finding w & b for linear regression:

Normal Equation

→ Normal equation
- Only for linear regression
- Solve for w, b without iterations

Disadvantages
- Doesn't generalize to other learning algorithms.
- Slow when number of features is large (> 10,000)

What you need to know
- Normal equation method may be used in machine learning libraries that implement linear regression.
- Gradient descent is the recommended method for finding parameters w,b

Loading [MathJax]/extensions/Safe.js

The Need for Speed: vector vs for loop

We utilized the NumPy library because it improves speed memory efficiency. Let's demonstrate:

In [39]:
```
import numpy as np
import time
```

In [40]:
```
def my_dot(a, b):
    """
   Compute the dot product of two vectors

    Args:
      a (ndarray (n,)):  input vector
      b (ndarray (n,)):  input vector with same dimension as a

    Returns:
      x (scalar):
    """
    x=0
    for i in range(a.shape[0]):
        x = x + a[i] * b[i]
    return x
```

In [41]:
```
np.random.seed(1)
a = np.random.rand(10000000)  # very large arrays
b = np.random.rand(10000000)

tic = time.time()  # capture start time
c = np.dot(a, b)
toc = time.time()  # capture end time

print(f"np.dot(a, b) =  {c:.4f}")
print(f"Vectorized version duration: {1000*(toc-tic):.4f} ms ")

tic = time.time()  # capture start time
c = my_dot(a,b)
toc = time.time()  # capture end time

print(f"my_dot(a, b) =  {c:.4f}")
print(f"loop version duration: {1000*(toc-tic):.4f} ms ")

del(a);del(b)  #remove these big arrays from memory
```

```
np.dot(a, b) =  2501072.5817
Vectorized version duration: 22.6665 ms
my_dot(a, b) =  2501072.5817
loop version duration: 3097.2767 ms
```

Lab: Multiple linear regression

Goals

- Extend our regression model routines to support multiple features
  - Extend data structures to support multiple features
  - Rewrite prediction, cost and gradient routines to support multiple features
  - Utilize NumPy `np.dot` to vectorize their implementations for speed and simplicity

Loading [MathJax]/extensions/Safe.js

In [74]:
```
import copy, math
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('deeplearning.mplstyle')
np.set_printoptions(precision=2)  # reduced display precision on numpy arrays
```

Problem Statement

You will use the motivating example of housing price prediction. The training dataset contains three examples with four features (size, bedrooms, floors and, age) shown in the table below. Note that, unlike the earlier labs, size is in sqft rather than 1000 sqft. This causes an issue, which you will solve in the next lab!

Size (sqft)	Number of Bedrooms	Number of floors	Age of Home	Price (1000s dollars)
2104	5	1	45	460
1416	3	2	40	232
852	2	1	35	178

You will build a linear regression model using these values so you can then predict the price for other houses. For example, a house with 1200 sqft, 3 bedrooms, 1 floor, 40 years old.

Please run the following code cell to create your `X_train` and `y_train` variables.

In [75]:
```
X_train = np.array([[2104, 5, 1, 45], [1416, 3, 2, 40], [852, 2, 1, 35]])
y_train = np.array([460, 232, 178])
```

Matrix X containing our examples

Similar to the table above, examples are stored in a NumPy matrix `X_train`. Each row of the matrix represents one example. When you have m training examples (m is three in our example), and there are n features (four in our example), $\mathbf{X}$ is a matrix with dimensions (m, n) (m rows, n columns).

$$\mathbf{X} = \begin{pmatrix} x^{(0)}_0 & x^{(0)}_1 & \cdots & x^{(0)}_{n-1} \\ x^{(1)}_0 & x^{(1)}_1 & \cdots & x^{(1)}_{n-1} \\ \cdots \\ x^{(m-1)}_0 & x^{(m-1)}_1 & \cdots & x^{(m-1)}_{n-1} \end{pmatrix} $$ notation:

- $\mathbf{x}^{(i)}$ is vector containing example i. $\mathbf{x}^{(i)}$ $ = (x^{(i)}_0, x^{(i)}_1, \cdots,x^{(i)}_{n-1})$
- $x^{(i)}_j$ is element j in example i. The superscript in parenthesis indicates the example number while the subscript represents an element.

Display the input data.

In [76]:
```
#data is stored in numpy array/matrix
print(f"X Shape: {X_train.shape}, X Type:{type(X_train)})")
print(X_train)
print(f"y Shape: {y_train.shape}, y Type:{type(y_train)})")
print(y_train)
```

Loading [MathJax]/extensions/Safe.js

```
X Shape: (3, 4), X Type:<class 'numpy.ndarray'>)
[[2104    5    1   45]
 [1416    3    2   40]
 [ 852    2    1   35]]
y Shape: (3,), y Type:<class 'numpy.ndarray'>)
[460 232 178]
```

Parameter vector w, b

- $\mathbf{w}$ is a vector with n elements.
  - Each element contains the parameter associated with one feature.
  - in our dataset, n is 4.
  - notionally, we draw this as a column vector

$$\mathbf{w} = \begin{pmatrix} w_0 \\ w_1 \\ \cdots\\ w_{n-1} \end{pmatrix} $$

- b is a scalar parameter.

  For demonstration, $\mathbf{w}$ and b will be loaded with some initial selected values that are near the optimal. $\mathbf{w}$ is a 1-D NumPy vector.

In [77]:
```
b_init = 785.1811367994083
w_init = np.array([ 0.39133535, 18.75376741, -53.36032453, -26.42131618])
print(f"w_init shape: {w_init.shape}, b_init type: {type(b_init)}")
```

```
w_init shape: (4,), b_init type: <class 'float'>
```

In theabove example code, the initial values of w and b are chosen arbitrarily. It is common practice to initialize these values randomly or with small values close to zero when training a linear regression model.

Model Prediction With Multiple Variables

The model's prediction with multiple variables is given by the linear model:

$$ f_{\mathbf{w},b}(\mathbf{x}) = w_0x_0 + w_1x_1 +... + w_{n-1}x_{n-1} + b \tag{1}$$
or in vector notation: $$ f_{\mathbf{w},b}(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b \tag{2} $$ where $\cdot$ is a vector `dot product`

Single Prediction, vector

Recall from the Python/Numpy lab that NumPy `np.dot()` [link] can be used to perform a vector dot product.

In [78]:
```
def predict(x, w, b):
    """
    single predict using linear regression
    Args:
      x (ndarray): Shape (n,) example with multiple features
      w (ndarray): Shape (n,) model parameters
      b (scalar):             model parameter

    Returns:
      p (scalar):  prediction
    """
```

Loading [MathJax]/extensions/Safe.js

```
    p = np.dot(x, w) + b
    return p
```

In [79]:
```
# get a row from our training data
x_vec = X_train[0,:]
print(f"x_vec shape {x_vec.shape}, x_vec value: {x_vec}")

# make a prediction
f_wb = predict(x_vec,w_init, b_init)
print(f"f_wb shape {f_wb.shape}, prediction: {f_wb}")
```

```
x_vec shape (4,), x_vec value: [2104    5    1   45]
f_wb shape (), prediction: 459.9999976194083
```

Compute Cost With Multiple Variables

The equation for the cost function with multiple variables $J(\mathbf{w},b)$ is: $$J(\mathbf{w},b) = \frac{1}{2m} \sum\limits_{i = 0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})^2 \tag{3}$$ where: $$ f_{\mathbf{w},b}(\mathbf{x}^{(i)}) = \mathbf{w} \cdot \mathbf{x}^{(i)} + b \tag{4} $$

In contrast to previous labs, $\mathbf{w}$ and $\mathbf{x}^{(i)}$ are vectors rather than scalars supporting multiple features.

In [80]:
```
def compute_cost(X, y, w, b):
    m = X.shape[0]
    cost = 0.0
    for _ in range (m):
        f_wb_i = np.dot(X[_], w) + b
        cost = cost + (f_wb_i - y[_])**2
    cost = cost / (2*m)
    return cost
```

In [81]:
```
# Compute and display cost using our pre-chosen optimal parameters.
cost = compute_cost(X_train, y_train, w_init, b_init)
print(f'Cost at optimal w : {cost}')
```

```
Cost at optimal w : 1.5578904045996674e-12
```

Gradient Descent With Multiple Variables

Gradient descent for multiple variables:

$$\begin{align*} \text{repeat}&\text{ until convergence:} \; \lbrace \newline\; & w_j = w_j - \alpha \frac{\partial J(\mathbf{w},b)}{\partial w_j} \tag{5} \; & \text{for j = 0..n-1}\newline &b\ \ = b - \alpha \frac{\partial J(\mathbf{w},b)}{\partial b} \newline \rbrace \end{align*}$$
where, n is the number of features, parameters w_j, b, are updated simultaneously and where

$$ \begin{align} \frac{\partial J(\mathbf{w},b)}{\partial w_j} &= \frac{1}{m} \sum\limits_{i = 0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})x_{j}^{(i)} \tag{6} \\ \frac{\partial J(\mathbf{w},b)}{\partial b} &= \frac{1}{m} \sum\limits_{i = 0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)}) \tag{7} \end{align} $$

- m is the number of training examples in the data set
- $f_{\mathbf{w},b}(\mathbf{x}^{(i)})$ is the model's prediction, while $y^{(i)}$ is the target value

Loading [MathJax]/extensions/Safe.js

5.1 Compute Gradient with Multiple Variables

An implementation for calculating the equations (6) and (7) is below. There are many ways to implement this. In this version, there is an

- outer loop over all m examples.
  - $\frac{\partial J(\mathbf{w},b)}{\partial b}$ for the example can be computed directly and accumulated
  - in a second loop over all n features:
    - $\frac{\partial J(\mathbf{w},b)}{\partial w_j}$ is computed for each w_j.

In [82]:
```
def compute_gradient(X, y, w, b):
    m,n = X.shape  #(number of rows(examples), number of columns(features))
    dj_dw = np.zeros((n,))
    dj_db = 0.

    for i in range(m):
        err = (np.dot(X[i], w) + b) - y[i]
        for j in range(n):
            dj_dw[j] = dj_dw[j] + err * X[i, j]
        dj_db = dj_db + err
    dj_dw = dj_dw / m
    dj_db = dj_db / m

    return dj_db, dj_dw
```

In the code snippet you provided, dj_db is initialized as a scalar value (0) rather than an array of zeros. The reason for this is that dj_db represents the partial derivative of the cost function with respect to the bias term b, which is a scalar value.

In linear regression, the partial derivative of the cost function with respect to b is computed by summing the individual contributions for each training example. Since the derivative with respect to b is a scalar value, there is no need to create an array of zeros for dj_db.

On the other hand, dj_dw represents the partial derivatives of the cost function with respect to each weight w. Since w is a vector of weights associated with each feature, dj_dw is initialized as a 1D array of zeros with a length equal to the number of features (n).

In this code, m represents the number of training examples, and n represents the number of features.

The outer loop for i in range(m) iterates over each training example. For each example, it calculates the prediction error by subtracting the predicted value (np.dot(X[i], w) + b) from the actual target value y[i]. This error represents the difference between the predicted and actual output for that training example.

The inner loop for j in range(n) iterates over each feature. For each feature, it calculates the partial derivative of the cost function with respect to w[j] by accumulating the product of the prediction error err and the corresponding feature value X[i, j] in the dj_dw array.

After the loops, the accumulated derivatives in dj_dw and dj_db are divided by the number of training examples m to calculate the average derivatives. This step normalizes the derivatives and ensures they are consistent regardless of the number of training examples.

By using the nested loops, you ensure that you update the derivatives for each training example and each

Loading [MathJax]/extensions/Safe.js

contributing to the overall gradients used for weight and bias updates during optimization algorithms like gradient descent.

In [83]:
```
#Compute and display gradient
tmp_dj_db, tmp_dj_dw = compute_gradient(X_train, y_train, w_init, b_init)
print(f'dj_db at initial w,b: {tmp_dj_db}')
print(f'dj_dw at initial w,b: \n {tmp_dj_dw}')
```

```
dj_db at initial w,b: -1.6739251122999121e-06
dj_dw at initial w,b:
 [-2.73e-03 -6.27e-06 -2.22e-06 -6.92e-05]
```

5.2 Gradient Descent With Multiple Variables

The routine below implements equation (5) above.

In [89]:
```
def gradient_descent(X, y, w_in, b_in, compute_cost, compute_gradient, alpha, num_iters)
    # An array to store cost J and w's at each iteration primarily for graphing later
    J_history = []
    w = copy.deepcopy(w_in)  #avoid modifying global w within function
    b = b_in

    for i in range(num_iters):

        # Calculate the gradient and update the parameters
        dj_db,dj_dw = compute_gradient(X, y, w, b)   ##None

        # Update Parameters using w, b, alpha and gradient
        w = w - alpha * dj_dw               ##None
        b = b - alpha * dj_db               ##None

        # Save cost J at each iteration
        if i<100000:      # prevent resource exhaustion
            J_history.append( compute_cost(X, y, w, b))

        # Print cost every at intervals 10 times or as many iterations if < 10
        if i% math.ceil(num_iters / 10) == 0:
            print(f"Iteration {i:4d}: Cost {J_history[-1]:8.2f}   ")

    return w, b, J_history #return final w,b and J history for graphing
```

By assigning the deep copy to w, you ensure that w is a separate copy of the w_in array and any modifications made to w do not affect w_in.

Taking only the last element (J_history[-1]) is commonly done when you want to monitor the convergence of the algorithm or analyze the final performance of the model. It provides information about how well the optimization process has minimized the cost function.

In [90]:
```
# initialize parameters
initial_w = np.zeros_like(w_init)
initial_b = 0.
# some gradient descent settings
iterations = 1000
alpha = 5.0e-7
# run gradient descent
w_final, b_final, J_hist = gradient_descent(X_train, y_train, initial_w, initial_b,
                                                    compute_cost, compute_gradient,
                                                    alpha, iterations)
found by gradient descent: {b_final:0.2f},{w_final} ")
```

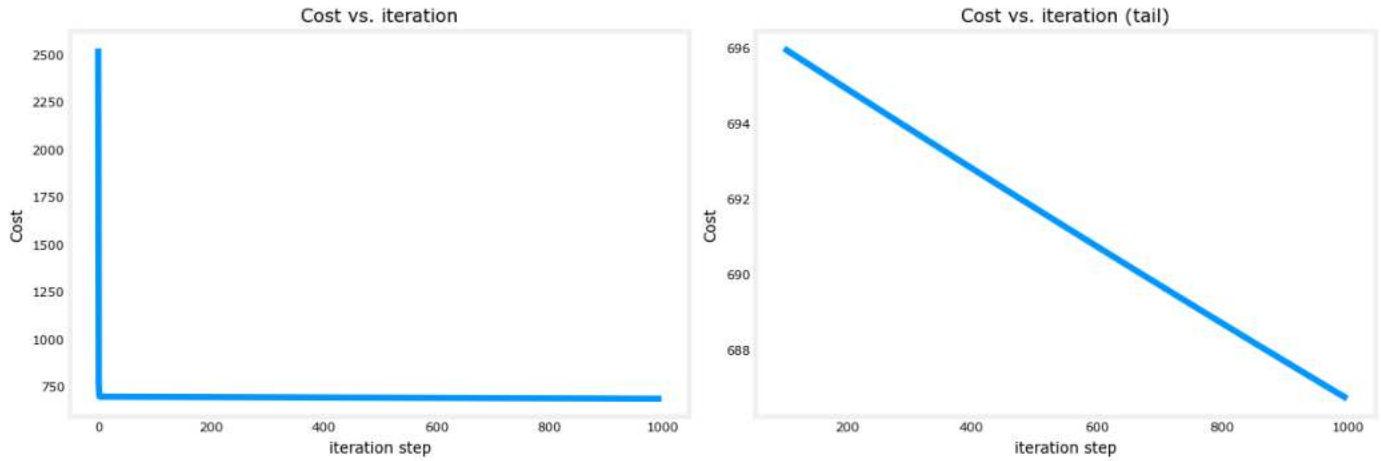
Loading [MathJax]/extensions/Safe.js

```
m,_ = X_train.shape
for i in range(m):
    print(f"prediction: {np.dot(X_train[i], w_final) + b_final:0.2f}, target value: {y_t
```

```
Iteration    0: Cost  2529.46
Iteration  100: Cost   695.99
Iteration  200: Cost   694.92
Iteration  300: Cost   693.86
Iteration  400: Cost   692.81
Iteration  500: Cost   691.77
Iteration  600: Cost   690.73
Iteration  700: Cost   689.71
Iteration  800: Cost   688.70
Iteration  900: Cost   687.69
b,w found by gradient descent: -0.00,[ 0.2   0.   -0.01 -0.07]
prediction: 426.19, target value: 460
prediction: 286.17, target value: 232
prediction: 171.47, target value: 178
```

The line initial_w = np.zeros_like(w_init) initializes a new array initial_w with the same shape and data type as the w_init array. It creates a new array of zeros that has the same dimensions as w_init.

In [91]:
```
# plot cost versus iteration
fig, (ax1, ax2) = plt.subplots(1, 2, constrained_layout=True, figsize=(12, 4))
ax1.plot(J_hist)
ax2.plot(100 + np.arange(len(J_hist[100:])), J_hist[100:])
ax1.set_title("Cost vs. iteration");  ax2.set_title("Cost vs. iteration (tail)")
ax1.set_ylabel('Cost')             ;  ax2.set_ylabel('Cost')
ax1.set_xlabel('iteration step')   ;  ax2.set_xlabel('iteration step')
plt.show()
```



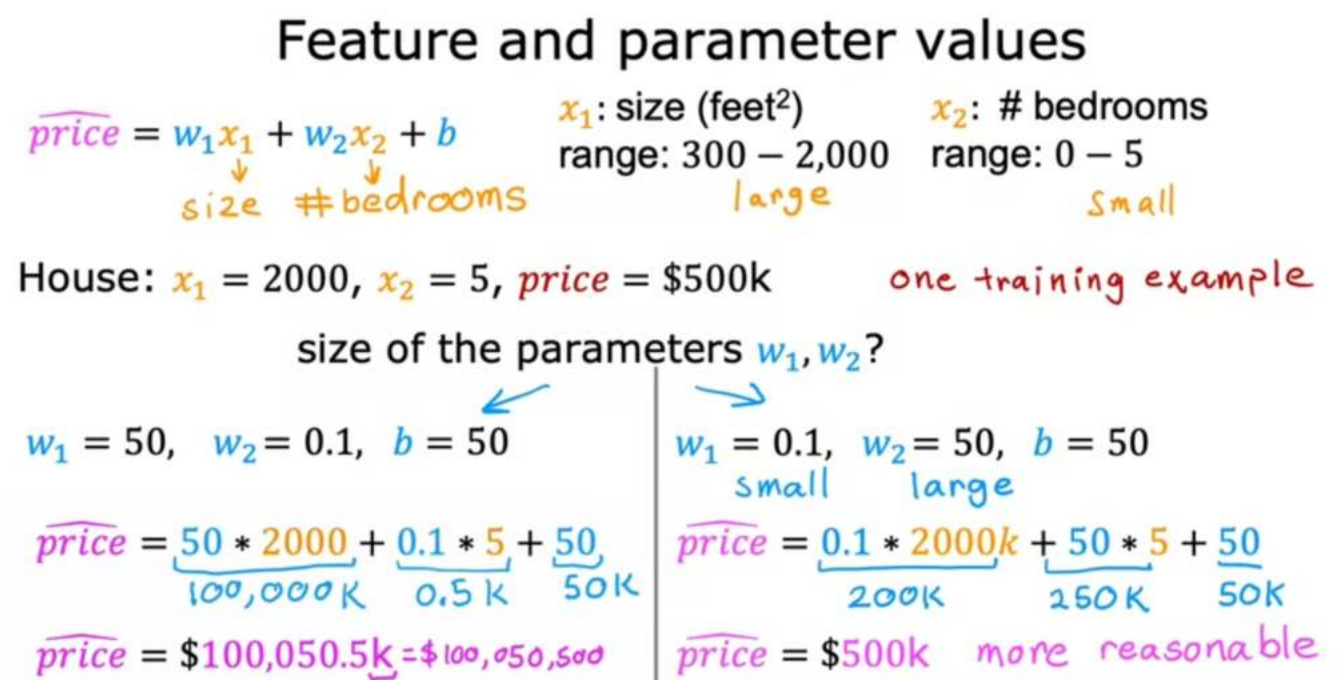
These results are not inspiring! Cost is still declining and our predictions are not very accurate. The next lab will explore how to improve on this.

Gradient Descent in Practice

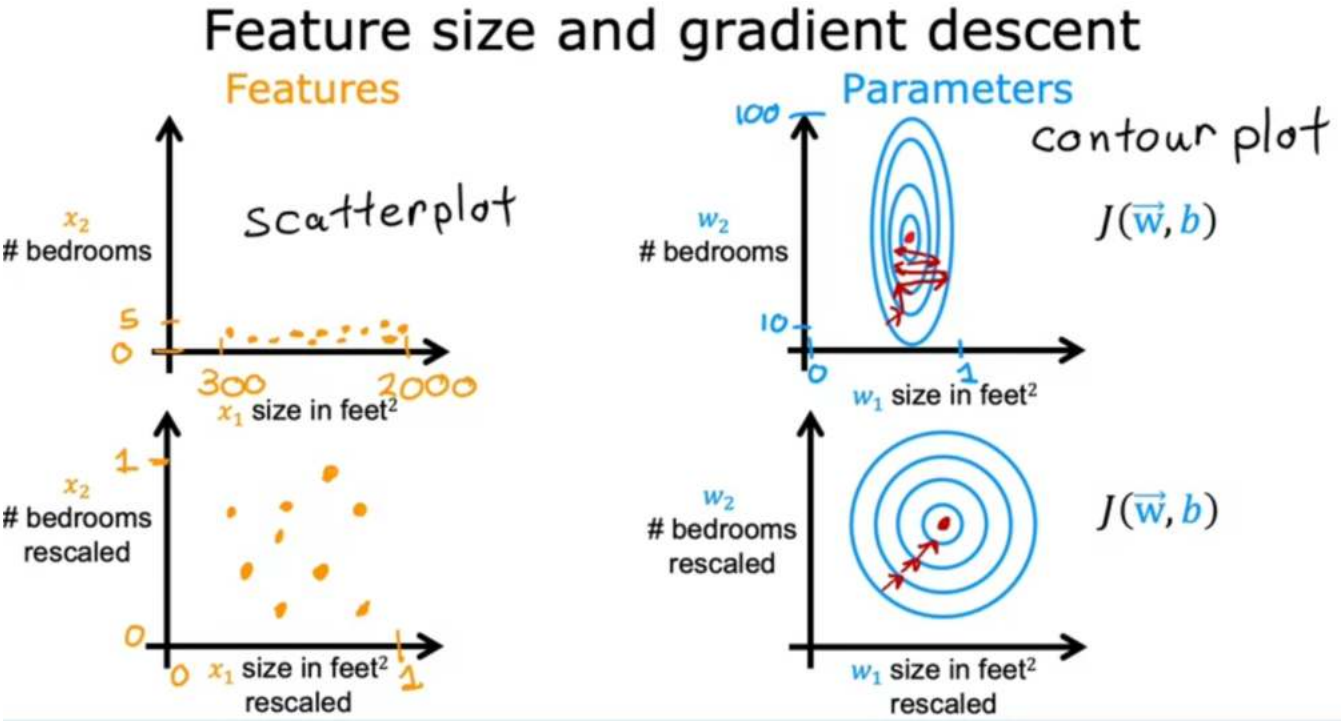
Feature scaling

Feature scaling able the gradient descent to run much faster

Loading [MathJax]/extensions/Safe.js



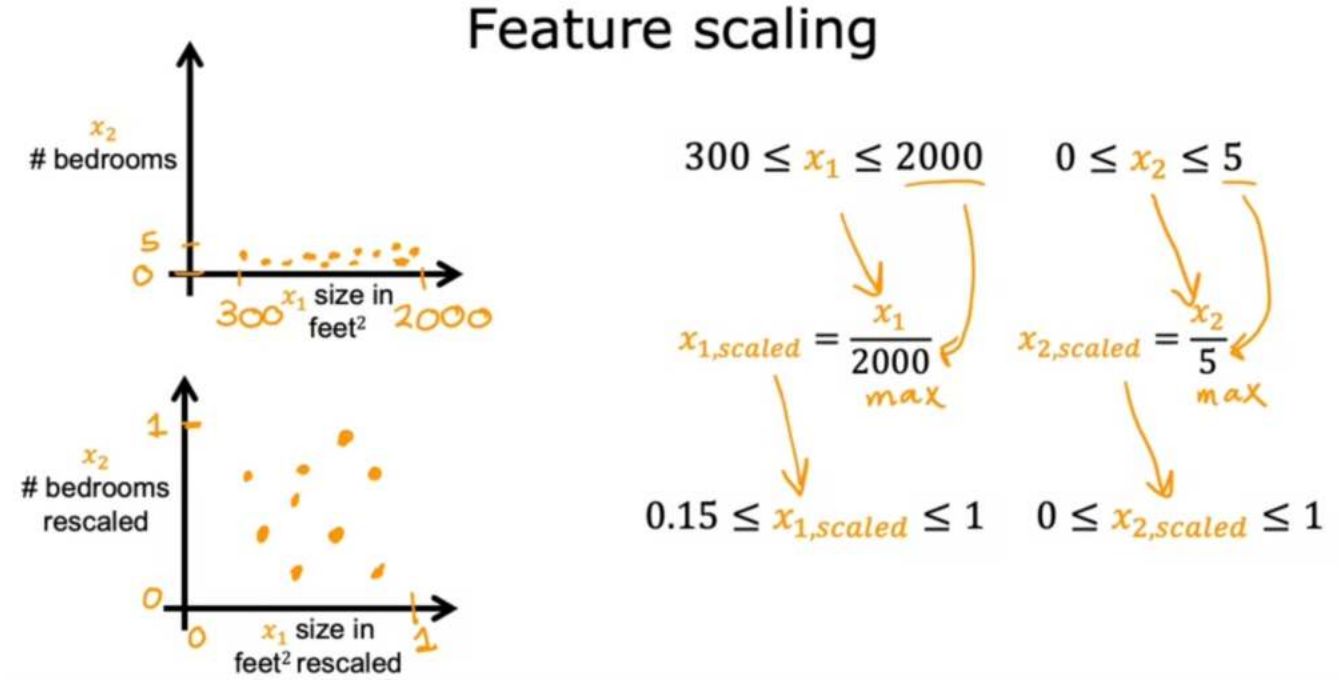
When a possible range of values of a feature is large like the size and square feet which go all the way up to 2000. It is more likely that a good model will learn to choose a relatively small parameter value, like 0.1. Likewise, when the possible values of the feature are small, like the number of bedrooms, then a reasonable value for its parameter will be relatively large like 50



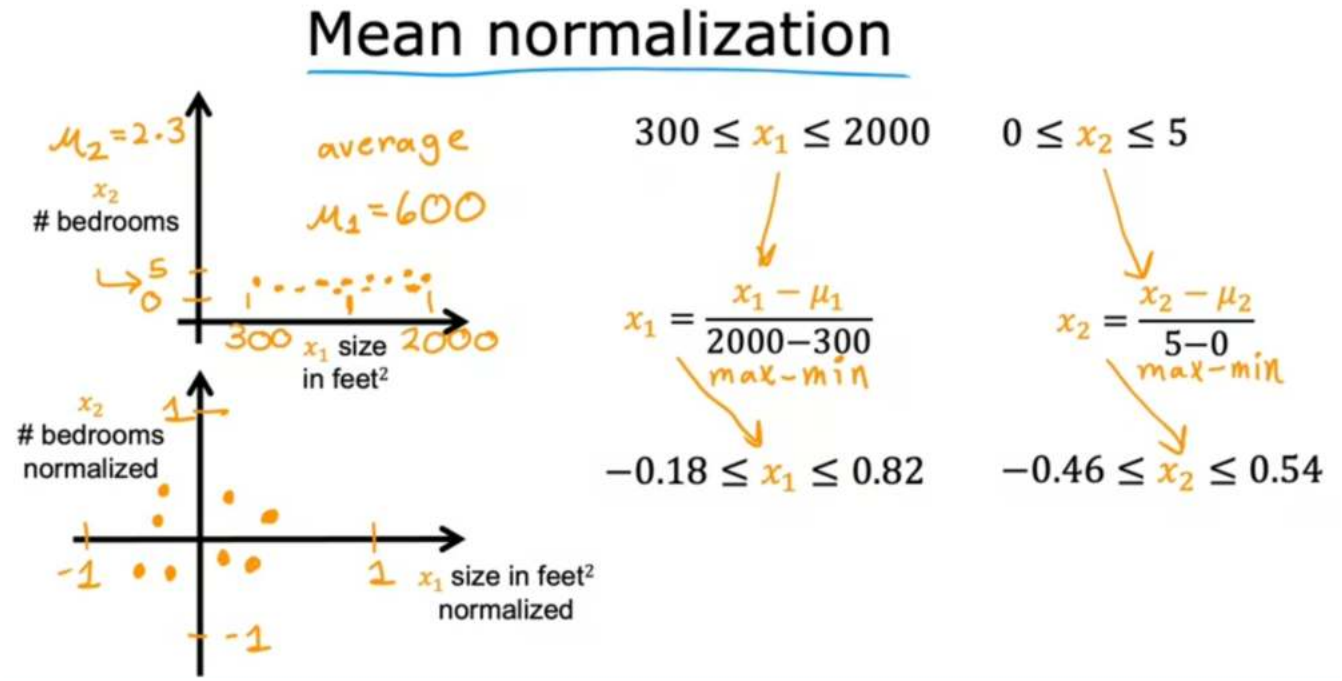
How do you actually scale features?

Well, if x_1 ranges from 3-2,000, one way to get a scale version of x*1 is to take each original x1* value and divide by 2,000, the maximum of the range. The scale x_1 will range from 0.15 up to one. Similarly, since x_2 ranges from 0-5, you can calculate a scale version of x_2 by taking each original x_2 and dividing by five, which is again the maximum.

Loading [MathJax]/extensions/Safe.js

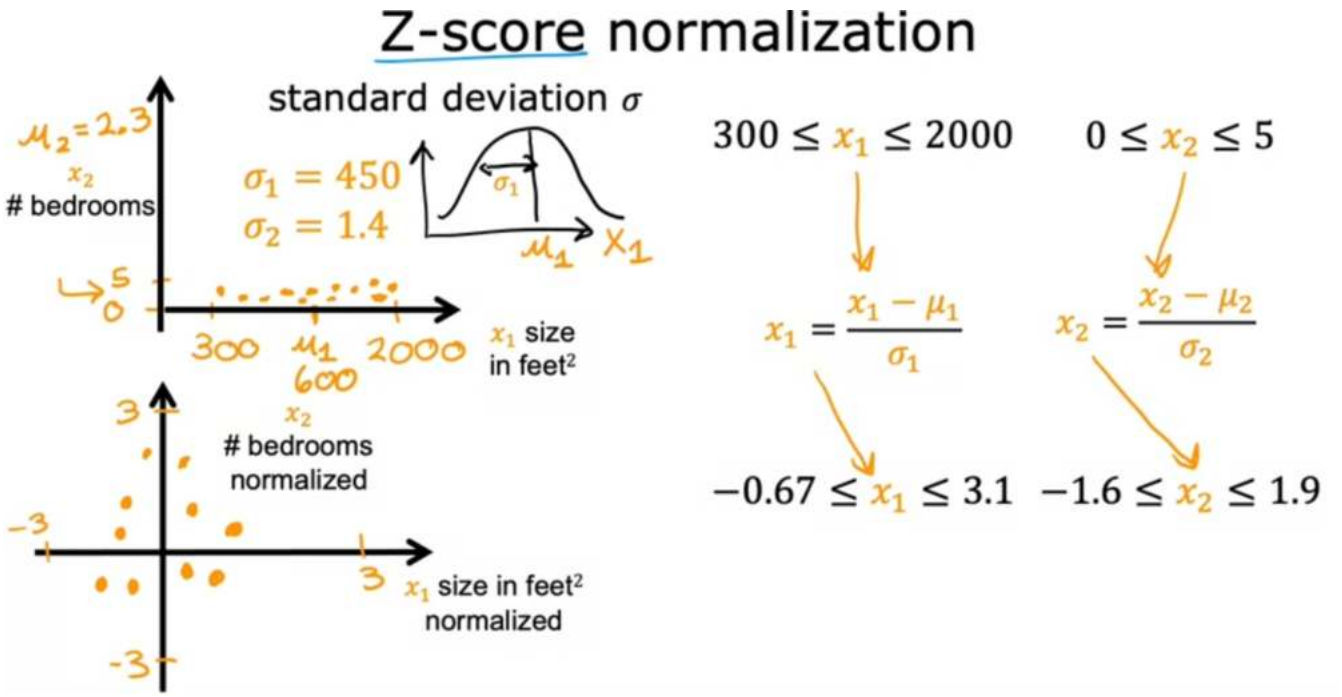


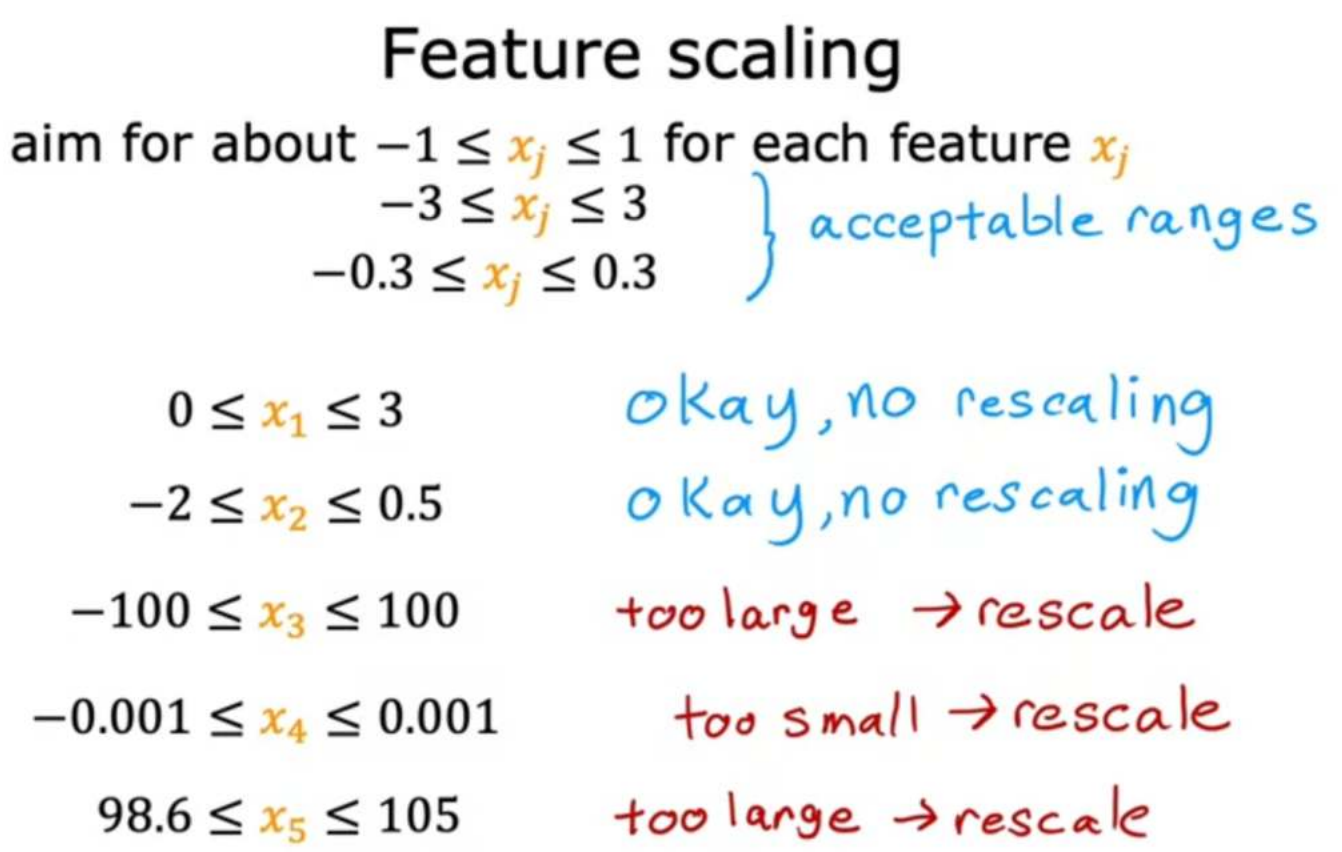
In addition to dividing by the maximum, you can also do what's called mean normalization. What this looks like is, you start with the original features and then you re-scale them so that both of them are centered around zero. Whereas before they only had values greater than zero, now they have both negative 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.



Loading [MathJax]/extensions/Safe.js

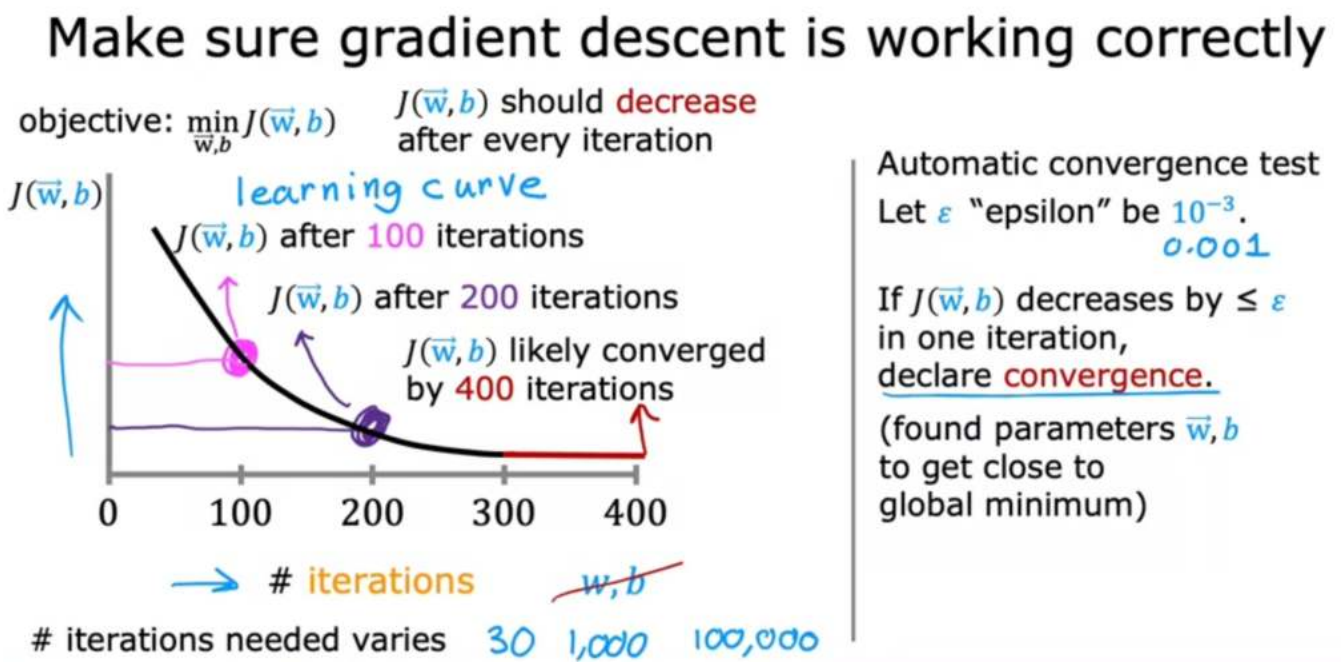
There's one last common re-scaling method call Z-score normalization. To implement Z-score normalization, you need to calculate something called the standard deviation of each feature. To implement a Z-score normalization, you first calculate the mean Mu, as well as the standard deviation, which is often denoted by the lowercase Greek alphabet Sigma of each feature. For instance, maybe feature 1 has a standard deviation of 450 and mean 600, then to Z-score normalize x_1, take each x_1, subtract Mu_1, and then divide by the standard deviation, which I'm going to denote as Sigma 1. What you may find is that the Z-score normalized x_1 now ranges from negative 0.67-3.1. Similarly, if you calculate the second features standard deviation to be 1.4 and mean to be 2.3, then you can compute x_2 minus Mu_2 divided by Sigma_2, and in this case, the Z-score normalized by x_2 might now range from negative 1.6-1.9.





Loading [MathJax]/extensions/Safe.js

Checking gradient descent for convergence



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

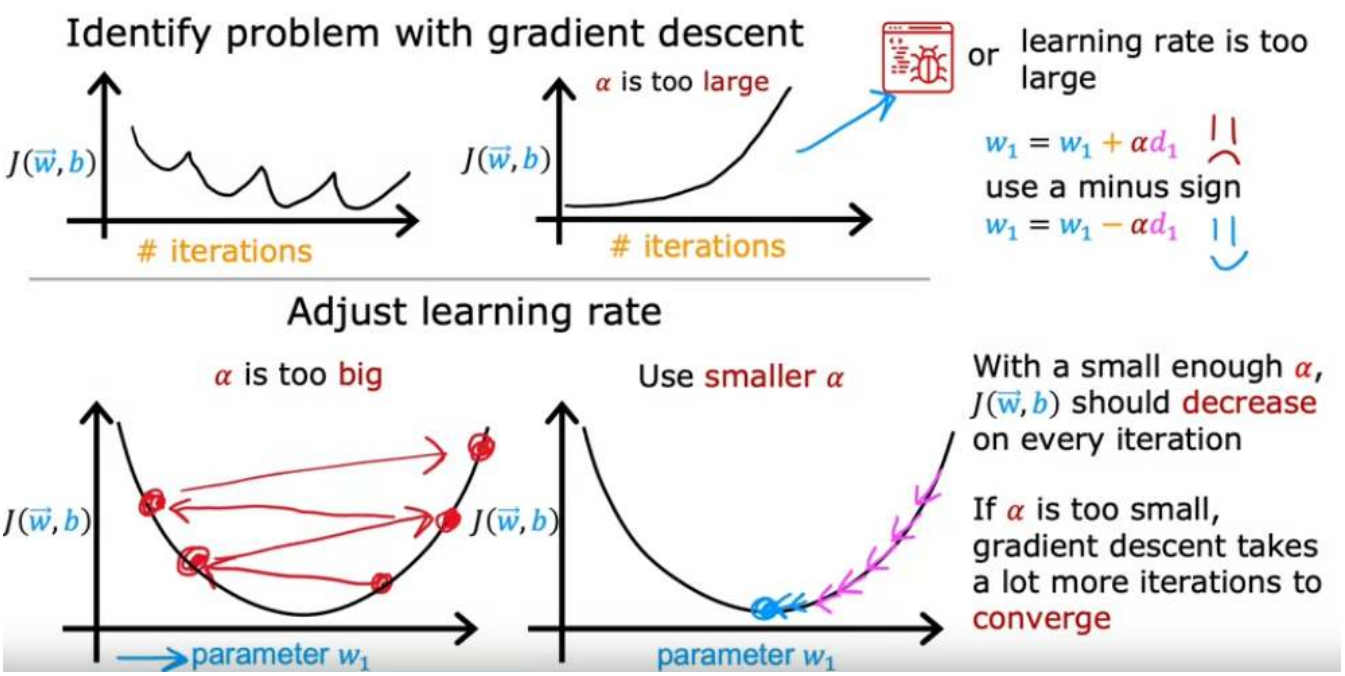
Choosing the learning rate

The linear algorithm run much better with an appropirate choice of learning rate.

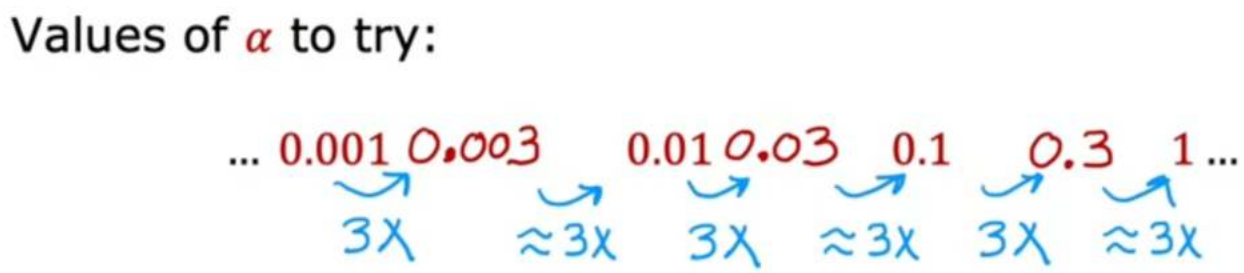
If it too small it will run very slowly & if it is too large, it may not even converge.

Concretely, if you plot the cost for a number of iterations and notice that the costs sometimes goes up and sometimes goes down, you should take that as a clear sign that gradient descent is not working properly. This could mean that there's a bug in the code. Or sometimes it could mean that your learning rate is too large.

Loading [MathJax]/extensions/Safe.js



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. One important trade-off is that if your learning rate is too small, then gradient descents can take a lot of iterations to converge.



Lab: Feature scaling and learning rate

Goals

In this lab you will:

- Utilize the multiple variables routines developed in the previous lab
- run Gradient Descent on a data set with multiple features
- explore the impact of the *learning rate alpha* on gradient descent
- improve performance of gradient descent by *feature scaling* using z-score normalization

Loading [MathJax]/extensions/Safe.js

In [93]:
```
import numpy as np
import matplotlib.pyplot as plt
from lab_utils_multi import  load_house_data, run_gradient_descent
from lab_utils_multi import  norm_plot, plt_equal_scale, plot_cost_i_w
from lab_utils_common import dlc
np.set_printoptions(precision=2)
plt.style.use('deeplearning.mplstyle')
```

Notation

General Notation	Description	Python (if applicable)
a	scalar, non bold	
$\mathbf{a}$	vector, bold	
$\mathbf{A}$	matrix, bold capital	
Regression		
$\mathbf{X}$	training example maxtrix	`X_train`
$\mathbf{y}$	training example targets	`y_train`
$\mathbf{x}^{(i)}$, $y^{(i)}$	i_{th}Training Example	`X[i]` , `y[i]`
m	number of training examples	`m`
n	number of features in each example	`n`
$\mathbf{w}$	parameter: weight,	`w`
b	parameter: bias	`b`
$f_{\mathbf{w},b}(\mathbf{x}^{(i)})$	The result of the model evaluation at $\mathbf{x}^{(i)}$ parameterized by $\mathbf{w},b$: $f_{\mathbf{w},b}(\mathbf{x}^{(i)}) = \mathbf{w} \cdot \mathbf{x}^{(i)}+b$	`f_wb`
$\frac{\partial J(\mathbf{w},b)}{\partial w_j}$	the gradient or partial derivative of cost with respect to a parameter w_j	`dj_dw[j]`
$\frac{\partial J(\mathbf{w},b)}{\partial b}$	the gradient or partial derivative of cost with respect to a parameter b	`dj_db`

Problem Statement

As in the previous labs, you will use the motivating example of housing price prediction. The training data set contains many examples with 4 features (size, bedrooms, floors and age) shown in the table below. Note, in this lab, the Size feature is in sqft while earlier labs utilized 1000 sqft. This data set is larger than the previous lab.

We would like to build a linear regression model using these values so we can then predict the price for other houses - say, a house with 1200 sqft, 3 bedrooms, 1 floor, 40 years old.

Dataset:

Size (sqft)	Number of Bedrooms	Number of floors	Age of Home	Price (1000s dollars)
952	2	1	65	271.5

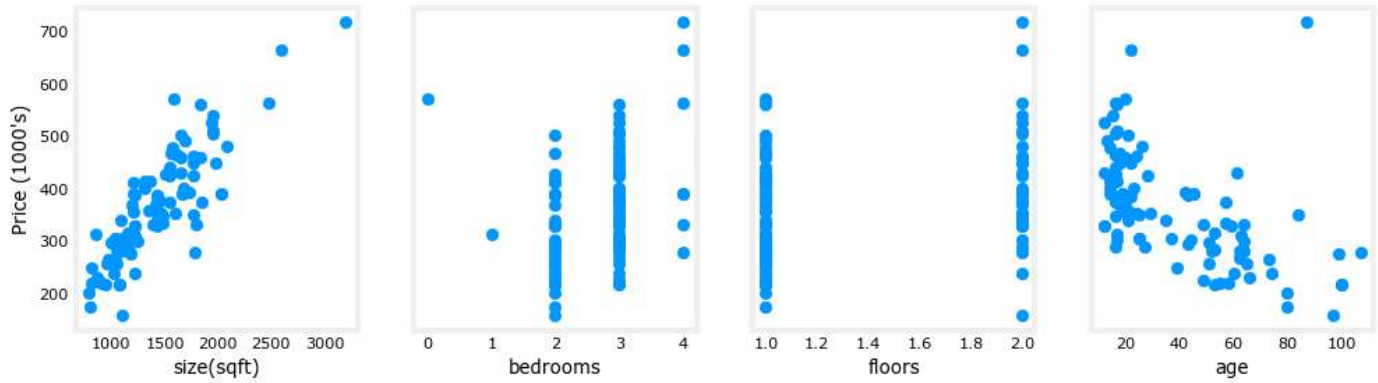
Loading [MathJax]/extensions/Safe.js

Size (sqft)	Number of Bedrooms	Number of floors	Age of Home	Price (1000s dollars)
1244	3	2	64	232
1947	3	2	17	509.8
...	...	...	...	...

In [96]:
```
# load the dataset
X_train, y_train = load_house_data()
X_features = ['size(sqft)','bedrooms','floors','age']
```

Let's view the dataset and its features by plotting each feature versus price.

In [99]:
```
fig, ax = plt.subplots(1, 4, figsize = (12, 3), sharey = True)
for i in range(len(ax)):
    ax[i].scatter(X_train[:,i], y_train)
    ax[i].set_xlabel(X_features[i])
ax[0].set_ylabel("Price (1000's)")
plt.show()
```



Plotting each feature vs. the target, price, provides some indication of which features have the strongest influence on price. Above, increasing size also increases price. Bedrooms and floors don't seem to have a strong impact on price. Newer houses have higher prices than older houses.

Let's break down the code step by step:

1. fig, ax = plt.subplots(1, 4, figsize=(12, 3), sharey=True): This line creates a figure and four subplots arranged in a 1x4 grid. The fig variable represents the entire figure, and ax is an array containing the four subplots. The figsize=(12, 3) argument sets the size of the figure, and sharey=True ensures that all subplots share the same y-axis.

1. for i in range(len(ax)):: This loop iterates over each subplot index, ranging from 0 to the length of the ax array (which is 4 in this case).

1. ax[i].scatter(X_train[:,i], y_train): This line plots a scatter plot on the i-th subplot. It uses the scatter function from Matplotlib, where X_train[:,i] represents the values of the i-th feature in the training data, and y_train represents the target variable values. It plots the feature values on the x-axis and the corresponding target variable values on the y-axis.

1. ax[i].set_xlabel(X_features[i]): This line sets the x-axis label of the i-th subplot to the name of the i-th feature. It assumes that there is a list or array X_features containing the names of the features.

1. ax[0].set_ylabel("Price (1000's)"): This line sets the y-axis label of the first subplot to "Price (1000's)". It represents the label for the target variable.

Loading [MathJax]/extensions/Safe.js

1. plt.show(): This line displays the plot with all the subplots.

Gradient Descent With Multiple Variables

Here are the equations you developed in the last lab on gradient descent for multiple variables.:

$$\begin{align*} \text{repeat}&\text{ until convergence:} \; \lbrace \newline\; & w_j := w_j - \alpha \frac{\partial J(\mathbf{w},b)}{\partial w_j} \tag{1} \; & \text{for j = 0..n-1}\newline &b\ \ := b - \alpha \frac{\partial J(\mathbf{w},b)}{\partial b} \newline \rbrace \end{align*}$$
where, n is the number of features, parameters w_j, b, are updated simultaneously and where

$$ \begin{align} \frac{\partial J(\mathbf{w},b)}{\partial w_j} &= \frac{1}{m} \sum\limits_{i = 0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})x_{j}^{(i)} \tag{2} \\ \frac{\partial J(\mathbf{w},b)}{\partial b} &= \frac{1}{m} \sum\limits_{i = 0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)}) \tag{3} \end{align} $$

- m is the number of training examples in the data set
- $f_{\mathbf{w},b}(\mathbf{x}^{(i)})$ is the model's prediction, while $y^{(i)}$ is the target value

Learning Rate

The lectures discussed some of the issues related to setting the learning rate α. The learning rate controls the size of the update to the parameters. See equation (1) above. It is shared by all the parameters.

Let's run gradient descent and try a few settings of α on our data set

α = 9.9e-7

In [100…

```
#set alpha to 9.9e-7
_, _, hist = run_gradient_descent(X_train, y_train, 10, alpha = 9.9e-7)
```

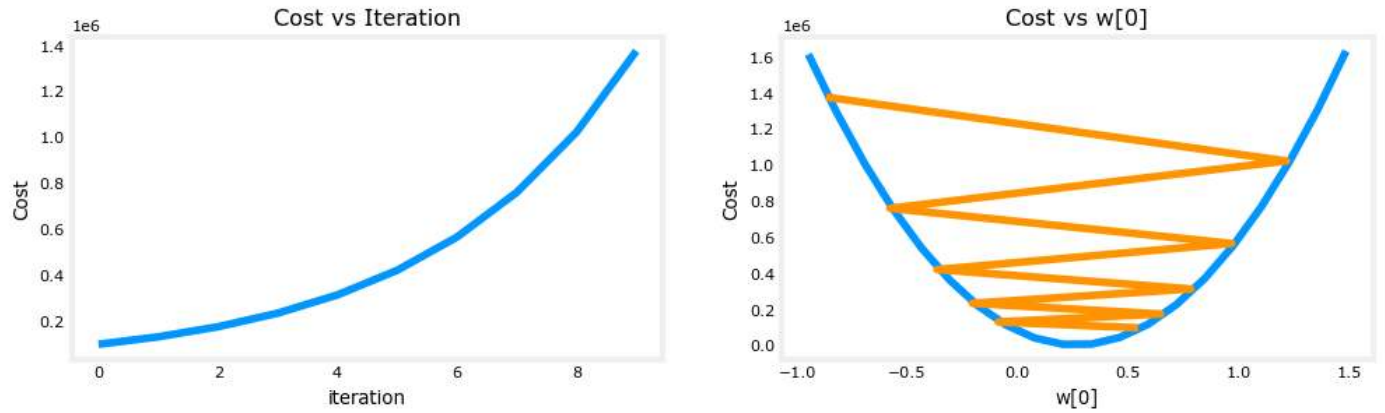
Loading [MathJax]/extensions/Safe.js

```
Iteration Cost          w0       w1       w2       w3       b       djdw0    djdw1    dj
dw2    djdw3    djdb
---------------------|--------|--------|--------|--------|--------|--------|--------|---
------|--------|--------|
        0 9.55884e+04  5.5e-01  1.0e-03  5.1e-04  1.2e-02  3.6e-04 -5.5e+05 -1.0e+03 -5.
2e+02 -1.2e+04 -3.6e+02
        1 1.28213e+05 -8.8e-02 -1.7e-04 -1.0e-04 -3.4e-03 -4.8e-05  6.4e+05  1.2e+03  6.
2e+02  1.6e+04  4.1e+02
        2 1.72159e+05  6.5e-01  1.2e-03  5.9e-04  1.3e-02  4.3e-04 -7.4e+05 -1.4e+03 -7.
0e+02 -1.7e+04 -4.9e+02
        3 2.31358e+05 -2.1e-01 -4.0e-04 -2.3e-04 -7.5e-03 -1.2e-04  8.6e+05  1.6e+03  8.
3e+02  2.1e+04  5.6e+02
        4 3.11100e+05  7.9e-01  1.4e-03  7.1e-04  1.5e-02  5.3e-04 -1.0e+06 -1.8e+03 -9.
5e+02 -2.3e+04 -6.6e+02
        5 4.18517e+05 -3.7e-01 -7.1e-04 -4.0e-04 -1.3e-02 -2.1e-04  1.2e+06  2.1e+03  1.
1e+03  2.8e+04  7.5e+02
        6 5.63212e+05  9.7e-01  1.7e-03  8.7e-04  1.8e-02  6.6e-04 -1.3e+06 -2.5e+03 -1.
3e+03 -3.1e+04 -8.8e+02
        7 7.58122e+05 -5.8e-01 -1.1e-03 -6.2e-04 -1.9e-02 -3.4e-04  1.6e+06  2.9e+03  1.
5e+03  3.8e+04  1.0e+03
        8 1.02068e+06  1.2e+00  2.2e-03  1.1e-03  2.3e-02  8.3e-04 -1.8e+06 -3.3e+03 -1.
7e+03 -4.2e+04 -1.2e+03
        9 1.37435e+06 -8.7e-01 -1.7e-03 -9.1e-04 -2.7e-02 -5.2e-04  2.1e+06  3.9e+03  2.
0e+03  5.1e+04  1.4e+03
w,b found by gradient descent: w: [-0.87 -0.   -0.   -0.03], b: -0.00
```

use _ as a placeholder variable when you don't need to use or assign a particular value.

It appears the learning rate is too high. The solution does not converge. Cost is *increasing* rather than decreasing. Let's plot the result:

In [101…
```
plot_cost_i_w(X_train, y_train, hist)
```



The plot on the right shows the value of one of the parameters, w_0. At each iteration, it is overshooting the optimal value and as a result, cost ends up *increasing* rather than approaching the minimum. Note that this is not a completely accurate picture as there are 4 parameters being modified each pass rather than just one. This plot is only showing w_0 with the other parameters fixed at benign values. In this and later plots you may notice the blue and orange lines being slightly off.

α = 9e-7

Let's try a bit smaller value and see what happens.

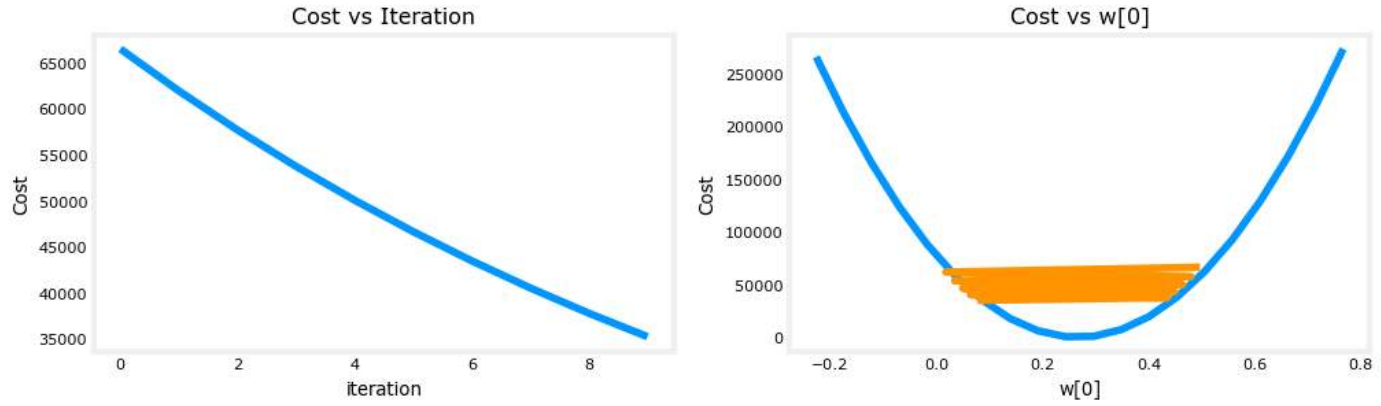
In [102…
```
#set alpha to 9e-7
_,_,hist = run_gradient_descent(X_train, y_train, 10, alpha = 9e-7)
```

Loading [MathJax]/extensions/Safe.js

```
Iteration Cost          w0       w1       w2       w3       b       djdw0    djdw1    dj
dw2    djdw3    djdb
---------------------|--------|--------|--------|--------|--------|--------|--------|---
------|--------|--------|
        0 6.64616e+04  5.0e-01  9.1e-04  4.7e-04  1.1e-02  3.3e-04 -5.5e+05 -1.0e+03 -5.
2e+02 -1.2e+04 -3.6e+02
        1 6.18990e+04  1.8e-02  2.1e-05  2.0e-06 -7.9e-04  1.9e-05  5.3e+05  9.8e+02  5.
2e+02  1.3e+04  3.4e+02
        2 5.76572e+04  4.8e-01  8.6e-04  4.4e-04  9.5e-03  3.2e-04 -5.1e+05 -9.3e+02 -4.
8e+02 -1.1e+04 -3.4e+02
        3 5.37137e+04  3.4e-02  3.9e-05  2.8e-06 -1.6e-03  3.8e-05  4.9e+05  9.1e+02  4.
8e+02  1.2e+04  3.2e+02
        4 5.00474e+04  4.6e-01  8.2e-04  4.1e-04  8.0e-03  3.2e-04 -4.8e+05 -8.7e+02 -4.
5e+02 -1.1e+04 -3.1e+02
        5 4.66388e+04  5.0e-02  5.6e-05  2.5e-06 -2.4e-03  5.6e-05  4.6e+05  8.5e+02  4.
5e+02  1.2e+04  2.9e+02
        6 4.34700e+04  4.5e-01  7.8e-04  3.8e-04  6.4e-03  3.2e-04 -4.4e+05 -8.1e+02 -4.
2e+02 -9.8e+03 -2.9e+02
        7 4.05239e+04  6.4e-02  7.0e-05  1.2e-06 -3.3e-03  7.3e-05  4.3e+05  7.9e+02  4.
2e+02  1.1e+04  2.7e+02
        8 3.77849e+04  4.4e-01  7.5e-04  3.5e-04  4.9e-03  3.2e-04 -4.1e+05 -7.5e+02 -3.
9e+02 -9.1e+03 -2.7e+02
        9 3.52385e+04  7.7e-02  8.3e-05 -1.1e-06 -4.2e-03  8.9e-05  4.0e+05  7.4e+02  3.
9e+02  1.0e+04  2.5e+02
w,b found by gradient descent: w: [ 7.74e-02  8.27e-05 -1.06e-06 -4.20e-03], b: 0.00
```

Cost is decreasing throughout the run showing that alpha is not too large.

In [103…
```
plot_cost_i_w(X_train, y_train, hist)
```



On the left, you see that cost is decreasing as it should. On the right, you can see that w_0 is still oscillating around the minimum, but it is decreasing each iteration rather than increasing. Note above that `dj_dw[0]` changes sign with each iteration as `w[0]` jumps over the optimal value. This alpha value will converge. You can vary the number of iterations to see how it behaves.

α = 1e-7

Let's try a bit smaller value for α and see what happens.

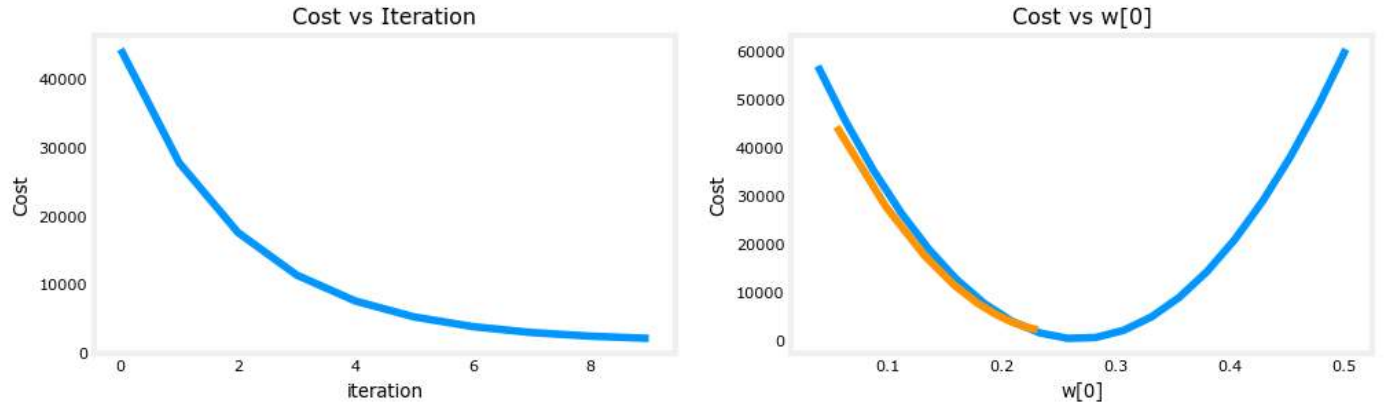
In [104…
```
#set alpha to 1e-7
_,_,hist = run_gradient_descent(X_train, y_train, 10, alpha = 1e-7)
```

Loading [MathJax]/extensions/Safe.js

```
Iteration Cost          w0       w1       w2       w3       b       djdw0    djdw1    dj
dw2    djdw3    djdb
---------------------|--------|--------|--------|--------|--------|--------|--------|---
------|--------|--------|
        0 4.42313e+04  5.5e-02  1.0e-04  5.2e-05  1.2e-03  3.6e-05 -5.5e+05 -1.0e+03 -5.
2e+02 -1.2e+04 -3.6e+02
        1 2.76461e+04  9.8e-02  1.8e-04  9.2e-05  2.2e-03  6.5e-05 -4.3e+05 -7.9e+02 -4.
0e+02 -9.5e+03 -2.8e+02
        2 1.75102e+04  1.3e-01  2.4e-04  1.2e-04  2.9e-03  8.7e-05 -3.4e+05 -6.1e+02 -3.
1e+02 -7.3e+03 -2.2e+02
        3 1.13157e+04  1.6e-01  2.9e-04  1.5e-04  3.5e-03  1.0e-04 -2.6e+05 -4.8e+02 -2.
4e+02 -5.6e+03 -1.8e+02
        4 7.53002e+03  1.8e-01  3.3e-04  1.7e-04  3.9e-03  1.2e-04 -2.1e+05 -3.7e+02 -1.
9e+02 -4.2e+03 -1.4e+02
        5 5.21639e+03  2.0e-01  3.5e-04  1.8e-04  4.2e-03  1.3e-04 -1.6e+05 -2.9e+02 -1.
5e+02 -3.1e+03 -1.1e+02
        6 3.80242e+03  2.1e-01  3.8e-04  1.9e-04  4.5e-03  1.4e-04 -1.3e+05 -2.2e+02 -1.
1e+02 -2.3e+03 -8.6e+01
        7 2.93826e+03  2.2e-01  3.9e-04  2.0e-04  4.6e-03  1.4e-04 -9.8e+04 -1.7e+02 -8.
6e+01 -1.7e+03 -6.8e+01
        8 2.41013e+03  2.3e-01  4.1e-04  2.1e-04  4.7e-03  1.5e-04 -7.7e+04 -1.3e+02 -6.
5e+01 -1.2e+03 -5.4e+01
        9 2.08734e+03  2.3e-01  4.2e-04  2.1e-04  4.8e-03  1.5e-04 -6.0e+04 -1.0e+02 -4.
9e+01 -7.5e+02 -4.3e+01
w,b found by gradient descent: w: [2.31e-01 4.18e-04 2.12e-04 4.81e-03], b: 0.00
```

Cost is decreasing throughout the run showing that α is not too large.

In [105…
```
plot_cost_i_w(X_train,y_train,hist)
```



On the left, you see that cost is decreasing as it should. On the right you can see that w_0 is decreasing without crossing the minimum. Note above that `dj_w0` is negative throughout the run. This solution will also converge, though not quite as quickly as the previous example.

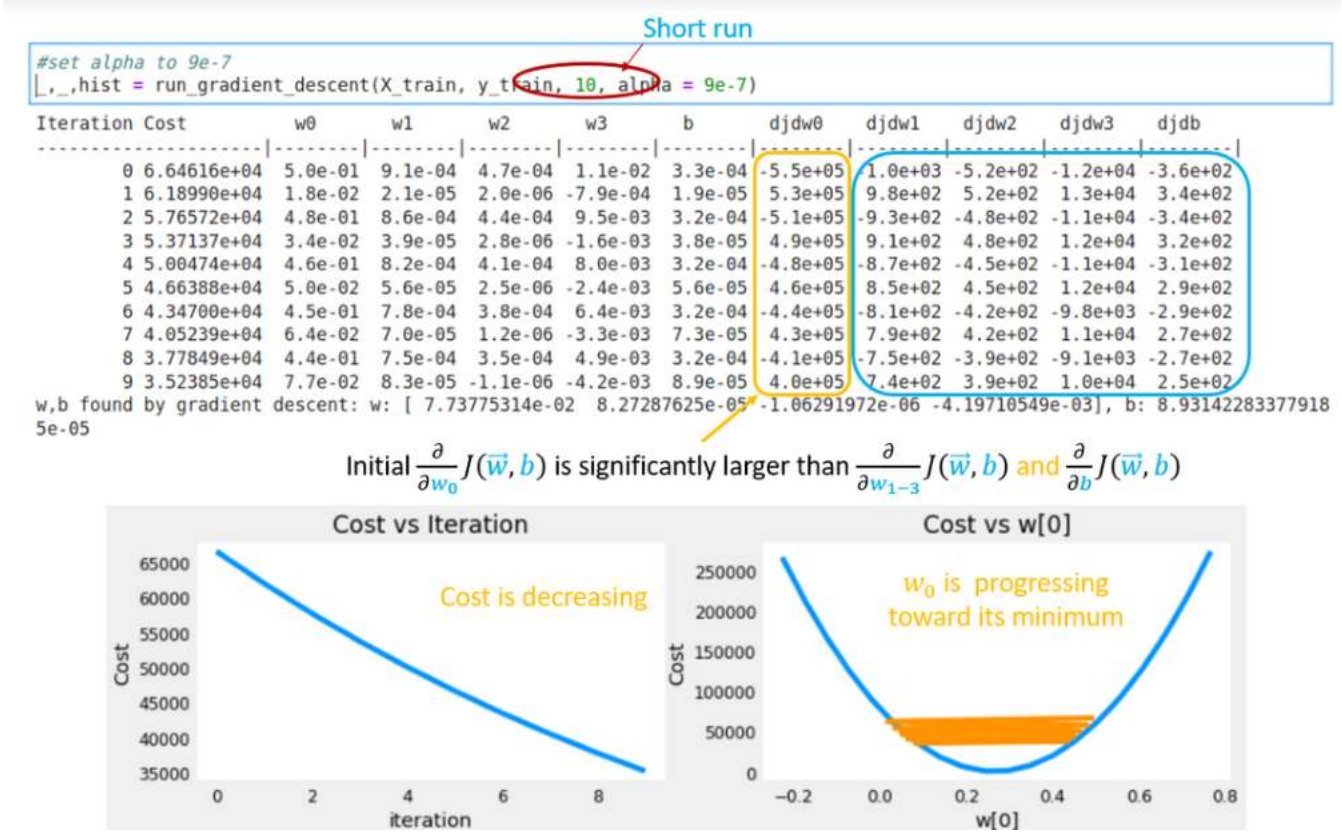
Feature Scaling

The lectures described the importance of rescaling the dataset so the features have a similar range. If you are interested in the details of why this is the case, click on the 'details' header below. If not, the section below will walk through an implementation of how to do feature scaling.

Details

Let's look again at the situation with = 9e-7. This is pretty close to the maximum value we can set to without diverging. This is a short run showing the first few iterations:

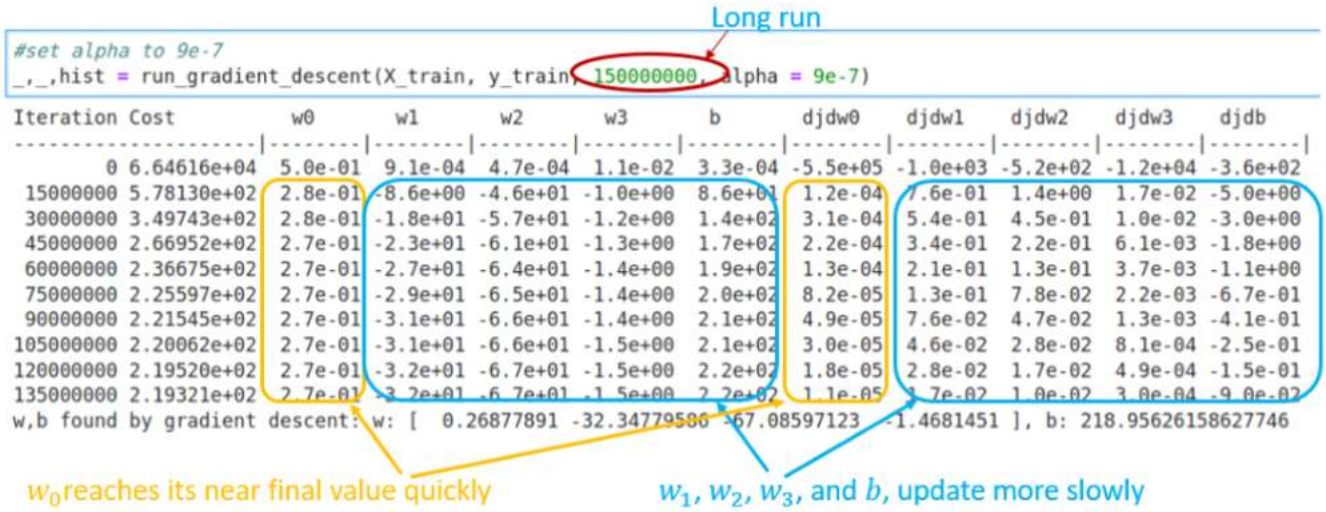
Loading [MathJax]/extensions/Safe.js



Above, while cost is being decreased, its clear that *w*0

is making more rapid progress than the other parameters due to its much larger gradient.

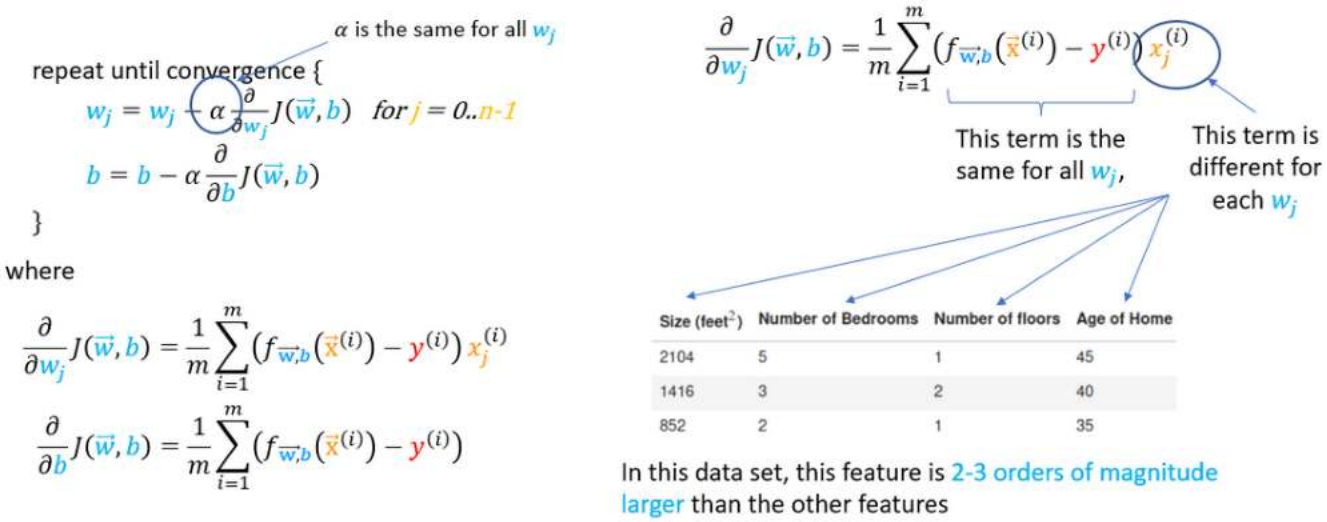
The graphic below shows the result of a very long run with = 9e-7. This takes several hours.



Above, you can see cost decreased slowly after its initial reduction. Notice the difference between w0 and w1,w2,w3 as well as dj_dw0 and dj_dw1-3. w0 reaches its near final value very quickly and dj_dw0 has quickly decreased to a small value showing that w0 is near the final value. The other parameters were reduced much more slowly.

Why is this? Is there something we can improve? See below:

Loading [MathJax]/extensions/Safe.js



The figure above shows why w's are updated unevenly.

- α is shared by all parameter updates (w's and b).
- the common error term is multiplied by the features for the w's. (not b).
- the features vary significantly in magnitude making some features update much faster than others. In this case, w_0 is multiplied by 'size(sqft)', which is generally > 1000, while w_1 is multiplied by 'number of bedrooms', which is generally 2-4.

The solution is Feature Scaling.

The lectures discussed three different techniques:

- Feature scaling, essentially dividing each positive feature by its maximum value, or more generally, rescale each feature by both its minimum and maximum values using (x-min)/(max-min). Both ways normalizes features to the range of -1 and 1, where the former method works for positive features which is simple and serves well for the lecture's example, and the latter method works for any features.
- Mean normalization: $x_i := \dfrac{x_i - \mu_i}{max - min} $
- Z-score normalization which we will explore below.

Loading [MathJax]/extensions/Safe.js

z-score normalization

After z-score normalization, all features will have a mean of 0 and a standard deviation of 1.

To implement z-score normalization, adjust your input values as shown in this formula: $$x^{(i)}_j = \dfrac{x^{(i)}_j - \mu_j}{\sigma_j} \tag{4}$$ where j selects a feature or a column in the $\mathbf{X}$ matrix. $µ_j$ is the mean of all the values for feature (j) and σ_j is the standard deviation of feature (j). $$ \begin{align} \mu_j &= \frac{1}{m} \sum_{i=0}^{m-1} x^{(i)}_j \tag{5}\\ \sigma^2_j &= \frac{1}{m} \sum_{i=0}^{m-1} (x^{(i)}_j - \mu_j)^2 \tag{6} \end{align} $$

> **Implementation Note:** When normalizing the features, it is important to store the values used for normalization - the mean value and the standard deviation used for the computations. After learning the parameters from the model, we often want to predict the prices of houses we have not seen before. Given a new x value (living room area and number of bed- rooms), we must first normalize x using the mean and standard deviation that we had previously computed from the training set.

Implementation

In [107…

```
def zscore_normalize_features(X):
    """
    computes  X, zcore normalized by column

    Args:
      X (ndarray (m,n))     : input data, m examples, n features

    Returns:
      X_norm (ndarray (m,n)): input normalized by column
      mu (ndarray (n,))     : mean of each feature
      sigma (ndarray (n,))  : standard deviation of each feature
    """
    # find the mean of each column/feature
    mu     = np.mean(X, axis=0)                 # mu will have shape (n,)
    # find the standard deviation of each column/feature
    sigma  = np.std(X, axis=0)                  # sigma will have shape (n,)
    # element-wise, subtract mu for that column from each example, divide by std for tha
    X_norm = (X - mu) / sigma

    return (X_norm, mu, sigma)
```

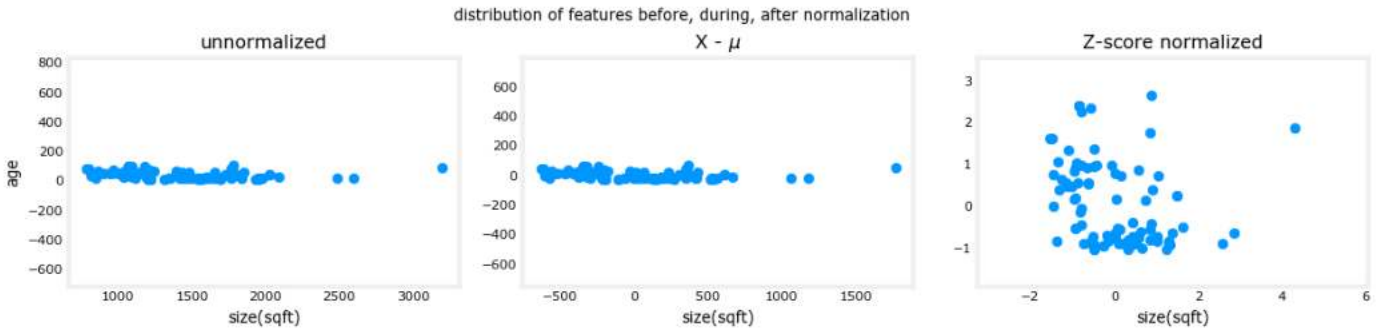
Let's look at the steps involved in Z-score normalization. The plot below shows the transformation step by step.

In [108…

```
mu     = np.mean(X_train,axis=0)
sigma  = np.std(X_train,axis=0)
X_mean = (X_train - mu)
X_norm = (X_train - mu)/sigma

fig,ax=plt.subplots(1, 3, figsize=(12, 3))
ax[0].scatter(X_train[:,0], X_train[:,3])
ax[0].set_xlabel(X_features[0]); ax[0].set_ylabel(X_features[3]);
ax[0].set_title("unnormalized")
ax[0].axis('equal')

ax[1].scatter(X_mean[:,0], X_mean[:,3])
ax[1].set_xlabel(X_features[0]); ax[0].set_ylabel(X_features[3]);
```

Loading [MathJax]/extensions/Safe.js

```
ax[1].set_title(r"X - $\mu$")
ax[1].axis('equal')

ax[2].scatter(X_norm[:,0], X_norm[:,3])
ax[2].set_xlabel(X_features[0]); ax[0].set_ylabel(X_features[3]);
ax[2].set_title(r"Z-score normalized")
ax[2].axis('equal')
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
fig.suptitle("distribution of features before, during, after normalization")
plt.show()
```



The plot above shows the relationship between two of the training set parameters, "age" and "size(sqft)". *These are plotted with equal scale*.

- Left: Unnormalized: The range of values or the variance of the 'size(sqft)' feature is much larger than that of age
- Middle: The first step removes the mean or average value from each feature. This leaves features that are centered around zero. It's difficult to see the difference for the 'age' feature, but 'size(sqft)' is clearly around zero.
- Right: The second step divides by the standard deviation. This leaves both features centered at zero with a similar scale.

Let's normalize the data and compare it to the original data.

In [109…

```
# normalize the original features
X_norm, X_mu, X_sigma = zscore_normalize_features(X_train)
print(f"X_mu = {X_mu}, \nX_sigma = {X_sigma}")
print(f"Peak to Peak range by column in Raw        X:{np.ptp(X_train,axis=0)}")
print(f"Peak to Peak range by column in Normalized X:{np.ptp(X_norm,axis=0)}")
```

```
X_mu = [1.42e+03 2.72e+00 1.38e+00 3.84e+01],
X_sigma = [411.62   0.65   0.49  25.78]
Peak to Peak range by column in Raw        X:[2.41e+03 4.00e+00 1.00e+00 9.50e+01]
Peak to Peak range by column in Normalized X:[5.85 6.14 2.06 3.69]
```

The peak to peak range of each column is reduced from a factor of thousands to a factor of 2-3 by normalization.

The term "peak to peak" refers to the difference between the maximum and minimum values of a dataset or a specific variable/column.

In the given code, the lines np.ptp(X_train, axis=0) and np.ptp(X_norm, axis=0) are calculating the peak-to-peak range for each column in the original and normalized features, respectively.

- np.ptp(X_train, axis=0) calculates the peak-to-peak range for each column in the X_train dataset before normalization.

Loading [MathJax]/extensions/Safe.js

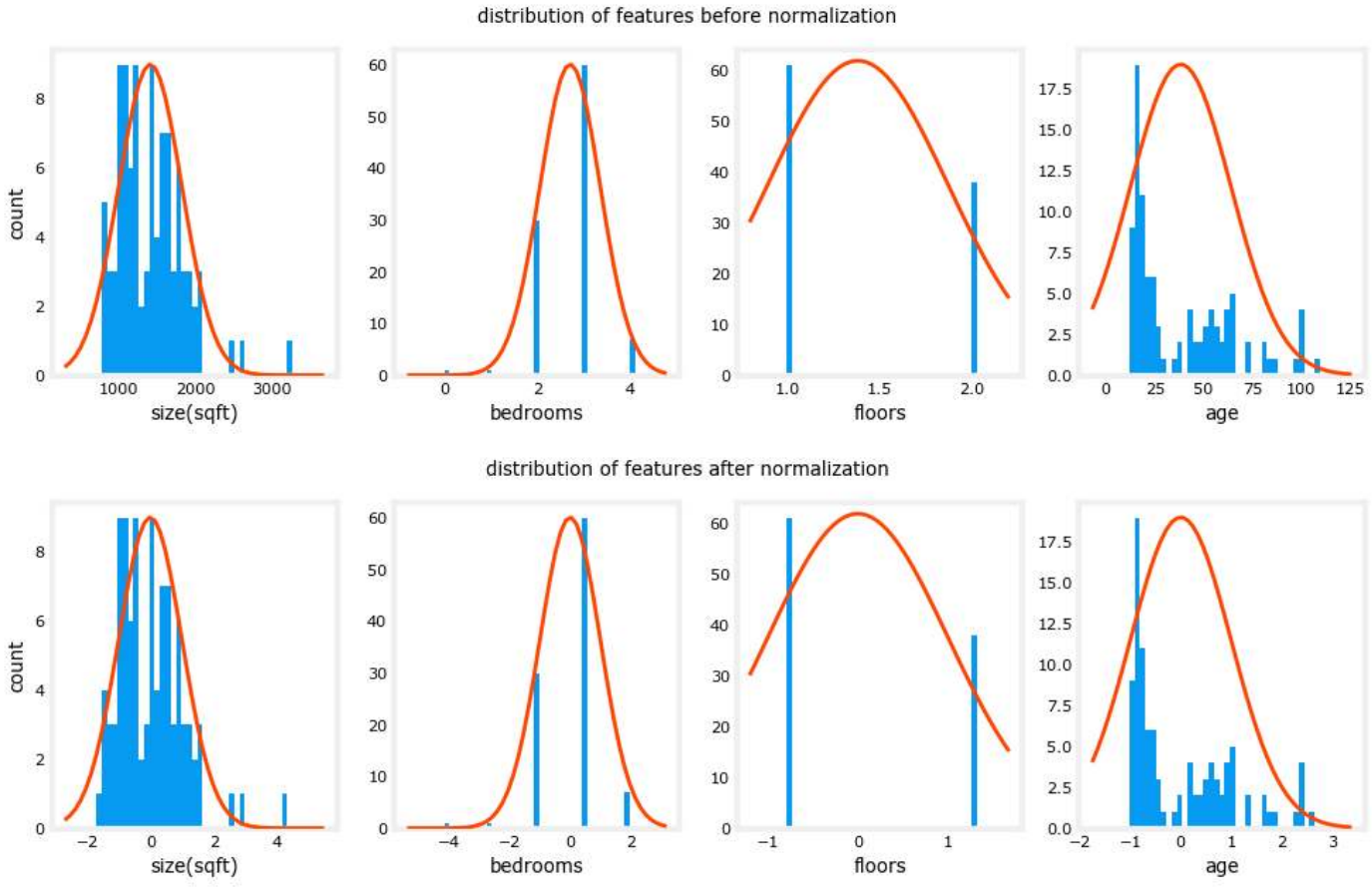
- np.ptp(X_norm, axis=0) calculates the peak-to-peak range for each column in the X_norm dataset after normalization.

The resulting values will provide information about the range or spread of the data in each column. It indicates the difference between the maximum and minimum values in each column and helps to understand the variability or amplitude of the data.

By comparing the peak-to-peak ranges before and after normalization, you can observe the impact of the normalization process on the data and see if it has compressed or stretched the data range in each column.

In [110…

```
fig,ax=plt.subplots(1, 4, figsize=(12, 3))
for i in range(len(ax)):
    norm_plot(ax[i],X_train[:,i],)
    ax[i].set_xlabel(X_features[i])
ax[0].set_ylabel("count");
fig.suptitle("distribution of features before normalization")
plt.show()
fig,ax=plt.subplots(1,4,figsize=(12,3))
for i in range(len(ax)):
    norm_plot(ax[i],X_norm[:,i],)
    ax[i].set_xlabel(X_features[i])
ax[0].set_ylabel("count");
fig.suptitle("distribution of features after normalization")

plt.show()
```



Notice, above, the range of the normalized data (x-axis) is centered around zero and roughly +/- 2. Most importantly, the range is similar for each feature.

Let's re-run our gradient descent algorithm with normalized data. Note the **vastly larger value of alpha**. This will speed up gradient descent.

Loading [MathJax]/extensions/Safe.js

In [111…
```
w_norm, b_norm, hist = run_gradient_descent(X_norm, y_train, 1000, 1.0e-1, )
```

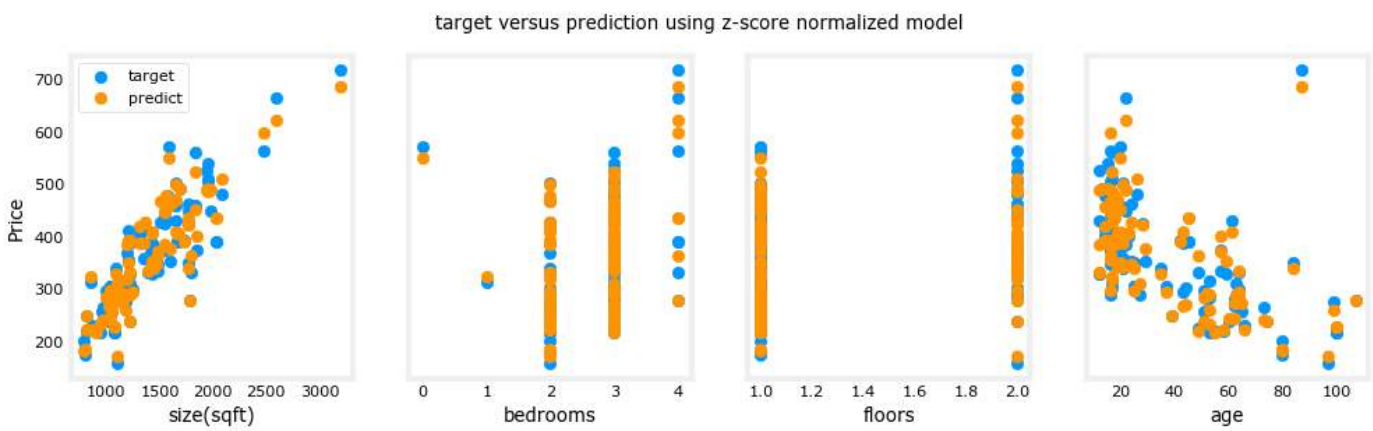
```
Iteration Cost          w0       w1       w2       w3       b       djdw0    djdw1    dj
dw2    djdw3    djdb
----------------------|--------|--------|--------|--------|--------|--------|--------|---
-----|--------|--------|
        0 5.76170e+04  8.9e+00  3.0e+00  3.3e+00 -6.0e+00  3.6e+01 -8.9e+01 -3.0e+01 -3.
3e+01  6.0e+01 -3.6e+02
      100 2.21086e+02  1.1e+02 -2.0e+01 -3.1e+01 -3.8e+01  3.6e+02 -9.2e-01  4.5e-01  5.
3e-01 -1.7e-01 -9.6e-03
      200 2.19209e+02  1.1e+02 -2.1e+01 -3.3e+01 -3.8e+01  3.6e+02 -3.0e-02  1.5e-02  1.
7e-02 -6.0e-03 -2.6e-07
      300 2.19207e+02  1.1e+02 -2.1e+01 -3.3e+01 -3.8e+01  3.6e+02 -1.0e-03  5.1e-04  5.
7e-04 -2.0e-04 -6.9e-12
      400 2.19207e+02  1.1e+02 -2.1e+01 -3.3e+01 -3.8e+01  3.6e+02 -3.4e-05  1.7e-05  1.
9e-05 -6.6e-06 -2.7e-13
      500 2.19207e+02  1.1e+02 -2.1e+01 -3.3e+01 -3.8e+01  3.6e+02 -1.1e-06  5.6e-07  6.
2e-07 -2.2e-07 -2.7e-13
      600 2.19207e+02  1.1e+02 -2.1e+01 -3.3e+01 -3.8e+01  3.6e+02 -3.7e-08  1.9e-08  2.
1e-08 -7.3e-09 -2.6e-13
      700 2.19207e+02  1.1e+02 -2.1e+01 -3.3e+01 -3.8e+01  3.6e+02 -1.2e-09  6.2e-10  6.
9e-10 -2.4e-10 -2.6e-13
      800 2.19207e+02  1.1e+02 -2.1e+01 -3.3e+01 -3.8e+01  3.6e+02 -4.1e-11  2.1e-11  2.
3e-11 -8.1e-12 -2.6e-13
      900 2.19207e+02  1.1e+02 -2.1e+01 -3.3e+01 -3.8e+01  3.6e+02 -1.4e-12  6.9e-13  7.
7e-13 -2.7e-13 -2.6e-13
w,b found by gradient descent: w: [110.56 -21.27 -32.71 -37.97], b: 363.16
```

The scaled features get very accurate results much, much faster!. Notice the gradient of each parameter is tiny by the end of this fairly short run. A learning rate of 0.1 is a good start for regression with normalized features. Let's plot our predictions versus the target values.

Note, the prediction is made using the normalized feature while the plot is shown using the original feature values.

In [112…
```
#predict target using normalized features
m = X_norm.shape[0]
yp = np.zeros(m)
for i in range(m):
    yp[i] = np.dot(X_norm[i], w_norm) + b_norm

    # plot predictions and targets versus original features
fig,ax=plt.subplots(1,4,figsize=(12, 3),sharey=True)
for i in range(len(ax)):
    ax[i].scatter(X_train[:,i],y_train, label = 'target')
    ax[i].set_xlabel(X_features[i])
    ax[i].scatter(X_train[:,i],yp,color=dlc["dlorange"], label = 'predict')
ax[0].set_ylabel("Price"); ax[0].legend();
fig.suptitle("target versus prediction using z-score normalized model")
plt.show()
```

Loading [MathJax]/extensions/Safe.js



The results look good. A few points to note:

- with multiple features, we can no longer have a single plot showing results versus features.
- when generating the plot, the normalized features were used. Any predictions using the parameters learned from a normalized training set must also be normalized.

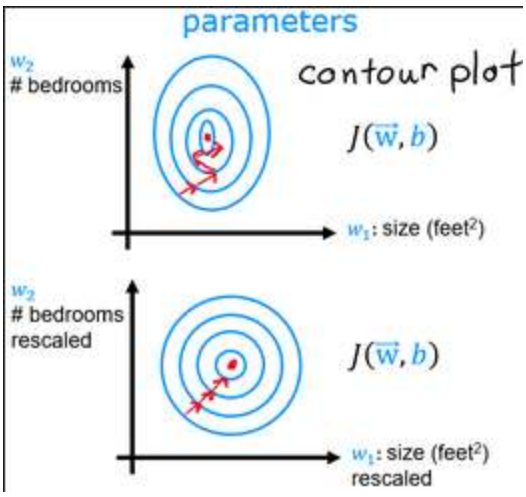
Prediction The point of generating our model is to use it to predict housing prices that are not in the data set. Let's predict the price of a house with 1200 sqft, 3 bedrooms, 1 floor, 40 years old. Recall, that you must normalize the data with the mean and standard deviation derived when the training data was normalized.

In [113…

```
# First, normalize out example.
x_house = np.array([1200, 3, 1, 40])
x_house_norm = (x_house - X_mu) / X_sigma
print(x_house_norm)
x_house_predict = np.dot(x_house_norm, w_norm) + b_norm
print(f" predicted price of a house with 1200 sqft, 3 bedrooms, 1 floor, 40 years old =
```

```
[-0.53  0.43 -0.79  0.06]
 predicted price of a house with 1200 sqft, 3 bedrooms, 1 floor, 40 years old = $318709
```

So, we can see that by using normalize we have an accurate predictions



Cost Contours

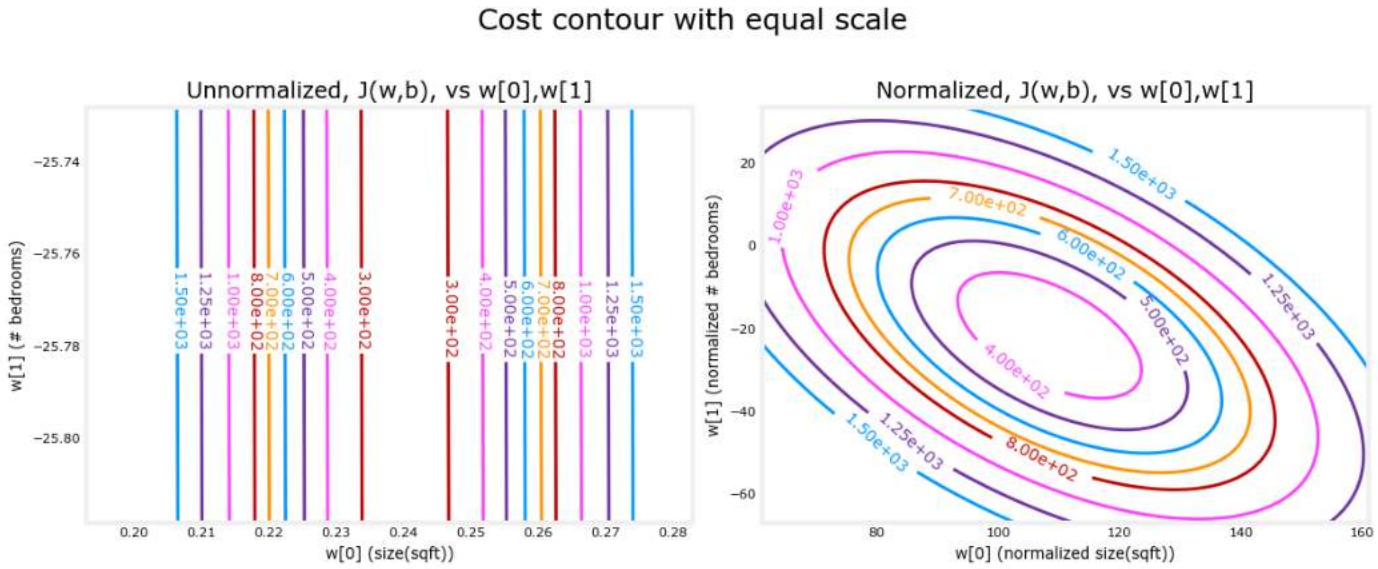
Another way to view feature scaling is in terms of the cost contours. When feature scales do not match, the plot of cost versus parameters in a contour plot is asymmetric.

In the plot below, the scale of the parameters is matched. The left plot is the cost contour plot of w[0], the square feet versus w[1], the number of bedrooms before normalizing the features. The plot is so asymmetric, the curves completing the contours are not visible. In contrast, when the features are

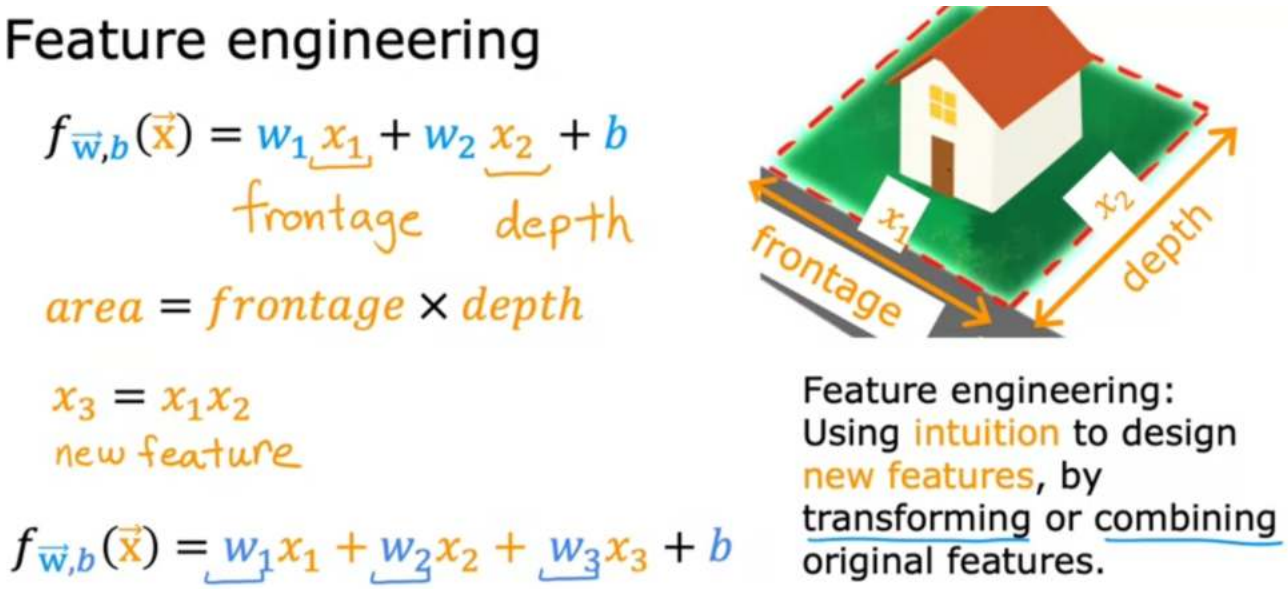
Loading [MathJax]/extensions/Safe.js

normalized, the cost contour is much more symmetric. The result is that updates to parameters during gradient descent can make equal progress for each parameter.

In [114…
```
plt_equal_scale(X_train, X_norm, y_train)
```



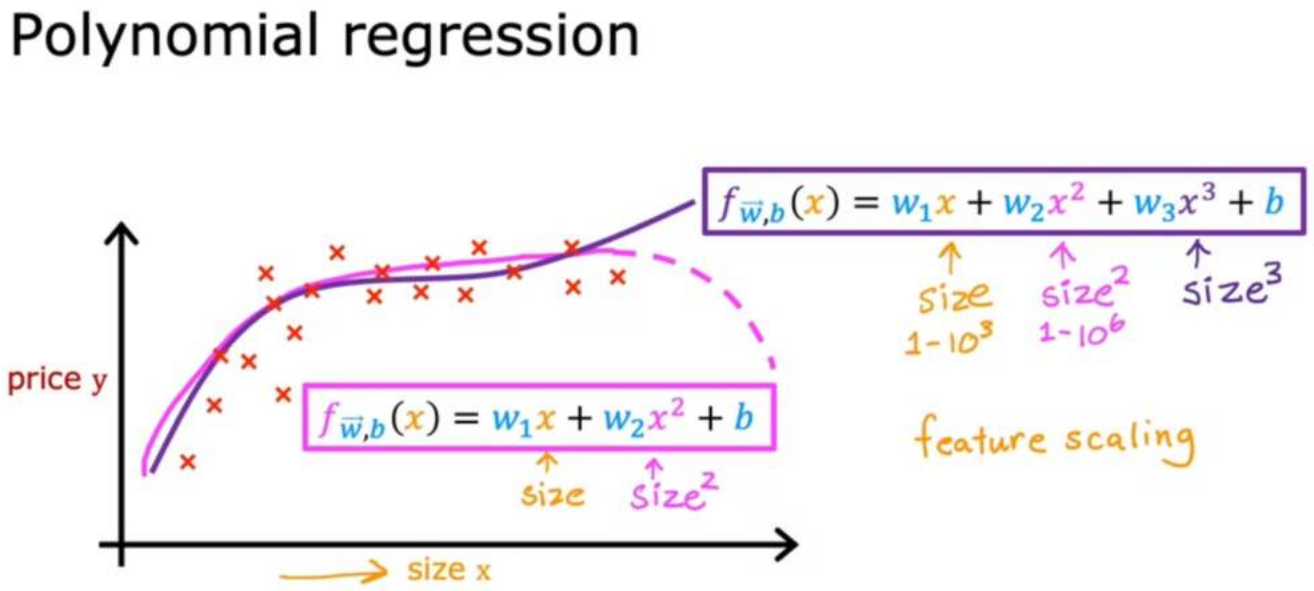
Feature engineering



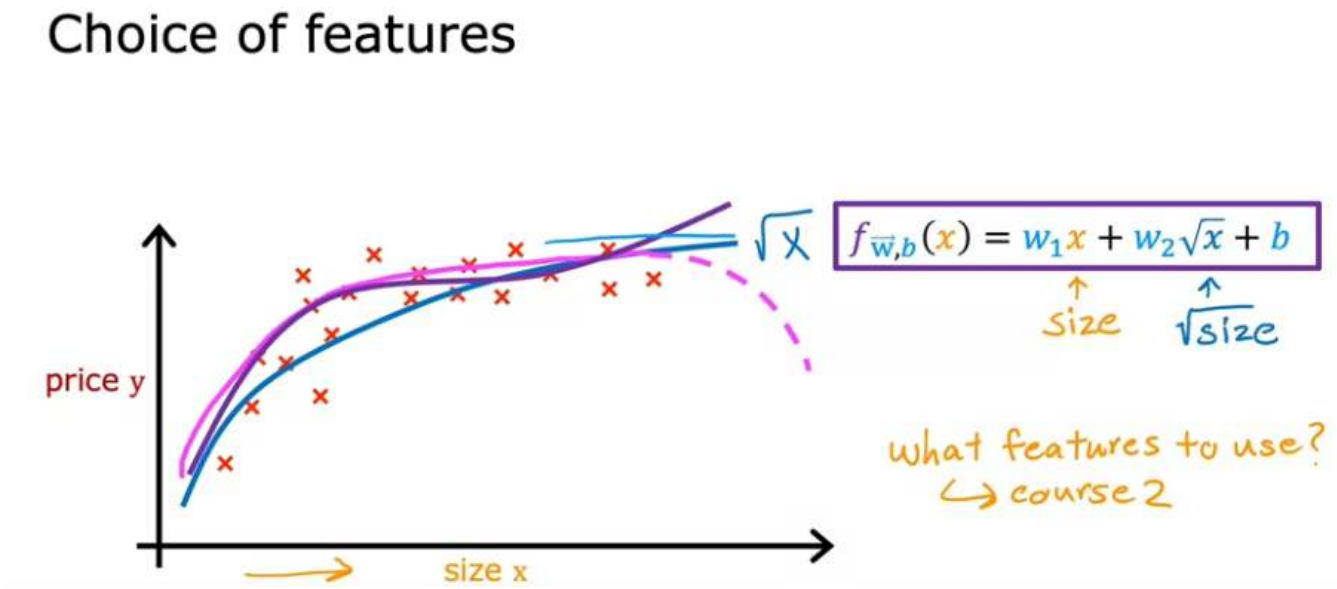
Polynomial regression

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

Loading [MathJax]/extensions/Safe.js



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.



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

lab: Feature engineering and Polynomial regression

Goals

In this lab you will:

- explore feature engineering and polynomial regression which allows you to use the machinery of linear regression to fit very complicated, even very non-linear functions.

In [115…

```
import numpy as np
import matplotlib.pyplot as plt
```

Loading [MathJax]/extensions/Safe.js

```
from lab_utils_multi import zscore_normalize_features, run_gradient_descent_feng
np.set_printoptions(precision=2)  # reduced display precision on numpy arrays
```

Feature Engineering and Polynomial Regression Overview

Out of the box, linear regression provides a means of building models of the form: $$f_{\mathbf{w},b} = w_0x_0 + w_1x_1+ ... + w_{n-1}x_{n-1} + b \tag{1}$$ What if your features/data are non-linear or are combinations of features? For example, Housing prices do not tend to be linear with living area but penalize very small or very large houses resulting in the curves shown in the graphic above. How can we use the machinery of linear regression to fit this curve? Recall, the 'machinery' we have is the ability to modify the parameters $\mathbf{w}$, $\mathbf{b}$ in (1) to 'fit' the equation to the training data. However, no amount of adjusting of $\mathbf{w}$,$\mathbf{b}$ in (1) will achieve a fit to a non-linear curve.

Polynomial Features

Above we were considering a scenario where the data was non-linear. Let's try using what we know so far to fit a non-linear curve. We'll start with a simple quadratic: $y = 1+x^2$

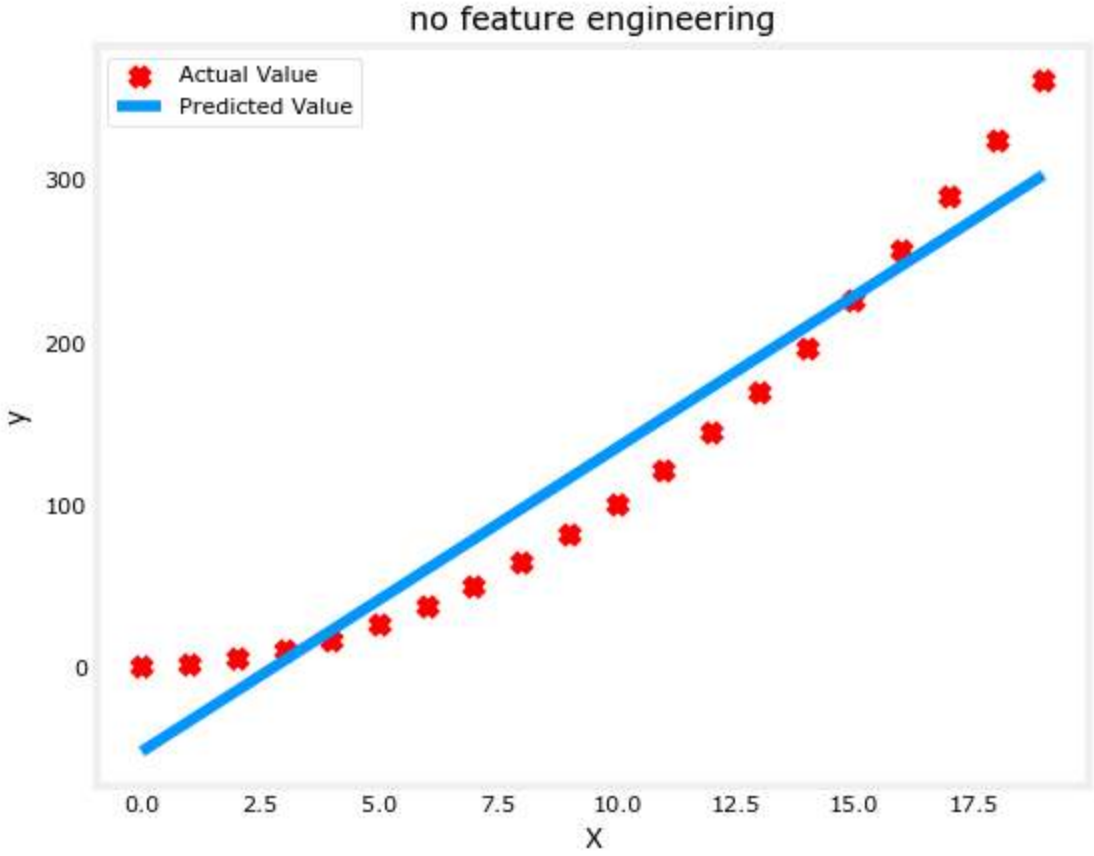
You're familiar with all the routines we're using. They are available in the lab*utils.py file for review. We'll use [`np.c..]`* which is a NumPy routine to concatenate along the column boundary.

In [116…

```
# create target data
x = np.arange(0, 20, 1)
y = 1 + x**2
X = x.reshape(-1, 1)

model_w,model_b = run_gradient_descent_feng(X,y,iterations=1000, alpha = 1e-2)

plt.scatter(x, y, marker='x', c='r', label="Actual Value"); plt.title("no feature engine
plt.plot(x,X@model_w + model_b, label="Predicted Value");  plt.xlabel("X"); plt.ylabel("
```

```
Iteration         0, Cost: 1.65756e+03
Iteration       100, Cost: 6.94549e+02
Iteration       200, Cost: 5.88475e+02
Iteration       300, Cost: 5.26414e+02
Iteration       400, Cost: 4.90103e+02
Iteration       500, Cost: 4.68858e+02
Iteration       600, Cost: 4.56428e+02
Iteration       700, Cost: 4.49155e+02
Iteration       800, Cost: 4.44900e+02
Iteration       900, Cost: 4.42411e+02
w,b found by gradient descent: w: [18.7], b: -52.0834
```

Loading [MathJax]/extensions/Safe.js



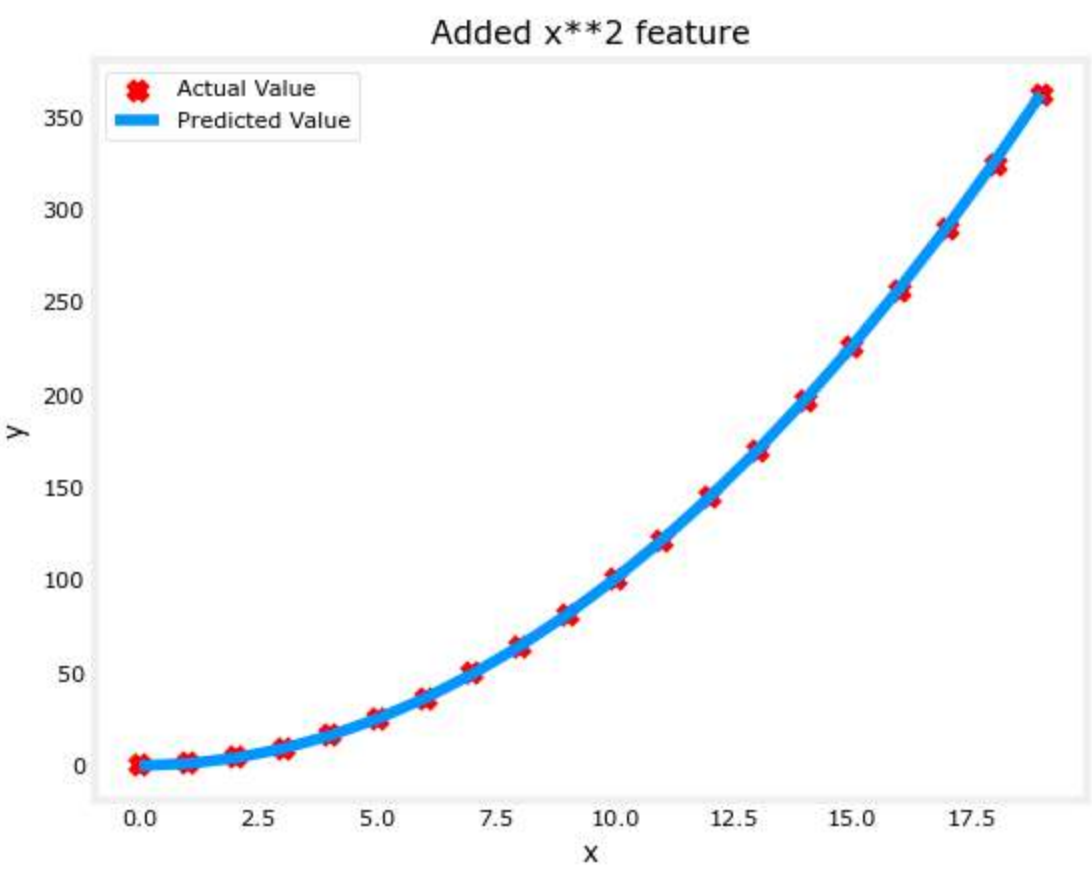
Well, as expected, not a great fit. What is needed is something like $y= w_0x_0^2 + b$, or a **polynomial feature**. To accomplish this, you can modify the *input data* to *engineer* the needed features. If you swap the original data with a version that squares the x value, then you can achieve $y= w_0x_0^2 + b$. Let's try it. Swap `X` for `X**2` below:

In [117…
```
# create target data
x = np.arange(0, 20, 1)
y = 1 + x**2

# Engineer features
X = x**2      #<-- added engineered feature
```

In [118…
```
X = X.reshape(-1, 1)  #X should be a 2-D Matrix
model_w,model_b = run_gradient_descent_feng(X, y, iterations=10000, alpha = 1e-5)

plt.scatter(x, y, marker='x', c='r', label="Actual Value"); plt.title("Added x**2 featur
plt.plot(x, np.dot(X,model_w) + model_b, label="Predicted Value"); plt.xlabel("x"); plt.
```

```
Iteration         0, Cost: 7.32922e+03
Iteration      1000, Cost: 2.24844e-01
Iteration      2000, Cost: 2.22795e-01
Iteration      3000, Cost: 2.20764e-01
Iteration      4000, Cost: 2.18752e-01
Iteration      5000, Cost: 2.16758e-01
Iteration      6000, Cost: 2.14782e-01
Iteration      7000, Cost: 2.12824e-01
Iteration      8000, Cost: 2.10884e-01
Iteration      9000, Cost: 2.08962e-01
w,b found by gradient descent: w: [1.], b: 0.0490
```

Loading [MathJax]/extensions/Safe.js



Great! near perfect fit. Notice the values of $\mathbf{w}$ and b printed right above the graph: `w,b found by gradient descent: w: [1.], b: 0.0490`. Gradient descent modified our initial values of $\mathbf{w},b $ to be (1.0,0.049) or a model of $y=1*x_0^2+0.049$, very close to our target of $y=1*x_0^2+1$. If you ran it longer, it could be a better match.

Selecting Features

Above, we knew that an x^2 term was required. It may not always be obvious which features are required. One could add a variety of potential features to try and find the most useful. For example, what if we had instead tried : $y=w_0x_0 + w_1x_1^2 + w_2x_2^3+b$?

Run the next cells.

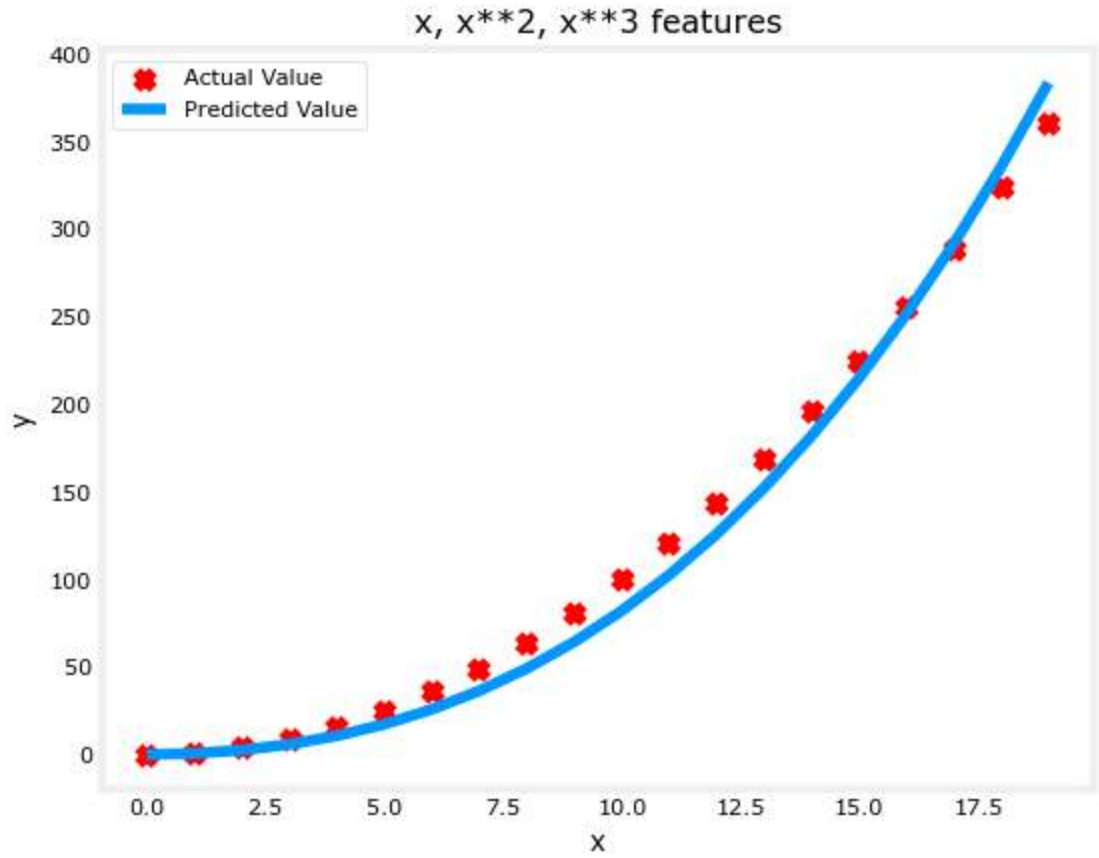
In [119…
```
# create target data
x = np.arange(0, 20, 1)
y = x**2

# engineer features .
X = np.c_[x, x**2, x**3]   #<-- added engineered feature
```

The **.c**_ function is a convenient way to concatenate arrays along the second axis, providing a concise and readable syntax for creating multi-dimensional arrays.

In [120…
```
model_w,model_b = run_gradient_descent_feng(X, y, iterations=10000, alpha=1e-7)

plt.scatter(x, y, marker='x', c='r', label="Actual Value"); plt.title("x, x**2, x**3 fea
plt.plot(x, X@model_w + model_b, label="Predicted Value"); plt.xlabel("x"); plt.ylabel("
```

Loading [MathJax]/extensions/Safe.js

```
Iteration         0, Cost: 1.14029e+03
Iteration      1000, Cost: 3.28539e+02
Iteration      2000, Cost: 2.80443e+02
Iteration      3000, Cost: 2.39389e+02
Iteration      4000, Cost: 2.04344e+02
Iteration      5000, Cost: 1.74430e+02
Iteration      6000, Cost: 1.48896e+02
Iteration      7000, Cost: 1.27100e+02
Iteration      8000, Cost: 1.08495e+02
Iteration      9000, Cost: 9.26132e+01
w,b found by gradient descent: w: [0.08 0.54 0.03], b: 0.0106
```



Note the value of $\mathbf{w}$, `[0.08 0.54 0.03]` and b is `0.0106`.This implies the model after fitting/training is: $$ 0.08x + 0.54x^2 + 0.03x^3 + 0.0106 $$ Gradient descent has emphasized the data that is the best fit to the x^2 data by increasing the w_1 term relative to the others. If you were to run for a very long time, it would continue to reduce the impact of the other terms.

> Gradient descent is picking the 'correct' features for us by emphasizing its associated parameter

Let's review this idea:

- Intially, the features were re-scaled so they are comparable to each other
- less weight value implies less important/correct feature, and in extreme, when the weight becomes zero or very close to zero, the associated feature is not useful in fitting the model to the data.
- above, after fitting, the weight associated with the x^2 feature is much larger than the weights for x or x^3 as it is the most useful in fitting the data.

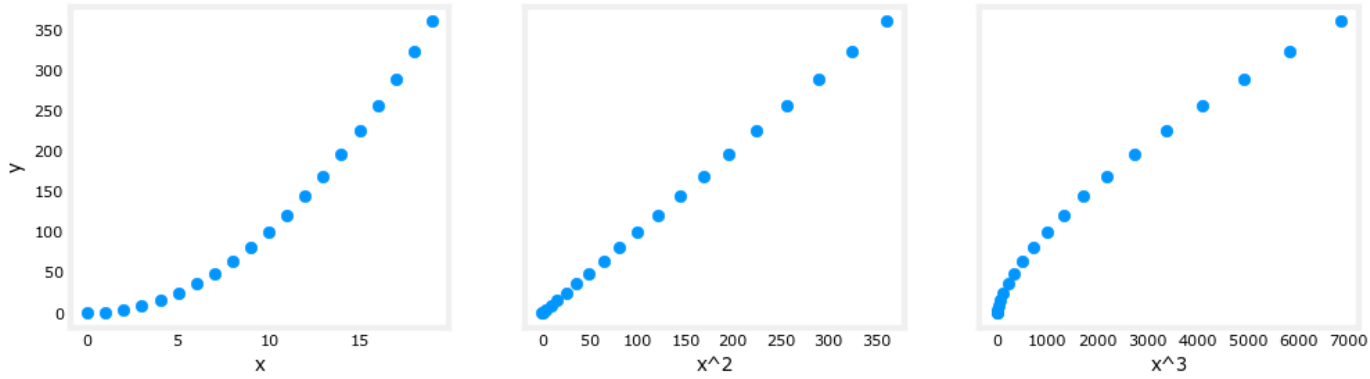
An Alternate View

Loading [MathJax]/extensions/Safe.js

Above, polynomial features were chosen based on how well they matched the target data. Another way to think about this is to note that we are still using linear regression once we have created new features. Given that, the best features will be linear relative to the target. This is best understood with an example.

In [121…
```
# create target data
x = np.arange(0, 20, 1)
y = x**2

# engineer features .
X = np.c_[x, x**2, x**3]   #<-- added engineered feature
X_features = ['x','x^2','x^3']
```

In [122…
```
fig,ax=plt.subplots(1, 3, figsize=(12, 3), sharey=True)
for i in range(len(ax)):
    ax[i].scatter(X[:,i],y)
    ax[i].set_xlabel(X_features[i])
ax[0].set_ylabel("y")
plt.show()
```



Above, it is clear that the x^2 feature mapped against the target value y is linear. Linear regression can then easily generate a model using that feature.

Scaling features

As described in the last lab, if the data set has features with significantly different scales, one should apply feature scaling to speed gradient descent. In the example above, there is x, x^2 and x^3 which will naturally have very different scales. Let's apply Z-score normalization to our example.

In [123…
```
# create target data
x = np.arange(0,20,1)
X = np.c_[x, x**2, x**3]
print(f"Peak to Peak range by column in Raw        X:{np.ptp(X,axis=0)}")

# add mean_normalization
X = zscore_normalize_features(X)
print(f"Peak to Peak range by column in Normalized X:{np.ptp(X,axis=0)}")
```

```
Peak to Peak range by column in Raw        X:[  19  361 6859]
Peak to Peak range by column in Normalized X:[3.3  3.18 3.28]
```

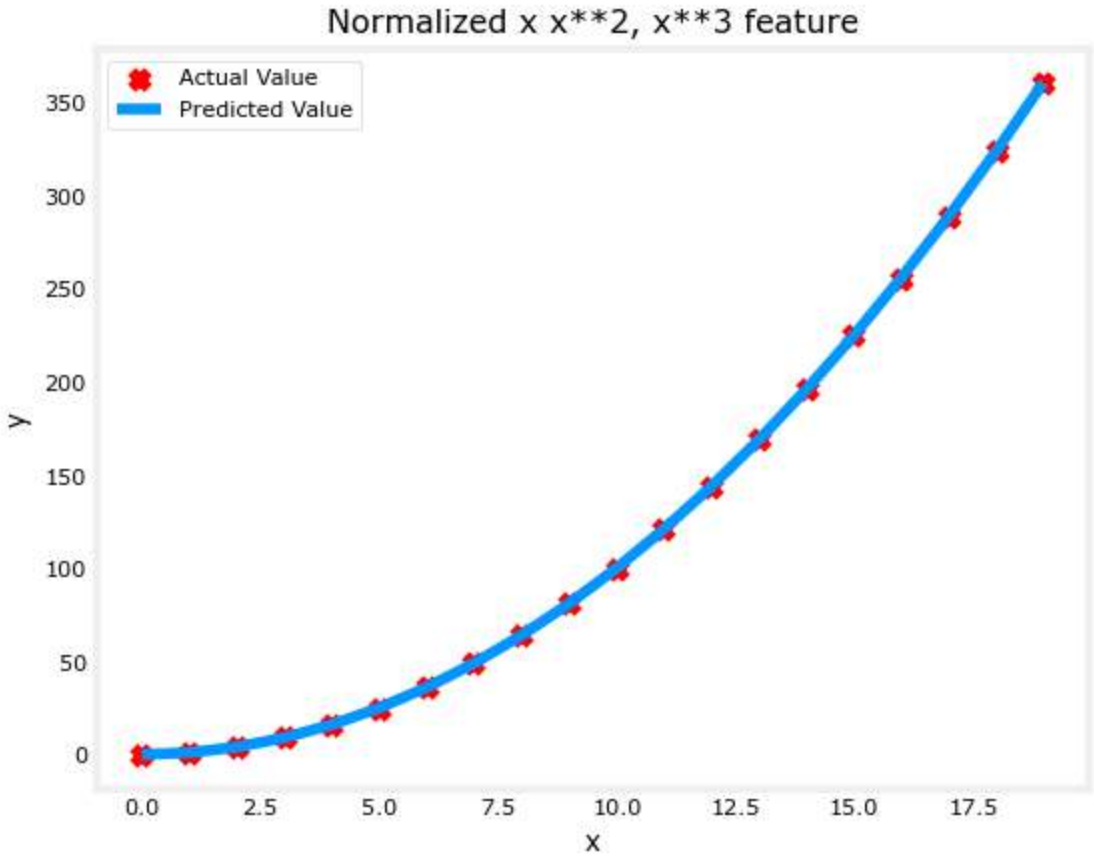
Now we can try again with a more aggressive value of alpha:

In [124…
```
x = np.arange(0,20,1)
y = x**2

X = np.c_[x, x**2, x**3]
```

Loading [MathJax]/extensions/Safe.js

```
X = zscore_normalize_features(X)

model_w, model_b = run_gradient_descent_feng(X, y, iterations=100000, alpha=1e-1)

plt.scatter(x, y, marker='x', c='r', label="Actual Value"); plt.title("Normalized x x**2
plt.plot(x,X@model_w + model_b, label="Predicted Value"); plt.xlabel("x"); plt.ylabel("y
```

```
Iteration         0, Cost: 9.42147e+03
Iteration     10000, Cost: 3.90938e-01
Iteration     20000, Cost: 2.78389e-02
Iteration     30000, Cost: 1.98242e-03
Iteration     40000, Cost: 1.41169e-04
Iteration     50000, Cost: 1.00527e-05
Iteration     60000, Cost: 7.15855e-07
Iteration     70000, Cost: 5.09763e-08
Iteration     80000, Cost: 3.63004e-09
Iteration     90000, Cost: 2.58497e-10
w,b found by gradient descent: w: [5.27e-05 1.13e+02 8.43e-05], b: 123.5000
```



Feature scaling allows this to converge much faster.

Note again the values of $\mathbf{w}$. The w_1 term, which is the x^2 term is the most emphasized.

Gradient descent has all but eliminated the x^3 term.

Complex Functions

With feature engineering, even quite complex functions can be modeled:

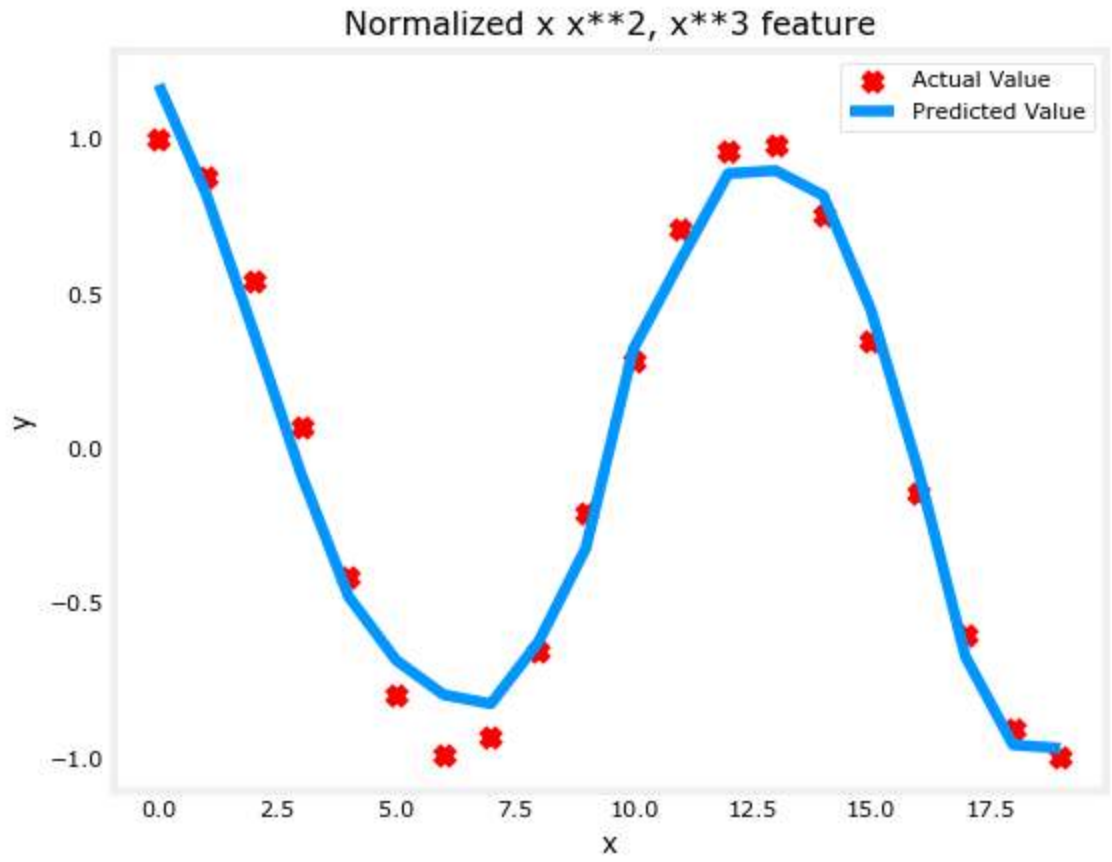
In [125…

```
x = np.arange(0,20,1)
y = np.cos(x/2)

X = np.c_[x, x**2, x**3,x**4, x**5, x**6, x**7, x**8, x**9, x**10, x**11, x**12, x**13]
X = zscore_normalize_features(X)

model_w,model_b = run_gradient_descent_feng(X, y, iterations=1000000, alpha = 1e-1)
```

Loading [MathJax]/extensions/Safe.js

```
plt.scatter(x, y, marker='x', c='r', label="Actual Value"); plt.title("Normalized x x**2
plt.plot(x,X@model_w + model_b, label="Predicted Value"); plt.xlabel("x"); plt.ylabel("y
```

```
Iteration         0, Cost: 2.24887e-01
Iteration    100000, Cost: 2.31061e-02
Iteration    200000, Cost: 1.83619e-02
Iteration    300000, Cost: 1.47950e-02
Iteration    400000, Cost: 1.21114e-02
Iteration    500000, Cost: 1.00914e-02
Iteration    600000, Cost: 8.57025e-03
Iteration    700000, Cost: 7.42385e-03
Iteration    800000, Cost: 6.55908e-03
Iteration    900000, Cost: 5.90594e-03
w,b found by gradient descent: w: [-1.61e+00 -1.01e+01  3.00e+01 -6.92e-01 -2.37e+01 -1.
51e+01  2.09e+01
 -2.29e-03 -4.69e-03  5.51e-02  1.07e-01 -2.53e-02  6.49e-02], b: -0.0073
```



In this lab you:

- learned how linear regression can model complex, even highly non-linear functions using feature engineering
- recognized that it is important to apply feature scaling when doing feature engineering

lab: Linear regression with scikit-learn

There is an open-source, commercially usable machine learning toolkit called scikit-learn. This toolkit contains implementations of many of the algorithms that you will work with in this course.

Goals

In this lab you will:

Loading [MathJax]/extensions/Safe.js
-learn to implement linear regression using Gradient Descent

In [126…
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import SGDRegressor
from sklearn.preprocessing import StandardScaler
from lab_utils_multi import  load_house_data
from lab_utils_common import dlc
np.set_printoptions(precision=2)
plt.style.use('deeplearning.mplstyle')
```

Gradient Descent

Scikit-learn has a gradient descent regression model sklearn.linear_model.SGDRegressor. Like your previous implementation of gradient descent, this model performs best with normalized inputs. sklearn.preprocessing.StandardScaler will perform z-score normalization as in a previous lab. Here it is referred to as 'standard score'.

Load the data set

In [127…
```
X_train, y_train = load_house_data()
X_features = ['size(sqft)','bedrooms','floors','age']
```

Scale/normalize the training data

In [128…
```
scaler = StandardScaler()
X_norm = scaler.fit_transform(X_train)
print(f"Peak to Peak range by column in Raw        X:{np.ptp(X_train,axis=0)}")
print(f"Peak to Peak range by column in Normalized X:{np.ptp(X_norm,axis=0)}")
```

```
Peak to Peak range by column in Raw        X:[2.41e+03 4.00e+00 1.00e+00 9.50e+01]
Peak to Peak range by column in Normalized X:[5.85 6.14 2.06 3.69]
```

Create and fit the regression model

In [129…
```
sgdr = SGDRegressor(max_iter=1000)
sgdr.fit(X_norm, y_train)
print(sgdr)
print(f"number of iterations completed: {sgdr.n_iter_}, number of weight updates: {sgdr.
```

```
SGDRegressor()
number of iterations completed: 116, number of weight updates: 11485.0
```

View parameters

Note, the parameters are associated with the *normalized* input data. The fit parameters are very close to those found in the previous lab with this data.

In [130…
```
b_norm = sgdr.intercept_
w_norm = sgdr.coef_
print(f"model parameters:                   w: {w_norm}, b:{b_norm}")
print( "model parameters from previous lab: w: [110.56 -21.27 -32.71 -37.97], b: 363.16"
```

```
model parameters:                   w: [110.   -20.96 -32.37 -38.08], b:[363.17]
model parameters from previous lab: w: [110.56 -21.27 -32.71 -37.97], b: 363.16
```

Loading [MathJax]/extensions/Safe.js

Make predictions

Predict the targets of the training data. Use both the `predict` routine and compute using w and b.

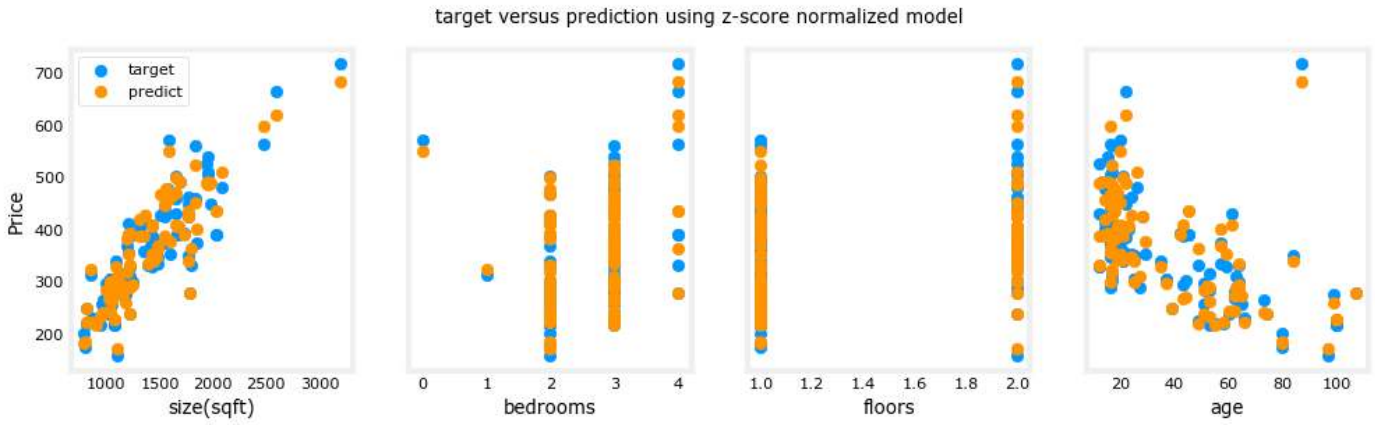
In [131…
```
# make a prediction using sgdr.predict()
y_pred_sgd = sgdr.predict(X_norm)
# make a prediction using w,b.
y_pred = np.dot(X_norm, w_norm) + b_norm
print(f"prediction using np.dot() and sgdr.predict match: {(y_pred == y_pred_sgd).all()}")

print(f"Prediction on training set:\n{y_pred[:4]}" )
print(f"Target values \n{y_train[:4]}")
```

```
prediction using np.dot() and sgdr.predict match: True
Prediction on training set:
[295.19 485.93 389.67 492.09]
Target values
[300.  509.8 394.  540. ]
```

Plot Results

Let's plot the predictions versus the target values.

In [132…
```
# plot predictions and targets vs original features
fig,ax=plt.subplots(1,4,figsize=(12,3),sharey=True)
for i in range(len(ax)):
    ax[i].scatter(X_train[:,i],y_train, label = 'target')
    ax[i].set_xlabel(X_features[i])
    ax[i].scatter(X_train[:,i],y_pred,color=dlc["dlorange"], label = 'predict')
ax[0].set_ylabel("Price"); ax[0].legend();
fig.suptitle("target versus prediction using z-score normalized model")
plt.show()
```



Programming Assignment: lab: Linear regression

Practice Lab: Linear Regression

Welcome to your first practice lab! In this lab, you will implement linear regression with one variable to predict profits for a restaurant franchise.

Outline

Loading [MathJax]/extensions/Safe.js

1 - Packages

First, let's run the cell below to import all the packages that you will need during this assignment.

- numpy is the fundamental package for working with matrices in Python.
- matplotlib is a famous library to plot graphs in Python.
- `utils.py` contains helper functions for this assignment. You do not need to modify code in this file.

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
from utils import *
import copy
import math
%matplotlib inline
```

2 - Problem Statement

Suppose you are the CEO of a restaurant franchise and are considering different cities for opening a new outlet.

- You would like to expand your business to cities that may give your restaurant higher profits.
- The chain already has restaurants in various cities and you have data for profits and populations from the cities.
- You also have data on cities that are candidates for a new restaurant.
  - For these cities, you have the city population.

Can you use the data to help you identify which cities may potentially give your business higher profits?

3 - Dataset

You will start by loading the dataset for this task.

- The `load_data()` function shown below loads the data into variables `x_train` and `y_train`
  - `x_train` is the population of a city
  - `y_train` is the profit of a restaurant in that city. A negative value for profit indicates a loss.
  - Both `X_train` and `y_train` are numpy arrays.

Loading [MathJax]/extensions/Safe.js

In [5]:
```
# load the dataset
x_train, y_train = load_data()
```

In [9]:
```
# print x_train
print('Type of x_train:' ,type(x_train))
print('First five elements of x_train are:\n', x_train[:5])
```

```
Type of x_train: <class 'numpy.ndarray'>
First five elements of x_train are:
 [6.1101 5.5277 8.5186 7.0032 5.8598]
```

`x_train` is a numpy array that contains decimal values that are all greater than zero.

- These values represent the city population times 10,000
- For example, 6.1101 means that the population for that city is 61,101

Now, let's print `y_train`

In [10]:
```
# print y_train
print('Type of y_train: ', type(y_train))
print('First five elements of y_train are: \n', y_train[:5])
```

```
Type of y_train:  <class 'numpy.ndarray'>
First five elements of y_train are:
 [17.592   9.1302 13.662  11.854   6.8233]
```

Similarly, `y_train` is a numpy array that has decimal values, some negative, some positive.

- These represent your restaurant's average monthly profits in each city, in units of \$10,000.
  - For example, 17.592 represents \$175,920 in average monthly profits for that city.
  - -2.6807 represents -\$26,807 in average monthly loss for that city.

Check the dimensions of your variables

Another useful way to get familiar with your data is to view its dimensions.

Please print the shape of `x_train` and `y_train` and see how many training examples you have in your dataset.

In [13]:
```
print('x_train shape', x_train.shape)
print('y_train shape', y_train.shape)
print('Number of training examples (m): ', len(x_train))
```

```
x_train shape (97,)
y_train shape (97,)
Number of training examples (m):  97
```

The city population array has 97 data points, and the monthly average profits also has 97 data points. These are NumPy 1D arrays.

Visualize your data

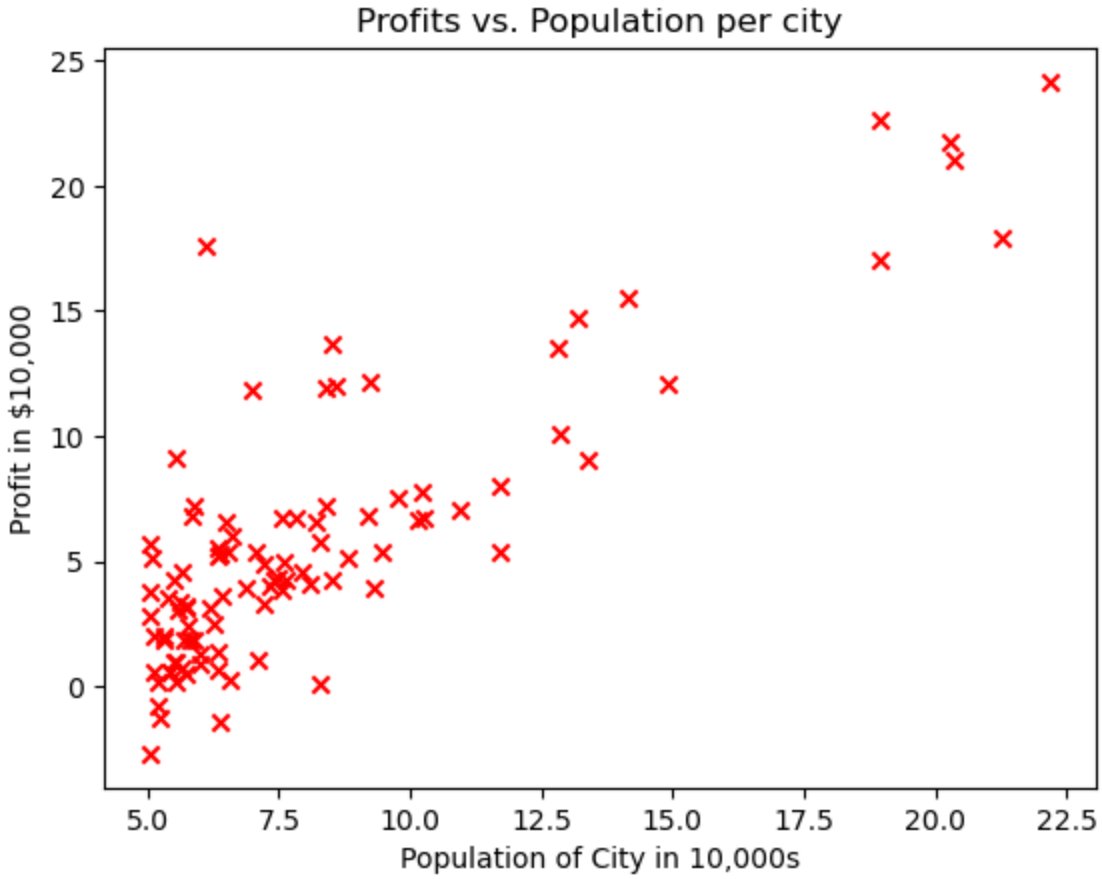
It is often useful to understand the data by visualizing it.

- For this dataset, you can use a scatter plot to visualize the data, since it has only two properties to plot (profit and population).

Loading [MathJax]/extensions/Safe.js

- Many other problems that you will encounter in real life have more than two properties (for example, population, average household income, monthly profits, monthly sales).When you have more than two properties, you can still use a scatter plot to see the relationship between each pair of properties.

In [14]:
```
# Create a scatter plot of the data. To change the markers to red "x",
# we used the 'marker' and 'c' parameters
plt.scatter(x_train, y_train, marker='x', c='r')

# Set the title
plt.title("Profits vs. Population per city")
# Set the y-axis label
plt.ylabel('Profit in $10,000')
# Set the x-axis label
plt.xlabel('Population of City in 10,000s')
plt.show()
```



Your goal is to build a linear regression model to fit this data.

- With this model, you can then input a new city's population, and have the model estimate your restaurant's potential monthly profits for that city.

4 - Refresher on linear regression

In this practice lab, you will fit the linear regression parameters (w,b) to your dataset.

- The model function for linear regression, which is a function that maps from `x` (city population) to `y` (your restaurant's monthly profit for that city) is represented as $$f_{w,b}(x) = wx + b$$

- To train a linear regression model, you want to find the best (w,b) parameters that fit your dataset.

Loading [MathJax]/extensions/Safe.js

    - To compare how one choice of (w,b) is better or worse than another choice, you can evaluate it with a cost function $J(w,b)$
        - J is a function of (w,b). That is, the value of the cost $J(w,b)$ depends on the value of (w,b).
    - The choice of (w,b) that fits your data the best is the one that has the smallest cost $J(w,b)$.

- To find the values (w,b) that gets the smallest possible cost $J(w,b)$, you can use a method called **gradient descent**.
    - With each step of gradient descent, your parameters (w,b) come closer to the optimal values that will achieve the lowest cost $J(w,b)$.

- The trained linear regression model can then take the input feature x (city population) and output a prediction $f_{w,b}(x)$ (predicted monthly profit for a restaurant in that city).

5 - Compute Cost

Gradient descent involves repeated steps to adjust the value of your parameter (w,b) to gradually get a smaller and smaller cost $J(w,b)$.

- At each step of gradient descent, it will be helpful for you to monitor your progress by computing the cost $J(w,b)$ as (w,b) gets updated.
- In this section, you will implement a function to calculate $J(w,b)$ so that you can check the progress of your gradient descent implementation.

Cost function

As you may recall from the lecture, for one variable, the cost function for linear regression $J(w,b)$ is defined as

$$J(w,b) = \frac{1}{2m} \sum\limits_{i = 0}^{m-1} (f_{w,b}(x^{(i)}) - y^{(i)})^2$$

- You can think of $f_{w,b}(x^{(i)})$ as the model's prediction of your restaurant's profit, as opposed to $y^{(i)}$, which is the actual profit that is recorded in the data.
- m is the number of training examples in the dataset

Model prediction

- For linear regression with one variable, the prediction of the model $f_{w,b}$ for an example $x^{(i)}$ is representented as:

$$ f_{w,b}(x^{(i)}) = wx^{(i)} + b$$
This is the equation for a line, with an intercept b and a slope w

Implementation

Please complete the `compute_cost()` function below to compute the cost $J(w,b)$.

Exercise 1

Loading [MathJax]/extensions/Safe.js

Complete the `compute_cost` below to:

- Iterate over the training examples, and for each example, compute:
  - The prediction of the model for that example $$ f_{wb}(x^{(i)}) = wx^{(i)} + b $$
  - The cost for that example $$cost^{(i)} = (f_{wb} - y^{(i)})^2$$
- Return the total cost over all examples $$J(\mathbf{w},b) = \frac{1}{2m} \sum\limits_{i = 0}^{m-1} cost^{(i)}$$
  - Here, m is the number of training examples and $\sum$ is the summation operator

If you get stuck, you can check out the hints presented after the cell below to help you with the implementation.

In [20]:
```
def compute_cost (x_train, y_train, w, b):

    # number of training examples
    m = x_train.shape[0]

    cost = 0

    for i in range(m):
        f_wb = np.dot(x_train[i], w) + b
        cost = cost + (f_wb - y_train[i])**2
    total_cost = cost / (2*m)

    return total_cost
```

check if your implementation was correct by running the following test code:

In [21]:
```
#Compute cost with some initial values for paramaters w, b
initial_w = 2
initial_b = 1

cost = compute_cost(x_train, y_train, initial_w, initial_b)
print(type(cost))
print(f'Cost at initial w: {cost:.3f}')

# Public tests
from public_tests import *
compute_cost_test(compute_cost)
```

```
<class 'numpy.float64'>
Cost at initial w: 75.203
All tests passed!
```

Expected Output:

Cost at initial w: 75.203

6 - Gradient descent

In this section, you will implement the gradient for parameters w, b for linear regression.

As described in the lecture videos, the gradient descent algorithm is:

Loading [MathJax]/extensions/Safe.js

$$\begin{align*}& \text{repeat until convergence:} \; \lbrace \newline \; & \phantom {0000} b := b - \alpha \frac{\partial J(w,b)}{\partial b} \newline \; & \phantom {0000} w := w - \alpha \frac{\partial J(w,b)}{\partial w} \tag{1} \; & \newline & \rbrace\end{align*}$$
where, parameters w, b are both updated simultaniously and where
$$ \frac{\partial J(w,b)}{\partial b} = \frac{1}{m} \sum\limits_{i = 0}^{m-1} (f_{w,b}(x^{(i)}) - y^{(i)}) \tag{2} $$ $$ \frac{\partial J(w,b)}{\partial w} = \frac{1}{m} \sum\limits_{i = 0}^{m-1} (f_{w,b}(x^{(i)}) -y^{(i)})x^{(i)} \tag{3} $$

- m is the number of training examples in the dataset
- $f_{w,b}(x^{(i)})$ is the model's prediction, while $y^{(i)}$, is the target value

You will implement a function called `compute_gradient` which calculates $\frac{\partial J(w)}{\partial w}$, $\frac{\partial J(w)}{\partial b}$

Exercise 2

Please complete the `compute_gradient` function to:

- Iterate over the training examples, and for each example, compute:
  - The prediction of the model for that example $$ f_{wb}(x^{(i)}) = wx^{(i)} + b $$
  - The gradient for the parameters w, b from that example $$ \frac{\partial J(w,b)}{\partial b}^{(i)} = (f_{w,b}(x^{(i)}) - y^{(i)}) $$ $$ \frac{\partial J(w,b)}{\partial w}^{(i)} = (f_{w,b}(x^{(i)}) -y^{(i)})x^{(i)} $$
- Return the total gradient update from all the examples $$ \frac{\partial J(w,b)}{\partial b} = \frac{1}{m} \sum\limits_{i = 0}^{m-1} \frac{\partial J(w,b)}{\partial b}^{(i)} $$

  $$ \frac{\partial J(w,b)}{\partial w} = \frac{1}{m} \sum\limits_{i = 0}^{m-1} \frac{\partial J(w,b)}{\partial w}^{(i)} $$
  - Here, m is the number of training examples and $\sum$ is the summation operator

If you get stuck, you can check out the hints presented after the cell below to help you with the implementation.

In [22]:
```
def compute_gradient (x_train, y_train, w, b):

    # number of training examples
    m = x_train.shape[0]

    #initilize dev of w & b
    dev_w = 0
    dev_b = 0

    for i in range(m):
        f_wb = np.dot(x_train[i], w) + b

        dev_w = dev_w + ((f_wb - y_train[i]) * x_train[i])
        dev_b = dev_b + (f_wb - y_train[i])

    total_dev_w = dev_w / m
    total_dev_b = dev_b / m
```

Loading [MathJax]/extensions/Safe.js

```
    return total_dev_w, total_dev_b
```

Run the cells below to check your implementation of the `compute_gradient` function with two different initializations of the parameters w,b.

In [23]:
```
# Compute and display gradient with w initialized to zeroes
initial_w = 0
initial_b = 0

tmp_dj_dw, tmp_dj_db = compute_gradient(x_train, y_train, initial_w, initial_b)
print('Gradient at initial w, b (zeros):', tmp_dj_dw, tmp_dj_db)

compute_gradient_test(compute_gradient)
```

```
Gradient at initial w, b (zeros): -65.32884974555672 -5.83913505154639
Using X with shape (4, 1)
All tests passed!
```

Expected Output:

Gradient at test w	-47.41610118 -4.007175051546391

In [24]:
```
# Compute and display cost and gradient with non-zero w
test_w = 0.2
test_b = 0.2
tmp_dj_dw, tmp_dj_db = compute_gradient(x_train, y_train, test_w, test_b)

print('Gradient at test w, b:', tmp_dj_dw, tmp_dj_db)
```

```
Gradient at test w, b: -47.41610118114435 -4.007175051546391
```

Expected Output:

Gradient at test w	-47.41610118 -4.007175051546391

2.6 Learning parameters using batch gradient descent

You will now find the optimal parameters of a linear regression model by using batch gradient descent. Recall batch refers to running all the examples in one iteration.

- You don't need to implement anything for this part. Simply run the cells below.
- A good way to verify that gradient descent is working correctly is to look at the value of $J(w,b)$ and check that it is decreasing with each step.
- Assuming you have implemented the gradient and computed the cost correctly and you have an appropriate value for the learning rate alpha, $J(w,b)$ should never increase and should converge to a steady value by the end of the algorithm.

In [27]:
```
def gradient_descent(x_train, y_train, w_in, b_in, compute_cost, compute_gradient, alpha

    # number of training examples
    m = x_train.shape[0]

    # An array to store cost J and w's at each iteration — primarily for graphing later
    J_hist = []
```

Loading [MathJax]/extensions/Safe.js

```
    w_hist = []

    w = copy.deepcopy(w_in) #avoid modifying global w within function
    b = b_in

    for i in range(num_iters):

        # Calculate the gradient and update the parameters
        total_dev_w, total_dev_b = compute_gradient(x_train, y_train, w, b)

        # Update Parameters using w, b, alpha and gradient
        w = w - alpha * total_dev_w
        b = b - alpha * total_dev_b

        # Save cost J at each iteration
        if i<100000:      # prevent resource exhaustion
            cost = compute_cost(x_train, y_train, w, b)
            J_hist.append(cost)

       # Print cost every at intervals 10 times or as many iterations if < 10
        if i% math.ceil(num_iters/10) == 0:
            w_hist.append(w)
            print(f"Iteration {i:4}: Cost {float(J_hist[-1]):8.2f}   ")

    return w, b, J_hist, w_hist #return w and J,w history for graphing
```

In [28]:

```
# initialize fitting parameters. Recall that the shape of w is (n,)
initial_w = 0.
initial_b = 0.

# some gradient descent settings
iterations = 1500
alpha = 0.01

w,b,_,_ = gradient_descent(x_train ,y_train, initial_w, initial_b,
                     compute_cost, compute_gradient, alpha, iterations)
print("w,b found by gradient descent:", w, b)
```

```
Iteration    0: Cost     6.74
Iteration  150: Cost     5.31
Iteration  300: Cost     4.96
Iteration  450: Cost     4.76
Iteration  600: Cost     4.64
Iteration  750: Cost     4.57
Iteration  900: Cost     4.53
Iteration 1050: Cost     4.51
Iteration 1200: Cost     4.50
Iteration 1350: Cost     4.49
w,b found by gradient descent: 1.166362350335582 -3.63029143940436
```

Expected Output:

w, b found by gradient descent	1.16636235 -3.63029143940436

We will now use the final parameters from gradient descent to plot the linear fit.

Recall that we can get the prediction for a single example $f(x^{(i)})= wx^{(i)}+b$.

To calculate the predictions on the entire dataset, we can loop through all the training examples and calculate the prediction for each example. This is shown in the code block below.

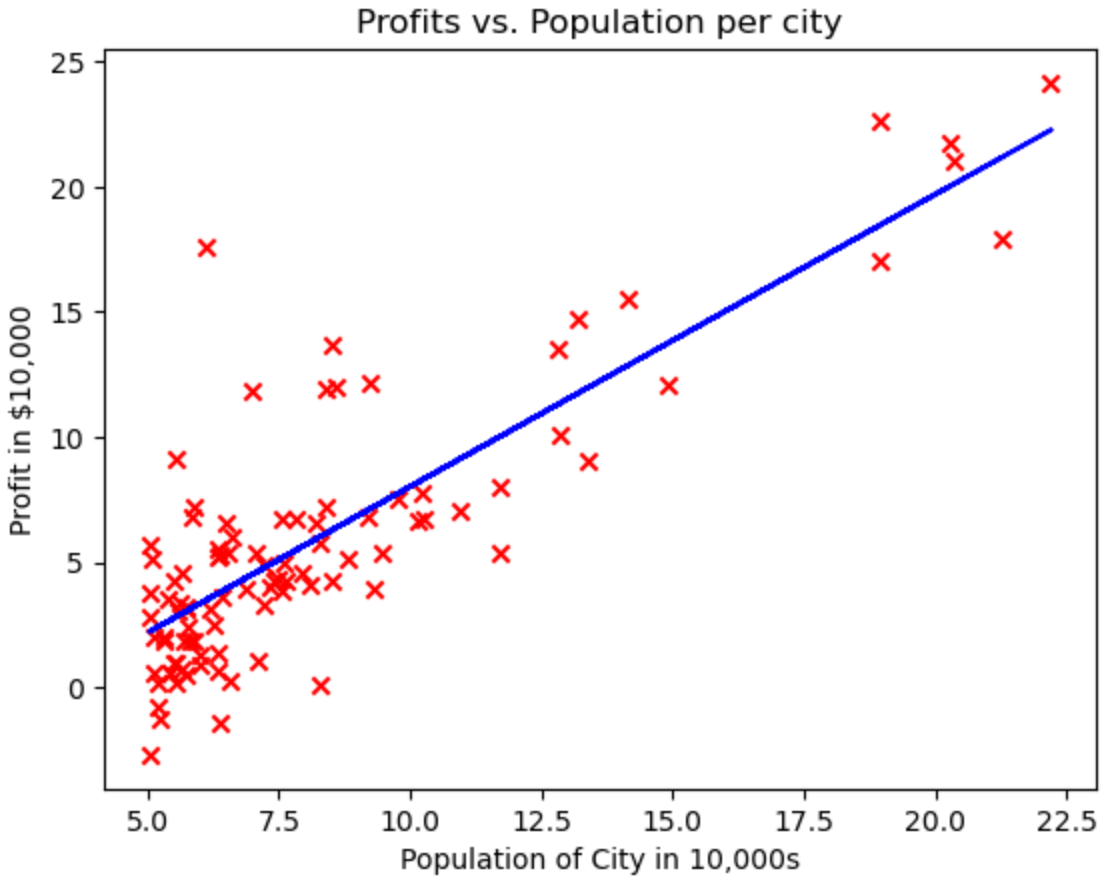
Loading [MathJax]/extensions/Safe.js

In [29]:
```
m = x_train.shape[0]
predicted = np.zeros(m)

for i in range(m):
    predicted[i] = w * x_train[i] + b
```

We will now plot the predicted values to see the linear fit.

In [30]:
```
# Plot the linear fit
plt.plot(x_train, predicted, c = "b")

# Create a scatter plot of the data.
plt.scatter(x_train, y_train, marker='x', c='r')

# Set the title
plt.title("Profits vs. Population per city")
# Set the y-axis label
plt.ylabel('Profit in $10,000')
# Set the x-axis label
plt.xlabel('Population of City in 10,000s')
```

Out[30]:
```
Text(0.5, 0, 'Population of City in 10,000s')
```



Your final values of w,b can also be used to make predictions on profits. Let's predict what the profit would be in areas of 35,000 and 70,000 people.

- The model takes in population of a city in 10,000s as input.
- Therefore, 35,000 people can be translated into an input to the model as `np.array([3.5])`
- Similarly, 70,000 people can be translated into an input to the model as `np.array([7.])`

Loading [MathJax]/extensions/Safe.js

In [31]:
```
predict1 = 3.5 * w + b
print('For population = 35,000, we predict a profit of $%.2f' % (predict1*10000))

predict2 = 7.0 * w + b
print('For population = 70,000, we predict a profit of $%.2f' % (predict2*10000))
```

```
For population = 35,000, we predict a profit of $4519.77
For population = 70,000, we predict a profit of $45342.45
```

Expected Output:

For population = 35,000, we predict a profit of	$4519.77
For population = 70,000, we predict a profit of	$45342.45

Congratulations on completing this practice lab on linear regression! Next week, you will create models to solve a different type of problem: classification. See you there!

In [ ]:

Loading [MathJax]/extensions/Safe.js