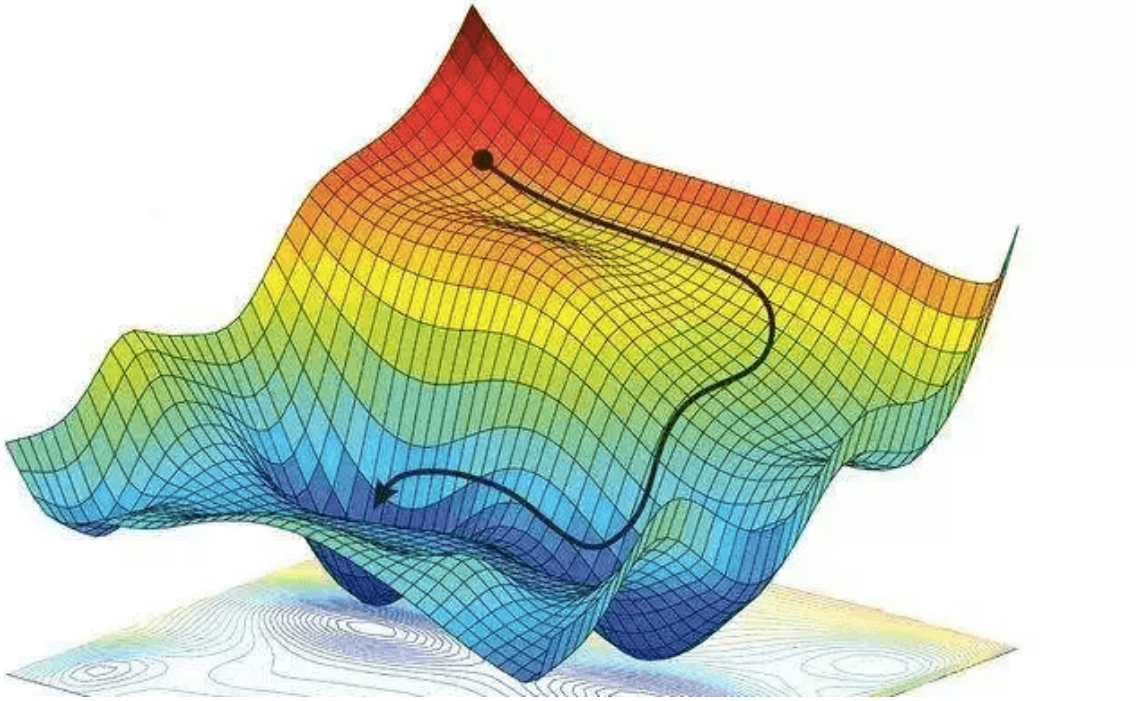
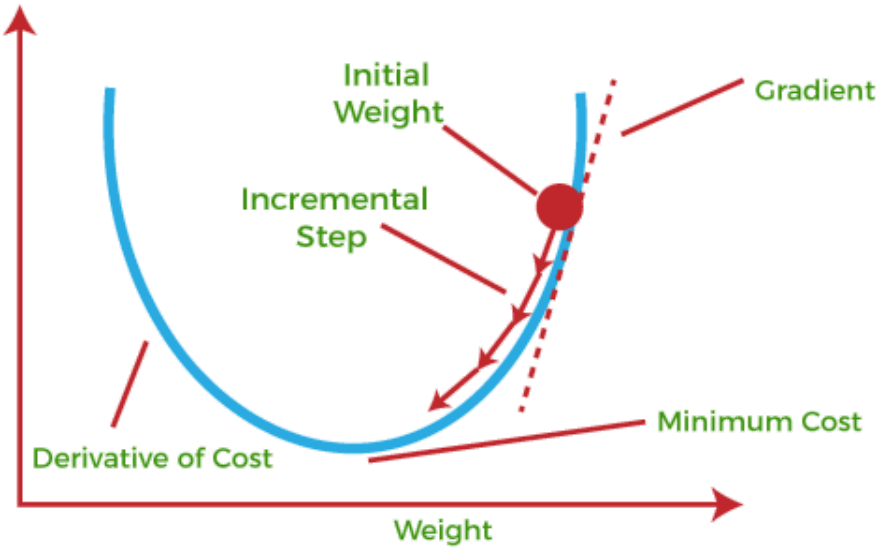
What is Gradient Descent ?

Gradient descent is an iterative optimization algorithm used to find the **minimum of a function**. It is widely used in machine learning and other optimization problems. The goal of gradient descent is to update the parameters of a model **iteratively by taking steps proportional to the negative gradient of the function being minimized**

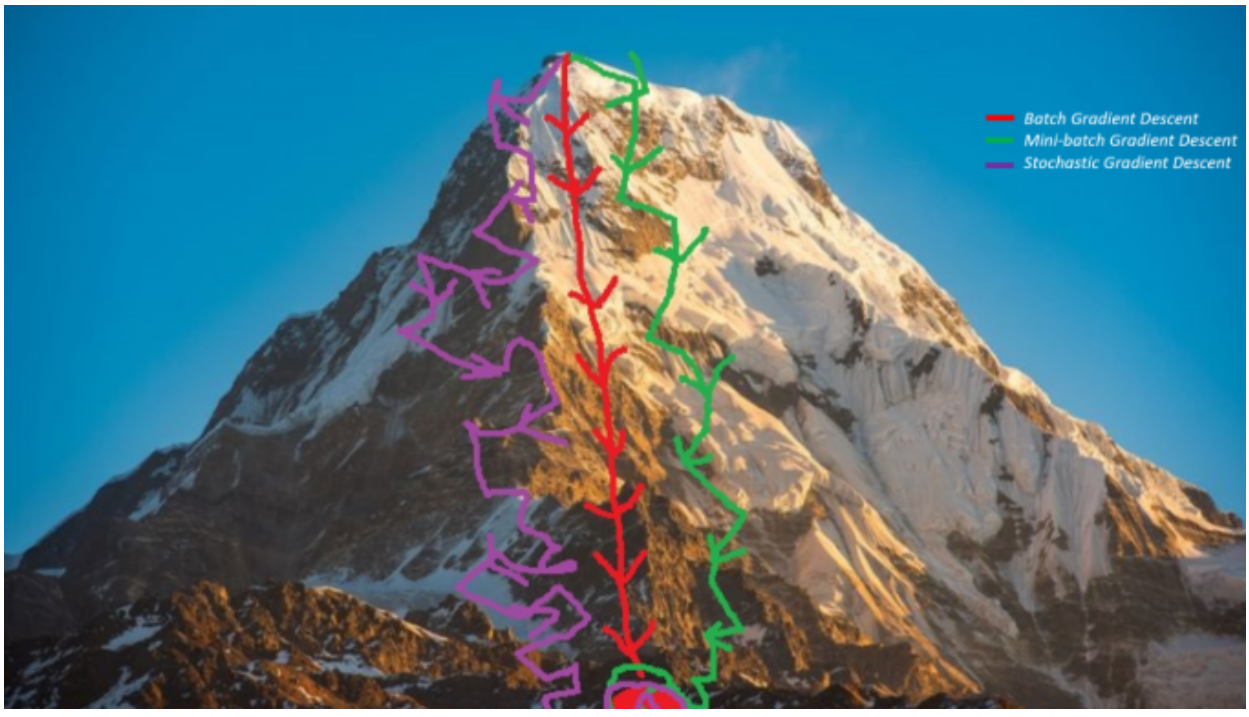


- gradient descent (also often called steepest descent) is a first-order iterative optimization algorithm for finding a local minimum of a differentiable function. The idea is to take repeated steps in the opposite direction of the gradient (or approximate gradient) of the function at the current point, because this is the direction of steepest descent. Conversely, stepping in the direction of the gradient will lead to a local maximum of that function; the procedure is then known as gradient ascent. It is particularly useful in machine learning for minimizing the cost or loss function. Gradient descent should not be confused with local search algorithms, although both are iterative methods for optimization.



Types of Gradient descent

1. Batch Gradient Descent (BGD)
2. Stochastic Gradient Descent (SGD)
3. Mini-batch Gradient Descent

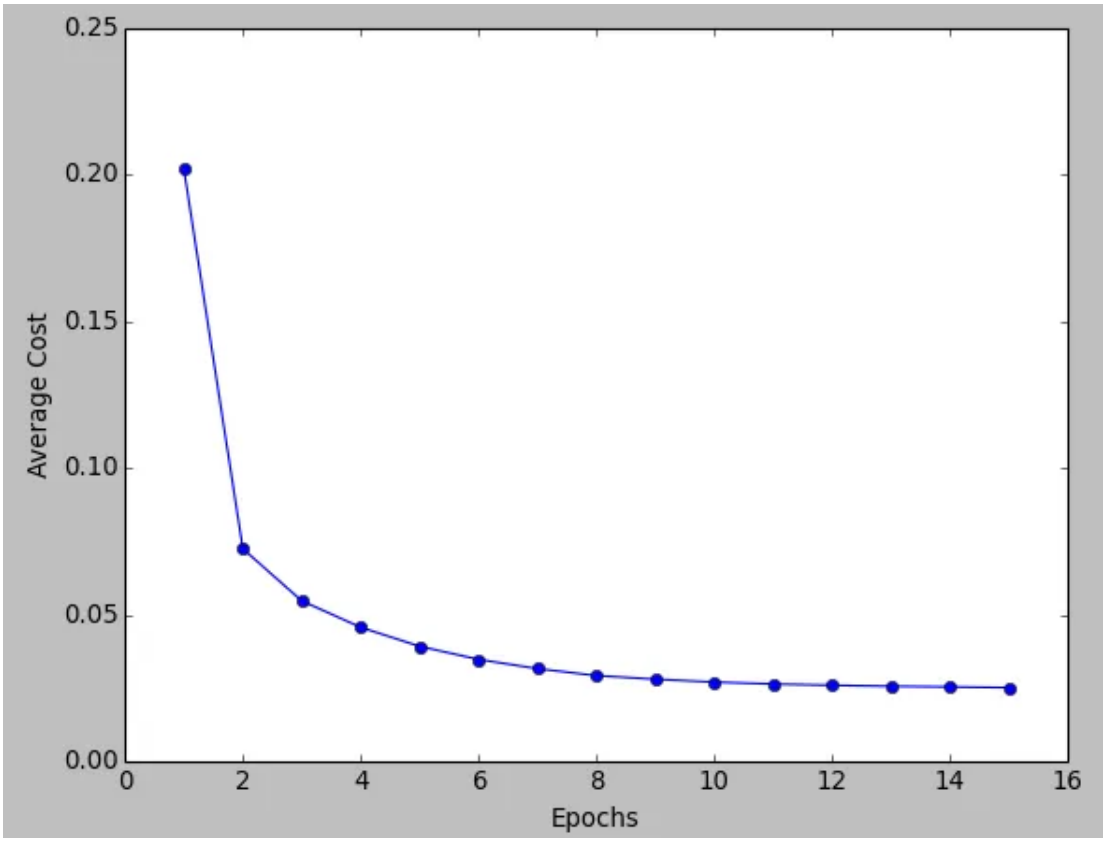


There are several variations of gradient descent that have been developed to address different challenges and improve the convergence speed of the algorithm. Here are some common types of gradient descent:

1. **Batch Gradient Descent (BGD)**:

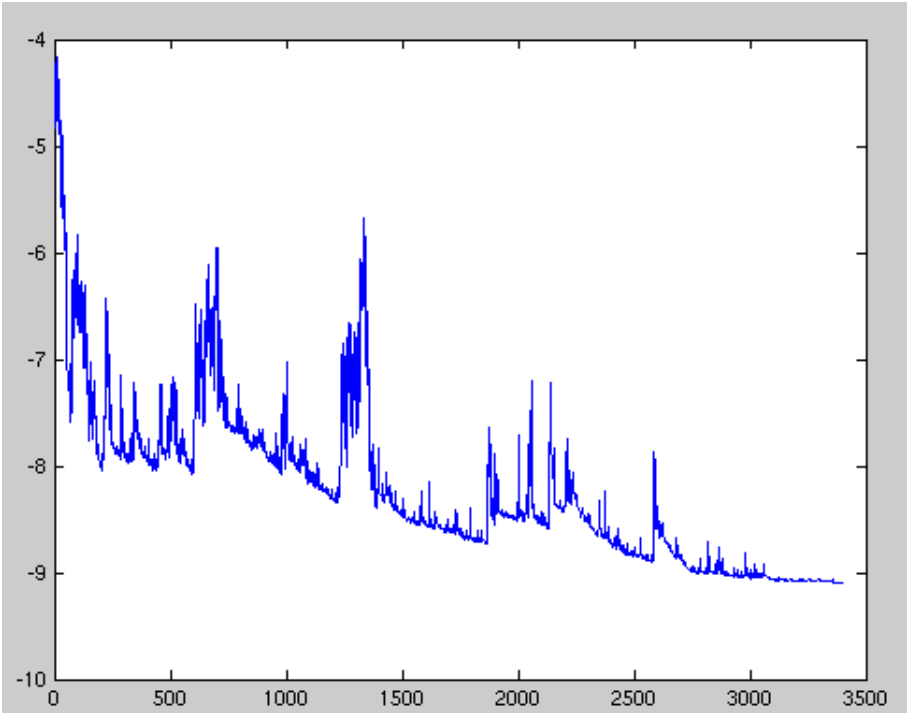
- In batch gradient descent, the entire training dataset is used to compute the gradient at each iteration.

- It calculates the average gradient over all training examples before updating the parameters.
- BGD can be computationally expensive, especially for large datasets, as it requires evaluating the entire dataset for each iteration.



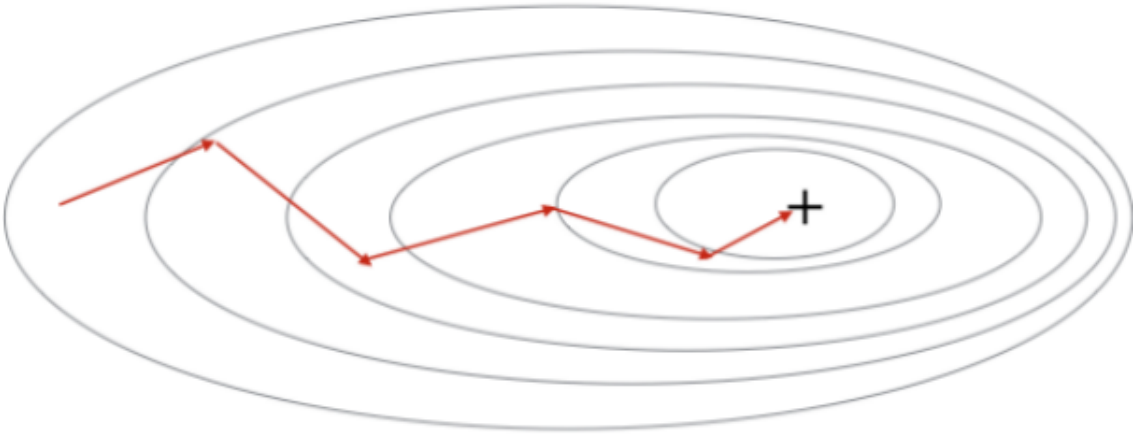
2. **Stochastic Gradient Descent (SGD)**:

- In stochastic gradient descent, only one randomly selected training example is used to compute the gradient at each iteration.
- It updates the parameters based on the gradient of a single example, resulting in faster iterations.
- SGD introduces more noise due to the high variance of the gradient estimate, but it can escape shallow local minima and handle large datasets more efficiently.



3. **Mini-batch Gradient Descent**:

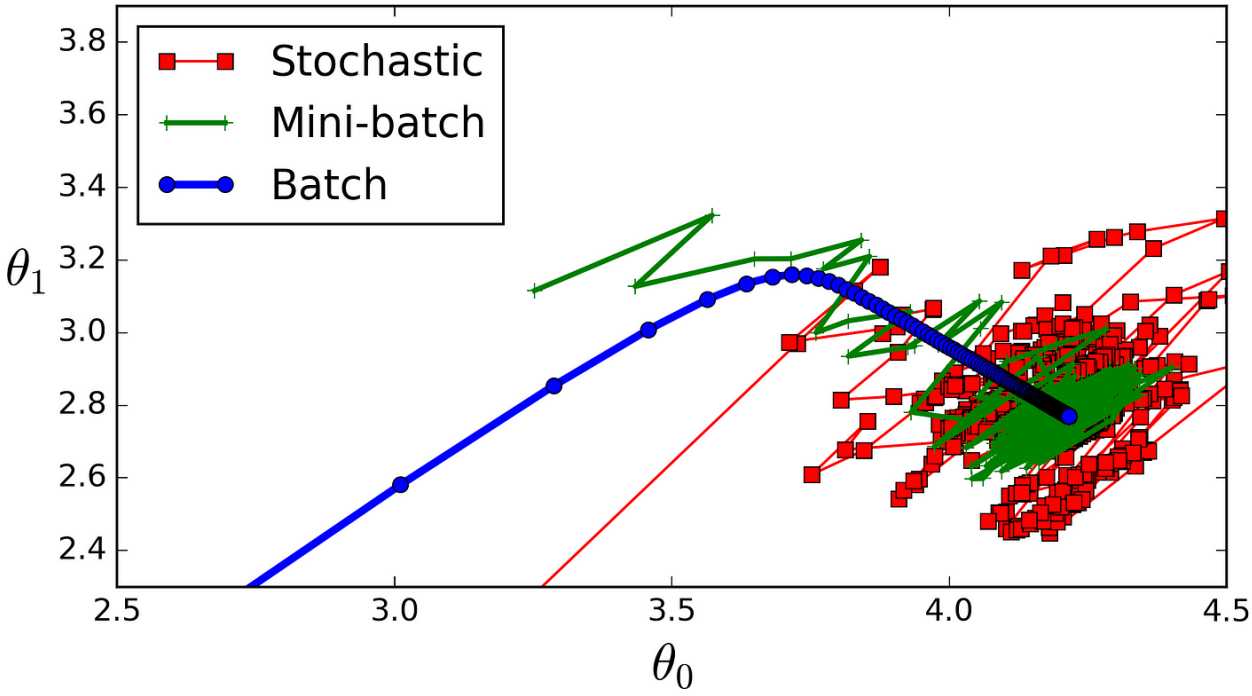
- Mini-batch gradient descent is a compromise between BGD and SGD.
- It computes the gradient using a small subset or mini-batch of training examples at each iteration.
- Mini-batch size is typically chosen to be between 10 and 1,000, providing a balance between stability and computational efficiency.



Here is a table that summarizes the key differences between the three types of gradient descent:

Type	Pros	Cons
Batch gradient descent	* Most accurate * Can converge to a global minimum	* Most computationally expensive * Can be slow to converge for large datasets

Type	Pros	Cons
Stochastic gradient descent	* Least computationally expensive * Can be fast to converge for large datasets	* Can be less accurate than batch gradient descent * Can be more sensitive to noise in the data
	* A compromise between batch gradient descent and stochastic	



The best type of gradient descent to use depends on the specific problem you are trying to solve.

- If accuracy is your top priority, then batch gradient descent is the best option.
- If you are working with a large dataset and speed is important, then stochastic gradient descent is a good choice.
- And if you want a balance between accuracy and speed, then mini-batch gradient descent is a good option.

Step by step Gradient Descent Implementation

The intuition behind gradient descent is to find the minimum of a function by iteratively moving in the direction of steepest descent. Let's explore the intuition behind this algorithm:

1. **Visualizing the cost function**:
    - Consider a scenario where we have a function with multiple parameters (e.g., a machine learning model with weights and biases).
- We can visualize this function as a landscape, where the height of the landscape represents the value of the function.
- The goal is to find the lowest point (minimum) on this landscape, which corresponds to the optimal parameter values.

2. **Steepest descent**:

- To find the minimum, we want to move downhill in the direction that decreases the function value the most.
- The steepest descent direction is given by the negative gradient of the function, which points in the direction of the greatest increase.
- By taking the negative gradient, we move in the opposite direction, i.e., the direction of steepest descent.

3. **Iterative updates**:

- Gradient descent performs iterative updates to move closer to the minimum.
- Starting from an initial point, we compute the gradient at that point, which indicates the direction of steepest ascent.
- We update the parameters by moving in the direction opposite to the gradient, multiplied by a small step size (learning rate).
- This step is repeated until convergence or a maximum number of iterations is reached.

4. **Adjusting step size:**

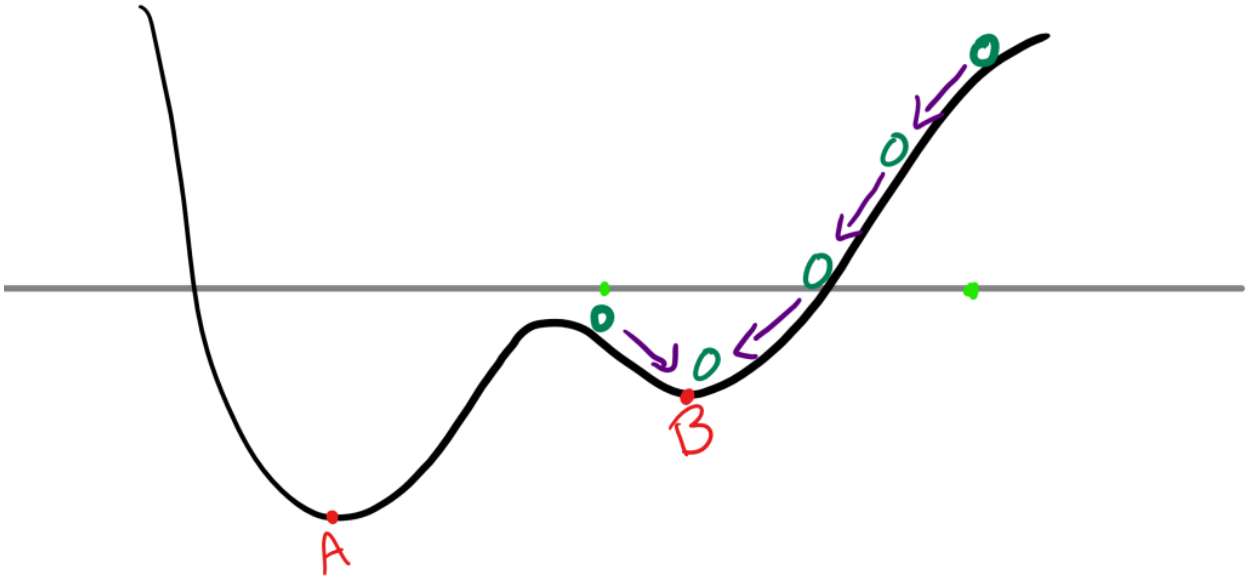
- The learning rate determines the size of the steps taken in each iteration.
- If the learning rate is too large, we might overshoot the minimum and diverge.
- If the learning rate is too small, convergence might be slow.
- Finding an appropriate learning rate is crucial for the algorithm's success.

5. **Convergence**:

- As we iterate and update the parameters, we gradually approach the minimum.
- Convergence occurs when further updates do not significantly reduce the function value.
- At convergence, the algorithm terminates, and the current parameter values are considered optimal or near-optimal.

The key intuition behind gradient descent is that by iteratively moving in the direction of steepest descent, we can navigate the landscape of the function and find its minimum. This process allows us to optimize various machine learning models and solve optimization problems efficiently.

Additionally, gradient descent is inspired by the idea of mimicking the behavior of a ball rolling down a hill. The ball naturally follows the steepest descent, ultimately settling at the bottom of the hill, which corresponds to the minimum of the function.

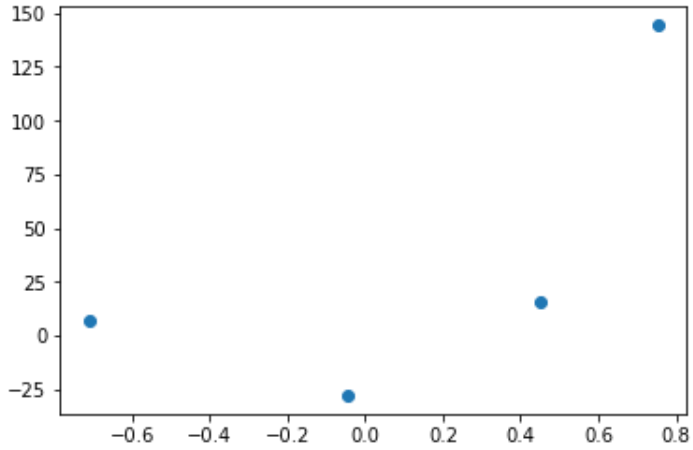


In [1]:
```
# Code

from sklearn.datasets import make_regression
import numpy as np
```

In [2]:
```
X, y = make_regression(
    n_samples=4,
    n_features=1,
    n_informative=1,
    n_targets=1,
    noise=80,
    random_state=13
)
```

In [3]:
```
import matplotlib.pyplot as plt

# Scatter plot of X and y
plt.scatter(X,y) # linear
```

Out[3]:
```
<matplotlib.collections.PathCollection at 0x1d0152e8490>
```



In [4]:
```
# Lets apply OLS
from sklearn.linear_model import LinearRegression
```

In [5]:
```
# Fits a linear regression model to the given data.

reg = LinearRegression()
reg.fit(X,y)
```

Out[5]:
```
LinearRegression()
```

In [6]:
```
reg.coef_
```

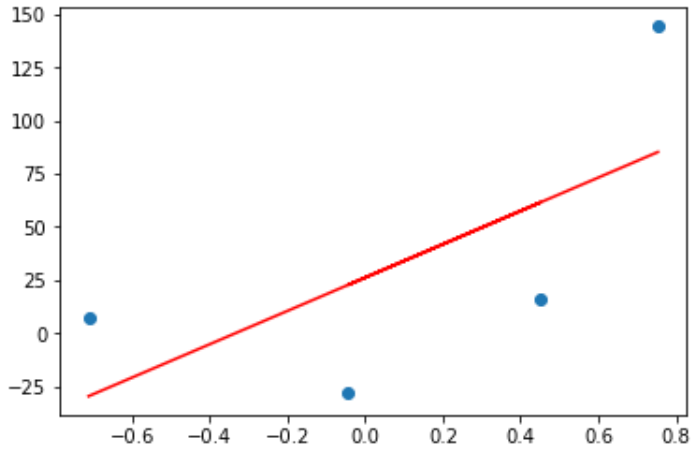
Out[6]:
```
array([78.35063668])
```

In [7]:
```
reg.intercept_
```

Out[7]:
```
26.15963284313262
```

In [8]:
```
plt.scatter(X,y)
plt.plot(X,reg.predict(X),color='red')
```

Out[8]:
```
[<matplotlib.lines.Line2D at 0x1d016730b20>]
```

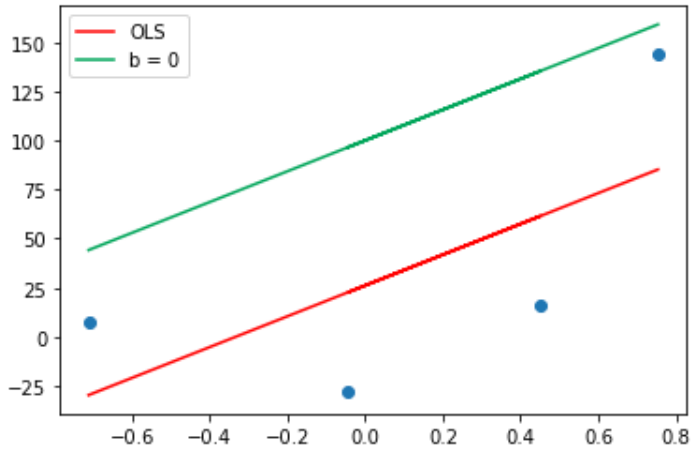


Apply Gradient descent

Lets apply Gradient Descent assuming slope is constant m = 78.35

In [9]:
```
# And let's assume the starting value for intercept b = 0


# Focal cell: calculate y_pred based on X
y_pred = ((78.35 * X) + 100).reshape(4)
```

In [10]:
```
# Plotting y_values

plt.scatter(X,y)
plt.plot(X,reg.predict(X),color='red',label='OLS')
plt.plot(X,y_pred,color='#00a65a',label='b = 0')
plt.legend()
plt.show()
```



$$\frac{\partial}{\partial \mathrm{b}} = \frac{2}{N}\sum_{i=1}^{N} -(y_i - (mx_i - b))$$

In [11]:
```
m = 78.35
b = 0

# Calculate the slope of the loss function
loss_slope = -2 * np.sum(y - m * X.ravel() - b)

# Print the slope of the loss function
loss_slope
```

Out[11]: -209.27763408209216

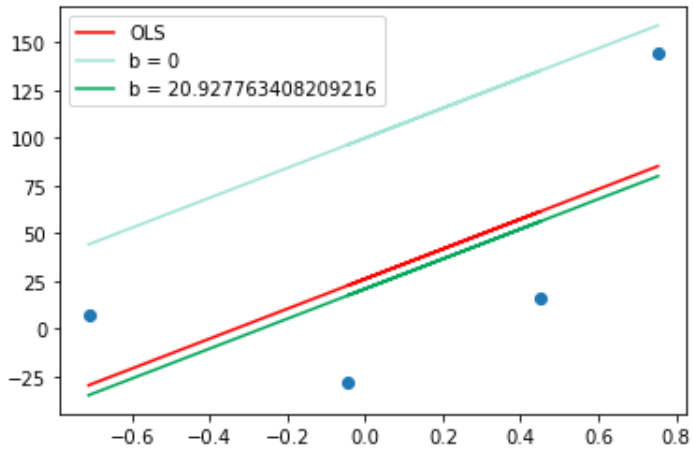
In [12]:
```
# Step size


# Lets take learning rate = 0.1
lr = 0.1

step_size = loss_slope*lr
step_size
```

Out[12]: -20.927763408209216

In [13]:
```

# Calculating the new intercept (b)

b = b - step_size # 0-(-20.927763408209216)

b
```

Out[13]: 20.927763408209216

In [14]:
```
y_pred1 = ((78.35 * X) + b).reshape(4) # Updated 'b' value

plt.scatter(X,y)
plt.plot(X,reg.predict(X),color='red',label='OLS')

plt.plot(X,y_pred,color='#A3E4D7',label='b = 0')
plt.plot(X,y_pred1,color='#00a65a',label='b = {}'.format(b))

plt.legend()
plt.show()
```



In [15]:
```
# Iteration 2

loss_slope = -2 * np.sum(y - m*X.ravel() - b) # Here b= 20.92
loss_slope
```

Out[15]: -41.85552681641843

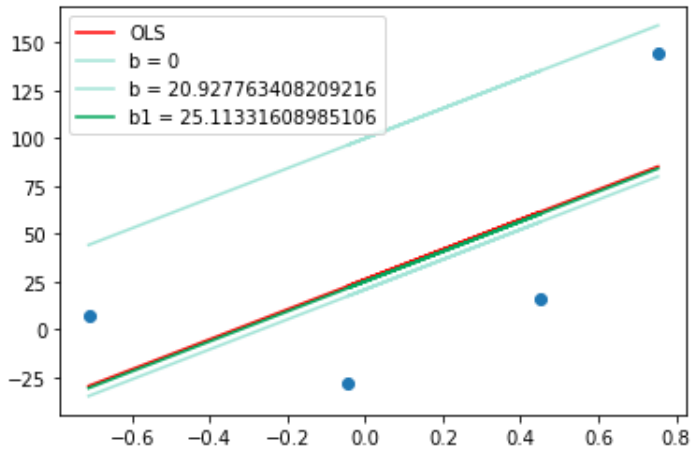
In [16]:
```
# Caluculate Step_size

step_size = loss_slope*lr # step_size * learning rate
step_size
```

Out[16]: -4.185552681641844

In [17]:
```
# Calculating the new intercept (b)

b1 = b - step_size #(20.92 - -4.18555)
b1
```

Out[17]: 25.11331608985106

In [18]:
```
y_pred2 = ((78.35 * X) + b1).reshape(4) # Updated 'b1' value

plt.scatter(X,y)
plt.plot(X,reg.predict(X),color='red',label='OLS')

plt.plot(X,y_pred,color='#A3E4D7',label='b = 0')
plt.plot(X,y_pred1,color='#A3E4D7',label='b = {}'.format(b))
plt.plot(X,y_pred2,color='#00a65a',label='b1 = {}'.format(b1))

plt.legend()
plt.show()
```



In [19]:
```
# Iteration 3
loss_slope = -2 * np.sum(y - m*X.ravel() - b1) # b1= 25.11
loss_slope
```

Out[19]: -8.371105363283675

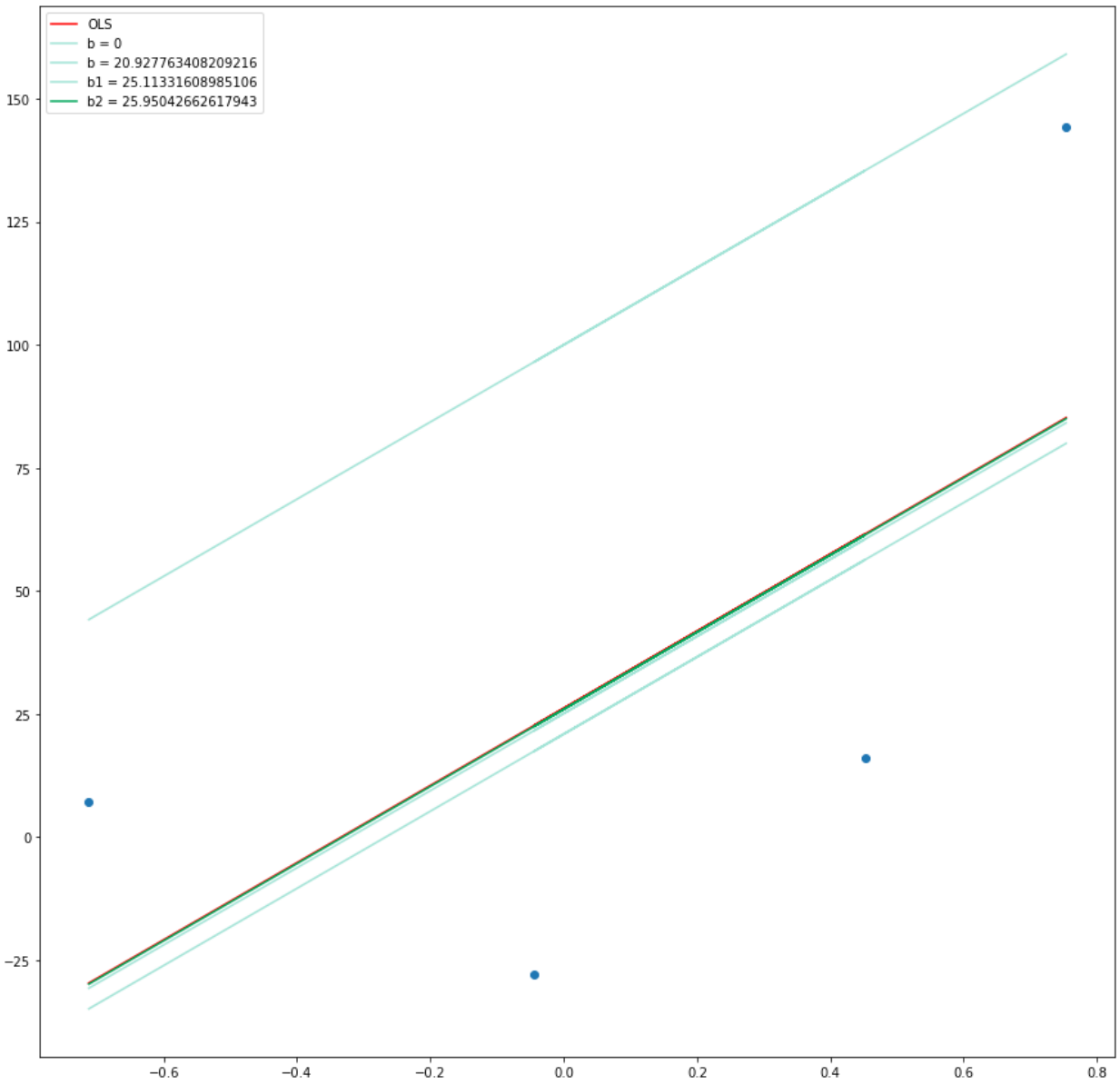
In [20]:
```
# Caluculate Step_size

step_size = loss_slope*lr # step_size * learning rate
step_size
```

Out[20]: -0.8371105363283675

In [21]:
```
# Calculating the new intercept (b1)

b2 = b1 - step_size #(25.11 - -0.83711 )
b2
```

Out[21]: 25.95042662617943

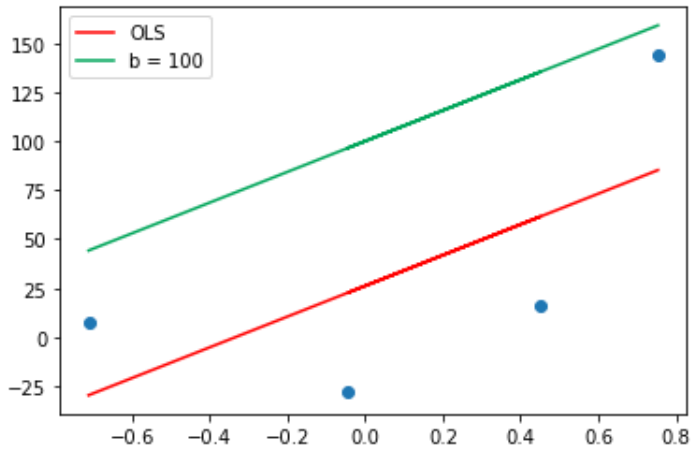
In [22]:
```
y_pred3 = ((78.35 * X) + b2).reshape(4) # updated 'b2' = 25.95

plt.figure(figsize=(15,15))
plt.scatter(X,y)
plt.plot(X,reg.predict(X),color='red',label='OLS')

plt.plot(X,y_pred,color='#A3E4D7',label='b = 0')


plt.plot(X,y_pred1,color='#A3E4D7',label='b = {}'.format(b))
plt.plot(X,y_pred2,color='#A3E4D7',label='b1 = {}'.format(b1))
plt.plot(X,y_pred3,color='#00a65a',label='b2 = {}'.format(b2))


plt.legend()
plt.show()
```



The beauty of gradient descent is that it takes near to the answer no matter how huge or small the

In [23]:
```
# Lets apply Gradient Descent assuming slope is constant m = 78.35
# and let's assume the starting value for intercept b = 100


# Focal cell: calculate y_pred based on X
y_pred = ((78.35 * X) + 100).reshape(4)
```

In [24]:
```
# Plotting y_values

plt.scatter(X,y)
plt.plot(X,reg.predict(X),color='red',label='OLS')
plt.plot(X,y_pred,color='#00a65a',label='b = 100')
plt.legend()
plt.show()
```



$$\frac{\partial}{\partial b} = \frac{2}{N}\sum_{i=1}^{N} -(y_i - (mx_i - b))$$

In [25]:
```
m = 78.35
b = 100

# Calculate the slope of the loss function
loss_slope = -2 * np.sum(y - m * X.ravel() - b)

# Print the slope of the loss function
loss_slope
```

Out[25]: 590.7223659179078

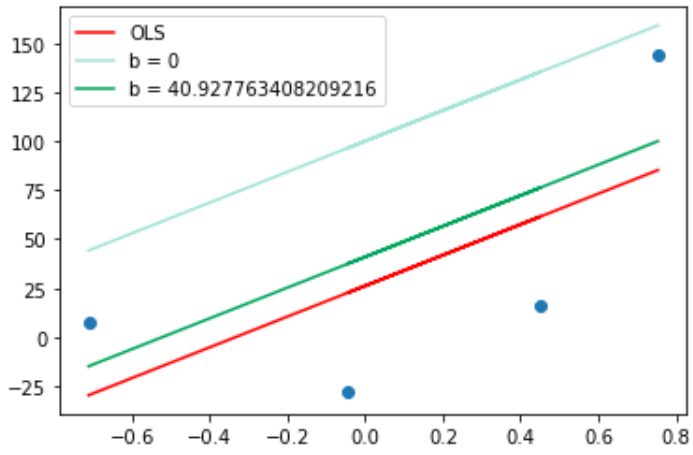
In [26]:
```
# Step size


# Lets take learning rate = 0.1
lr = 0.1

step_size = loss_slope*lr
step_size
```

Out[26]: 59.072236591790784

In [27]:
```

# Calculating the new intercept (b)

b = b - step_size # 100-(59.072236591790784 )

b
```

Out[27]: 40.927763408209216

In [28]:
```
y_pred1 = ((78.35 * X) + b).reshape(4) # Updated 'b' value

plt.scatter(X,y)
plt.plot(X,reg.predict(X),color='red',label='OLS')

plt.plot(X,y_pred,color='#A3E4D7',label='b = 0')
plt.plot(X,y_pred1,color='#00a65a',label='b = {}'.format(b))
plt.legend()
plt.show()
```



In [29]:
```
# Iteration 2

loss_slope = -2 * np.sum(y - m*X.ravel() - b) # Here b= 40.92
loss_slope
```

Out[29]: 118.14447318358157

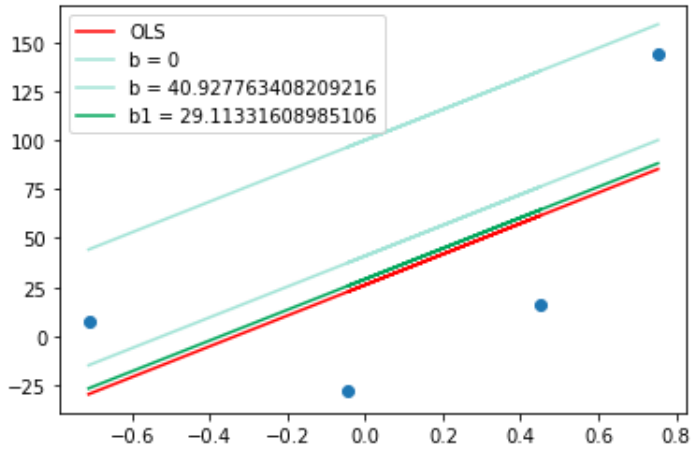
In [30]:
```
# Caluculate Step_size

step_size = loss_slope*lr # step_size * learning rate
step_size
```

Out[30]:
```
11.814447318358157
```

In [31]:
```
# Calculating the new intercept (b)

b1 = b - step_size #(40.92 - 11.814447318358157  )
b1
```

Out[31]:
```
29.11331608985106
```

In [32]:
```
y_pred2 = ((78.35 * X) + b1).reshape(4) # Updated 'b1' value

plt.scatter(X,y)
plt.plot(X,reg.predict(X),color='red',label='OLS')

plt.plot(X,y_pred,color='#A3E4D7',label='b = 0')
plt.plot(X,y_pred1,color='#A3E4D7',label='b = {}'.format(b))
plt.plot(X,y_pred2,color='#00a65a',label='b1 = {}'.format(b1))

plt.legend()
plt.show()
```



In [33]:
```
# Iteration 3
loss_slope = -2 * np.sum(y - m*X.ravel() - b1) # b1= 29.11
loss_slope
```

Out[33]:
```
23.62889463671634
```

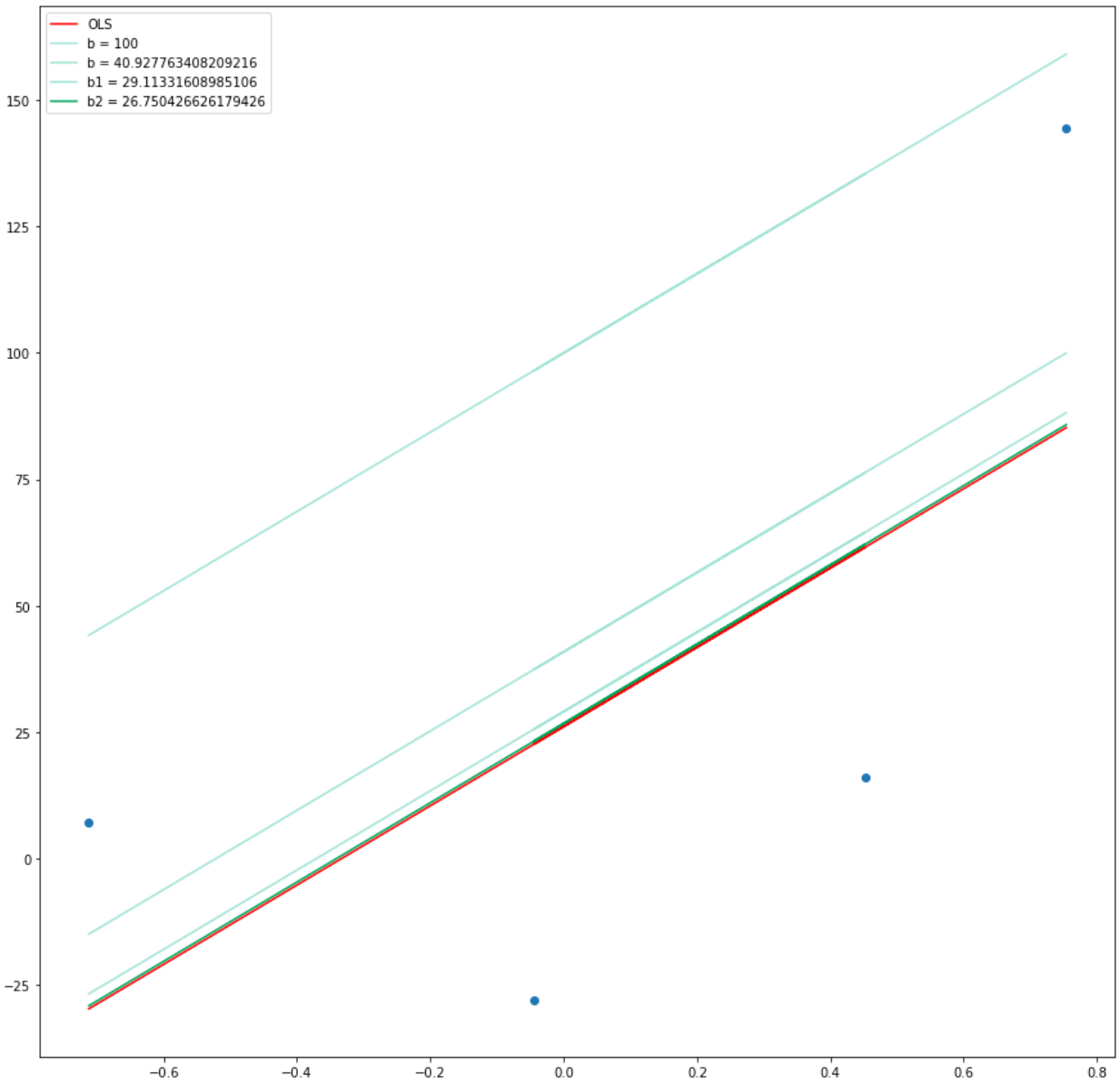
In [34]:
```
# Caluculate Step_size

step_size = loss_slope*lr # step_size * learning rate
step_size
```

Out[34]:
```
2.362889463671634
```

In [35]:
```
# Calculating the new intercept (b1)

b2 = b1 - step_size #(29.11 - 2.362889463671634 )
b2
```

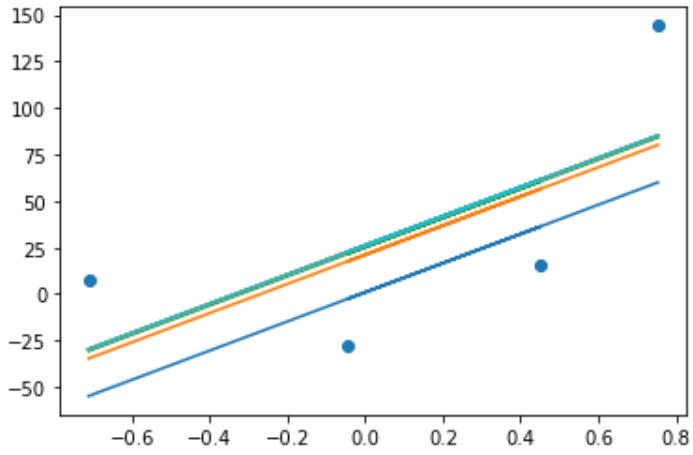
Out[35]:
```
26.750426626179426
```

In [36]:
```
y_pred3 = ((78.35 * X) + b2).reshape(4) # updated 'b2' = 26.75

plt.figure(figsize=(15,15))
plt.scatter(X,y)
plt.plot(X,reg.predict(X),color='red',label='OLS')

plt.plot(X,y_pred,color='#A3E4D7',label='b = 100')

plt.plot(X,y_pred1,color='#A3E4D7',label='b = {}'.format(b))
plt.plot(X,y_pred2,color='#A3E4D7',label='b1 = {}'.format(b1))
plt.plot(X,y_pred3,color='#00a65a',label='b2 = {}'.format(b2))



plt.legend()
plt.show()
```



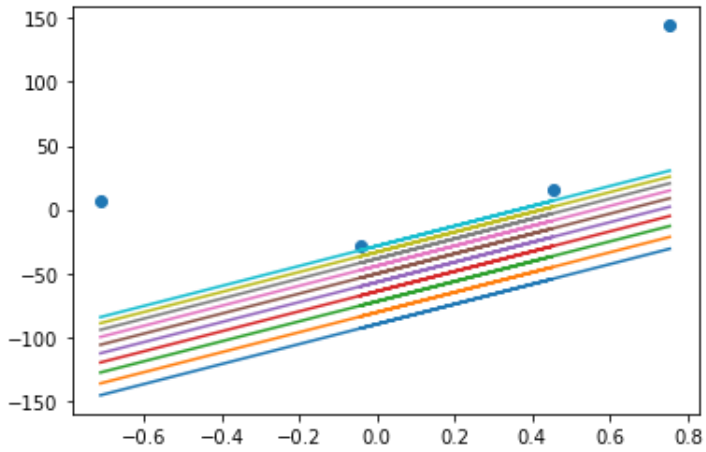
In [37]:
```
# Loop with -100 , epochs = 100 and learning_rate = 0.1

b = -100
m = 78.35
lr = 0.1

epochs = 10

for i in range(epochs):
    loss_slope = -2 * np.sum(y - m*X.ravel() - b)
    b = b - (lr * loss_slope)

    y_pred = m * X + b

    plt.plot(X,y_pred)

plt.scatter(X,y)
```

Out[37]:
```
<matplotlib.collections.PathCollection at 0x1d016833760>
```



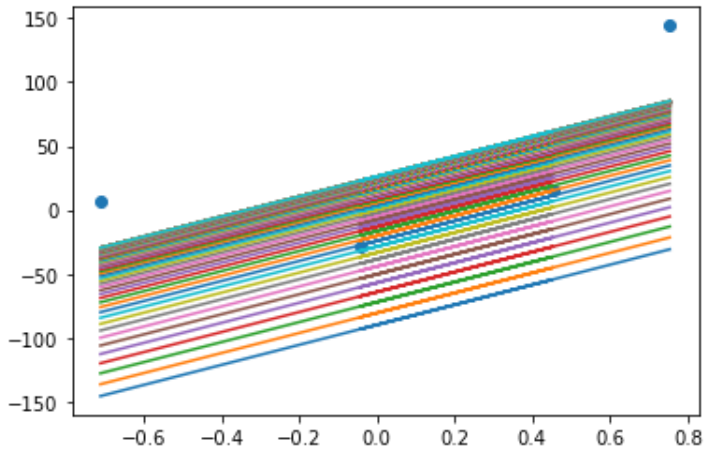
In [38]:
```
#  What happens , when we decrease Learning rate

b = -100
m = 78.35
lr = 0.01 # Learning is very small here

epochs = 10

for i in range(epochs):
    loss_slope = -2 * np.sum(y - m*X.ravel() - b)
    b = b - (lr * loss_slope)

    y_pred = m * X + b

    plt.plot(X,y_pred)

plt.scatter(X,y)

# we cant reach to Right answer
```

Out[38]: <matplotlib.collections.PathCollection at 0x1d016a95fd0>



In [39]:
```
#  To solve this , we have to increase Epochs to 100

b = -100
m = 78.35
lr = 0.01

epochs = 100

for i in range(epochs):
    loss_slope = -2 * np.sum(y - m*X.ravel() - b)
    b = b - (lr * loss_slope)

    y_pred = m * X + b

    plt.plot(X,y_pred)

plt.scatter(X,y)
```

Out[39]:
```
<matplotlib.collections.PathCollection at 0x1d016ada550>
```



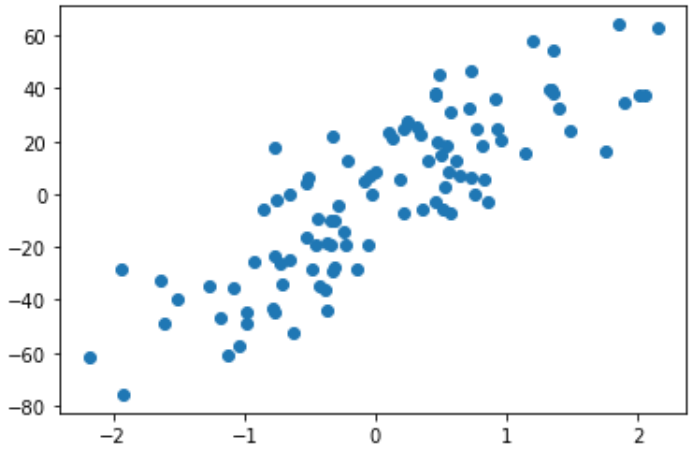
Creating GDRegressor Class

In [40]:
```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import cross_val_score
```

In [41]:
```
from sklearn.datasets import make_regression

# Generate a regression dataset
X, y = make_regression(
    n_samples=100,
    n_features=1,
    n_informative=1,
    n_targets=1,
    noise=20,
    random_state=13
)
```

In [42]:
```
plt.scatter(X,y)
```

Out[42]:
```
<matplotlib.collections.PathCollection at 0x1d016ca2e20>
```



In [43]:
```
# Importing the necessary module
from sklearn.model_selection import train_test_split

# Splitting the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=2
```

In [44]:
```
from sklearn.linear_model import LinearRegression

# Create an instance of LinearRegression
lr = LinearRegression()
```

In [45]:
```
lr.fit(X_train,y_train)

print(lr.coef_)
```

```
[28.12597332]
```

In [46]:
```
print(lr.intercept_)
```

```
-2.271014426178382
```

In [47]:
```
# Predict y values using the lr model on the X_test data
y_pred = lr.predict(X_test)

# Import the r2_score function from sklearn.metrics
from sklearn.metrics import r2_score

# Calculate the R2 score using the y_test and y_pred values
r2_score(y_test, y_pred)
```

Out[47]:
```
0.6345158782661013
```

In [48]:
```
print(lr.coef_)
```

```
[28.12597332]
```

In [49]:
```
m = 28.12
```

In [50]:
```
class GDRegressor:

    def __init__(self,learning_rate,epochs):
        self.m = 28.12
        self.b = -120
        self.lr = learning_rate
        self.epochs = epochs

    def fit(self ,X,y):
        # Calculate the b using GD
        for i in range(self.epochs):
            loss_slope = -2 * np.sum(y - self.m*X.ravel()- self.b)
            self.b = self.b -(self.lr *loss_slope)
        print(self.b)
```

In [51]:
```
gd = GDRegressor(0.1,10)
```

In [52]:
```
gd.fit(X,y) # Here Learning rate is too High , thats the reason , we got this result
```

```
-721554187522014.0
```

In [53]:
```
# lr =0.01

gd = GDRegressor(0.01,10)
```

In [54]:
```
gd.fit(X,y)
```

```
-120.0
```

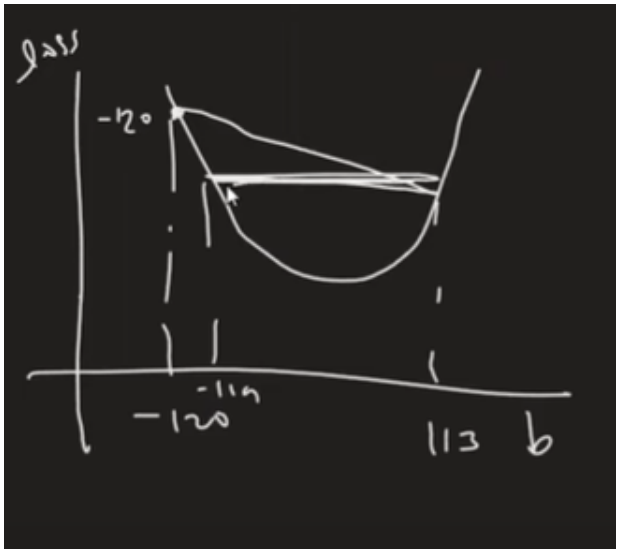
In [55]:
```
# change in code

class GDRegressor:

    def __init__(self,learning_rate,epochs):
        self.m = 28.12
        self.b = -120
        self.lr = learning_rate
        self.epochs = epochs

    def fit(self ,X,y):
        # Calculate the b using GD
        for i in range(self.epochs):
            loss_slope = -2 * np.sum(y - self.m*X.ravel()- self.b)
            self.b = self.b -(self.lr *loss_slope)
            print(loss_slope,self.b) # added line
        print(self.b)
```

In [56]:
```
gd = GDRegressor(0.01,10)
```

In [57]:
```
gd.fit(X,y)
```

```
-23537.64116001603 115.37641160016031
23537.64116001603 -120.0
-23537.64116001603 115.37641160016031
23537.64116001603 -120.0
-23537.64116001603 115.37641160016031
23537.64116001603 -120.0
-23537.64116001603 115.37641160016031
23537.64116001603 -120.0
-23537.64116001603 115.37641160016031
23537.64116001603 -120.0
-120.0
```

Problem



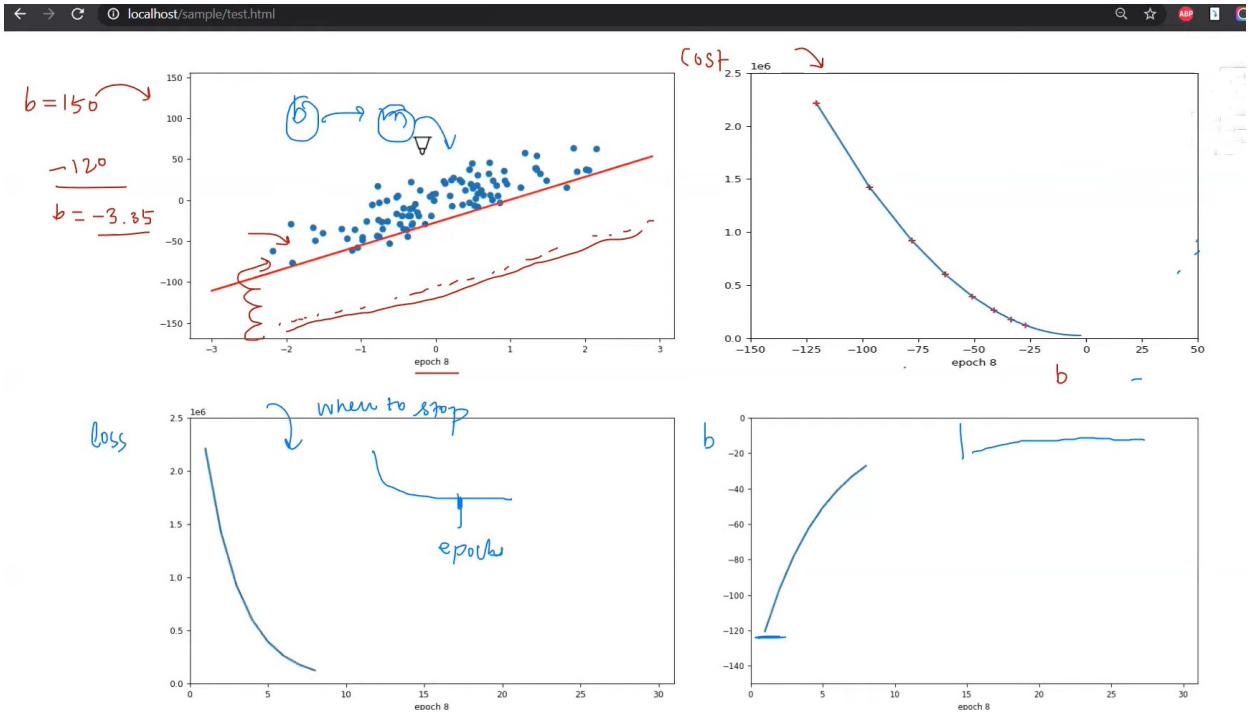
solution

In [58]:
```
# Decrease Learing rate and increase Epochs

gd =GDRegressor(0.001,100) #learning_rate =0.001 , epochs =100
```

In [59]:
```
gd.fit(X,y)
```

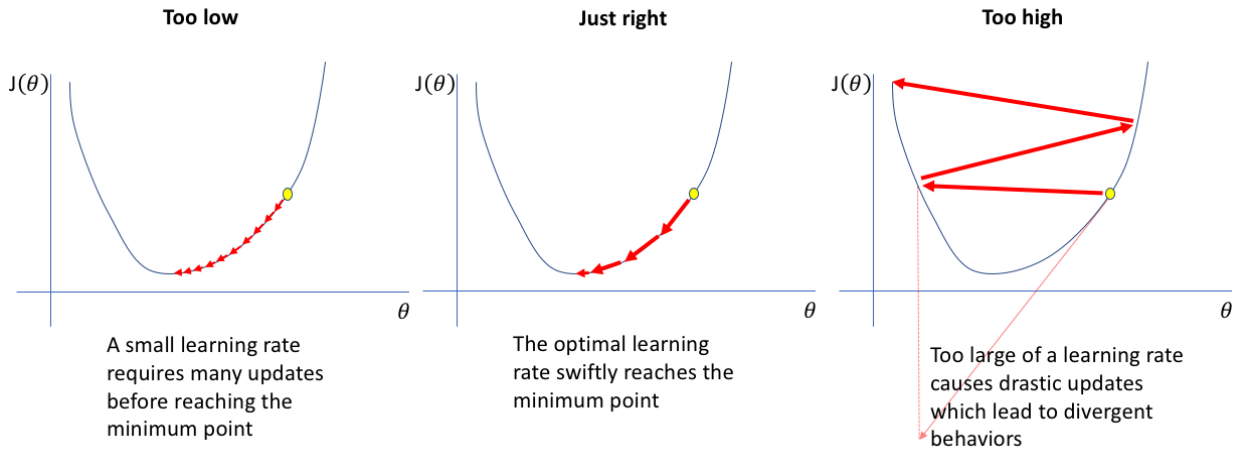
```
-23537.64116001603 -96.46235883998396
-18830.11292801282 -77.63224591197114
-15064.090342410256 -62.568155569560886
-12051.272273928204 -50.51688329563268
-9641.017819142568 -40.87586547649011
-7712.814255314051 -33.16305122117606
-6170.251404251242 -26.99279981692482
-4936.201123400993 -22.056598693523824
-3948.9608987207935 -18.107637794803033
-3159.168718976635 -14.948469075826399
-2527.3349751813084 -12.42113410064509
-2021.8679801450467 -10.399266120500045
-1617.4943841160375 -8.781771736384007
-1293.99550729283 -7.4877762290911765
-1035.1964058342637 -6.4525798232569125
-828.157124667411 -5.624422698589502
-662.5256997339287 -4.961896998855573
-530.0205597871429 -4.4318764390684295
-424.01644782971437 -4.007859991238715
-339.2131582637714 -3.6686468329749435
-271.37052661101717 -3.3972763063639264
-217.09642128881381 -3.1801798850751126
-173.677137031051 -3.0065027480440616
-138.94170962484083 -2.867561038419221
-111.15336769987276 -2.7564076707193483
-88.92269415989811 -2.66748497655945
-71.13815532791858 -2.5963468212315317
-56.910524262334796 -2.539436296969197
-45.52841940986784 -2.4939078775593293
-36.422735527894424 -2.4574851420314348
-29.138188422315324 -2.4283469536091196
-23.310550737852424 -2.405036402871267
-18.648440590281957 -2.386387962280985
-14.918752472225457 -2.3714692098087595
-11.935001977780324 -2.359534207830979
-9.548001582224288 -2.349986206248755
-7.638401265779464 -2.3423478049829756
-6.1107210126235145 -2.336237083970352
-4.888576810098989 -2.331348507160253
-3.9108614480791175 -2.327437645712174
-3.128689158463253 -2.3243089565537107
-2.502951326770642 -2.3218060052269403
-2.0023610614164724 -2.319803644165524
-1.6018888491333385 -2.3182017553163905
-1.281511079306597 -2.316920244237084
-1.0252088634453287 -2.3158950353736385
-0.8201670907562288 -2.315074868282882
-0.6561336726047671 -2.3144187346102774
-0.5249069380840368 -2.313893827672193
-0.419925550466985 -2.3134739021217263
-0.3359404403737898 -2.3131379616813526
-0.268752352298975 -2.3128692093290537
-0.2150018818393936 -2.3126542074472143
-0.1720015054712007 -2.312482205941743
-0.1376012043770345 -2.312344604737366
-0.11008096350155938 -2.3122345237738644
-0.08806477080128161 -2.312146459003063
-0.07045181664108213 -2.312076007186422
-0.056361453312916865 -2.312019645733109
-0.04508916265030649 -2.311974556570459
-0.03607133012025088 -2.3119384852403386
```

```
-0.028857064096321494 -2.311909628176242
-0.0230856512768014 -2.3118865425249653
-0.01846852102165286 -2.3118680740039435
-0.014774816817066494 -2.3118532991871263
-0.011819853453864937 -2.3118414793336726
-0.009455882762985368 -2.3118320234509095
-0.007564706210459349 -2.311824458744699
-0.006051764968219686 -2.311818406979731
-0.004841411974787491 -2.311813565567756
-0.0038731295797447274 -2.3118096924381764
-0.0030985036637218855 -2.3118065939345125
-0.0024788029310300885 -2.3118041151315816
-0.001983042344846808 -2.3118021320892366
-0.0015864338758433405 -2.3118005456553608
-0.00126914710060076 -2.31179927650826
-0.0010153176805332 01 -2.3117982611905794
-0.0008122541442929787 -2.311797448936435
-0.0006498033154684 89 -2.3117967991331194
-0.0005198426524160027 -2.311796279290467
-0.0004158741217992201 -2.311795863416345
-0.000332699297587169 -2.3117955307170477
-0.00026615943816210574 -2.3117952645576096
-0.00021292755040036582 -2.3117950516300594
-0.0001703420404339795 -2.311794881288019
-0.0001362736322789715 -2.3117947450143865
-0.00010901890571801687 -2.3117946359954806
-8.721512465115211e-05 -2.311794548780356
-6.977209962855113e-05 -2.3117944790082565
-5.581767975826324e-05 -2.3117944231905767
-4.465414388477029e-05 -2.3117943785364328
-3.572331508649995e-05 -2.311794342813118
-2.857865209193733e-05 -2.311794314234466
-2.2862921525756974e-05 -2.3117942913715446
-1.8290337379767152e-05 -2.3117942730812073
-1.4632270023184901e-05 -2.3117942584489373
-1.1705815950335818e-05 -2.3117942467431214
-9.364652811427732e-06 -2.311794237378469
-7.491722101349296e-06 -2.311794229886747
-5.99337784024101e-06 -2.311794223893369
-2.311794223893369
```



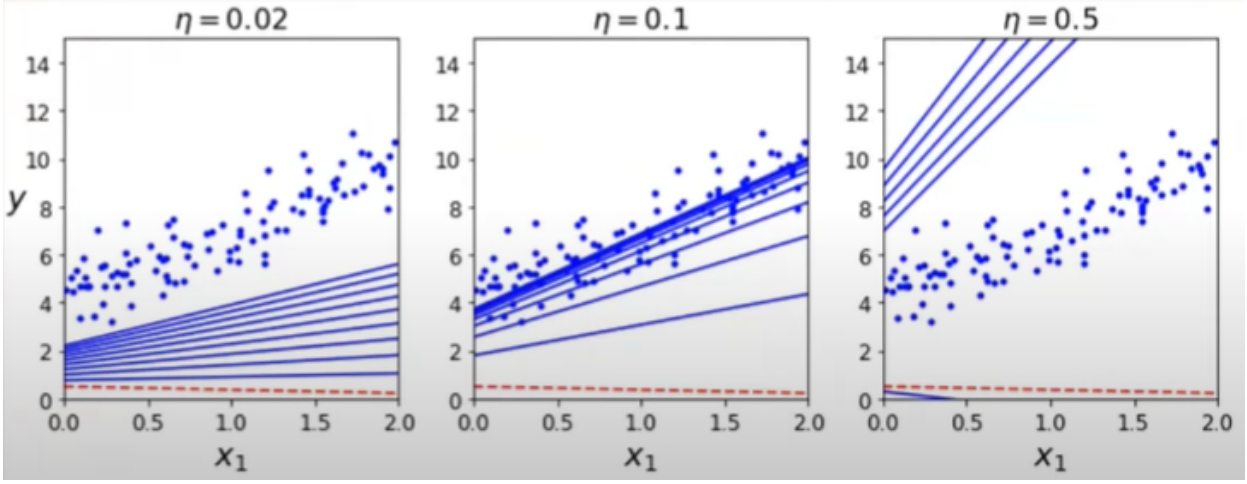
Effect of Learning rate

The learning rate is a hyperparameter that determines the step size at each iteration during the training process of a machine learning algorithm, such as gradient descent. It plays a crucial role in determining how quickly or slowly the algorithm converges to the optimal solution.



The effect of the learning rate on the training process can be summarized as follows:

1. **Large Learning Rate:** A large learning rate can cause the algorithm to take large steps during each iteration. This may result in overshooting the optimal solution, leading to divergence or instability. The algorithm may fail to converge, and the loss function may oscillate or increase instead of decreasing.

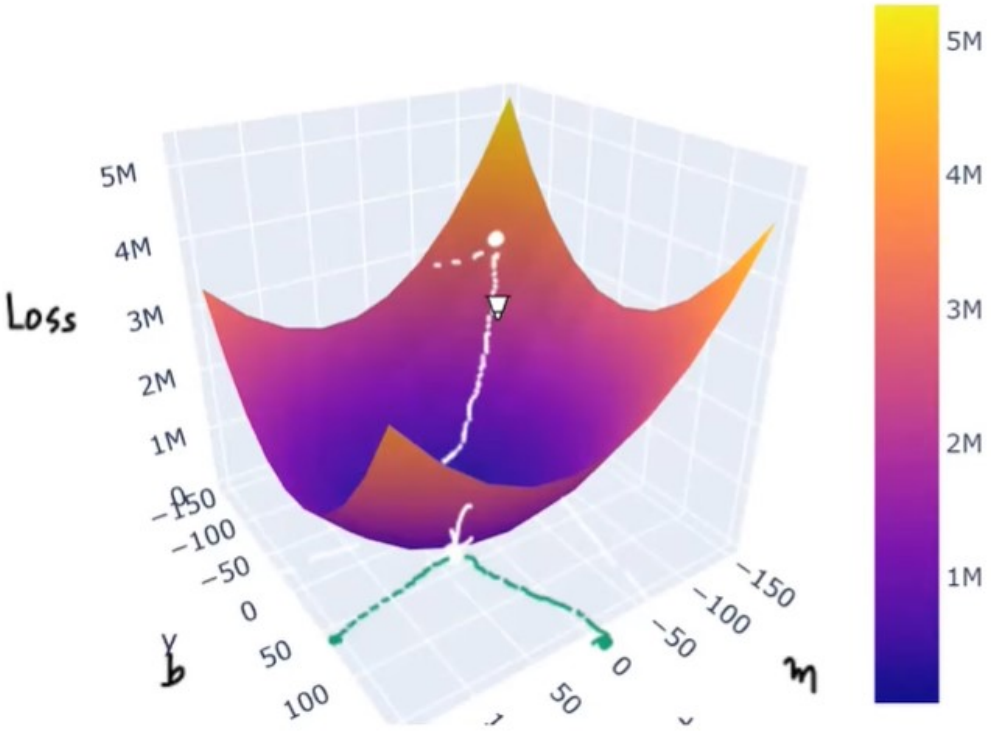


2. **Small Learning Rate:** On the other hand, a small learning rate means taking small steps during each iteration. While this can ensure stability and prevent divergence, it can also result in slow convergence. The algorithm may require a larger number of iterations to reach the optimal solution, increasing the training time.

3. **Appropriate Learning Rate:** An appropriate learning rate strikes a balance between convergence speed and stability. It allows the algorithm to make meaningful progress towards the optimal solution without overshooting or oscillating too much. Finding the optimal learning rate often involves experimentation and tuning. Techniques such as learning rate schedules, adaptive learning rates, or automatic tuning methods can help in finding a suitable learning rate during the training process.

It's important to note that the optimal learning rate depends on various factors, including the specific algorithm, the problem domain, and the dataset. Therefore, it is recommended to experiment with different learning rates to find the one that works best for your specific scenario.

loss function



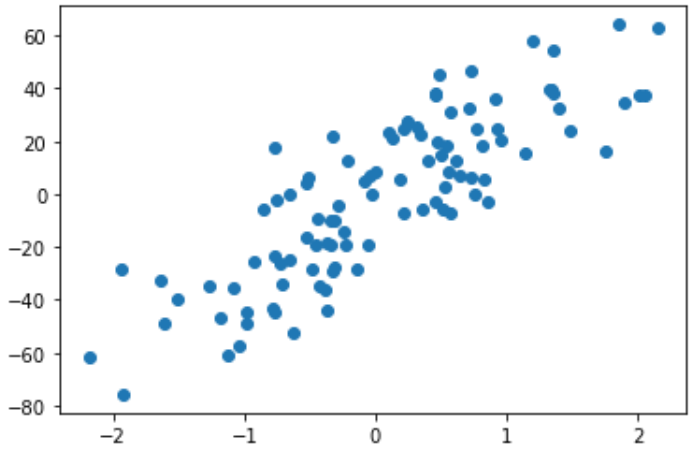
Adding 'm' into the mix

In [60]:
```
from sklearn.datasets import make_regression
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import cross_val_score
```

In [61]:
```
# Focal cell:
X, y = make_regression(
    n_samples=100,
    n_features=1,
    n_informative=1,
    n_targets=1,
    noise=20,
    random_state=13
)
```

In [63]:
```
plt.scatter(X,y)
```

Out[63]: <matplotlib.collections.PathCollection at 0x1d016780a00>



In [64]:
```
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.2,random_state=2)
```

In [65]:
```
from sklearn.linear_model import LinearRegression
```

In [66]:
```
lr = LinearRegression()
```

In [69]:
```
# Fit the linear regression model on the training data
lr.fit(X_train, y_train)

# Print the coefficients of the linear regression model
print(lr.coef_)

# Print the intercept of the linear regression model
print(lr.intercept_)
```

```
[28.12597332]
-2.271014426178382
```

In [68]:
```
 # Predict the target values using the lr model
y_pred = lr.predict(X_test)

# Import the r2_score function from sklearn.metrics
from sklearn.metrics import r2_score

# Calculate the R-squared score between y_test and y_pred
r2_score(y_test, y_pred)
```

Out[68]: 0.6345158782661013

In [72]:
```
class GDRegressor:

    def __init__(self,learning_rate,epochs):
        self.m = 100
        self.b = -120
        self.lr = learning_rate
        self.epochs = epochs

    def fit(self,X,y):
        # calcualte the b using GD
        for i in range(self.epochs):
            loss_slope_b = -2 * np.sum(y - self.m*X.ravel() - self.b)
            loss_slope_m = -2 * np.sum((y - self.m*X.ravel() - self.b)*X.ravel())

            self.b = self.b - (self.lr * loss_slope_b)
            self.m = self.m - (self.lr * loss_slope_m)
        print(self.m,self.b)
```

In [73]:
```
gd =GDRegressor(0.001,100)
```

In [74]:
```
gd.fit(X,y)
```

```
27.828091872608653 -2.2947448944994893
```

In [75]:
```
# Converging in 50 epochs

gd = GDRegressor(0.001,50)
```

In [86]:
```
gd.fit(X_train,y_train) # result
```

```
28.125973319843975 -2.27101442919929
```

In [87]:
```
# Adding Predict function


class GDRegressor:

    def __init__(self,learning_rate,epochs):
        self.m = 100
        self.b = -120
        self.lr = learning_rate
        self.epochs = epochs

    def fit(self,X,y):
        # calcualte the b using GD
        for i in range(self.epochs):
            loss_slope_b = -2 * np.sum(y - self.m*X.ravel() - self.b)
            loss_slope_m = -2 * np.sum((y - self.m*X.ravel() - self.b)*X.ravel())

            self.b = self.b - (self.lr * loss_slope_b)
            self.m = self.m - (self.lr * loss_slope_m)
        print(self.m,self.b)

    def predict(self,X):
        return self.m * X + self.b
```

In [88]:
```
gd = GDRegressor(0.001,50)
```

In [89]:
```
gd.fit(X_train,y_train)
```

```
28.159367347119066 -2.3004574196824854
```

In [90]:
```
gd.predict(X_test)
```

Out[90]:
```
array([[ 21.09732023],
       [ 18.02962184],
       [ 18.23238093],
       [ -8.13929374],
       [ 15.71827044],
       [ 58.2529665 ],
       [-15.07783741],
       [-11.37125581],
       [-15.85557195],
       [ -8.84542293],
       [-23.62986105],
       [ 14.77224759],
       [ 12.68984073],
       [-23.94450805],
       [-22.36092777],
       [-16.71707058],
       [ 24.53064978],
       [-17.03243073],
       [ -3.92053306],
       [-12.06068528]])
```

In [91]:
```
y_pred = gd.predict(X_test)
```

In [83]:
```
from sklearn.metrics import r2_score
r2_score(y_test,y_pred)
```

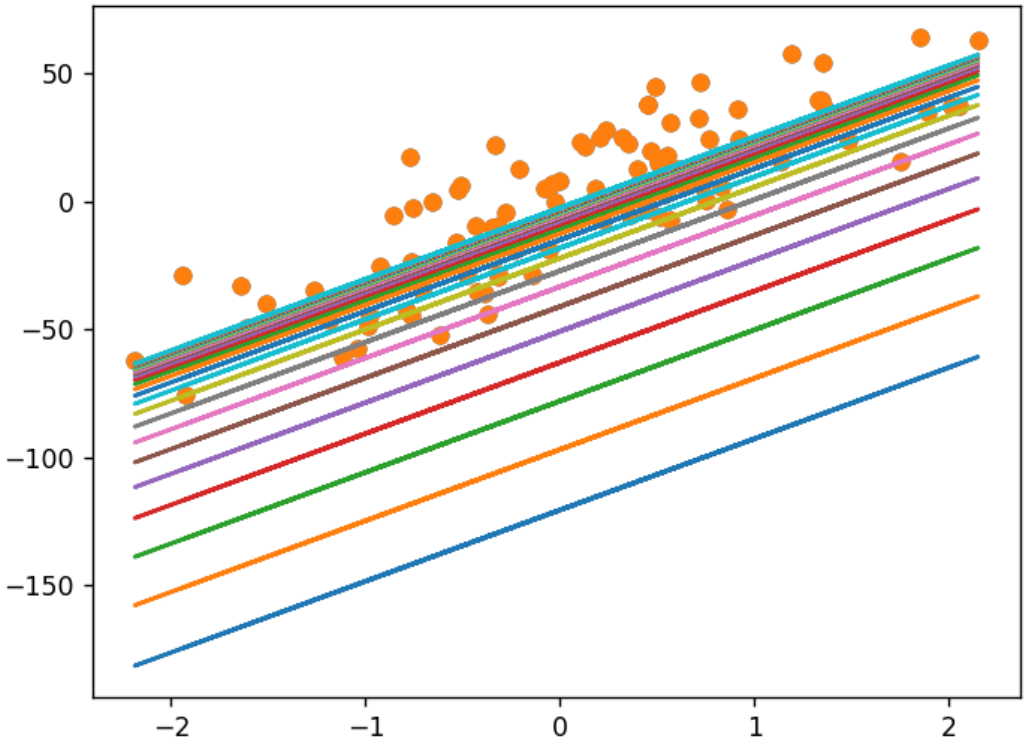
Out[83]:
```
0.6343842836315579
```

Animation code

In [93]:
```
%matplotlib notebook
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
import matplotlib.animation as animation
```

In [94]:
```
# Importing the necessary module
from sklearn.datasets import make_regression

# Generating the dataset for regression
X, y = make_regression(
    n_samples=100,
    n_features=1,
    n_informative=1,
    n_targets=1,
    noise=20,
    random_state=13
)
```

In [95]:
```
# Plotting a scatter plot
plt.scatter(X, y)

# Displaying the plot
plt.show()
```



In [96]:
```
from sklearn.linear_model import LinearRegression

# Create a LinearRegression object
reg = LinearRegression()

# Fit the model to the data
reg.fit(X, y)

# Print the coefficients
print(reg.coef_)

# Print the intercept
print(reg.intercept_)
```

```
[27.82809103]
-2.29474455867698
```

In [97]:
```
b = -150
m = 27.82
lr = 0.001
all_b = []
all_cost = []

epochs = 30

for i in range(epochs):
    slope = 0
    cost = 0
    for j in range(X.shape[0]):
        slope = slope - 2*(y[j] - (m * X[j]) -  b)
        cost = cost + (y[j] - m * X[j] -b) ** 2

    b = b - (lr * slope)
    all_b.append(b)
    all_cost.append(cost)
    y_pred = m * X + b
    plt.plot(X,y_pred)
plt.scatter(X,y)
```

Out[97]:
```
<matplotlib.collections.PathCollection at 0x1d016f5f1f0>
```

In [98]:
```
# Flatten the array 'all_b'
all_b = np.array(all_b).ravel()
```

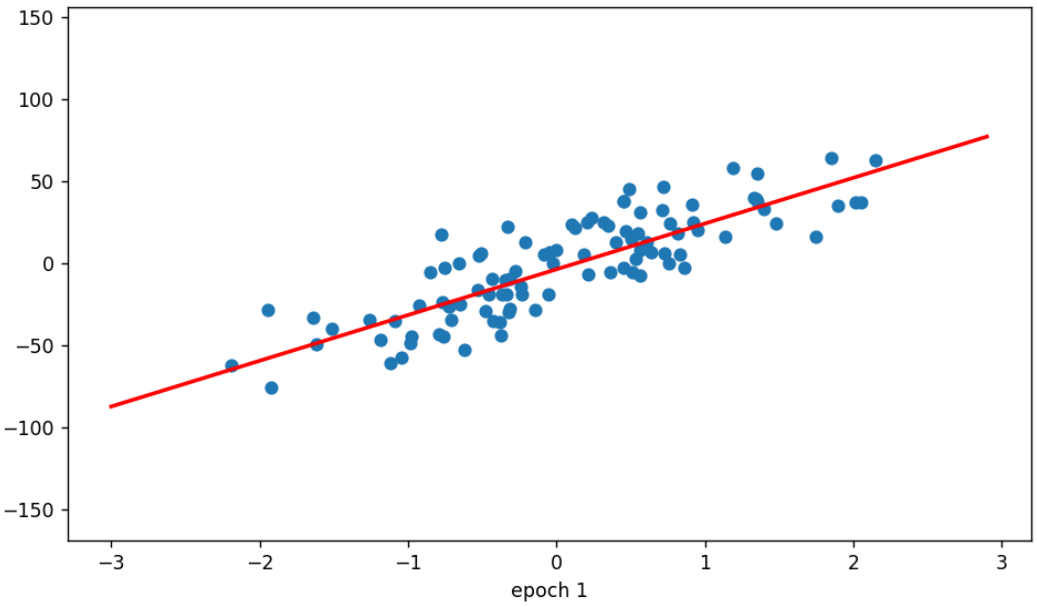
In [99]:
```
all_b
```

Out[99]:
```
array([-120.4588544 ,  -96.82593791,  -77.91960473,  -62.79453818,
        -50.69448494,  -41.01444235,  -33.27040827,  -27.07518102,
        -22.11899921,  -18.15405376,  -14.98209741,  -12.44453232,
        -10.41448025,   -8.7904386 ,   -7.49120528,   -6.45181862,
         -5.62030929,   -4.95510183,   -4.42293586,   -3.99720308,
         -3.65661686,   -3.38414789,   -3.16617271,   -2.99179256,
         -2.85228845,   -2.74068515,   -2.65140252,   -2.57997641,
         -2.52283553,   -2.47712282])
```

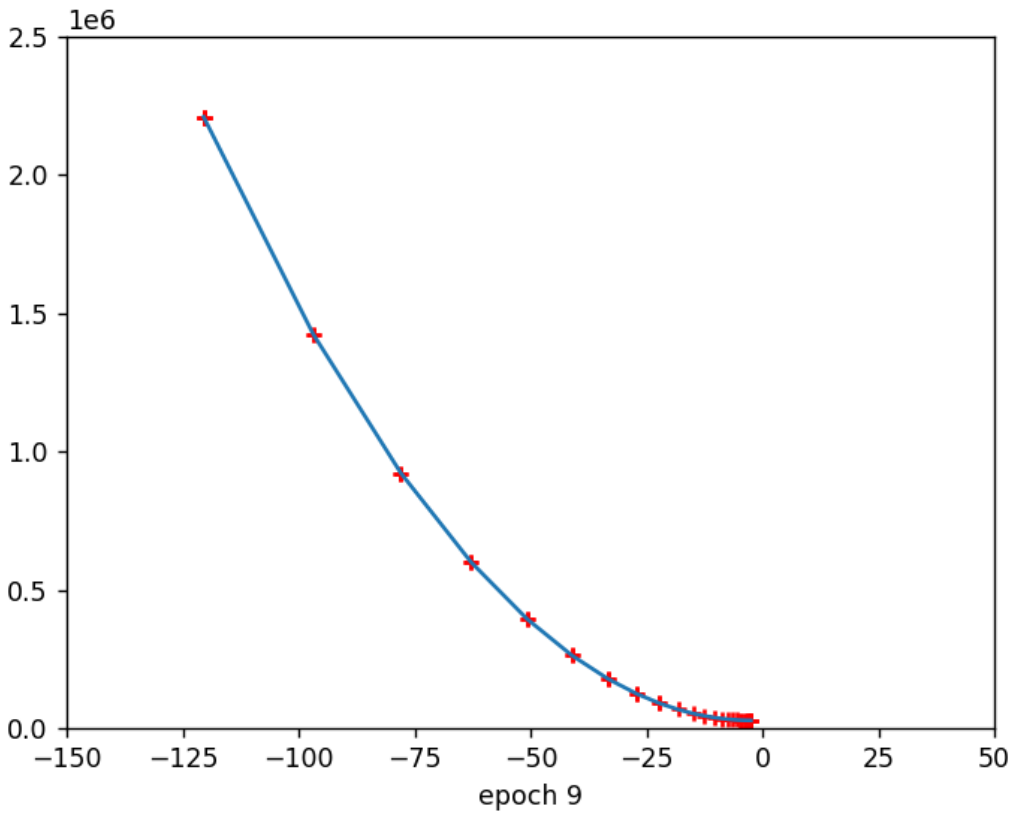
In [100]:
```
# Flatten the 'all_cost' array
all_cost = np.array(all_cost).ravel()
all_cost
```

Out[100]:
```
array([2210040.49020261, 1424629.13499295,  921965.86765877,
        600261.37656489,  394370.50226481,  262600.34271276,
        178267.44059944,  124294.38324692,   89751.62654131,
         67644.26224972,   53495.5491031 ,   44440.37268926,
         38645.05978441,   34936.0595253 ,   32562.29935947,
         31043.09285334,   30070.80068942,   29448.5337045 ,
         29050.28283416,   28795.40227714,   28632.27872065,
         28527.87964449,   28461.06423575,   28418.30237416,
         28390.93478274,   28373.41952423,   28362.20975879,
         28355.0355089 ,   28350.44398898,   28347.50541622])
```

In [101]:
```
fig, ax = plt.subplots(figsize=(9,5))
#fig.set_tight_layout(True)

x_i = np.arange(-3, 3, 0.1)
y_i = x_i*m -150
ax.scatter(X, y)
line, = ax.plot(x_i, x_i*50 - 4, 'r-', linewidth=2)

def update(i):
    label = 'epoch {0}'.format(i + 1)
    line.set_ydata(x_i*m + all_b[i])
    ax.set_xlabel(label)
    # return line, ax

anim = FuncAnimation(fig, update, repeat=True, frames=epochs, interval=500)

#f = r"animation.gif"
#writergif = animation.PillowWriter(fps=2)
#anim.save(f, writer=writergif)
```



In [102]:
```
import matplotlib.pyplot as plt
import matplotlib.animation
import numpy as np

fig, ax = plt.subplots()
ax.plot(all_b, all_cost)
x, y = [],[]
sc = ax.scatter(x,y,color='red',marker='+')
plt.xlim(-150,50)
plt.ylim(0,2500000)

def animate(i):
    label = 'epoch {0}'.format(i + 1)
    x.append(all_b[i])
    y.append(all_cost[i])
    sc.set_offsets(np.c_[x,y])
    ax.set_xlabel(label)

ani = matplotlib.animation.FuncAnimation(fig, animate,
                frames=30, interval=500, repeat=True)
plt.show()

f = r"animation3.gif"
writergif = animation.PillowWriter(fps=2)
ani.save(f, writer=writergif)
```



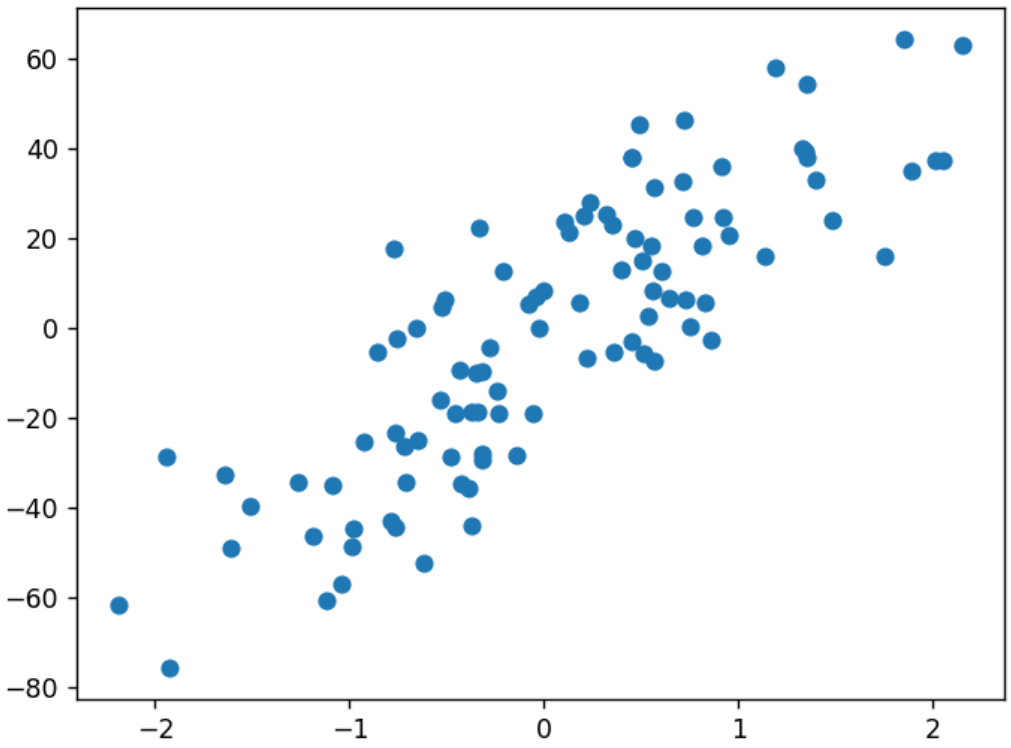
Gradient-descent-animation(both-m-and-b)

In [104]:
```
from sklearn.datasets import make_regression

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

In [105]:
```
%matplotlib notebook
from matplotlib.animation import FuncAnimation
import matplotlib.animation as animation
```

In [106]:
```
X,y = make_regression(n_samples=100, n_features=1, n_informative=1, n_targets=1,noise=2(
```

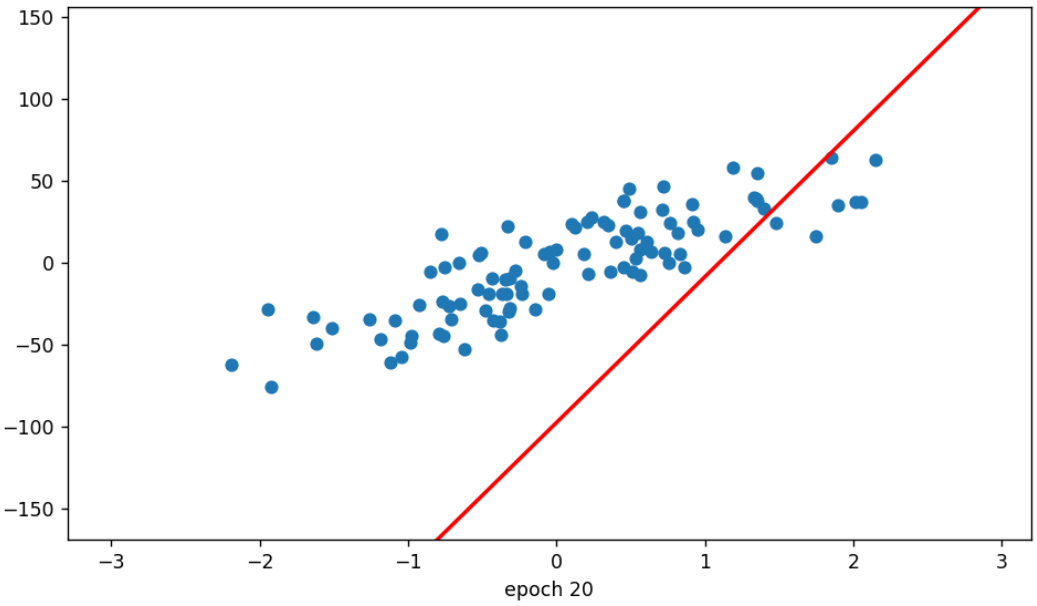
In [107]:
```
plt.scatter(X,y)
```



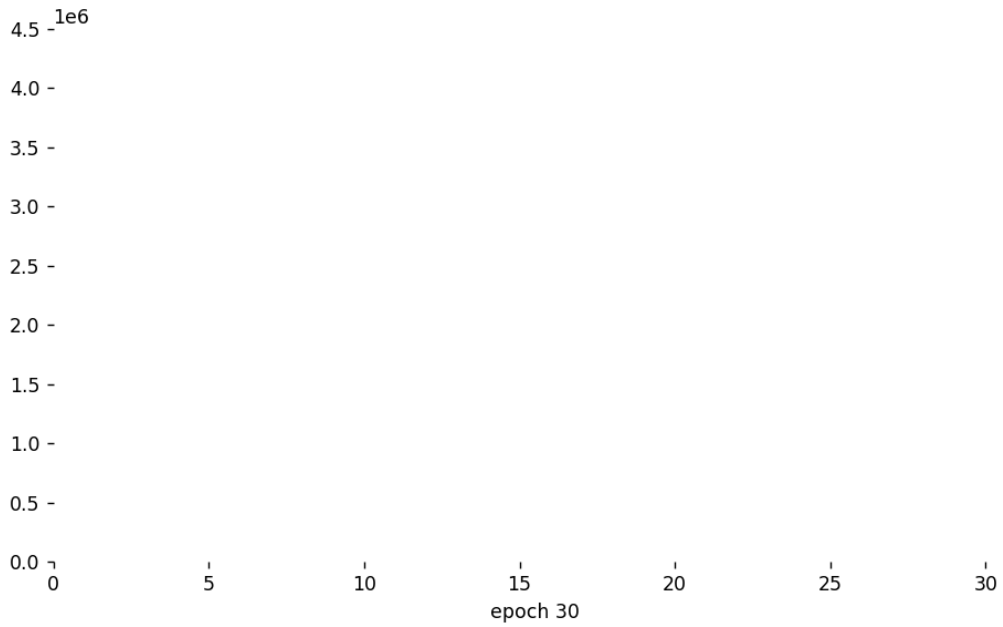
Out[107]: <matplotlib.collections.PathCollection at 0x1d0170f36d0>

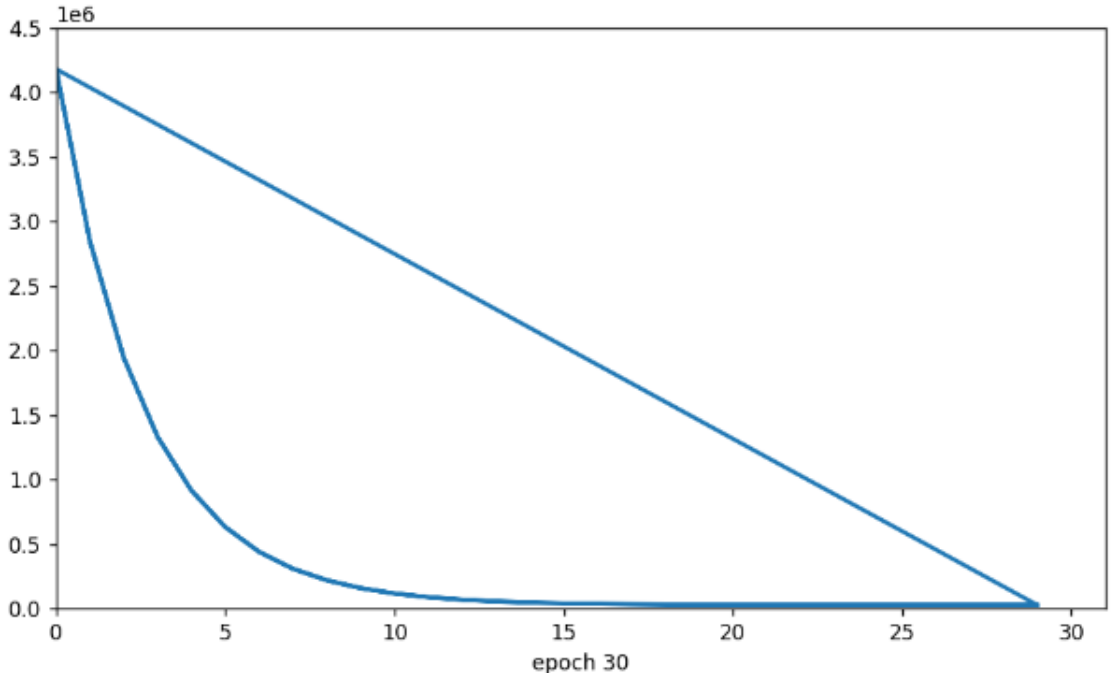
In [114]:
```
b = -120
m = 100
lr = 0.001
all_b = []
all_m = []
all_cost = []

epochs = 30

for i in range(epochs):
    slope_b = 0
    slope_m = 0
    cost = 0
    for j in range(X.shape[0]):
        slope_b = slope_b - 2*(y[j] - (m * X[j]) -  b)
        slope_m = slope_m - 2*(y[j] - (m * X[j]) -  b)*X[j]
        cost = cost + (y[j] - m * X[j] -b) ** 2

    b = b - (lr * slope_b)
    m = m - (lr * slope_m)
    all_b.append(b)
    all_m.append(m)
    all_cost.append(cost)
```

In [115]:
```
fig, ax = plt.subplots(figsize=(9,5))
#fig.set_tight_layout(True)

x_i = np.arange(-3, 3, 0.1)
y_i = x_i*(-27) -150
ax.scatter(X, y)
line, = ax.plot(x_i, x_i*50 - 4, 'r-', linewidth=2)

def update(i):
    label = 'epoch {0}'.format(i + 1)
    line.set_ydata(x_i*all_m[i] + all_b[i])
    ax.set_xlabel(label)
    # return line, ax

anim = FuncAnimation(fig, update, repeat=True, frames=epochs, interval=500)

#f = r"animation4.gif"
#writergif = animation.PillowWriter(fps=2)
#anim.save(f, writer=writergif)
```

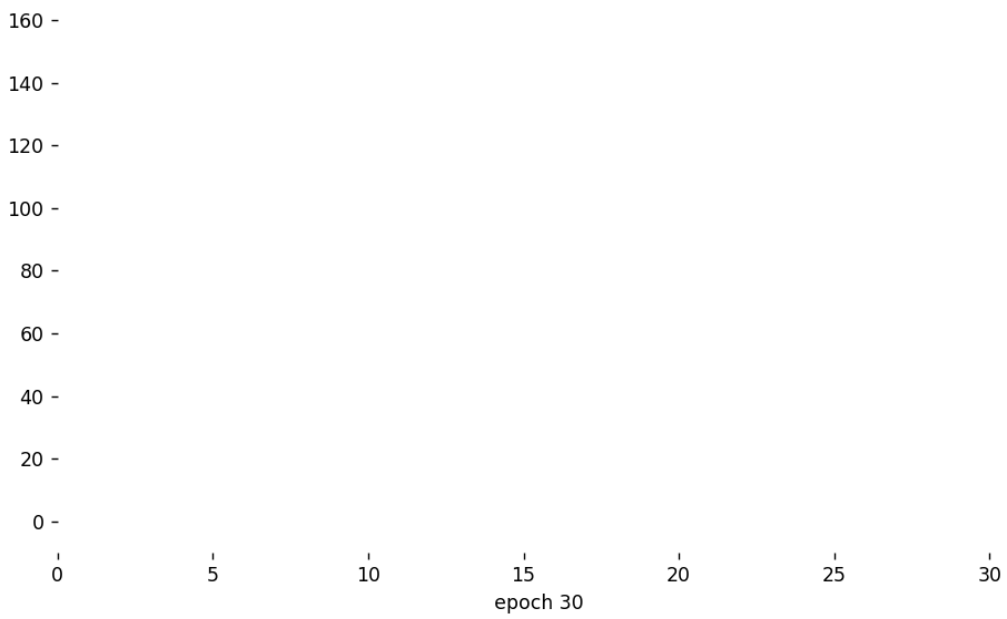


In [116]:
```
# Cost function
# creating a blank window
# for the animation
num_epochs = list(range(0,30))
fig = plt.figure(figsize=(9,5))
axis = plt.axes(xlim =(0, 31), ylim =(0, 4500000))

line, = axis.plot([], [], lw = 2)

xdata, ydata = [], []

# animation function
def animate(i):
    label = 'epoch {0}'.format(i + 1)
    xdata.append(num_epochs[i])
    ydata.append(all_cost[i])
    line.set_data(xdata, ydata)
    axis.set_xlabel(label)

    return line,

# calling the animation function
anim = animation.FuncAnimation(fig, animate, frames = 30,repeat=False, interval = 500)

f = r"animation5.gif"
writergif = animation.PillowWriter(fps=2)
anim.save(f, writer=writergif)
```




1e6
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
5
10
15
20
25
30
epoch 30

In [117]:
```
# intercept graph
num_epochs = list(range(0,30))
fig = plt.figure(figsize=(9,5))
axis = plt.axes(xlim =(0, 31), ylim =(-10, 160))

line, = axis.plot([], [], lw = 2)

xdata, ydata = [], []

# animation function
def animate(i):
    label = 'epoch {0}'.format(i + 1)
    xdata.append(num_epochs[i])
    ydata.append(all_b[i])
    line.set_data(xdata, ydata)
    axis.set_xlabel(label)

    return line,

# calling the animation function
anim = animation.FuncAnimation(fig, animate, frames = 30,repeat=False, interval = 500)

f = r"animation6.gif"
writergif = animation.PillowWriter(fps=2)
anim.save(f, writer=writergif)
```



Gradient-descent-3d

In [122]:
```
from sklearn.datasets import make_regression
import matplotlib.pyplot as plt
import numpy as np
```

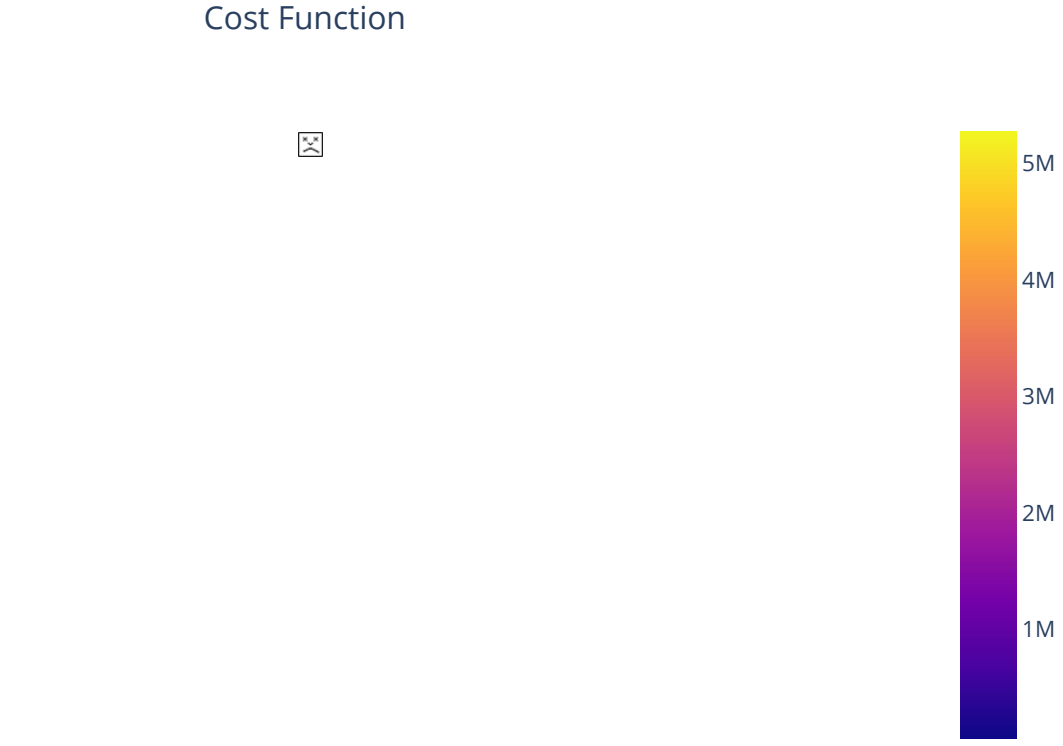
In [123]:
```
X,y = make_regression(n_samples=100, n_features=1, n_informative=1, n_targets=1,noise=2
```

In [124]:
```
plt.scatter(X,y)
```

Out[124]:
```
<matplotlib.collections.PathCollection at 0x1d019b22610>
```

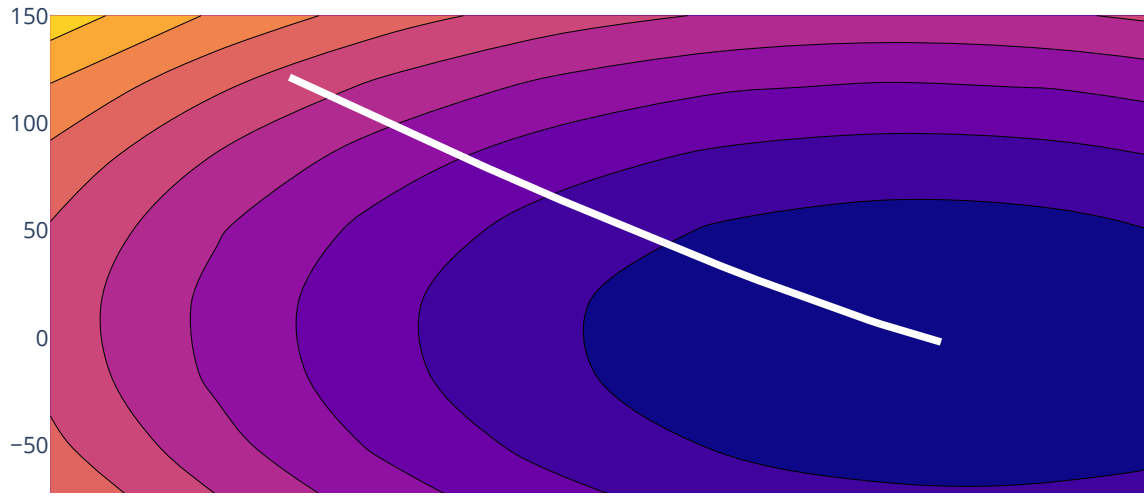
In [125]:
```
m_arr = np.linspace(-150, 150, 10)
b_arr = np.linspace(-150, 150, 10)
mGrid, bGrid = np.meshgrid(m_arr,b_arr)

final = np.vstack((mGrid.ravel().reshape(1,100),bGrid.ravel().reshape(1,100))).T

z_arr = []

for i in range(final.shape[0]):
    z_arr.append(np.sum((y - final[i,0]*X.reshape(100) - final[i,1])**2))

z_arr = np.array(z_arr).reshape(10,10)
```

In [127]:
```
fig.show()

fig.write_html("cost_function.html")
b = 150
m = -127.82
lr = 0.001
all_b = []
all_m = []
all_cost = []

epochs = 30

for i in range(epochs):
    slope_b = 0
    slope_m = 0
    cost = 0
    for j in range(X.shape[0]):
        slope_b = slope_b - 2*(y[j] - (m * X[j]) -  b)
        slope_m = slope_m - 2*(y[j] - (m * X[j]) -  b)*X[j]
        cost = cost + (y[j] - m * X[j] -b) ** 2

    b = b - (lr * slope_b)
    m = m - (lr * slope_m)
    all_b.append(b)
    all_m.append(m)
    all_cost.append(cost)
```

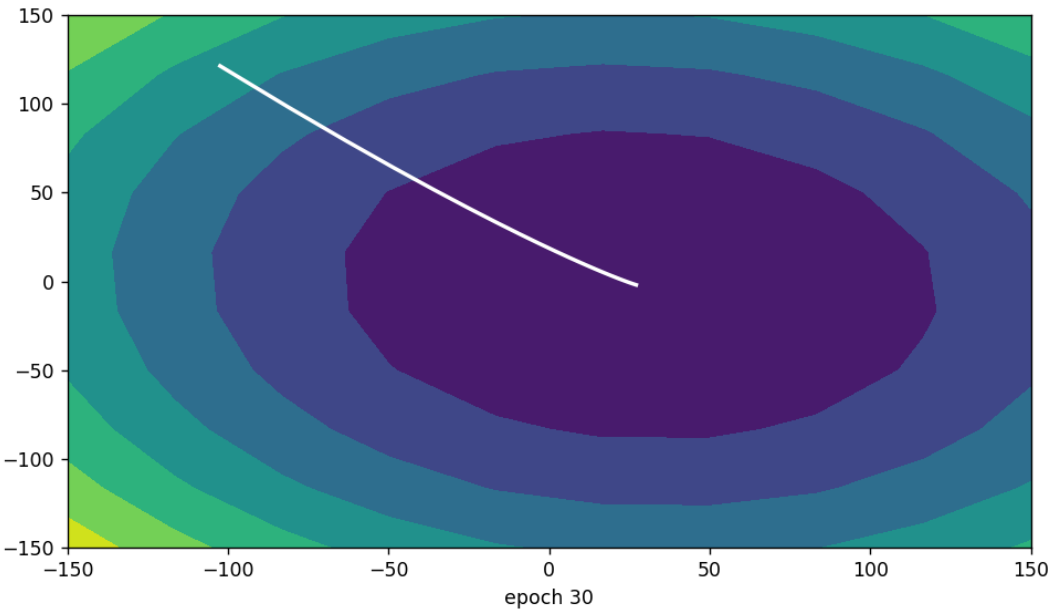


In [128]:
```
import plotly.express as px

fig = px.scatter_3d(x=np.array(all_m).ravel(), y=np.array(all_b).ravel(), z=np.array(al

fig.add_trace(go.Surface(x = m_arr, y = b_arr, z =z_arr*100 ))

fig.show()

fig.write_html("cost_function2.html")
```

In [129]:
```
import plotly.graph_objects as go

fig = go.Figure(go.Scatter(x=np.array(all_m).ravel(), y=np.array(all_b).ravel(), name='
                           line=dict(color='#fff', width=4)))

fig.add_trace(go.Contour(z=z_arr,x=m_arr,y=b_arr))
fig.show()
```



In [131]:
```
%matplotlib notebook
from matplotlib.animation import FuncAnimation
import matplotlib.animation as animation

# intercept graph
num_epochs = list(range(0,30))
fig = plt.figure(figsize=(9,5))
axis = plt.axes(xlim =(-150, 150), ylim =(-150, 150))

axis.contourf(m_arr, b_arr, z_arr)

line, = axis.plot([], [], lw = 2,color='white')

xdata, ydata = [], []

# animation function
def animate(i):
    label = 'epoch {0}'.format(i + 1)
    xdata.append(all_m[i])
    ydata.append(all_b[i])
    line.set_data(xdata, ydata)
    axis.set_xlabel(label)

    return line,

# calling the animation function
anim = animation.FuncAnimation(fig, animate, frames = 30,repeat=False, interval = 500)

#f = r"animation8.gif"
#writergif = animation.PillowWriter(fps=2)
#anim.save(f, writer=writergif)
```



Pratical Learning rate

1. When we have Small Learning rate

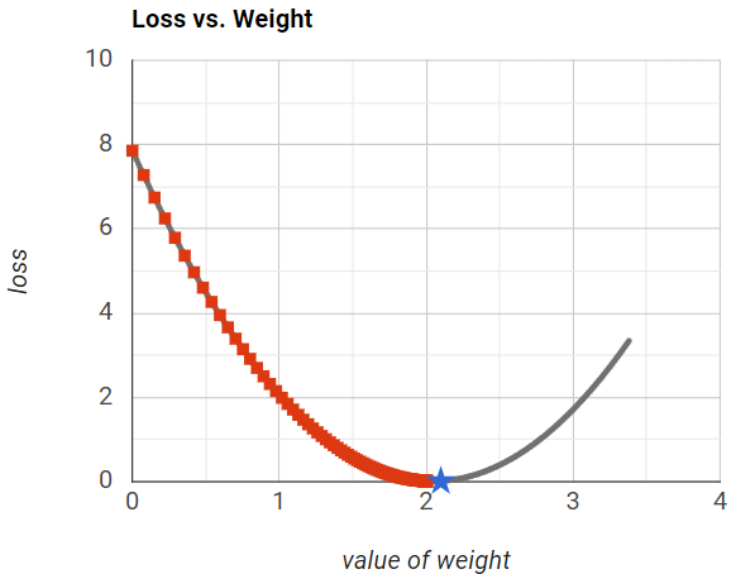
Experiment with different learning rates and see how they affect the number of steps required to reach the minimum of the loss curve. Try the exercises below the graph.

Set learning rate: 0.01

Execute single step: Step 103

Reset the graph: Reset

⚠ Gradient descent is taking too long to reach the minimum loss. Try again with a different learning rate.



2. When Learning rate is normal

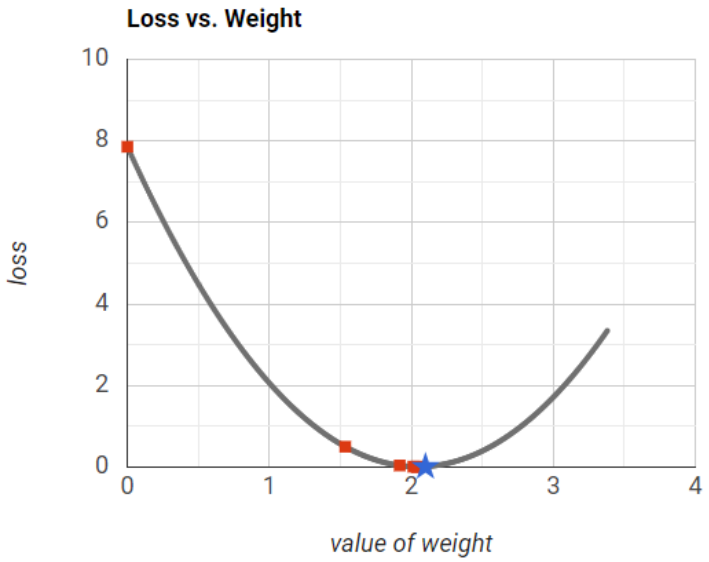
Experiment with different learning rates and see how they affect the number of steps required to reach the minimum of the loss curve. Try the exercises below the graph.

Set learning rate: 0.20

Execute single step: Step 6

Reset the graph: Reset

✓ This model has reached minimal loss. Is it possible to achieve similar results with fewer steps?

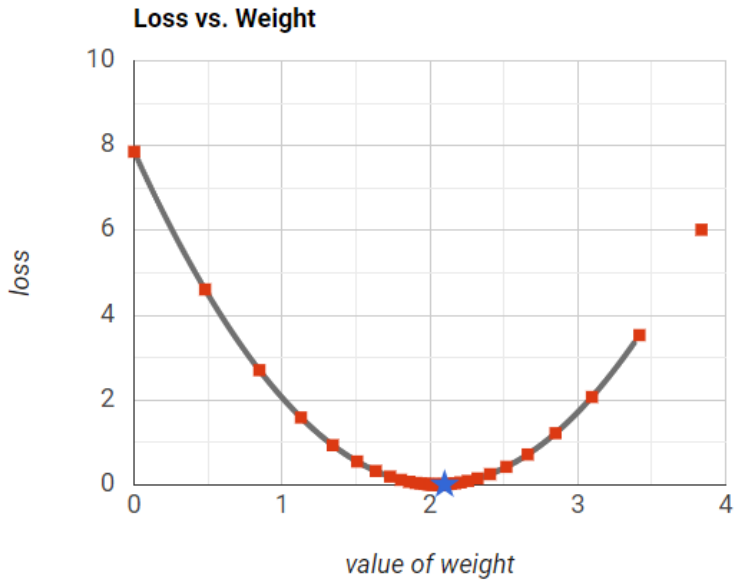


3. When Learning rate is High

Experiment with different learning rates and see how they affect the number of steps required to reach the minimum of the loss curve. Try the exercises below the graph.

Set learning rate:		0.50
Execute single step:	Step	37
Reset the graph:	Reset	

✓ This model has reached minimal loss. Is it possible to achieve similar results with fewer steps?



4.When Learning rate is Too High

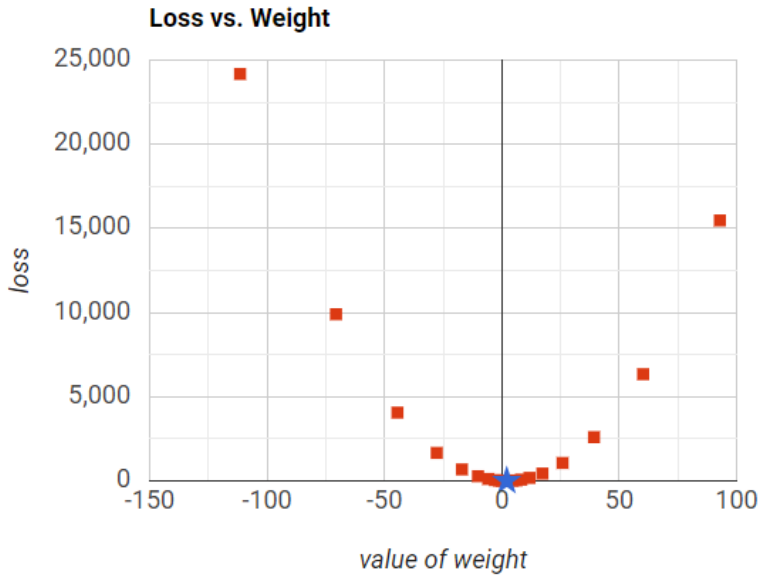
Experiment with different learning rates and see how they affect the number of steps required to reach the minimum of the loss curve. Try the exercises below the graph.

Set learning rate: 0.60

Execute single step: Step 18

Reset the graph: Reset

⚠ Gradient descent will never reach the minimum loss. Try again with a different learning rate.



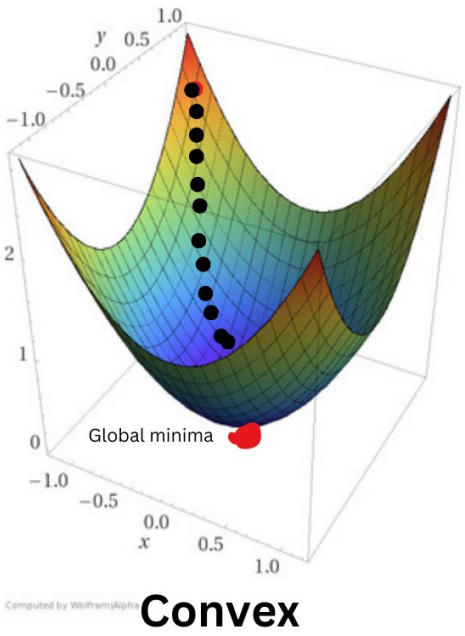
https://developers.google.com/machine-learning/crash-course/fitter/graph (https://developers.google.com/machine-learning/crash-course/fitter/graph)

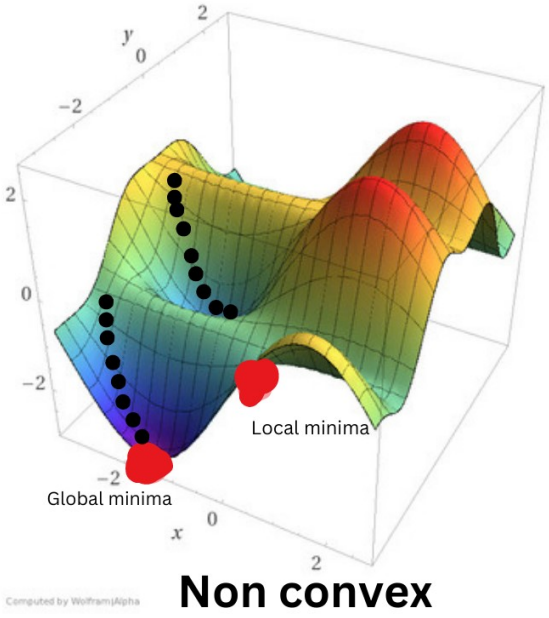
Convex and Non convex initializations

In the context of gradient descent, the concepts of convex and non-convex are related to the optimization landscape and the behavior of the algorithm during the training process. Let's explore each concept:

1. **Convex Initialization:** Convex initialization refers to starting the optimization process with initial parameter values that result in a convex optimization landscape. In a convex landscape, the objective function is a convex function, meaning that it has a single global minimum and no local minima. Convex optimization landscapes are desirable because gradient descent is guaranteed to converge to the global minimum in such cases, regardless of the initialization. Convex initialization can lead to faster and more stable convergence.
2. **Non-convex Initialization:** Non-convex initialization, on the other hand, refers to starting the optimization process with initial parameter values that result in a non-convex optimization landscape. In a non-convex landscape, the objective function can have multiple local minima, making the

optimization problem more challenging. Non-convex initialization can lead to slower convergence and

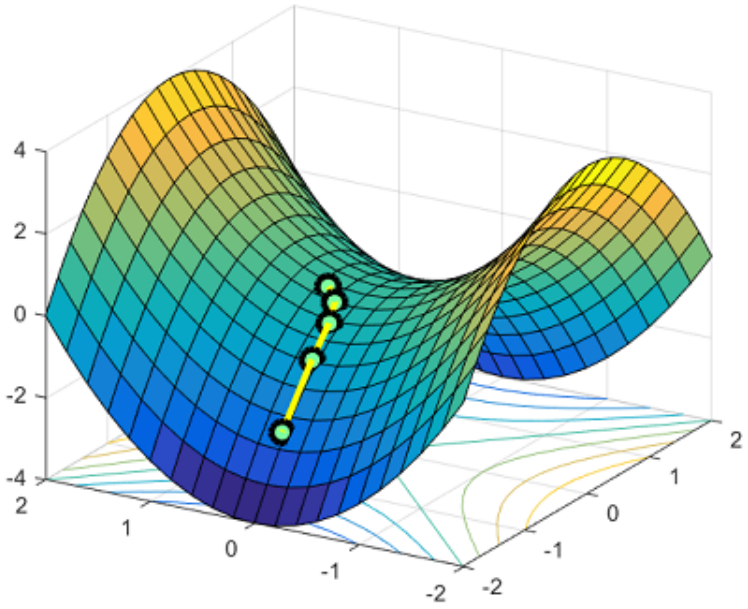




Saddle Point

A saddle point is a critical point in the optimization landscape where the gradients are zero, but it is not a local minimum or maximum. At a saddle point, the objective function may have a flat region in some dimensions and a steep region in others. Saddle points can pose challenges for gradient descent because the algorithm may converge slowly in the flat regions or get stuck due to the presence of zero gradients. In high-dimensional spaces, saddle points are more prevalent than local minima.

- The interplay between non-convex initialization and saddle points can impact the behavior of gradient descent. If the optimization landscape contains many saddle points, gradient descent may get trapped in these regions, leading to slower convergence. However, it's important to note that not all saddle points are problematic. In fact, saddle points can serve as critical points where the algorithm can explore different directions and potentially escape suboptimal solutions.
- Several techniques have been developed to address the challenges associated with non-convex optimization landscapes and saddle points. These include using advanced optimization algorithms, such as momentum-based methods, adaptive learning rate techniques, or second-order optimization methods. Additionally, initialization strategies, such as random initialization, can help to escape saddle points and find better solutions.
- In practice, the specific behavior of gradient descent with respect to convex and non-convex initializations and saddle points depends on the problem, the data, and the optimization algorithm used. Exploring different initialization strategies and optimizing techniques can help improve the convergence and performance of gradient descent in non-convex landscapes.



In mathematics, a local minimum of a function is a point where the function value is smaller than at nearby points, but possibly greater than at a distant point. A global minimum is a point where the function value is smaller than at all other feasible points.

A saddle point of a function is a point where the function value is higher in some directions than in others. Saddle points are not minima or maxima, but they can be important in optimization problems.

Here is a table that summarizes the differences between local and global minima and saddle points:

Feature	Local minimum	Global minimum	Saddle point
Function value	Smaller than nearby points	Smaller than all other feasible points	Higher in some directions than in others
Stability	Stable	Stable	Unstable
Importance in optimization	Can be a good starting point for optimization algorithms	The goal of optimization algorithms	Can be a local optimum

Here are some examples of local and global minima and saddle points:

- The function $f(x) = x^2$ has a global minimum at $x = 0$.
- The function $f(x) = x^3$ has a local minimum at $x = 0$ and a global minimum at $x = -\infty$.
- The function $f(x, y) = x^2 + y^2$ has a saddle point at $(0, 0)$.

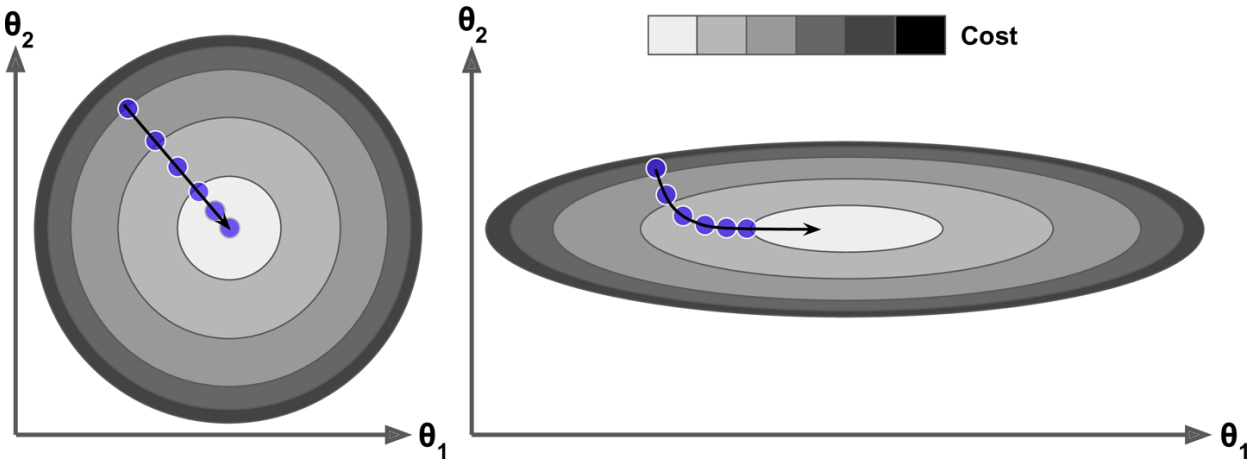
In machine learning, local and global minima and saddle points are important concepts in the field of optimization. Optimization algorithms are used to find the minimum of a function, and they can get stuck in local minima or saddle points. There are a number of techniques that can be used to avoid local minima and saddle points, such as using multiple starting points or using a more sophisticated optimization algorithm.

Effect of Data

Data scaling is the process of normalizing the range of features in a dataset. This is done to ensure that all features have a similar scale, which can help to improve the performance of machine learning algorithms.

In gradient descent, the step size of the algorithm is determined by the scale of the features. If the features are not scaled, the step size may be too large for some features and too small for others. This can lead to the algorithm converging to a suboptimal solution.

Scaling the features can help to improve the performance of gradient descent in a number of ways. First, it can help to ensure that the algorithm converges to a global minimum rather than a local minimum. Second, it can help to improve the stability of the algorithm, making it less likely to diverge or oscillate. Third, it can help to improve the accuracy of the model, making it more likely to generalize well to new data.

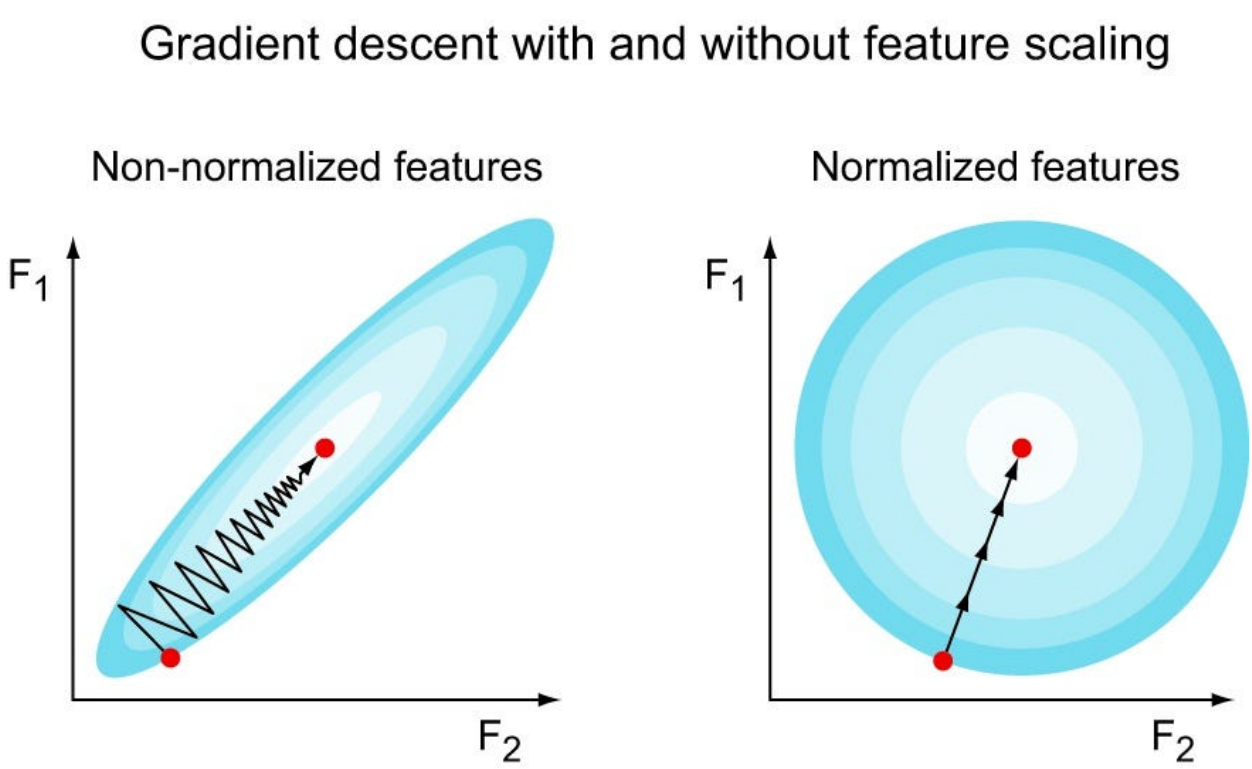


There are two main methods of scaling data: **Normalization** and **Standardization**.

Normalization is the process of transforming the data so that it has a mean of 0 and a standard deviation of 1. This can be done by subtracting the mean from each feature and then dividing by the standard deviation.

Standardization is the process of transforming the data so that it has a mean of 0 and a standard deviation of 1, but it also centers the data around the median. This can be done by subtracting the median from each feature and then dividing by the interquartile range.

The best method of scaling data depends on the specific machine learning algorithm that is being used. However, in general, normalization is a good choice for most algorithms.



Here are some of the effects of data scaling on gradient descent:

- **Convergence speed:** Scaling the features can help to improve the convergence speed of gradient descent. This is because the step size of the algorithm will be more consistent when the features are scaled.
- **Accuracy:** Scaling the features can help to improve the accuracy of the model. This is because the model will be less sensitive to noise when the features are scaled.
- **Stability:** Scaling the features can help to improve the stability of the algorithm. This is because the algorithm will be less likely to diverge or oscillate when the features are scaled.

Overall, data scaling can be a helpful technique for improving the performance of gradient descent. If you are using gradient descent to train a machine learning model, it is a good idea to try scaling the data and see if it improves the performance of the model.

In [ ]: