Stochastic gradient descent (often abbreviated **SGD**) is an iterative method for optimizing an objective function with suitable smoothness properties (e.g. differentiable or subdifferentiable). It can be regarded as a stochastic approximation of gradient descent optimization, since it replaces the actual gradient (calculated from the entire data set) by an estimate thereof (calculated from a randomly selected subset of the data). Especially in high-dimensional optimization problems this reduces the computational burden, achieving faster iterations in trade for a lower convergence rate.[1]

Stochastic gradient descent is a very popular and common algorithm used in various Machine Learning algorithms, most importantly forms the basis of Neural Networks.

Stochastic Gradient Descent (SGD) is a simple yet very efficient approach to fitting linear classifiers and regressors under convex loss functions such as (linear) Support Vector Machines and Logistic Regression. Even though SGD has been around in the machine learning community for a long time, it has received a considerable amount of attention just recently in the context of large-scale learning.

SGD has been successfully applied to large-scale and sparse machine learning problems often encountered in text classification and natural language processing. Given that the data is sparse, the classifiers in this module easily scale to problems with more than 10^5 training examples and more than 10^5 features.

Strictly speaking, SGD is merely an optimization technique and does not correspond to a specific family of machine learning models. It is only a *way* to train a model. Often, an instance of `SGDClassifier` or `SGDRegressor` will have an equivalent estimator in the scikit-learn API, potentially using a different optimization technique. For example, using `SGDClassifier(loss='log')` results in logistic regression, i.e. a model equivalent to `LogisticRegression` which is fitted via SGD instead of being fitted by one of the other solvers in `LogisticRegression`. Similarly, `SGDRegressor(loss='squared_loss', penalty='l2')` and `Ridge` solve the same optimization problem, via different means.

The advantages of Stochastic Gradient Descent are:

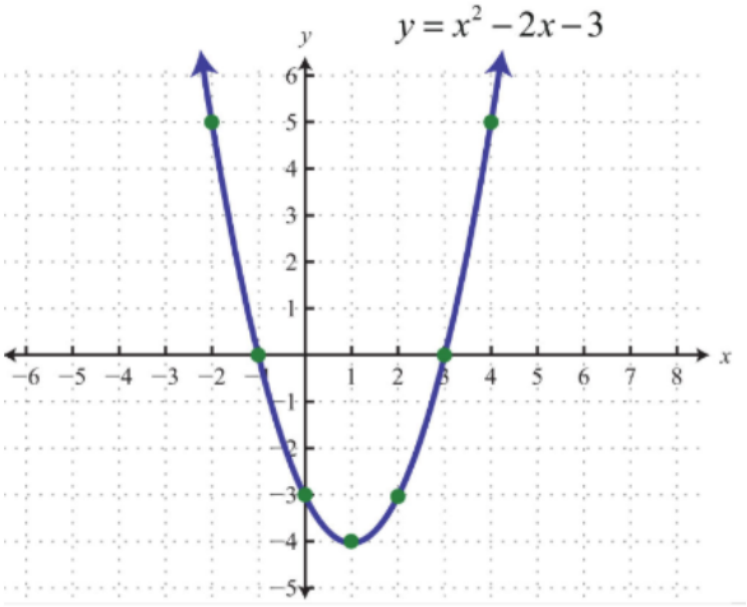
- Efficiency.
- Ease of implementation (lots of opportunities for code tuning).

The disadvantages of Stochastic Gradient Descent include:

- SGD requires a number of hyperparameters such as the regularization parameter and the number of iterations.
- SGD is sensitive to feature scaling.

What is the objective of Gradient Descent?

Gradient, in plain terms means slope or slant of a surface. So gradient descent literally means descending a slope to reach the lowest point on that surface. Let us imagine a two dimensional graph, such as a parabola in the figure below.



https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html#sklearn.linear_model.SGDClassifier

https://towardsdatascience.com/regression-explained-in-simple-terms-dccbcad96f61

https://towardsdatascience.com/stochastic-gradient-descent-clearly-explained-53d239905d31

In the above graph, the lowest point on the parabola occurs at x = 1. The objective of gradient descent algorithm is to find the value of "x" such that "y" is minimum. "y" here is termed as the objective function that the gradient descent algorithm operates upon, to descend to the lowest point.

Gradient Descent - the Algorithm

- Here we use linear regression problem to explain gradient descent algorithm.
- The objective of regression is to minimize the sum of squared residuals.
- We know that a function reaches its minimum value when the slope is equal to 0.
- By using this technique, we solved the linear regression problem and learnt the weight vector. The same problem can be solved by gradient descent technique.

"Gradient descent is an iterative algorithm, that starts from a random point on a function and travels down its slope in steps until it reaches the lowest point of that function."

This algorithm is useful in cases where the optimal points cannot be found by equating the slope of the function to 0. In the case of linear regression, you can mentally map the sum of squared residuals as the function "y" and the weight vector as "x" in the parabola above.

How to move down in steps?

This is the crux of the algorithm. The general idea is to start with a random point (in our parabola example start with a random "x") and find a way to update this point with each iteration such that we descend the slope.

The steps of the algorithm are

1. Find the slope of the objective function **with respect to each parameter/feature**. In other words, compute the gradient of the function.
2. Pick a random initial value for the parameters. (To clarify, in the parabola example, differentiate "y" with respect to "x". If we had mor features like x1, x2 etc., we take the partial derivative of "y" with resp to each of the features.)
3. Update the gradient function by plugging in the parameter values.
4. Calculate the step sizes for each feature as : **step size = gradient * learning rate.**
5. Calculate the new parameters as : **new params = old params -step size**
6. Repeat steps 3 to 5 until gradient is almost 0.

The **"Learning Rate"** mentioned above is a flexible parameter which heavily influences the convergence of the algorithm.

LARGER LEARNING RATES MAKE THE ALGORITHM TAKE HUGE STEPS DOWN THE SLOPE and ***it might jump across the minimum point thereby missing it***.

So, **it is always good to stick to low learning rate such as 0.01**.

It can also be mathematically shown that:

The GD Algorithm takes:

- ***Larger steps down the slope if the starting point is high above and***
- ***Baby steps as it reaches closer to the destination to be careful not to miss it and also be quick enough.***

Stochastic Gradient Descent (SGD)

There are a few downsides of the gradient descent algorithm.

We need to take a closer look at the amount of computation we make for each iteration of the algorithm.
Say we have 10,000 data points and 10 features.

The sum of squared residuals consists of as many terms as there are data points, so 10000 terms in our case.

We need to compute the derivative of this function with respect to each of the features, so in effect we will be doing 10000 * 10 = 100,000 computations per iteration.

It is common to take 1000 iterations, in effect we have 100,000 * 1000 = 100000000 computations to complete the algorithm.

That is pretty much an overhead and hence gradient descent is slow on huge data.

Stochastic gradient descent comes to our rescue !! "Stochastic", in plain terms means "random".

Where can we potentially induce randomness in our gradient descent algorithm??

Yes, you might have guessed it right !! It is while selecting data points at each step to calculate the derivatives. SGD randomly picks one data point from the whole data set at each iteration to reduce the computations enormously.

It is also common to sample a small number of data points instead of just one point at each step and that is called "mini-batch" gradient descent. Mini-batch tries to strike a balance between the goodness of gradient descent and speed of SGD.

Have some function $J(\theta_0, \theta_1)$

Want $\min_{\theta_0,\theta_1} J(\theta_0, \theta_1)$

Outline:

- Start with some θ_0, θ_1
- Keep changing θ_0, θ_1 to reduce $J(\theta_0, \theta_1)$ until we hopefully end up at a minimum

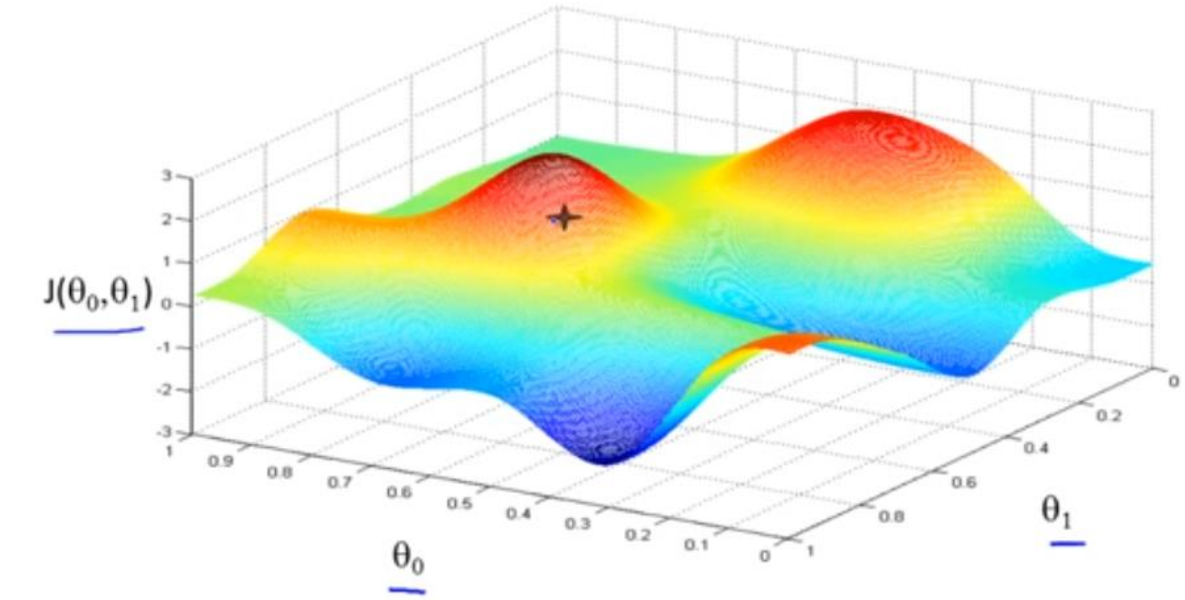
Generalization

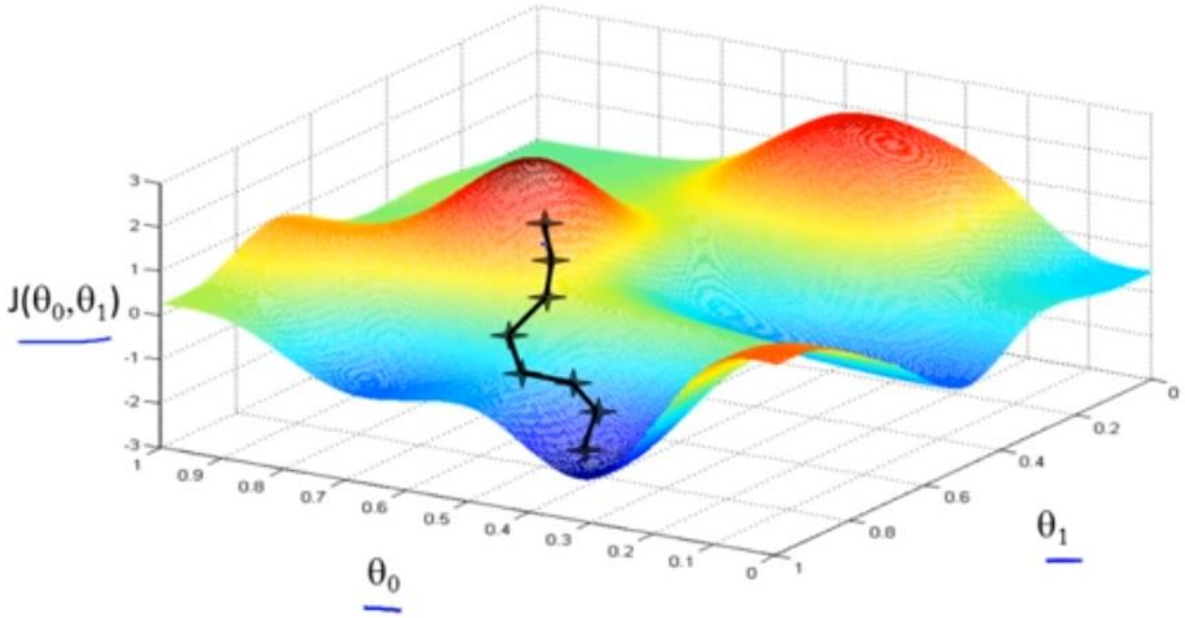
Have some function $J(\theta_0, \theta_1)$ $J(\theta_0, \theta_1, \theta_2, \ldots, \theta_n)$

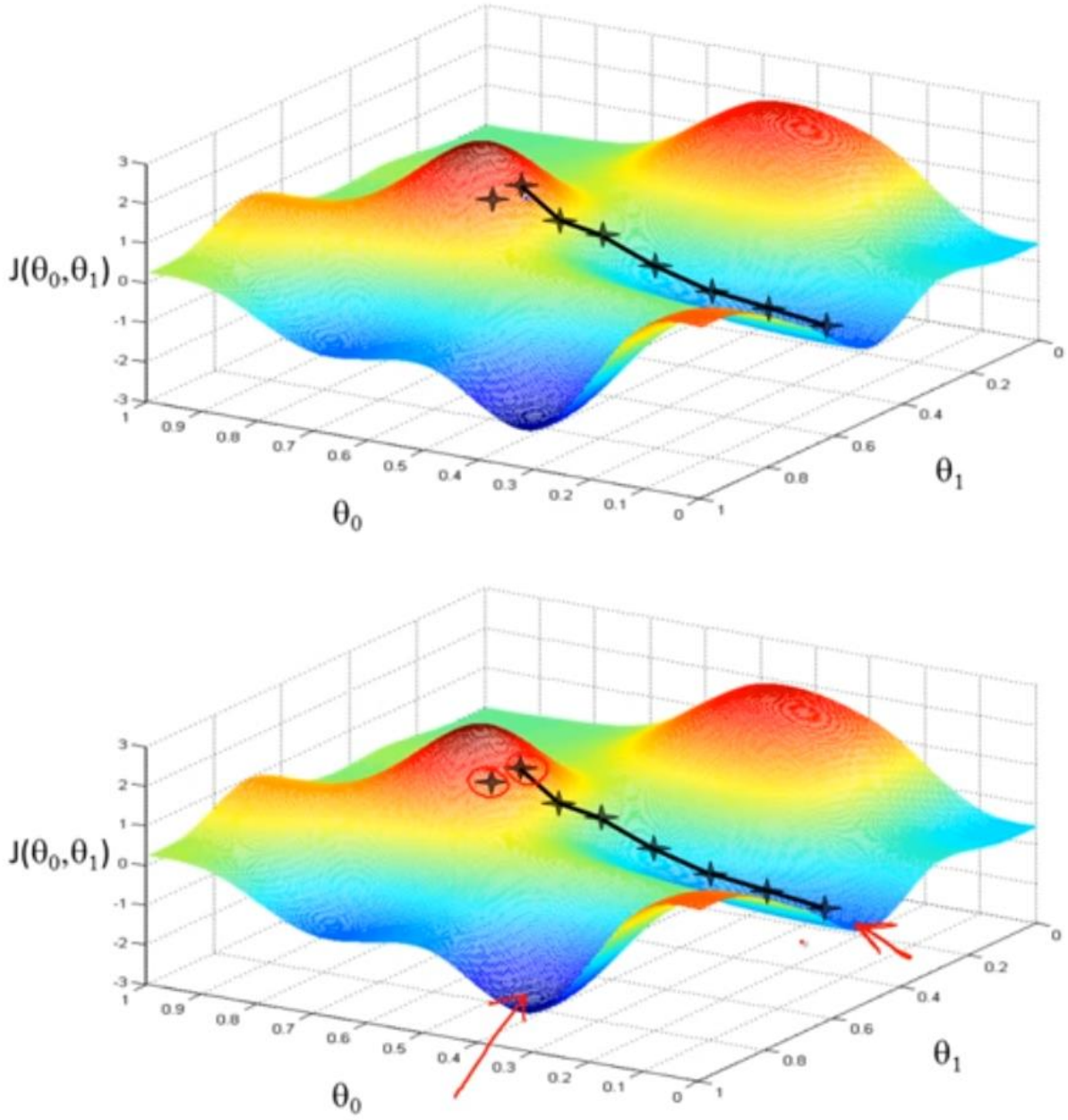
Want $\min_{\theta_0,\theta_1} J(\theta_0, \theta_1)$ $\min_{\theta_0 \ldots \theta_n} J(\theta_0, \ldots, \theta_n)$

Outline:

- Start with some θ_0, θ_1 (Say $\theta_0 = 0, \theta_1 = 0$)
- Keep changing θ_0, θ_1 to reduce $J(\theta_0, \theta_1)$ until we hopefully end up at a minimum


J(θ0,θ1)
θ0
θ1


J(θ0,θ1)
θ0
θ1


J(θ0,θ1)
θ0
θ1

Gradient descent algorithm

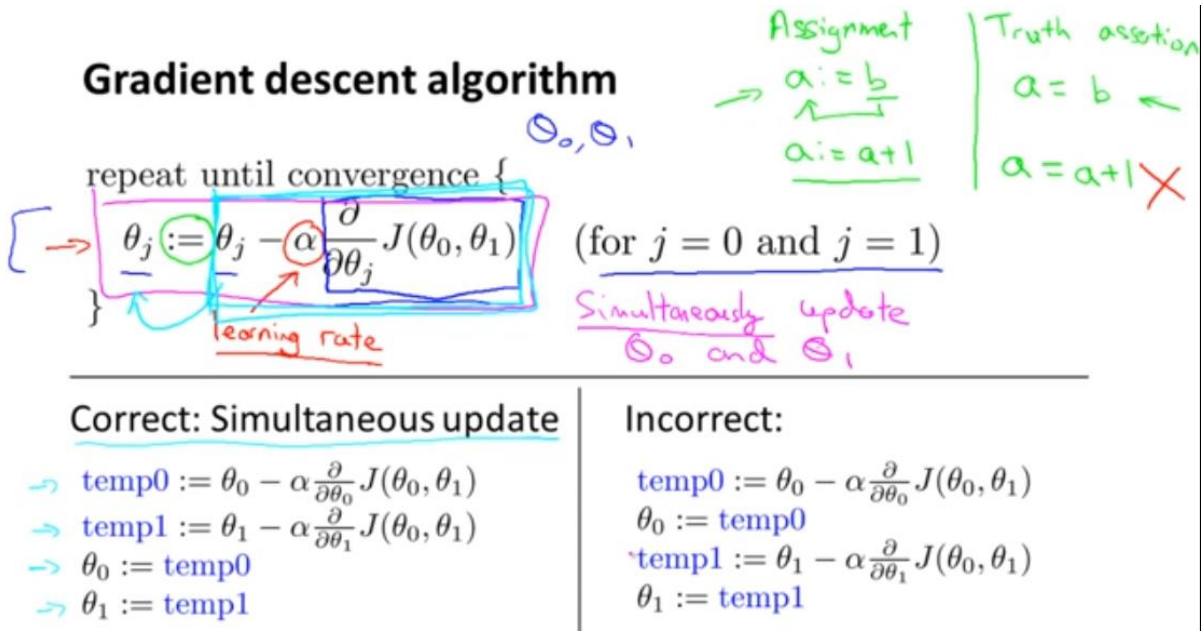
repeat until convergence {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \quad \text{(for } j = 0 \text{ and } j = 1\text{)}$$

}

Correct: Simultaneous update

$\text{temp0} := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$
$\text{temp1} := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$
$\theta_0 := \text{temp0}$
$\theta_1 := \text{temp1}$



Correct: Simultaneous update

$\text{temp0} := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$
$\text{temp1} := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$
$\theta_0 := \text{temp0}$
$\theta_1 := \text{temp1}$

Incorrect:

$\text{temp0} := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$
$\theta_0 := \text{temp0}$
$\text{temp1} := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$
$\theta_1 := \text{temp1}$

Gradient descent algorithm

repeat until convergence {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

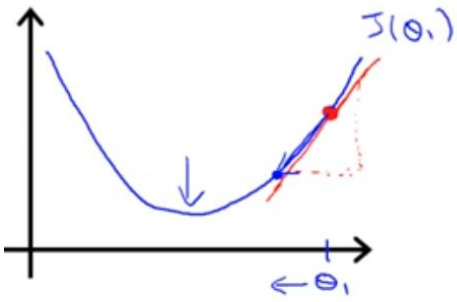
(simultaneously update $j = 0$ and $j = 1$)

}

learning rate

derivative

$\min_{\theta_1} J(\theta_1)$ $\quad \theta_1 \in \mathbb{R}$.

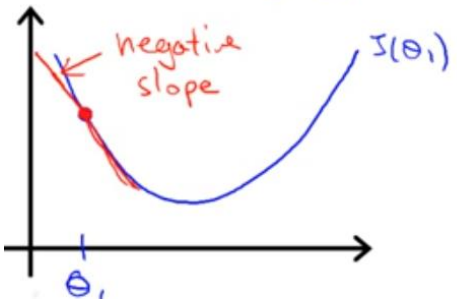


$(\theta_1 \in \mathbb{R})$

$$\theta_1 := \theta_1 - \alpha \frac{d}{d\theta_1} J(\theta_1)$$

$\frac{d}{d\theta_1} J(\theta_1) \geq 0$

$\theta_1 := \theta_1 - \alpha \cdot$ (positive number)

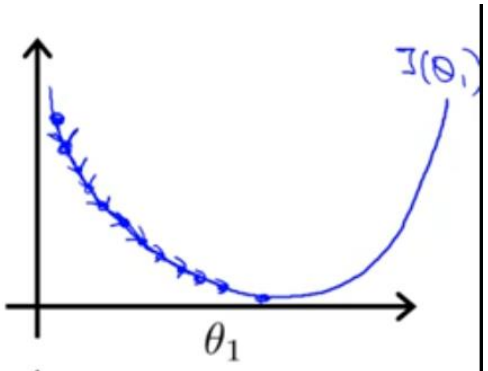


$\frac{d}{d\theta_1} J(\theta_1) \leq 0$

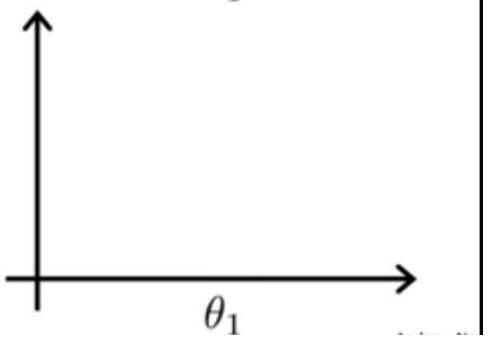
$\theta_1 := \theta_1 - \alpha$ (negative number)

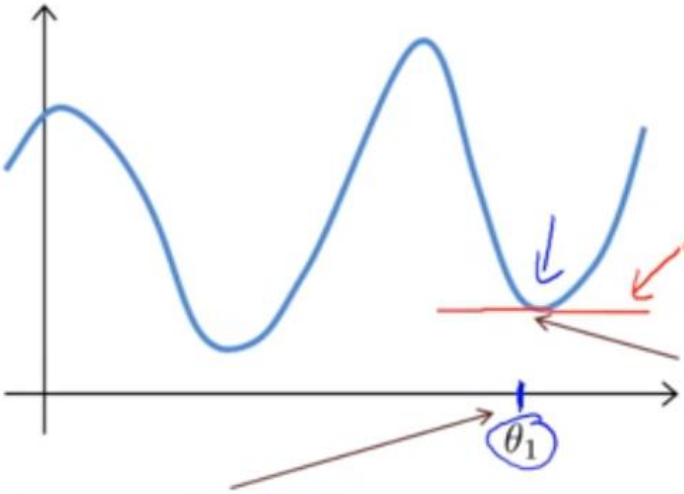
$$\theta_1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_1)$$

If α is too small, gradient descent can be slow.



If α is too large, gradient descent can overshoot the minimum. It may fail to converge, or even diverge.





$$\theta_1 := \theta_1 - \alpha \frac{d}{d\theta_1} J(\theta_1)$$

$= 0$

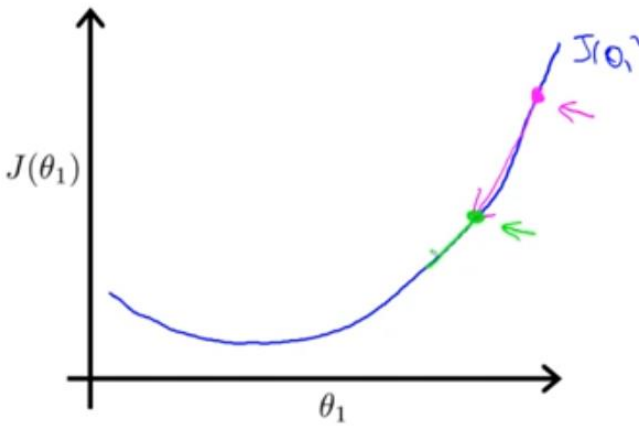
$\theta_1 := \theta_1 - \alpha \cdot 0$

$\theta_1 := \theta_1$

Gradient descent can converge to a local minimum, even with the learning rate α fixed.

$$\theta_1 := \theta_1 - \alpha \frac{d}{d\theta_1} J(\theta_1)$$

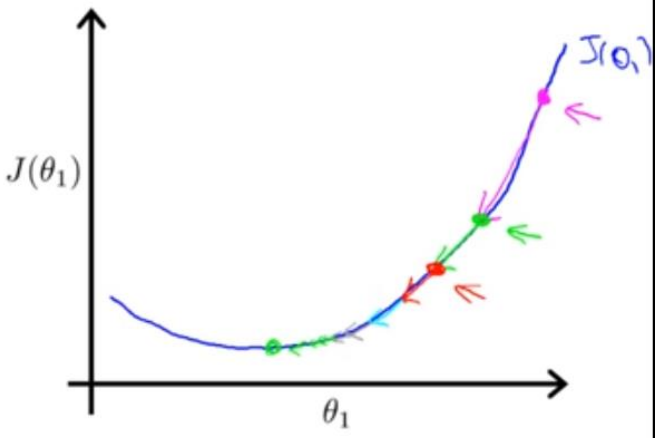
As we approach a local minimum, gradient descent will automatically take smaller steps. So, no need to decrease α over time.



Gradient descent can converge to a local minimum, even with the learning rate α fixed.

$$\theta_1 := \theta_1 - \alpha \frac{d}{d\theta_1} J(\theta_1)$$

As we approach a local minimum, gradient descent will automatically take smaller steps. So, no need to decrease α over time.



Gradient descent algorithm

repeat until convergence {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

$$(\text{for } j = 1 \text{ and } j = 0)$$

}

Linear Regression Model

$$h_\theta(x) = \theta_0 + \theta_1 x$$

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

Gradient descent algorithm

repeat until convergence {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

(for $j = 1$ and $j = 0$)

}

Linear Regression Model

$$h_\theta(x) = \theta_0 + \theta_1 x$$

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

$$\min_{\theta_0, \theta_1} J(\theta_0, \theta_1)$$

$$\frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) = \frac{2}{2\theta_j} \cdot \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

$$= \frac{2}{2\theta_j} \frac{1}{2m} \sum_{i=1}^{m} \left(\theta_0 + \theta_1 x^{(i)} - y^{(i)} \right)^2$$

$$\frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) = \frac{2}{2\theta_j} \cdot \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

$$= \frac{2}{2\theta_j} \frac{1}{2m} \sum_{i=1}^{m} \left(\theta_0 + \theta_1 x^{(i)} - y^{(i)} \right)^2$$

$$\theta_0 \; j = 0 : \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)$$

$$\theta_1 \; j = 1 : \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) \cdot x^{(i)}$$

Gradient descent algorithm

repeat until convergence {

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) \cdot x^{(i)}$$

}

Gradient descent algorithm

$\frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$

repeat until convergence {

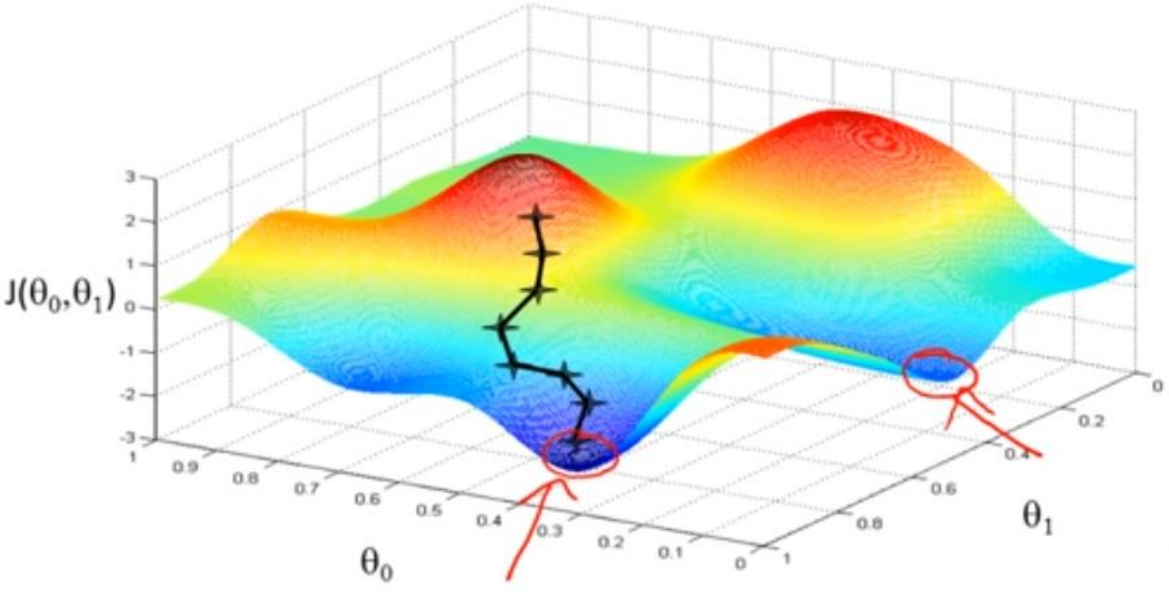
$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) \cdot x^{(i)}$$

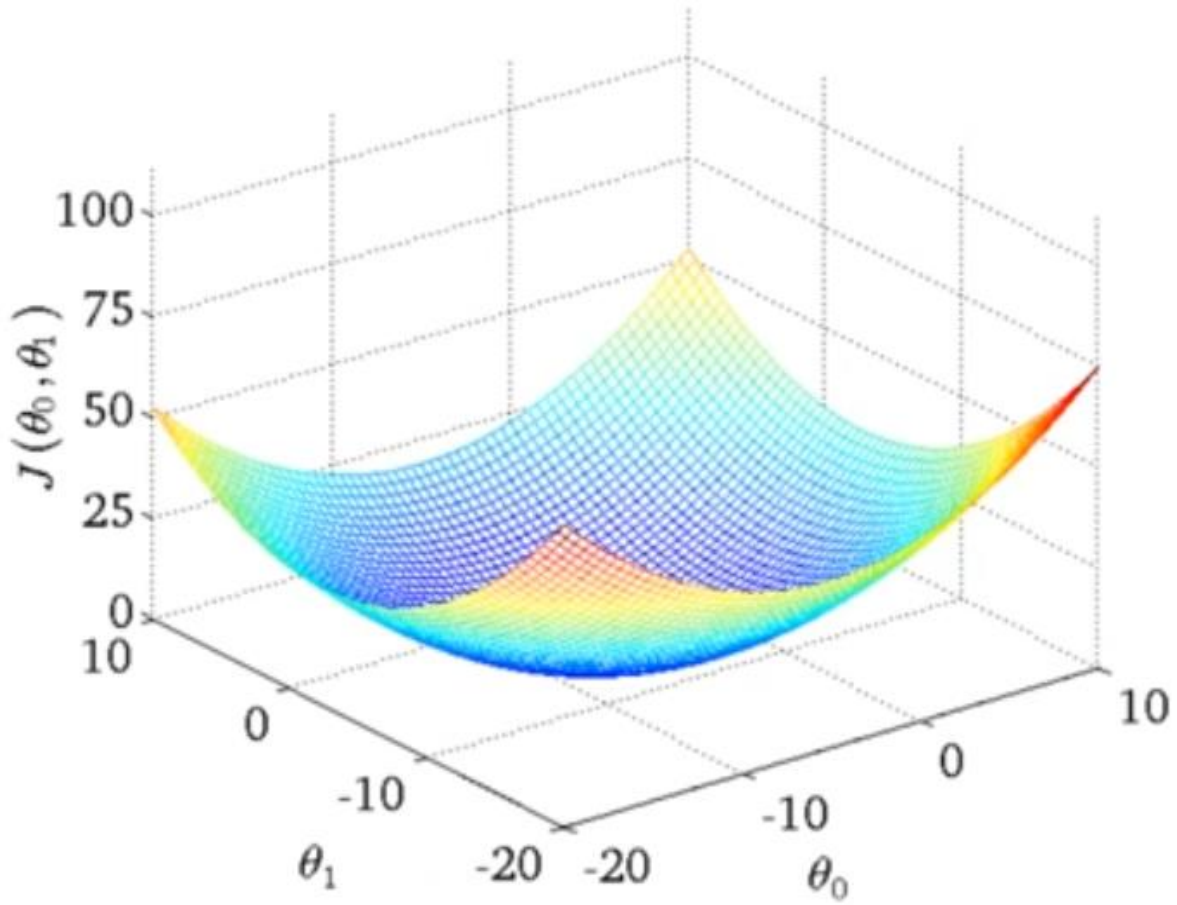
}

update θ_0 and θ_1 simultaneously

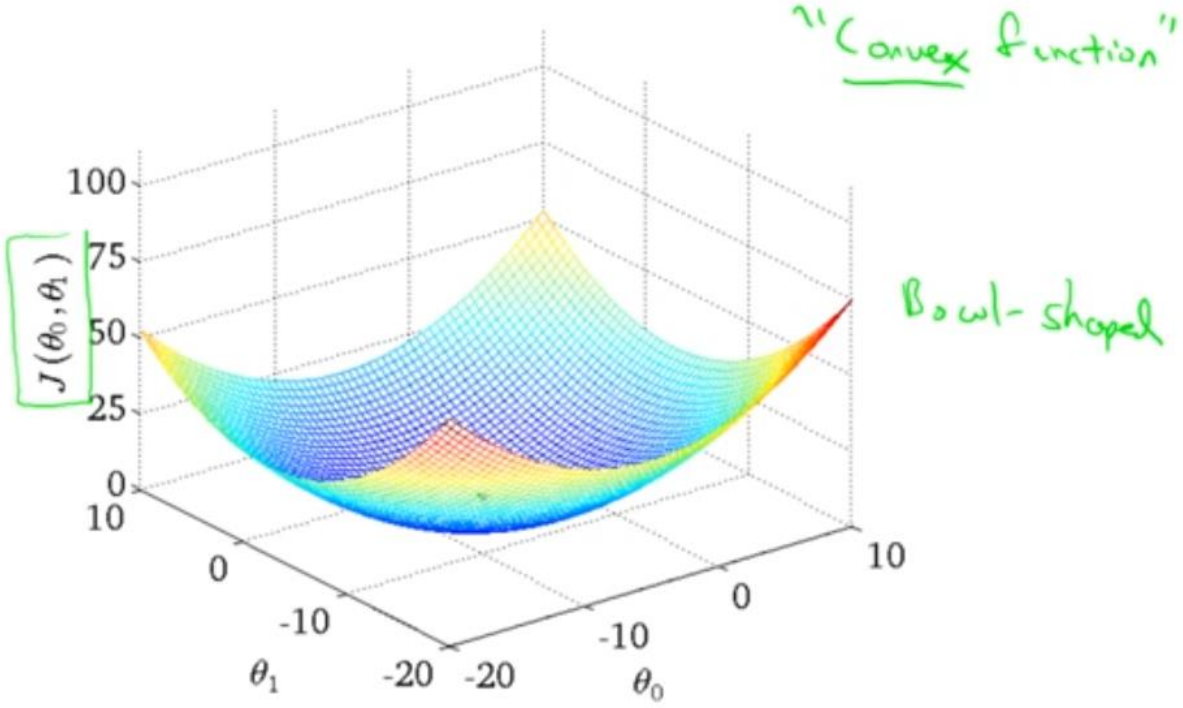
$\frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$



A bow shaped function:



A Convex Function represents the bow shape.



"Batch" Gradient Descent

"Batch": Each step of gradient descent uses all the training examples.

$$\sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)$$

Mean Squared Error Function

1. Difference between
   actual value of y & predicted value of y

$$(y_i - \bar{y}_i)$$

actual value predicted value

$$\bar{y}_i = mx_i + c$$

2. Square the Difference

$$(y_i - \bar{y}_i)^2$$

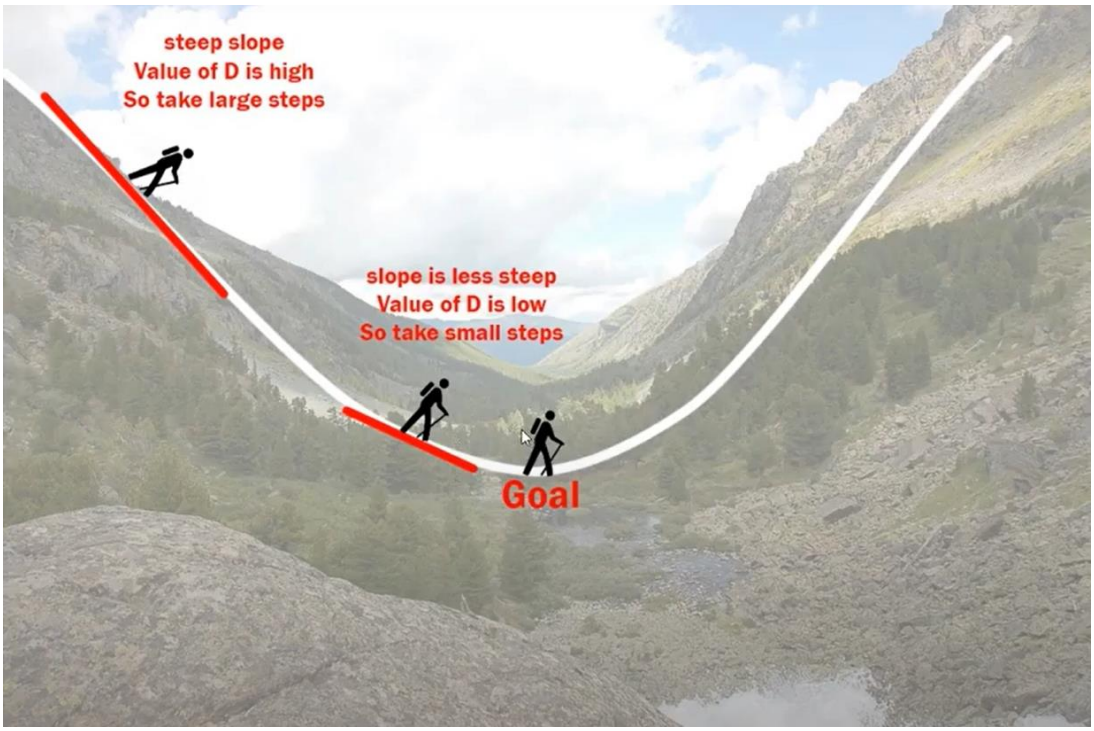
3. Find the mean of the squares

$$\frac{1}{n}\sum_{i=0}^{n}(y_i - \bar{y}_i)^2$$

$$E = \frac{1}{n}\sum_{i=0}^{n}(y_i - (mx_i + c))^2$$

Gradient Descent Algorithm

Gradient Descent is an iterative optimization algorithm to find the minimum of a function.
Here that function is our Loss Function.



Step 1

Initially

m = 0

c = 0

L = Learning Rate = 0.0001

Step 2

Calculate the partial derivative of loss function with respect to m & c

$$D_m = \frac{1}{n}\sum_{i=0}^{n} 2(y_i - (mx_i + c))(-x_i)$$

Partial Derivative wrt m, $D_m = \frac{-2}{n}\sum_{i=0}^{n} x_i(y_i - \bar{y}_i)$

Partial Derivative wrt c, $D_c = \frac{-2}{n}\sum_{i=0}^{n} (y_i - \bar{y}_i)$

Step 3

Update current value of m & c using these equations:

$$m = m - L \times D_m$$

$$c = c - L \times D_c$$

Step 4

Repeat Step 2 and Step 3 untill Loss = 0 (ideally)

Imagine a valley and a person with no sense of direction who wants to get to the bottom of the valley. He goes down the slope and takes large steps when the slope is steep and small steps when the slope is less steep. He decides his next position based on his current position and stops when he gets to the bottom of the valley which was his goal.
Let's try applying gradient descent to **m** and **c** and approach it step by step:

1. Initially let m = 0 and c = 0. Let L be our learning rate. This controls how much the value of **m** changes with each step. L could be a small value like 0.0001 for good accuracy.
2. Calculate the partial derivative of the loss function with respect to m, and plug in the current values of x, y, m and c in it to obtain the derivative value **D**.

$$D_m = \frac{1}{n}\sum_{i=0}^{n} 2(y_i - (mx_i + c))(-x_i)$$

$$D_m = \frac{-2}{n}\sum_{i=0}^{n} x_i(y_i - \bar{y}_i)$$

D_m is the value of the partial derivative with respect to **m**. Similarly lets find the partial derivative with respect to **c**, D_c :

$$D_c = \frac{-2}{n}\sum_{i=0}^{n}(y_i - \bar{y}_i)$$

3. Now we update the current value of **m** and **c** using the following equation:

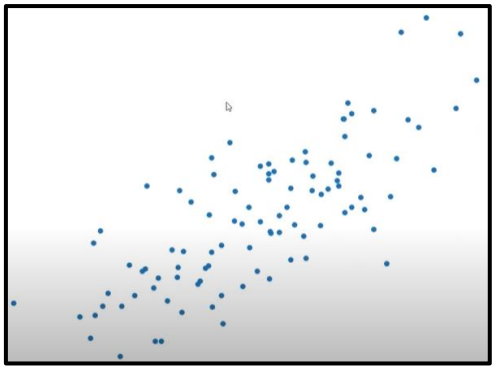
$$m = m - L \times D_m$$

$$c = c - L \times D_c$$

4. We repeat this process untill our loss function is a very small value or ideally 0 (which means 0 error or 100% accuracy). The value of **m** and **c** that we are left with now will be the optimum values.

Now going back to our analogy, **m** can be considered the current position of the person. **D** is equivalent to the steepness of the slope and **L** can be the speed with which he moves. Now the new value of **m** that we calculate using the above equation will be his next positon, and $L \times D$ will be the size of the steps he will take. When the slope is more steep (**D** is more) he takes longer steps and when it is less steep (**D** is less), he takes smaller steps. Finally he arrives at the bottom of the valley which corresponds to our loss = 0.
We repeat the same process above to find the value of **c** also. Now with the optimum value of **m** and **c** our model is ready to make predictions !

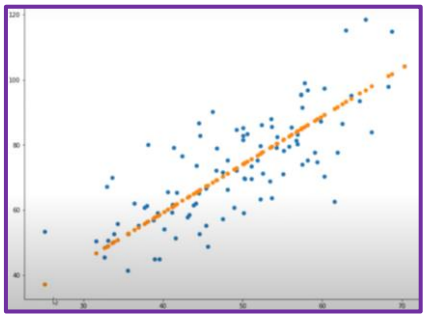
```
# Making the imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = (12.0, 9.0)

# Preprocessing input data
data = pd.read_csv('data.csv')
X = data.iloc[:, 0]
Y = data.iloc[:, 1]
plt.scatter(X, Y)
plt.show()
```



```
# Building the model
m = 0
c = 0

L = 0.0001
epochs = 1000

n = float(len(X))

for i in range(len(X)):
    Y_pred = m*X + c
    D_m = (-2/n) * sum(X * (Y - Y_pred))
    D_c = (-2/n) * sum(Y - Y_pred)
    m = m - L*D_m
    c = c - L*D_c

print(m, c)
```

```
1.4809284677644328 0.0362353540942316
```

```
# Making predictions
Y_pred = m*X + c

plt.scatter(X, Y)
plt.scatter(X, Y_pred)
plt.show()
```



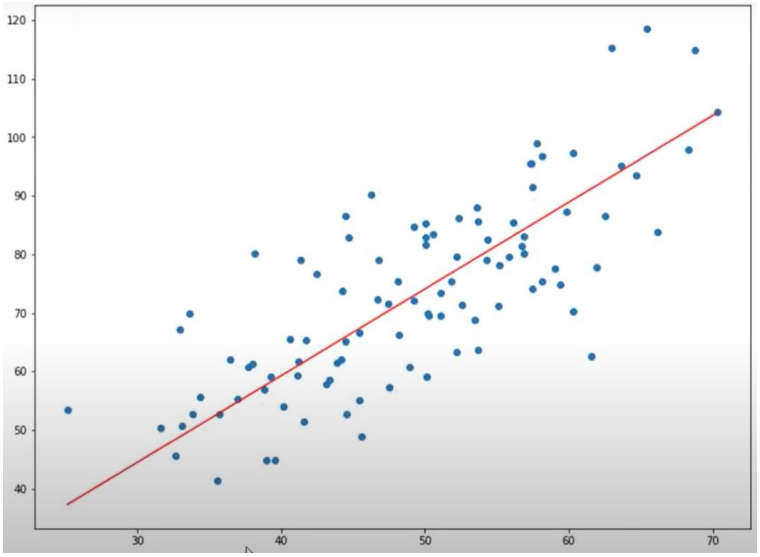
For each value of X, we have:

The **predicted value of Y**, on the **orange line (Regression Line)** and the **actual Y values** are shown in **blue dots**.

It is along this Regression Line that we get the least error for each and every value of X.

If you want to see the Regression Line instead of the scatter plot, we have to take the leftmost point, i.e. the min values of y_pred and X, the maximum values of y_pred and X, and draw a line between those two.

```
# Making predictions
Y_pred = m*X + c

plt.scatter(X, Y)
plt.plot([min(X), max(X)], [min(Y_pred), max(Y_pred)], color='red')
plt.show()
```



Advantages of Gradient Descent:

- Easy to learn and use
- Can be applied to any function

MLR – 22 SEP 2021

Home prices in Monroe Township, NJ (USA)

area	bedrooms	age	price
2600	3	20	550000
3000	4	15	565000
3200		18	610000
3600	3	30	595000
4000	5	8	760000

Given these home prices find out price of a home that has,

3000 sqr ft area, 3 bedrooms, 40 year old
2500 sqr ft area, 4 bedrooms, 5 year old

```
import pandas as pd
import numpy as np
from sklearn import linear_model
```

```
df = pd.read_csv("homeprices.csv")
df
```

	area	bedrooms	age	price
0	2600	3.0	20	550000
1	3000	4.0	15	565000
2	3200	NaN	18	610000
3	3600	3.0	30	595000
4	4000	5.0	8	760000

```
df.bedrooms.fillna(median_bedrooms)
```

```
0    3.0
1    4.0
2    3.0
3    3.0
4    5.0
Name: bedrooms, dtype: float64
```

```
df.bedrooms = df.bedrooms.fillna(median_bedrooms)
df
```

	area	bedrooms	age	price
0	2600	3.0	20	550000
1	3000	4.0	15	565000
2	3200	3.0	18	610000
3	3600	3.0	30	595000
4	4000	5.0	8	760000

```
reg = linear_model.LinearRegression()
reg.fit(df[['area','bedrooms','age']],df.price)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
```

```
reg.coef_
```

```
array([   137.25, -26025.  ,  -6825.  ])
```

```
reg.intercept_
```

```
383724.99999999983
```

```
reg.predict([[3000,3,40]])
```

```
array([ 444400.])
```

```
137.25*3000+-26025*3+-6825*40+383724.99999999983
```

```
444399.9999999998
```


```
reg = linear_model.LinearRegression()
reg.fit(df[['area','bedrooms','age']],df.price)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
```

```
reg.coef_
```

```
array([   137.25, -26025.  ,  -6825.  ])
```

```
reg.intercept_
```

```
383724.99999999983
```

```
reg.predict([[3000,3,40]])
```

```
array([ 444400.])
```

```
137.25*3000+-26025*3+-6825*40+383724.99999999983
```

```
444399.9999999998
```


```
reg.predict([[2500,4,5]])
```

```
array([ 588625.])
```