I have seen most people when hear the term Gradient then they try to finish the topic without understanding Maths behind it. In this tutorial, I will explain your Gradient descent from a very ground level, pick you up with simple maths examples, and make gradient descent completely consumable for you.

Gradient Descent is a first-order optimization technique used to find the local minimum or optimize the loss function. It is also known as the parameter optimization technique.

It is easy to find the value of slope and intercept using a closed-form solution But when you work in Multidimensional data then the technique is so costly and takes a lot of time Thus it fails here. So, the new technique came as Gradient Descent which finds the minimum very fast.

The intuition behind Gradient Descent

Ordinary Least Squares

$f(x) = b_0 + w_0x$

b_0 = intercept

w_0-weight or slope

Your goal is to minimize the difference between the prediction $f(\mathbf{x})$ and the actual data y. This difference is called the residual.

As you've already learned, linear regression and the ordinary least squares method start with the observed values of the inputs $\mathbf{x} = (x_1, ..., x_r)$ and outputs y. They define a linear function $f(\mathbf{x}) = b_0 + b_1x_1 + \cdots + b_rx_r$, which is as close as possible to y.

This is an optimization problem.

It finds the values of weights $b_0, b_1, ..., b_r$ that minimize:

The sum of squared residuals SSR = $\Sigma_i(y_i - f(\mathbf{x}_i))^2$ or the mean squared error MSE = SSR / n. Here, n is the total number of observations (data) and $i = 1, ..., n$.

You can also use the cost function C = SSR / $(2n)$, which is mathematically more convenient than SSR or MSE.

$$\mathbf{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

$\mathbf{MSE}$ = mean squared error
n = number of data points
Y_i = observed values
$\hat{Y}_i$ = predicted values

The most basic form of linear regression is simple linear regression.

It has only one set of inputs x and two weights: b_0 and b_1, or one of them is weight and other is intercept---➔ Y= $b_0 + w_0 * x$.

The equation of the regression line is $f(x) = b_0 + b_1x$. Although the optimal values of b_0 and b_1 can be calculated analytically, you'll use gradient descent to determine them.

Both SSR and MSE use the square of the difference between the actual and predicted outputs. The lower the difference, the more accurate the prediction. A difference of zero indicates that the prediction is equal to the actual data.

In SSR or MSE is minimized by adjusting the model parameters. For example, in linear regression, you want to find the function $f(\mathbf{x}) = b_0 + b_1x_1 + \cdots + b_rx_r$, so you need to determine the weights $b_0, b_1, ..., b_r$ that minimize SSR or MSE.

In a classification problem, the outputs y are categorical, often either 0 or 1. For example, you might try to predict whether an email is spam or not. In the case of binary outputs, it's convenient to minimize the cross-entropy function that also depends on the actual outputs y_i and the corresponding predictions $p(\mathbf{x}_i)$:

$$H = -\sum_i (y_i \log(p(\mathbf{x}_i)) + (1 - y_i) \log(1 - p(\mathbf{x}_i)))$$

Loss and cost Function

Loss and cost functions are used in machine learning to quantify the discrepancy between the actual values and the predicted value during the model training. Both are important components in machine learning.

A loss function is used to evaluate the performance of a model **on a single training observation**. It takes the predicted output from the model and compares it to the true target output, quantifying the discrepancy between the two.

Cost Function is commonly computed as the average or sum of the individual loss function values **across the full training dataset**. The cost function is a global measure of how well the model performs and is used by optimization methods such as gradient descent to iteratively adjust the model's parameters during training.

$$J(\theta) = \frac{1}{2m} \sum (h_\theta(x^{(i)}) - y^{(i)})^2$$

$$\text{Cost Function (J)} = (\frac{1}{n}) \sum_{i=1}^{n} (y_i - (wx_i + b))^2$$

Cost function

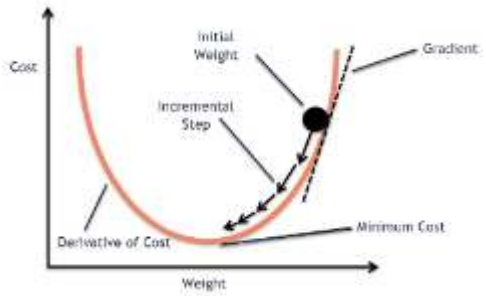
Goal is to minimize the cost function as much as possible. To achieve this goal, we'll use gradient descent method.

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

Parameters: θ_0, θ_1

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

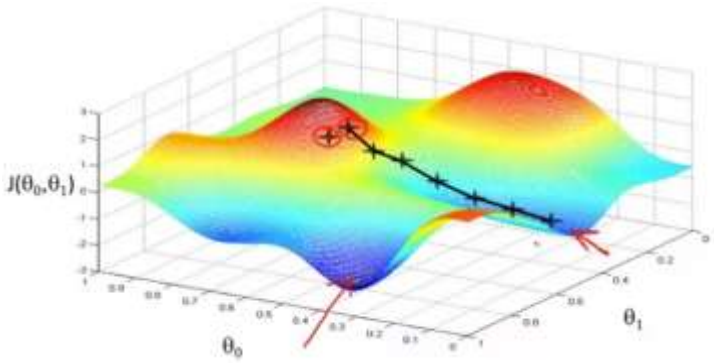
Goal: $\underset{\theta_0, \theta_1}{\text{minimize}}\ J(\theta_0, \theta_1)$



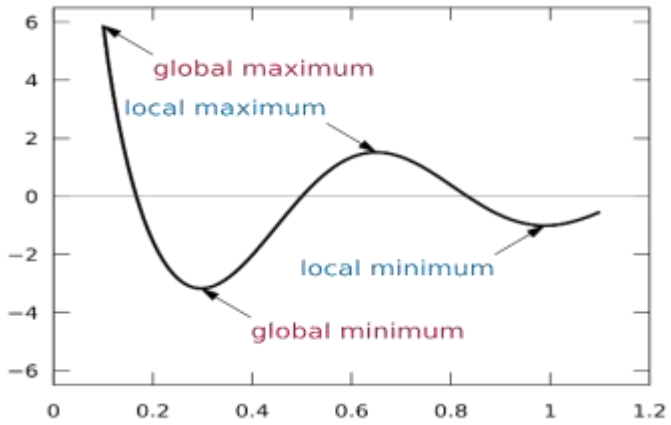
Gradient Descent is one of the most common methods used to optimize different **convex functions** in Machine Learning.

By moving in the opposite direction of the gradient, which is the negative gradient, during optimization, the algorithm aims to converge towards the optimal set of parameters that provide the best fit to the training data.

Gradient descent is a versatile optimization technique that can be utilized in various machine learning algorithms, such as linear regression, logistic regression, neural networks, and support vector machines.



For now, we should keep in mind that using a gradient descent algorithm we can keep changing the value of all our parameters such that we can reach a local minimum. while gradient descent is a powerful optimization algorithm, it has limitations in distinguishing between local and global minima, especially for complex, high-dimensional functions. Careful initialization and step size selection are important to try to avoid getting stuck in poor local minima.



A local minimum is a point where the function value is lower than all nearby points, but it may not be the lowest value globally. The global minimum is the absolute lowest value of the function across the entire domain. Gradient descent cannot reliably detect whether the minimum it finds is the global minimum or just a local minimum. Gradient descent is limited in its ability to escape local minima and find the global minimum.

We have to minimize the cost function to find the value of Theta (θ) in the regression line.

The method of gradient descent can be represented as follows:

minimize the cost function to find the value of Theta in the regression line.

Cost Function – "One Half Mean Squared Error":

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

Objective:

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

Update rules:

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

Derivatives:

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

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

$$\begin{aligned}\frac{\partial}{\partial\theta_i}J(\theta_i) &= \frac{\partial}{\partial\theta_i}\frac{1}{2}(h_\theta(x) - y)^2 \\ &= 2\frac{1}{2}(h_\theta(x) - y)\frac{\partial}{\partial\theta_i}(h_\theta(x) - y) \\ &= (h_\theta(x) - y)\frac{\partial}{\partial\theta_i}\sum_{i=1}^{m}(\theta_i x_i - y) \\ &= (h_\theta(x) - y)x_i\end{aligned}$$

m = size of data

h(x) = predicted value

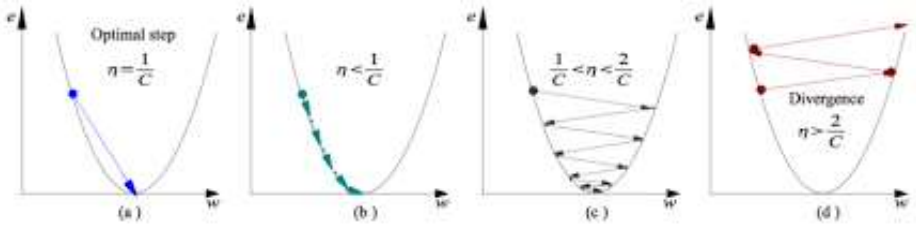
y = true target

learning rate

The scientists have gone one step further and add it one more variable to the derivate of the cost function J to accelerate the formula. This scalar makes the gradient more or less steep to coverage faster to the minimum point, and it's called learning rate.

Alpha is called Learning rate – a tuning parameter in the optimization process. It decides the length of the steps.

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

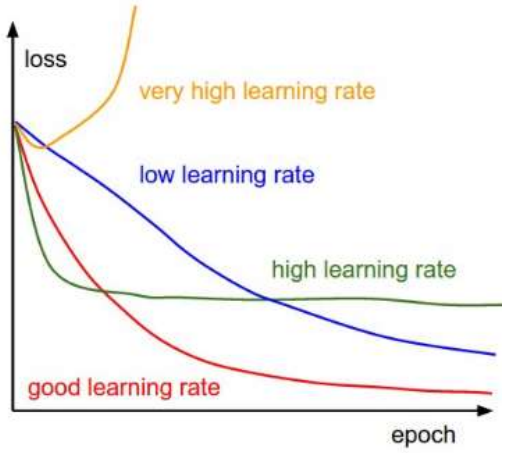


Learning rate is optimal, model converges to the minimum

Learning rate is too small, it takes more time but converges to the minimum

Learning rate is higher than the optimal value, it overshoots but converges ($1/C < \eta < 2/C$)

Learning rate is very large, it overshoots and diverges, moves away from the minima, performance decreases on learning



In the first graph, α is too small, that makes our method slow. On the other hand, in case that α is too large, the steps of each iteration will be that big that it will overshoot the minimum over and over. Therefore, it is suggested to try different values of α between -1 and 1 to find the sweet spot. Usually, the machine learning engineers use α=0.01 or α=0.1 and increase or decrease the learning rate from there.

keep in mind that Normalize your data: $z= (x_i - \mu)/\sigma$

Example:

To find the $\boldsymbol{\theta_0}$ and $\boldsymbol{\theta_1}$ parameter in linear regression using the gradient descent algorithm is

$\Theta_1 = (n * \Sigma xy - \Sigma x * \Sigma y) / (n * \Sigma x^2 - (\Sigma x)^2)$

$\Theta_0 = (1/m) * \Sigma(y^{(i)} - \theta_1 * x^{(i)})$ or $\Theta_0 = \bar{y} - \Theta_1 \bar{x}$

Linear regression equation: $h_\theta(x) = \theta_0 + \theta_1 * x$

i	x	y
1	0	2
2	-3	8
3	5	2
4	2	1

n = 4

Σxy = -12

Σx = 4

Σy = 13

$\Sigma x2$ = 38

$\Sigma \bar{y}$ = 3.25

$\Sigma \bar{x}$ = 1

$\Theta_1 = (n * \Sigma xy - \Sigma x * \Sigma y) / (n * \Sigma x^2 - (\Sigma x)^2)$

$\Theta_1 = (4 * -12 - 4 * 13) / (4 * 38 - 4^2) = -100 / 136 = -0.74$

$\Theta_0 = \bar{y} - \Theta_1\bar{x}$

$\Theta_0 = 3.25 + 0.74 = 3.99$

$h_\theta(x) = \theta_0 + \theta_1 * x = 3.99 - 0.74 * x$

Normal Equation

At this point in the article, I want to introduce the vectorization of the functions. The reason that we do such a thing is about the computational efficiency from programming libraries and make our formulas shorter and easier to read. Let's say that we have four cars whose prices we want to predict.

Our target variable(dependent variable) will be price, and the independent variable will be the horsepower of the car.

Normal Equation is an analytical approach used for optimization. It is an alternative for Gradient descent. Normal equation performs minimization without iteration. Normal equations directly compute the parameters of the model that minimizes the Sum of the squared difference between the actual term and the predicted term of the dataset without needing to choose any hyperparameters like learning rate or the number of iterations.

The normal equation to find the linear regression coefficients θ0 and θ1 is:

$\theta = (X^T X)^{-1} * X^T y$

Convert our features and target variables into matrices, and vectors:

$$x = \begin{bmatrix} 111 \\ 111 \\ 154 \\ 102 \end{bmatrix} \xrightarrow{\text{add intercept point}} \begin{bmatrix} 1 & 111 \\ 1 & 111 \\ 1 & 154 \\ 1 & 102 \end{bmatrix}, y = \begin{bmatrix} 13495.0 \\ 16500.0 \\ 16500 \\ 13950.0 \end{bmatrix}, \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$$

Multiply our features by our parameters:

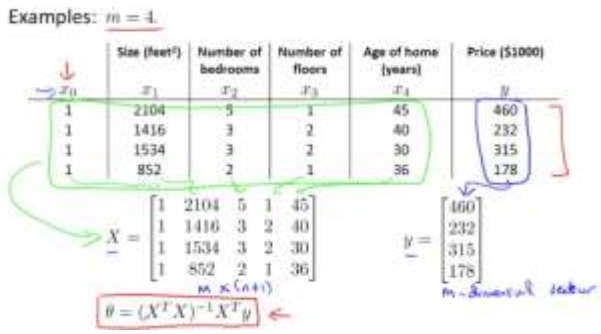
$$h_\theta(x) = \begin{bmatrix} \theta_0 & \theta_1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} = \theta^T x = \begin{bmatrix} \theta_0 + \theta_1 * 111 \\ \theta_0 + \theta_1 * 111 \\ \theta_0 + \theta_1 * 154 \\ \theta_0 + \theta_1 * 102 \end{bmatrix}$$

Where:

X is the design matrix of the input features

y is the vector of target values

x	y
1	1
2	3
3	5
4	7



Differences between Cost Function and Mean Square Error (MSE)

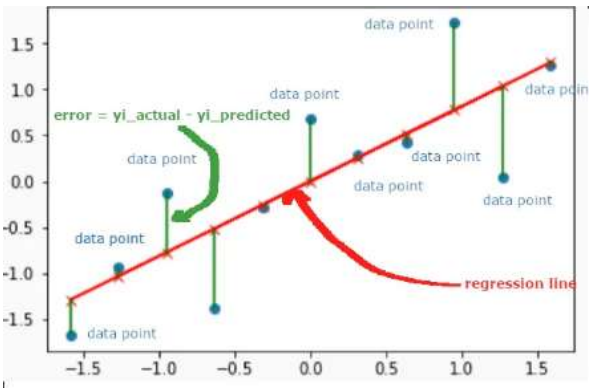
Objective: The primary objective of both cost functions and the MSE function is to measure the difference between predictions and actual values, but they might use different metrics or formulas to do so. The cost function is a broader term that encompasses various functions used to evaluate model performance, with MSE being a specific type of cost function used in regression problems

- The term "cost function" is more general and can refer to any function used to evaluate the performance of a model.
- The MSE function is a specific type of cost function used in regression problems.
- While MSE is a cost function, not all cost functions are MSE.
- Other cost functions might be used for different types of problems or models, such as cross-entropy for classification problems.

A sample function of MSE in python:

```
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
```



First, you need calculus to find the gradient of the cost function $C = \Sigma_i(y_i - b_0 - b_1x_i)^2 / (2n)$. Since you have two decision variables, b_0 and b_1, the gradient ∇C is a vector with two components:

The gradient of a function C of several independent variables $v_1, ..., v_r$ is denoted with $\nabla C(v_1, ..., v_r)$ and defined as the vector function of the partial derivatives of C with respect to each independent variable: $\nabla C = (\partial C/\partial v_1, ..., \partial C/v_r)$. The symbol ∇ is called nabla.

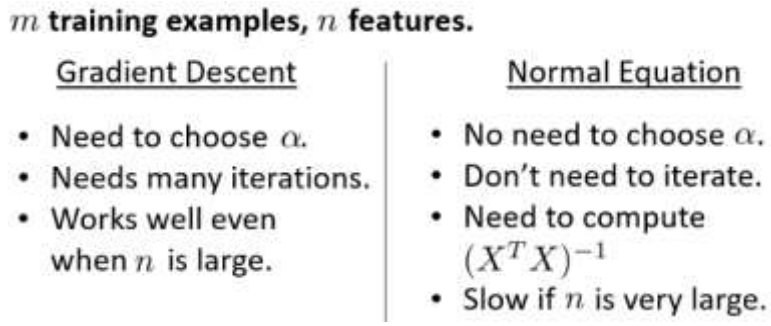
The nonzero value of the gradient of a function C at a given point defines the direction and rate of the fastest increase of C. When working with gradient descent, you're interested in the direction of the fastest decrease in the cost function. This direction is determined by the negative gradient, $-\nabla C$.

Challenges of Gradient Descent

While gradient descent is a powerful optimization algorithm, it can also present some challenges that can affect its performance. Some of these challenges include:

- Local Optima: Gradient descent can converge to local optima instead of the global optimum, especially if the cost function has multiple peaks and valleys.
- Learning Rate Selection: The choice of learning rate can significantly impact the performance of gradient descent. If the learning rate is too high, the algorithm may overshoot the minimum, and if it is too low, the algorithm may take too long to converge.
- Overfitting: Gradient descent can overfit the training data if the model is too complex or the learning rate is too high. This can lead to poor generalization performance on new data.
- Convergence Rate: The convergence rate of gradient descent can be slow for large datasets or high-dimensional spaces, which can make the algorithm computationally expensive.
- Saddle Points: In high-dimensional spaces, the gradient of the cost function can have saddle points, which can cause gradient descent to get stuck in a plateau instead of converging to a minimum.

To overcome these challenges, several variations of gradient descent algorithm have been developed, such as adaptive learning rate methods, momentum-based methods, and second-order methods. Additionally, choosing the right regularization method, model architecture, and hyperparameters can also help improve the performance of gradient descent algorithm.



Algorithm for Gradient Descent

Till now we have seen the parameters required for gradient descent.

Now let us map the parameters with the gradient descent algorithm and work on an example to better understand gradient descent.

Let us consider a parabolic equation $y=4X^2$. By looking at the equation we can identify that the parabolic function is minimum at x = 0 i.e. at x=0, y=0. Therefore x=0 is the local minima of the parabolic function $y=4X^2$. Now let us see the algorithm for gradient descent and how we can obtain the local minima by applying gradient descent:

```
repeat until convergence
{
    w = w - (learning_rate * (dJ/dw))
    b = b - (learning_rate * (dJ/db))
}
```

Step 1: Initializing all the necessary parameters and deriving the gradient function for the parabolic equation 4x2. The derivative of X^2 is 2x, so the derivative of the parabolic equation $4X^2$ will be 8x.

X_0 = 3 (random initialization of x)

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

learning_rate = 0.01 (to determine the step size while moving towards local minima)

Gradient = dy/dx = $d(4X^2)/dx = 8x$

Step 2: Let us perform 3 iterations of gradient descent:

For each iteration keep on updating the value of x based on the gradient descent formula.

```
Iteration 1:
    x1 = x0 - (learning_rate * gradient)
    x1 = 3 - (0.01 * (8 * 3))
    x1 = 3 - 0.24
    x1 = 2.76

Iteration 2:
    x2 = x1 - (learning_rate * gradient)
    x2 = 2.76 - (0.01 * (8 * 2.76))
    x2 = 2.76 - 0.2208
    x2 = 2.5392

Iteration 3:
    x3 = x2 - (learning_rate * gradient)
    x3 = 2.5392 - (0.01 * (8 * 2.5392))
    x3 = 2.5392 - 0.203136
    x3 = 2.3360
```

From the above three iterations of gradient descent, we can notice that the value of x is decreasing iteration by iteration and will slowly converge to 0 (local minima) by running the gradient descent for more iterations. Now you might have a question, for how many iterations we should run gradient descent?

We can set a **stopping threshold** i.e., when the difference between the previous and the present value of x becomes less than the stopping threshold, we stop the iterations.

We can set a **stopping threshold** i.e., when the **difference between the previous and the present value of x becomes less than the stopping threshold, we stop the iterations.**

When it comes to the implementation of gradient descent for machine learning algorithms and deep learning algorithms, we try to minimize the cost function in the algorithms using gradient descent. Now that we are clear with the gradient descent's internal working, let us look into the python implementation of gradient descent where we will be minimizing the cost function of the linear regression algorithm and finding the best fit line. In our case the parameters are below mentioned:

Prediction Function

The prediction function for the linear regression algorithm is a linear equation given by **y=wx+b**.

```
prediction_function (y) = (w * x) + b
Here, x is the independent variable
      y is the dependent variable
      w is the weight associated with input variable
      b is the bias
```

Partial Derivatives (Gradients)

$$\text{Cost Function (J)} = \left(\frac{1}{n}\right)\sum_{i=1}^{n}(y_i - (wx_i + b))^2$$

Calculating the partial derivatives for weight and bias using the cost function. We get:

$$\frac{dJ}{dw} = (\frac{-2}{n})\sum_{i=1}^{n} x_i * (y_i - (wx_i + b))$$

$$\frac{dJ}{db} = (\frac{-2}{n})\sum_{i=1}^{n} (y_i - (wx_i + b))$$

Parameter Updating

Updating the weight and bias by subtracting the multiplication of learning rates and their respective gradients.

```
w = w - (learning_rate * (dJ/dw))
b = b - (learning_rate * (dJ/db))
```

Python Implementation for Gradient Descent

For the implementation part, we will be writing two functions:

- **The first one will be the cost functions** that take the actual output and the predicted output as input and return the loss,
- **The second will be the actual gradient descent function** which takes the independent variable, and target variable as input and finds the best-fit line using the gradient descent algorithm.
- **The main function,** we will be initializing linearly related random data and applying the gradient descent algorithm to the data to find the best-fit line.

The iterations, learning_rate, and stopping threshold are the tuning parameters for the gradient descent algorithm and can be tuned by the user.

The iterations specify the number of times the update of parameters must be done, the stopping threshold is the minimum change of loss between two successive iterations to stop the gradient descent algorithm.

The optimal weight and bias found by using the gradient descent algorithm are later used to plot the best-fit line in the main function.