**Gradient Descent**

**Introduction to Gradient Descent**

**Gradient Descent** is an algorithm used to find the best solution to a problem by making small adjustments in the right direction. I**t’s like trying to find the lowest point in a hilly area by walking down the slope, step by step, until you reach the bottom.**

**For example:**

Imagine you're at the top of a hill and your goal is to find the lowest point in the valley. You can't see the entire valley from the top, but you can feel the slope under your feet.

1. **Start at the Top:** You begin at the top of the hill (this is like starting with random guesses for the model's parameters).
2. **Feel the Slope:** You look around to find out which direction the ground is sloping down. This is like calculating the **gradient**, which tells you the steepest way downhill.
3. **Take a Step Down:** Move in the direction where the slope is steepest (this is adjusting the model's parameters). **The bigger the slope, the bigger the step you take.**
4. **Repeat:** You keep repeating the process — feeling the slope and moving downhill — until you **reach the bottom of the valley (this is when the model has learned and minimized the error**).

The key idea is that, just like walking down a hill, Gradient Descent moves towards the "bottom" or **minimum** of the loss function, which represents the error in predictions.

Moving in opposite direction of the gradient allows the algorithm to **gradually descend towards lower values of the function and eventually reaching to the minimum of the function. These gradients guide the updates ensuring convergence towards the optimal parameter values.** Gradual steps used in descent is done by defining **learning rate**.

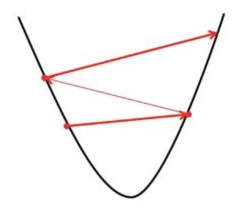
**What is Learning Rate?**

Learning rate is a important hyperparameter in gradient descent that controls how big or small the steps should be when going downwards in gradient for updating models parameters. It is essential to determines how quickly or slowly the algorithm converges toward minimum of cost function.

**1. If Learning rate is too small**: The algorithm will take tiny steps during iteration and converge very slowly. This can significantly increases training time and computational cost especially for large datasets. **This process is termed as vanishing gradient problem.**

Learning rate with small steps

**2. If Learning rate is too big: T**he algorithm may take huge steps leading overshooting the minimum of cost function without settling. It fail to converge causing the algorithm to oscillate. **This process is termed as exploding gradient problem.**

Learning rate with big steps

In image we can see point got oscillated from right to left with converging to minimum gradient value.

**To address these problems we have some technique that can be used:**

* **Weights Regularzations:**The initialization of weights can be adjusted to ensure that they are in an appropriate range. Using a different activation function such as the Rectified Linear Unit (ReLU) can help us to mitigate the vanishing gradient problem.
* **Gradient clipping:**Restrict the gradients to a predefined range to prevent them from becoming excessively large or small.
* **Batch normalization:**It can also help to address these problems by normalizing the input of each layer to prevent activation function from saturating and hence reducing vanishing and exploding gradient problems.

Choosing right learning rate can leads to **fast and stable convergence** improving the efficiency of the training process but sometimes vanishing and exploding gradient problem is unavoidable and to address these we have some techniques that we will discuss further in the article.

**Mathematics Behind Gradient Descent**

For simplicity let's consider a [linear regression](https://www.geeksforgeeks.org/ml-linear-regression/) model with a single input feature x*x* and target y*y*. The **loss function** (or cost function) for a single data point is defined as the[Mean Squared Error (MSE)](https://www.geeksforgeeks.org/mean-squared-error/):

J(w,b)=1n∑i=1n(yp−y)2*J*(*w*,*b*)=*n*1​∑*i*=1*n*​(*yp*​−*y*)2

Here:

* yp=x⋅w+b*yp*​=*x*⋅*w*+*b*: The predicted value.
* w*w*: Weight (slope of the line).
* b*b*: Bias (intercept).
* n*n*: Number of data points.

To optimize the model parameters w*w*, we compute the **gradient of the loss function** with respect to w*w*. This process involves taking the partial derivatives of J(w,b)*J*(*w*,*b*).

The gradient with respect to w*w* is:

∂J(w,b)∂w=∂∂w[1n∑i=1n(yp−y)2]∂*w*∂*J*(*w*,*b*)​=∂*w*∂​[*n*1​∑*i*=1*n*​(*yp*​−*y*)2]

\frac{\partial J(w, b)}{\partial w} = \frac{2}{n} \sum\_{i=1}^{n} (y\_p - y) \cdot \frac{\partial}{\partial w} \left( y\_p - y \right)

substitute yp=x⋅w+b*yp*​=*x*⋅*w*+*b*: ∂J(w,b)∂w=2n∑i=1n(yp−y)⋅∂∂w(x⋅w+b−y)∂*w*∂*J*(*w*,*b*)​=*n*2​∑*i*=1*n*​(*yp*​−*y*)⋅∂*w*∂​(*x*⋅*w*+*b*−*y*)

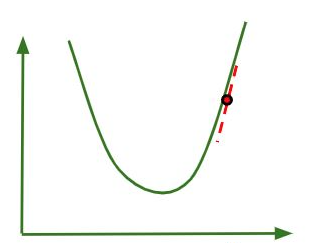
**Final Gradient with respect to**w*w*:

∂J(w,b)∂w=2n∑i=1n(yp−y)⋅x∂*w*∂*J*(*w*,*b*)​=*n*2​∑*i*=1*n*​(*yp*​−*y*)⋅*x*

**Gradient Descent Update:**

Once the gradients are calculated we update the parameters w*w* in the direction opposite to the gradient (to minimize the loss function):

1. For +ve gradient:

Gradient descent

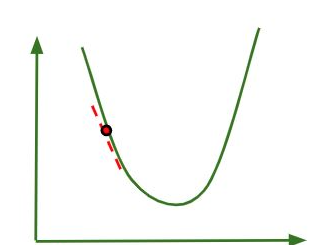
w=w−γ⋅∂J(w,b)∂w*w*=*w*−*γ*⋅∂*w*∂*J*(*w*,*b*)​

Here:

* γ*γ*: Learning rate (step size for each update).
* ∂w∂J(w,b)∂w∂*w*∂*w*∂*J*(*w*,*b*)​​: Gradients with respect to w*w*.

Since the gradient is positive subtracting it effectively decreasesw*w* and hence reducing cost function.

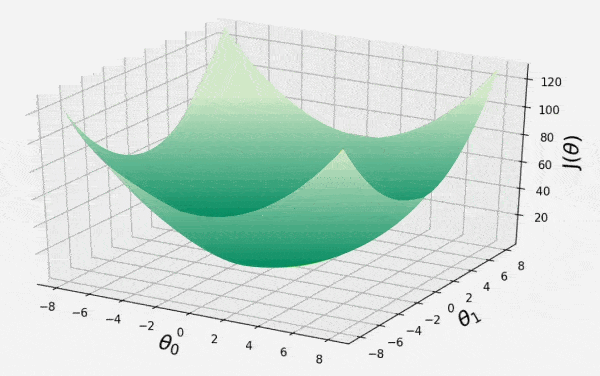
2. For -ve gradient:

Gradient descent

Since the gradient is negative subtracting it effectively **increases**w*w* so here we add it to reduce cost function.

**Working of Gradient Descent**

* **Step 1** we first initialize the parameters of the model randomly
* **Step 2** Compute the gradient of the cost function with respect to each parameter. It involves making partial differentiation of cost function with respect to the parameters.
* **Step 3**Update the parameters of the model by taking steps in the opposite direction of the model. Here we choose a [hyperparameter learning rate](https://www.geeksforgeeks.org/hyperparameter-tuning/) which is denoted by γ*γ*. It helps in deciding the step size of the gradient.
* **Step 4** Repeat steps 2 and 3 iteratively to get the best parameter for the defined model.

Gradient Descent

This animation shows iterative process of gradient descent as it traverses the 3D convex surface of cost function. Each step represents adjustment of model parameters to minimize the loss. It illustrates how the algorithm moves in opposite direction of descent to converge

**Pseudo code:**

t ← 0  
max\_iterations ← 1000  
w, b ← initialize randomly  
  
while t < max\_iterations do  
 t ← t + 1  
 w\_t+1 ← w\_t − γ ∇w\_t  
 b\_t+1 ← b\_t − γ ∇b\_t  
end  
  
Here:  
max\_iterations is the number of iteration we want to do to update our parameter   
W,b are the weights and bias parameter   
γ is the learning parameter

So now we learned what is gradient descent and how it works, now we will learn about its variations.

**Different Variants of Gradient Descent**

Types of gradient descent are:

1. **Batch Gradient Descent**: [Batch Gradient Descent](https://www.geeksforgeeks.org/difference-between-batch-gradient-descent-and-stochastic-gradient-descent/) computes gradients using the entire dataset in each iteration.
2. **Stochastic Gradient Descent (SGD)**: [SGD](https://www.geeksforgeeks.org/ml-stochastic-gradient-descent-sgd/) uses one data point per iteration to compute gradients, making it faster.
3. **Mini-batch Gradient Descent**: [Mini-batch Gradient Descent](https://www.geeksforgeeks.org/ml-mini-batch-gradient-descent-with-python/) combines batch and SGD by using small batches of data for updates.
4. **Momentum-based Gradient Descent**: [Momentum-based Gradient Descent](https://www.geeksforgeeks.org/ml-momentum-based-gradient-optimizer-introduction/) speeds up convergence by adding a fraction of the previous gradient to the current update.
5. **Adagrad**: [Adagrad](https://www.geeksforgeeks.org/intuition-behind-adagrad-optimizer/" \t "_blank)adjusts learning rates based on the historical magnitude of gradients.
6. **RMSprop**: [RMSprop](https://www.geeksforgeeks.org/rmsprop-optimizer-in-deep-learning/" \t "_blank) is similar to Adagrad but uses a moving average of squared gradients for learning rate adjustments.
7. **Adam**: [Adam](https://www.geeksforgeeks.org/adam-optimizer/) combines Momentum, Adagrad, and RMSprop by using moving averages of gradients and squared gradients.

*For understand their explanation and use-cases, please refer :* [*Types of Gradient Descent*](https://www.geeksforgeeks.org/different-variants-of-gradient-descent/)*.*

**Advantages Of Gradient Descent**

* **Flexibility: I**t can be used with various cost functions and can handle non-linear regression problems.
* **Scalability:** It is scalable to large datasets since it updates the parameters for each training example one at a time.
* **Convergence:** It can converge to global minimum of the cost function provided that the learning rate is set appropriately.

**Disadvantages Of Gradient Descent**

* **Sensitivity to Learning Rate**: The choice of learning rate is important in gradient descent as it can lead to vanishing or exploding gradient problem.
* **Sensitivity to initialization: It** can be sensitive to the initialization of the models parameters which can affect the convergence and the quality of the solution.
* **Local Minima:** It can get stuck in local minima, if the cost function has multiple local minima.
* **Time-consuming: It**can be time-consuming especially when dealing with large datasets.

Gradient Descent is a fundamental optimization technique used in machine learning. Understanding it allows us to make efficient and accurate model by reducing error made by model using cost function during their training phase, making gradient descent essential for building effective machine learning models.