Gradient Descent in Machine Learning

What is Gradient?

A gradient is nothing but a derivative that defines the effects on outputs of the function with a little bit of variation in inputs.

What is Gradient Descent?

Gradient Descent stands as a cornerstone orchestrating the intricate dance of model optimization. At its core, it is a numerical optimization algorithm that aims to find the optimal parameters—weights and biases—of a neural network by minimizing a defined cost function.

Gradient Descent (GD) is a widely used optimization algorithm in machine learning and deep learning that minimises the cost function of a neural network model during training. It works by iteratively adjusting the weights or parameters of the model in the direction of the negative gradient of the cost function until the minimum of the cost function is reached.

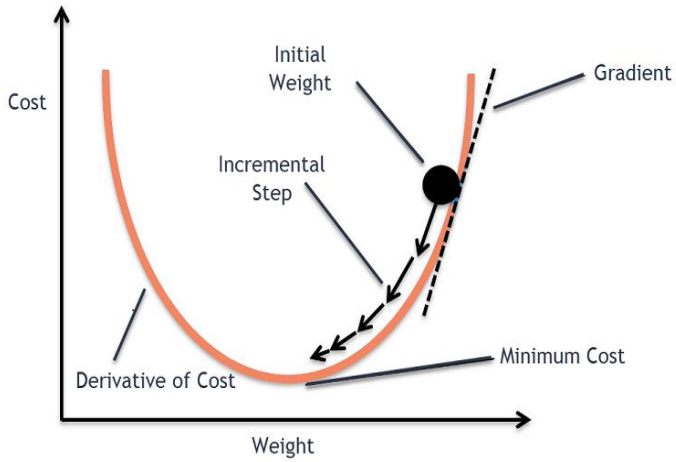
The learning happens during the **backpropagation** while training the neural network-based model. There is a term known as **Gradient Descent**, which is used to optimize the weight and biases based on the cost function. The cost function evaluates the difference between the actual and predicted outputs.

Gradient Descent is a fundamental optimization algorithm in machine learning used to minimize the cost or loss function during model training.

- It iteratively adjusts model parameters by moving in the direction of the steepest decrease in the cost function.
- The algorithm calculates gradients, representing the partial derivatives of the cost function concerning each parameter.

These gradients guide the updates, ensuring convergence towards the optimal parameter values that yield the lowest possible cost.

Gradient Descent is versatile and applicable to various machine learning models, including linear regression and neural networks. Its efficiency lies in navigating the parameter space efficiently, enabling models to learn patterns and make accurate predictions. Adjusting the **learning rate** is crucial to balance convergence speed and avoiding overshooting the optimal solution.



Define the loss function

$$\text{Loss function (J)} = \frac{1}{n}\sum(actual - predicted)^2$$

Here we are calculating the Mean Squared Error by taking the square of the difference between the actual and the predicted value and then dividing it by its length (i.e n = the Total number of output or target values) which is the mean of squared errors.

```
# Define the loss function
def Mean_Squared_Error(prediction, actual):
    error = (actual-prediction)**2
    return error.mean()


# Find the total mean squared error
loss = Mean_Squared_Error(y_p, y)
loss
```

As we can see from the above right now the Mean Squared Error is 30559.4473. All the steps which are done till now are known as forward propagation.

Now our task is to find the optimal value of weight w and bias b which can fit our model well by giving very less or minimum error as possible. i.e

$$\text{minimize } \frac{1}{n}\sum(actual - predicted)^2$$

Now to update the weight and bias value and find the optimal value of weight and bias we will do backpropagation. Here the Gradient Descent comes into the role to find the optimal value weight and bias.

How the Gradient Descent Algorithm Works

For the sake of complexity, we can write our loss function for the single row as below

$$J(w,b) = \frac{1}{n}(y_p - y)^2$$

In the above function x and y are our input data i.e constant. To find the optimal value of weight w and bias b. we partially differentiate with respect to w and b. This is also said that we will find the gradient of loss function J(w,b) with respect to w and b to find the optimal value of w and b.

Gradient of J(w,b) with respect to w

$$\begin{aligned}
J'_w &= \frac{\partial J(w,b)}{\partial w} \\
&= \frac{\partial}{\partial w}\left[\frac{1}{n}(y_p - y)^2\right] \\
&= \frac{2(y_p - y)}{n}\frac{\partial}{\partial w}\left[(y_p - y)\right] \\
&= \frac{2(y_p - y)}{n}\frac{\partial}{\partial w}\left[((xW^T + b) - y)\right] \\
&= \frac{2(y_p - y)}{n}\left[\frac{\partial(xW^T + b)}{\partial w} - \frac{\partial(y)}{\partial w}\right] \\
&= \frac{2(y_p - y)}{n}\left[x - 0\right] \\
&= \frac{1}{n}(y_p - y)[2x]
\end{aligned}$$

i.e

$$J'_w = \frac{\partial J(w,b)}{\partial w}$$
$$= J(w,b)[2x]$$

Gradient of J(w,b) with respect to b

$$J'_b = \frac{\partial J(w,b)}{\partial b}$$
$$= \frac{\partial}{\partial b}\left[\frac{1}{n}(y_p - y)^2\right]$$
$$= \frac{2(y_p - y)}{n}\frac{\partial}{\partial b}\left[(y_p - y)\right]$$
$$= \frac{2(y_p - y)}{n}\frac{\partial}{\partial b}\left[((xW^T + b) - y)\right]$$
$$= \frac{2(y_p - y)}{n}\left[\frac{\partial(xW^T + b)}{\partial b} - \frac{\partial(y)}{\partial b}\right]$$
$$= \frac{2(y_p - y)}{n}[1 - 0]$$
$$= \frac{1}{n}(y_p - y)[2]$$

i.e

$$J'_b = \frac{\partial J(w,b)}{\partial b}$$
$$= J(w,b)[2]$$

Here we have considered the linear regression. So that here the parameters are weight and bias only. But in a fully connected neural network model there can be multiple layers and multiple parameters. but the concept will be the same everywhere. And the below-mentioned formula will work everywhere.

$$Param = Param - \gamma \nabla J$$

Here,

- γ = Learning rate
- J = Loss function

- ∇ = Gradient symbol denotes the derivative of loss function J
- Param = weight and bias There can be multiple weight and bias values depending upon the complexity of the model and features in the dataset

In our case:

$$w = w - \gamma \nabla J(w, , b)$$
$$b = b - \gamma \nabla J(w, , b)$$

In the current problem, two input features, So, the weight will be two.

Implementations of the Gradient Descent algorithm for the above model

Steps:

1. Find the gradient using loss.backward()
2. Get the parameter using model.linear.weight and model.linear.bias
3. Update the parameter using the above-defined equation.
4. Again assign the model parameter to our model

How Does Gradient Descent Work?

1. It is is an optimization algorithm used to minimize the cost function of a model.
2. The cost function measures how well the model fits the training data and is defined based on the difference between the predicted and actual values.
3. The gradient of the cost function is the derivative with respect to the model's parameters and points in the direction of the steepest ascent.
4. The algorithm starts with an initial set of parameters and updates them in small steps to minimize the cost function.

5. In each iteration of the algorithm, the gradient of the cost function with respect to each parameter is computed.
6. The gradient tells us the direction of the steepest ascent, and by moving in the opposite direction, we can find the direction of the steepest descent.
7. The size of the step is controlled by the learning rate, which determines how quickly the algorithm moves towards the minimum.
8. The process is repeated until the cost function converges to a minimum, indicating that the model has reached the optimal set of parameters.
9. There are different variations of gradient descent, including batch gradient descent, stochastic gradient descent, and mini-batch gradient descent, each with its own advantages and limitations.
10. Efficient implementation of gradient descent is essential for achieving good performance in machine learning tasks. The choice of the learning rate and the number of iterations can significantly impact the performance of the algorithm.

Types of Gradient Descent

The choice of gradient descent algorithm depends on the problem at hand and the size of the dataset. Batch gradient descent is suitable for small datasets, while stochastic gradient descent algorithm is more suitable for

large datasets. Mini-batch is a good compromise between the two and is often used in practice.

Batch Gradient Descent

Batch gradient descent updates the model's parameters using the gradient of the entire training set. It calculates the average gradient of the cost function for all the training examples and updates the parameters in the opposite direction. Batch gradient descent guarantees convergence to the global minimum, but can be computationally expensive and slow for large datasets.

Stochastic Gradient Descent

Stochastic gradient descent updates the model's parameters using the gradient of one training example at a time. It randomly selects a training example, computes the gradient of the cost function for that example, and updates the parameters in the opposite direction. Stochastic gradient descent is computationally efficient and can converge faster than batch gradient descent. However, it can be noisy and may not converge to the global minimum.

Mini-Batch Gradient Descent

Mini-batch gradient descent updates the model's parameters using the gradient of a small subset of the training set, known as a mini-batch. It

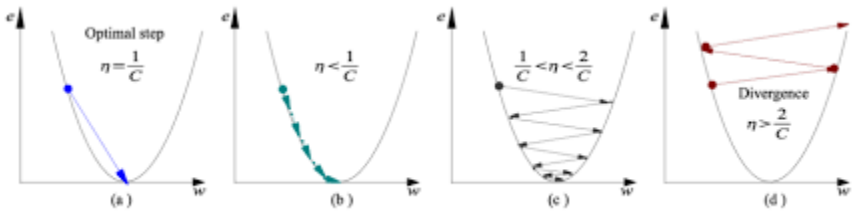
calculates the average gradient of the cost function for the mini-batch and updates the parameters in the opposite direction. Mini-batch gradient descent algorithm combines the advantages of both batch and stochastic gradient descent, and is the most commonly used method in practice. It is computationally efficient and less noisy than stochastic gradient descent, while still being able to converge to a good solution.

Alpha – The Learning Rate

We have the direction we want to move in, now we must decide the size of the step we must take.

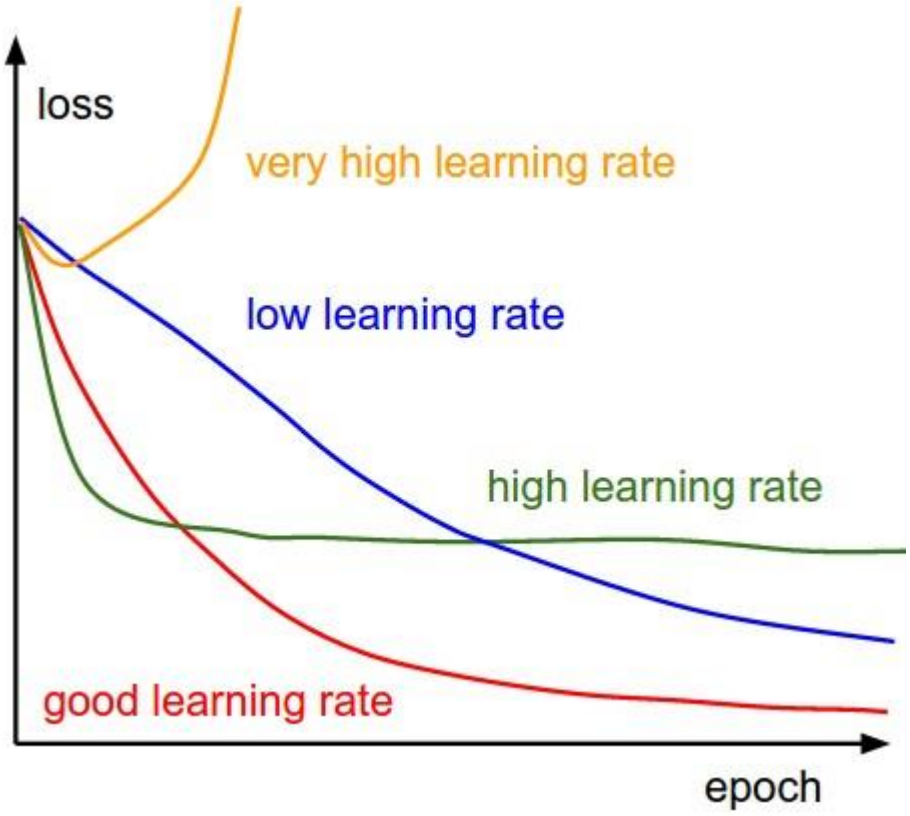
**It must be chosen carefully to end up with local minima.*

- If the learning rate is too high, we might OVERSHOOT the minima and keep bouncing, without reaching the minima
- If the learning rate is too small, the training might turn out to be too long



Source: Coursera

- 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

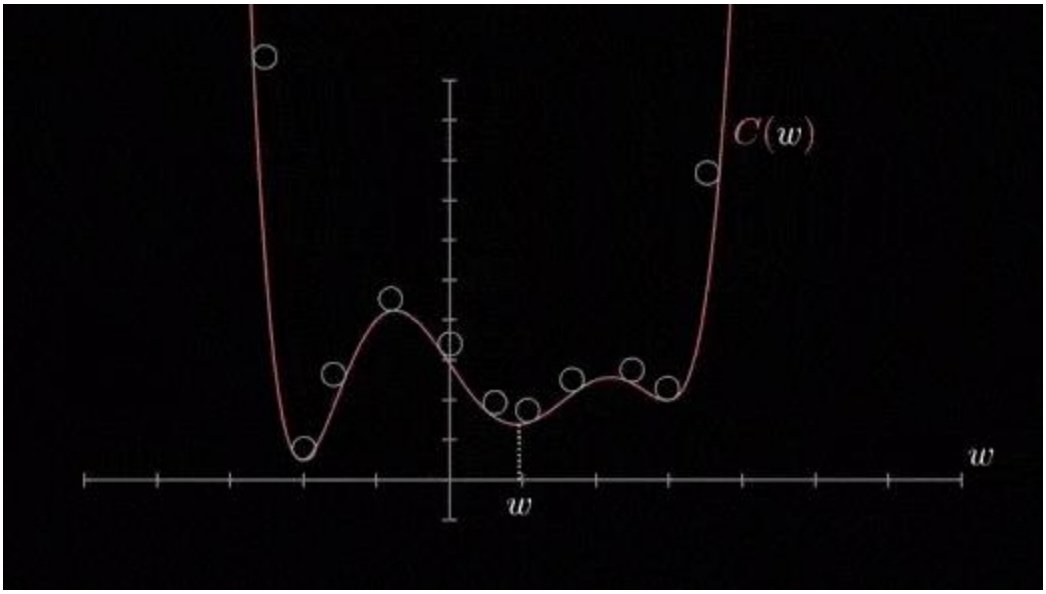


Source: researchgate

Note: *As the gradient decreases while moving towards the local minima, the size of the step decreases. So, the learning rate (alpha) can be constant over the optimization and need not be varied iteratively.*

Local Minima

The cost function may consist of many minimum points. The gradient may settle on any one of the minima, which depends on the initial point (i.e initial parameters(theta)) and the learning rate. Therefore, the optimization may converge to different points with different starting points and learning rate.



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:

1. Local Optima: Gradient descent can converge to local optima instead of the global optimum, especially if the cost function has multiple peaks and valleys.
2. 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.
3. 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.
4. 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.
5. 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.

Advantages & Disadvantages of gradient descent

Advantages of Gradient Descent

1. **Widely used:** Gradient descent and its variants are widely used in machine learning and optimization problems because they are effective and easy to implement.
2. **Convergence**: Gradient descent and its variants can converge to a global minimum or a good local minimum of the cost function, depending on the problem and the variant used.
3. **Scalability**: Many variants of gradient descent can be parallelized and are scalable to large datasets and high-dimensional models.
4. **Flexibility**: Different variants of gradient descent offer a range of trade-offs between accuracy and speed, and can be adjusted to optimize the performance of a specific problem.

Disadvantages of gradient descent:

1. **Choice of learning rate:** The choice of learning rate is crucial for the convergence of gradient descent and its variants. Choosing a learning rate that is too large can lead to oscillations or overshooting while choosing a learning rate that is too small can lead to slow convergence or getting stuck in local minima.
2. **Sensitivity to initialization:** Gradient descent and its variants can be sensitive to the initialization of the model's parameters, which can affect the convergence and the quality of the solution.
3. **Time-consuming:** Gradient descent and its variants can be time-consuming, especially when dealing with large datasets and high-dimensional models. The convergence speed can also vary depending on the variant used and the specific problem.
4. **Local optima:** Gradient descent and its variants can converge to a local minimum instead of the global minimum of the cost function, especially in non-convex problems. This can affect the quality of the solution, and techniques like random initialization and multiple restarts may be used to mitigate this issue.