# Gradient Descent Algorithm in Machine Learning

**Gradient descent** is the backbone of the learning process for various algorithms, including linear regression, logistic regression, support vector machines, and neural networks which serves as a fundamental optimization technique to minimize the cost function of a model by **iteratively adjusting the model parameters to reduce the difference between predicted and actual values, improving the model's performance.**

Let's see its role in machine learning:

**Prerequisites**: Understand the working and math of gradient descent.

## 1. Training Machine Learning Models

Neural networks are trained using Gradient Descent (or its variants) in combination with backpropagation. Backpropagation computes the gradients of the **loss function with respect to each parameter (weights and biases) in the network by applying the chain rule.** The process involves:

- **Forward Propagation**: Computes the output for a given input by passing data through the layers.
- **Backward Propagation**: Uses the chain rule to calculate gradients of the loss with respect to each parameter (weights and biases) across all layers.

***Gradients are then used by Gradient Descent to update the parameters layer-by-layer, moving toward minimizing the loss function.***

*Neural networks often use advanced variants of Gradient Descent. If you want to read more about variants, please refer : Gradient Descent Variants.*

## 2. Minimizing the Cost Function

The algorithm minimizes a cost function, which quantifies the error or loss of the model's predictions compared to the true labels for:

### A. Linear Regression

Gradient descent minimizes the Mean Squared Error (MSE) which serves as the loss function to find the best-fit line. Gradient Descent is used to iteratively update the weights (coefficients) and bias by computing the gradient of the MSE with respect to these parameters.

Since MSE is a convex function **gradient descent guarantees convergence to the global minimum if the learning rate is appropriately chosen.** For each iteration:

The algorithm computes the gradient of the MSE with respect to the weights and biases.

It updates the weights (w) and bias (b) using the formula:

- Calculating the gradient of the log-loss with respect to the weights.
- Updating weights and biases iteratively to maximize the likelihood of the correct classification:

$$w = w - \alpha \cdot \frac{\partial J(w,b)}{\partial w}, \quad b = b - \alpha \cdot \frac{\partial J(w,b)}{\partial b}$$

The formula is the **parameter update rule for gradient descent**, which adjusts the weights w and biases b to minimize a cost function. This process iteratively adjusts the line's slope and intercept to minimize the error.

### B. Logistic Regression

In logistic regression, gradient descent minimizes the **Log Loss (Cross-Entropy Loss)** to optimize the decision boundary for binary classification. Since the output is probabilistic (between 0 and 1), the sigmoid function is applied. The process involves:

- Calculating the gradient of the log-loss with respect to the weights.
- Updating weights and biases iteratively to maximize the likelihood of the correct classification:

$$w = w - \alpha \cdot \frac{\partial J(w)}{\partial w}$$

This adjustment shifts the decision boundary to separate classes more effectively.

## 3. Support Vector Machines (SVMs)

For SVMs, gradient descent optimizes the **hinge loss**, which ensures a maximum-margin hyperplane. The algorithm:

- Calculates gradients for the hinge loss and the regularization term (if used, such as L2 regularization).
- Updates the weights to maximize the margin between classes while minimizing misclassification penalties with same formula provided above.

- Gradient descent ensures the **optimal placement of the hyperplane to separate classes with the largest possible margin.**

# Gradient Descent Python Implementation

Diving further into the concept, let's understand with practical implementation.

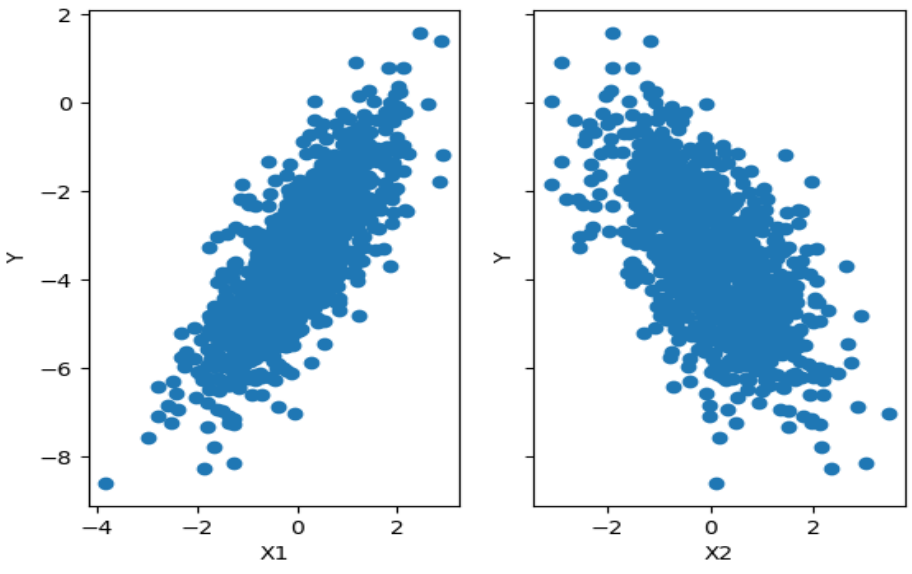
Import the necessary libraries

```
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
```

Set the input and output data

```
# set random seed for reproducibility
torch.manual_seed(42)

# set number of samples
num_samples = 1000

# create random features with 2 dimensions
x = torch.randn(num_samples, 2)

# create random weights and bias for the linear regression model
true_weights = torch.tensor([1.3, -1])
true_bias    = torch.tensor([-3.5])

# Target variable
y = x @ true_weights.T + true_bias

# Plot the dataset
fig, ax = plt.subplots(1, 2, sharey=True)
ax[0].scatter(x[:,0],y)
ax[1].scatter(x[:,1],y)

ax[0].set_xlabel('X1')
ax[0].set_ylabel('Y')
ax[1].set_xlabel('X2')
ax[1].set_ylabel('Y')
plt.show()
```

**Output:**




X vs Y

Let's first try with a linear model:

$$y_p = xW^T + b$$

```
# Define the model
class LinearRegression(nn.Module):
    def __init__(self, input_size, output_size):
        super(LinearRegression, self).__init__()
        self.linear = nn.Linear(input_size, output_size)

    def forward(self, x):
        out = self.linear(x)
        return out


# Define the input and output dimensions
input_size = x.shape[1]
output_size = 1

# Instantiate the model
model = LinearRegression(input_size, output_size)
```

We can manually set the model parameter

```
# create a random weight & bias tensor
weight = torch.randn(1, input_size)
bias   = torch.rand(1)

# create a nn.Parameter object from the weight & bias tensor
weight_param = nn.Parameter(weight)
bias_param   = nn.Parameter(bias)

# assign the weight & bias parameter to the linear layer
model.linear.weight = weight_param
model.linear.bias   = bias_param

weight, bias = model.parameters()
print('Weight :',weight)
print('bias :',bias)
```

Output:

```
Weight : Parameter containing:
tensor([[-0.3239,  0.5574]], requires_grad=True)
bias : Parameter containing:
tensor([0.5710], requires_grad=True)
```

Prediction

```
y_p = model(x)
y_p[:5]
```

Output:

```
tensor([[ 0.7760],
        [-0.8944],
        [-0.3369],
        [-0.3095],
        [ 1.7338]], grad_fn=<SliceBackward0>)
```

### Define the loss function

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
```

**Output**:

```
tensor(19.9126, grad_fn=<MeanBackward0>)
```

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.

### 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**

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

i.e

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

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

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

i.e

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

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,

- $\gamma$ = Learning rate
- J = Loss function
- $\nabla$ = 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:

$$
\begin{aligned}
w &= w - \gamma \nabla J(w,,b) \\
b &= b - \gamma \nabla J(w,,b)
\end{aligned}
$$

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

## Implementations of the Gradient Descent algorithm for 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.

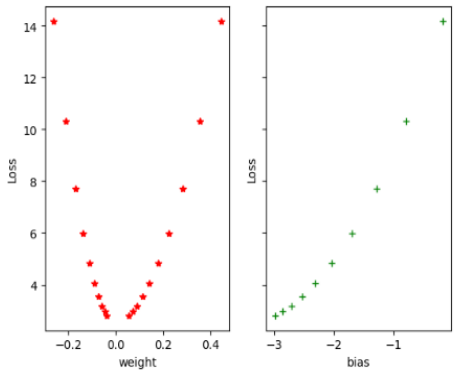
```
# Find the gradient using
loss.backward()
# Learning Rate
learning_rate = 0.001
# Model Parameter
w = model.linear.weight
b = model.linear.bias
# Matually Update the model parameter
w = w - learning_rate * w.grad
b = b - learning_rate * b.grad
# assign the weight & bias parameter to the linear layer
model.linear.weight = nn.Parameter(w)
model.linear.bias   = nn.Parameter(b)
```

Now Repeat this process till 1000 epochs

```
# Number of epochs
num_epochs = 1000

# Learning Rate
learning_rate = 0.01

# SUBPLOT WEIGHT & BIAS VS LOSSES
fig, (ax1, ax2) = plt.subplots(1, 2, sharey=True)

for epoch in range(num_epochs):
    # Forward pass
    y_p = model(x)
    loss = Mean_Squared_Error(y_p, y)

    # Backproogation
    # Find the fradient using
    loss.backward()

    # Learning Rate
    learning_rate = 0.001

    # Model Parameter
    w = model.linear.weight
    b = model.linear.bias

    # Matually Update the model parameter
    w = w - learning_rate * w.grad
    b = b - learning_rate * b.grad

    # assign the weight & bias parameter to the linear layer
    model.linear.weight = nn.Parameter(w)
    model.linear.bias   = nn.Parameter(b)

    if (epoch+1) % 100 == 0:
        ax1.plot(w.detach().numpy(),loss.item(),'r*-')
        ax2.plot(b.detach().numpy(),loss.item(),'g+-')
        print('Epoch [{}/{}], weight:{}, bias:{} Loss: {:.4f}'.format(
            epoch+1,num_epochs,
            w.detach().numpy(),
            b.detach().numpy(),
            loss.item()))

ax1.set_xlabel('weight')
ax2.set_xlabel('bias')
ax1.set_ylabel('Loss')
ax2.set_ylabel('Loss')
plt.show()
```

**Output**:

```
Epoch [100/1000], weight:[[-0.2618025   0.44433367]], bias:[-0.17722966] Loss: 14.1803
Epoch [200/1000], weight:[[-0.21144074  0.35393423]], bias:[-0.7892358] Loss: 10.3030
Epoch [300/1000], weight:[[-0.17063744  0.28172654]], bias:[-1.2897989] Loss: 7.7120
Epoch [400/1000], weight:[[-0.13759881  0.22408141]], bias:[-1.699218] Loss: 5.9806
Epoch [500/1000], weight:[[-0.11086453  0.17808875]], bias:[-2.0340943] Loss: 4.8235
Epoch [600/1000], weight:[[-0.08924612  0.14141548]], bias:[-2.3080034] Loss: 4.0502
Epoch [700/1000], weight:[[-0.0717768   0.11219224]], bias:[-2.5320508] Loss: 3.5333
Epoch [800/1000], weight:[[-0.0576706   0.08892148]], bias:[-2.7153134] Loss: 3.1878
Epoch [900/1000], weight:[[-0.04628877  0.07040432]], bias:[-2.8652208] Loss: 2.9569
Epoch [1000/1000], weight:[[-0.0371125   0.05568104]], bias:[-2.9878428] Loss: 2.8026
```




*Weight & Bias vs Losses - Geeksforgeeks*

From the above graph and data, we can observe the Losses are decreasing as per the weight and bias variations.

Now we have found the optimal weight and bias values. Print the optimal weight and bias and

```
w = model.linear.weight
b = model.linear.bias

print('weight(W) = {} \n  bias(b) = {}'.format(
  w.abs(),
  b.abs()))
```

**Output:**

```
weight(W) = tensor([[0.0371, 0.0557]], grad_fn=<AbsBackward0>)
  bias(b) = tensor([2.9878], grad_fn=<AbsBackward0>)
```

**Prediction**

```
pred =  x @ w.T + b
pred[:5]
```

**Output:**

```
tensor([[-2.9765],
        [-3.1385],
        [-3.0818],
        [-3.0756],
        [-2.8681]], grad_fn=<SliceBackward0>)
```

# Different Variants of Gradient Descent

There are several variants of gradient descent that differ in the way the step size or learning rate is chosen and the way the updates are made. Here are some popular variants:

**Batch Gradient Descent**

In batch gradient descent, To update the model parameter values like weight and bias, the entire training dataset is used to compute the gradient and update the parameters at each iteration. This can be slow for large datasets but may lead to a more accurate model. It is effective for convex or relatively smooth error manifolds because it moves directly toward an optimal solution by taking a large step in the direction of the negative gradient of the cost function. However, it can be slow for large datasets because it computes the gradient and updates the parameters using the entire training dataset at each iteration. This can result in longer training times and higher computational costs.

**Stochastic Gradient Descent (SGD)**

In SGD, only one training example is used to compute the gradient and update the parameters at each iteration. This can be faster than batch gradient descent but may lead to more noise in the updates.

**Mini-batch Gradient Descent**

**I**n Mini-batch gradient descent a small batch of training examples is used to compute the gradient and update the parameters at each iteration. This can be a good compromise between batch gradient descent and Stochastic Gradient Descent, as it can be faster than batch gradient descent and less noisy than Stochastic Gradient Descent.

**Momentum-based Gradient Descent**

In momentum-based gradient descent, Momentum is a variant of gradient descent that incorporates information from the previous weight updates to help the algorithm converge more quickly to the optimal solution. Momentum adds a term to the weight update that is proportional to the running average of the past gradients, allowing the algorithm to move more quickly in the direction of the optimal solution. The updates to the parameters are based on the current gradient and the previous updates. This can help prevent the optimization process from getting stuck in local minima and reach the global minimum faster.

**Nesterov Accelerated Gradient (NAG)**

Nesterov Accelerated Gradient (NAG) is an extension of Momentum Gradient Descent. It evaluates the gradient at a hypothetical position ahead of the current position based

on the current momentum vector, instead of evaluating the gradient at the current position. This can result in faster convergence and better performance.

**Adagrad**

In Adagrad, the learning rate is adaptively adjusted for each parameter based on the historical gradient information. This allows for larger updates for infrequent parameters and smaller updates for frequent parameters.

**RMSprop**

In RMSprop the learning rate is adaptively adjusted for each parameter based on the moving average of the squared gradient. This helps the algorithm to converge faster in the presence of noisy gradients.

**Adam**

Adam stands for adaptive moment estimation, it combines the benefits of Momentum-based Gradient Descent, Adagrad, and RMSprop the learning rate is adaptively adjusted for each parameter based on the moving average of the gradient and the squared gradient, which allows for faster convergence and better performance on non-convex optimization problems. It keeps track of two exponentially decaying averages the first-moment estimate, which is the exponentially decaying average of past gradients, and the second-moment estimate, which is the exponentially decaying average of past squared gradients. The first-moment estimate is used to calculate the momentum, and the second-moment estimate is used to scale the learning rate for each parameter. This is one of the most popular optimization algorithms for deep learning.

## 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.