Gradient Descent Algorithm in Machine Learning

Think about how a machine learns from the data in machine learning and deep learning during training. This involves a large amount of data.

Through the lens of this article, we will delve into the intricacies of minimizing the cost function, a pivotal task in training models.

Table of Content

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.

Gradient Descent Python Implementation

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

Import the necessary libraries

Python3

```
import  torch

import  torch.nn as nn

import  matplotlib.pyplot as plt
```

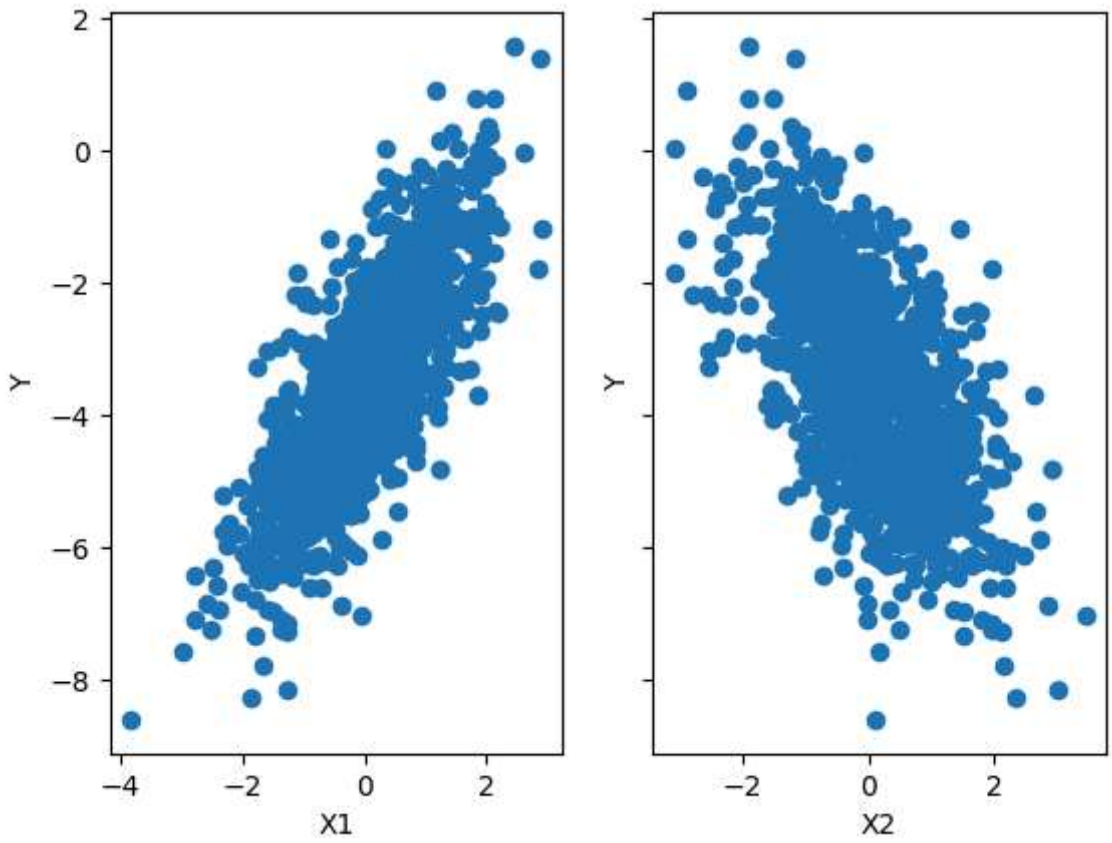
Set the input and output data

Python3

```
torch.manual_seed(``42``)

num_samples  =  1000

x  =  torch.randn(num_samples,  2``)
```

```
true_weights  =  torch.tensor([``1.3``,  -``1``])

true_bias    =  torch.tensor([``-``3.5``])

y  =  x @ true_weights.T  +  true_bias

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:

Python3

```
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

input_size  =  x.shape[``1``]

output_size  =  1

model  =  LinearRegression(input_size, output_size)
```

Note:

The number of weight values will be equal to the input size of the model, And the input size in deep Learning is the number of independent input features i.e we are putting inside the model

In our case, input features are two so, the input size will also be two, and the corresponding weight value will also be two.

We can manually set the model parameter

Python3

```
weight  =  torch.randn(``1``, input_size)

bias  =  torch.rand(``1``)

weight_param  =  nn.Parameter(weight)
```

```
bias_param   =  nn.Parameter(bias)

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

Python3

Output:

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

Define the loss function

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.

Python3

```
def  Mean_Squared_Error(prediction, actual):

   error  =  (actual``-``prediction)``*``*``2

   return  error.mean()

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.

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

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

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

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

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

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

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.

Here,

In our case:

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

```
# 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

Python3

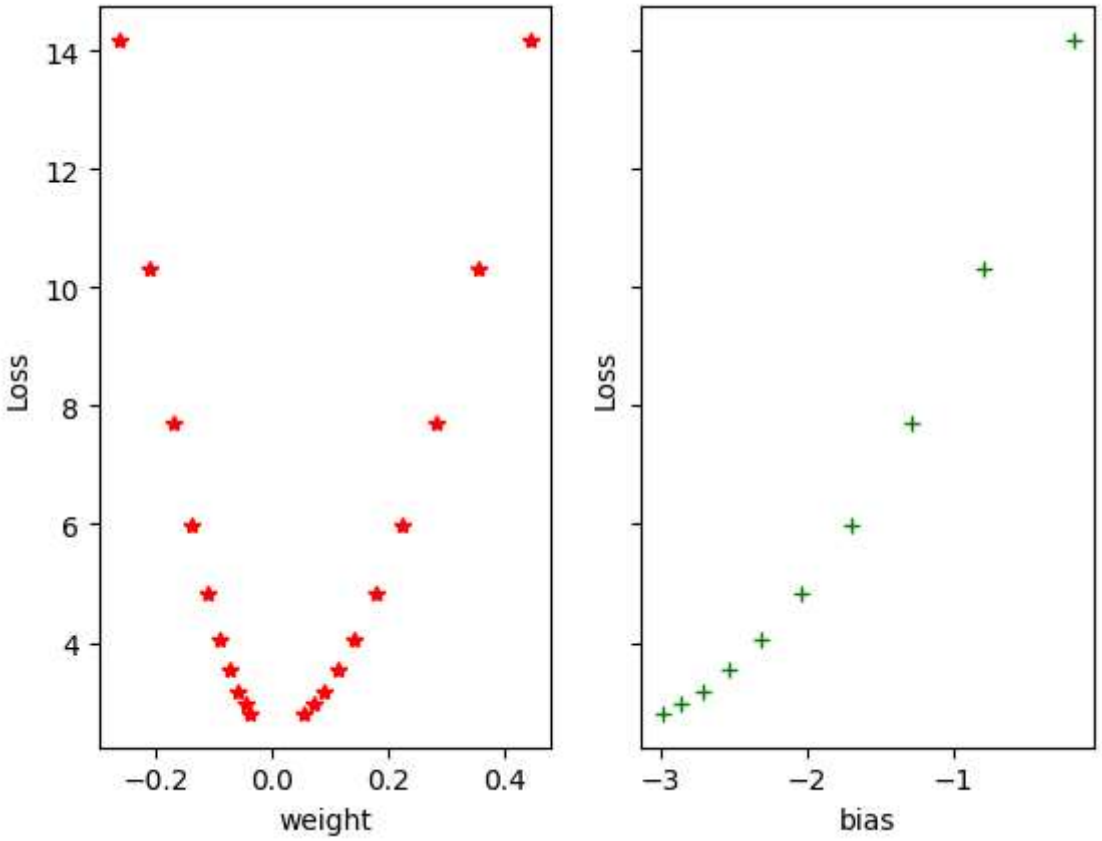
```
num_epochs  =  1000

learning_rate  =  0.01

fig, (ax1, ax2)  =  plt.subplots(``1``,  2``, sharey``=``True``)

for  epoch  in  range``(num_epochs):

  y_p  =  model(x)

  loss  =  Mean_Squared_Error(y_p, y)

  loss.backward()
```

```
    learning_rate  =  0.001

    w  =  model.linear.weight

    b  =  model.linear.bias

    w  =  w  -  learning_rate  *  w.grad

    b  =  b  -  learning_rate  *  b.grad

    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

Python3

```
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

Python3

```
pred  =  x @ w.T  +  b
```

```
pred[:``5``]
```

Output:

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

Gradient Descent Learning Rate

The learning rate is a critical hyperparameter in the context of gradient descent, influencing the size of steps taken during the optimization process to update the model parameters. Choosing an appropriate learning rate is crucial for efficient and effective model training.

When the learning rate is **too small**, the optimization process progresses very slowly. The model makes tiny updates to its parameters in each iteration, leading to sluggish convergence and potentially getting stuck in local minima.

On the other hand, an **excessively large learning rate** can cause the optimization algorithm to overshoot the optimal parameter values, leading to divergence or oscillations that hinder convergence.

Achieving the right balance is essential. A small learning rate might result in vanishing gradients and slow convergence, while a large learning rate may lead to overshooting and instability.

Vanishing and Exploding Gradients

Vanishing and exploding gradients are common problems that can occur during the training of deep neural networks. These problems can significantly slow down the training process or even prevent the network from learning altogether.

The vanishing gradient problem occurs when gradients become too small during backpropagation. The weights of the network are not considerably changed as a result, and the network is unable to discover the underlying patterns in the data. Many-layered deep neural networks are especially prone to this issue. The gradient values fall exponentially as they move backward through the layers, making it challenging to efficiently update the weights in the earlier layers.

The exploding gradient problem, on the other hand, occurs when gradients become too large during backpropagation. When this happens, the weights are updated by a large amount, which can cause the network to diverge or oscillate, making it difficult to converge to a good solution.

To address these problems the following technique 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 also help to mitigate the vanishing gradient problem.
- **Gradient clipping:** It involves limiting the maximum and minimum values of the gradient during backpropagation. This can prevent the gradients from becoming too large or too small and can help to stabilize the training process.
- **Batch normalization:** It can also help to address these problems by normalizing the input to each layer, which can prevent the activation function from saturating and help to reduce the vanishing and exploding gradient problems.

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

In 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)

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.

Conclusion

In the intricate landscape of machine learning and deep learning, the journey of model optimization revolves around the foundational concept of gradient descent and its diverse variants. Through the lens of this powerful optimization algorithm, we explored the intricacies of minimizing the cost function, a pivotal task in training models.

Gradient Descent In Machine Learning-FAQs

What is the fastest gradient descent algorithm?

> The fastest gradient descent algorithm depends on the specific problem being solved. However, some of the most commonly used variants include stochastic gradient descent (SGD), mini-batch gradient descent, and momentum-based gradient descent.

What is the gradient descent algorithm gda

> Gradient Descent Algorithm (GDA) is an iterative optimization algorithm used to find the minimum of a function. It works by repeatedly moving in the direction of the negative gradient of the function, which is the direction that leads to the steepest descent.

Why is gradient descent used?

> Gradient descent is used to find the minimum of a function. This is useful for many optimization problems, such as training machine learning models and minimizing the cost of a function.

Which is an example of gradient descent algorithm?

> An example of a gradient descent algorithm is linear regression. In linear regression, gradient descent is used to find the coefficients of a linear model that best fits a set of data points.

What is the formula of the gradient?

> The formula for the gradient of a function f(x) is: $$\nabla f(x) = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right]$$
> where x is a vector of input variables and f/x_i is the partial derivative of f with respect to x_i