Gradient Descent Method

Group 9

March 2023

1 Gradient Descent Method

Gradient descent is an iterative optimization algorithm used for minimizing the cost function in a model. The idea behind gradient descent is to update the model parameters in small steps towards the direction of steepest descent of the cost function. In other words, the algorithm aims to find the local minimum of the cost function by adjusting the model parameters in the direction of decreasing cost

Gradient Descent Method Explanation

To explain in brief about gradient descent, imagine that you are on a mountain and are blindfolded and your task is to come down from the mountain to the flat land without assistance.
The only assistance you have is a gadget which tells you the height from sea-level. What would be your approach be.
You would start to descend in some random direction and then ask the gadget what is the height now. If the gadget tells you that height and it is more than the initial height then you know you started in wrong direction.
You change the direction and repeat the process in many iterations untill you finally successfully descend down.

Well here is the analogy with machine learning terms now:
a. Size of Steps took in any direction = Learning rate
b. Gadget tells you height = Cost function
c. The direction of your steps = Gradients

Cost Function Gradients

The equation for calculating cost function and gradients are as shown below. Note: For other algorithms the cost function will be different and the gradients would have to be derived from the cost functions

Cost

$$J(\theta) = 1/2m \sum_{i=1}^{m} (h(\theta)^{(i)} - y^{(i)})^2 \tag{1}$$

Gradient

$$\frac{\partial J(\theta)}{\partial \theta_j} = 1/m \sum_{i=1}^{m} (h(\theta^{(i)} - y^{(i)}).X_j^{(i)} \tag{2}$$

Other Gradients

$$\theta_0 := \theta_0 - \alpha.(1/m. \sum_{i=1}^{m} (h(\theta^{(i)} - y^{(i)}).X_0^{(i)}) \tag{3}$$

$$\theta_1 := \theta_1 - \alpha.(1/m. \sum_{i=1}^{m} (h(\theta^{(i)} - y^{(i)}).X_1^{(i)}) \tag{4}$$

$$\theta_2 := \theta_2 - \alpha.(1/m. \sum_{i=1}^{m} (h(\theta^{(i)} - y^{(i)}).X_2^{(i)}) \tag{5}$$

$$\theta_j := \theta_j - \alpha.(1/m. \sum_{i=1}^{m} (h(\theta^{(i)} - y^{(i)}).X_0^{(i)}) \tag{6}$$

The goal of the gradient descent algorithm is to find the values of the slope m and the intercept b of the line that best fit the given data points.

The gradient descent algorithm minimizes the cost function, which is the sum of squared errors between the predicted y values and the actual y values.

The algorithm adjusts the values of m and b iterate until the cost function is minimized. The line represents the final values of m and b that give the best fit to the data points.

The purpose of plotting the line and the dotted points together is to visually assess how well the linear regression model fits the data.

If the line fits the points closely, then the model is a good fit for the data, and can be used to make predictions about new values of the independent variable.

Python code

```
from timeit import Timer
import numpy as np
import matplotlib.pyplot as  plt

def gradient_descent(x,y):
    m_curr=b_curr=0

    iterations = 100
    n = len(x)
    learning_rate = 0.08

    for i in range(0, iterations):
        y_predicted = m_curr * x + b_curr
        cost = (1/n) * sum([val**2 for val in (y-y_predicted)])
        md = -(2/n)*sum(x*(y-y_predicted))
        bd = -(2/n)*sum(y-y_predicted)
        m_curr = m_curr - learning_rate * md
        b_curr = b_curr - learning_rate * bd
        print ("m: {}, b: {}, cost: {} iteration: {}".format(m_curr,b_curr,cost, i))
        t = Timer("gradient_descent", "from __main__ import gradient_descent")
        print("Time taken: " +str(t.timeit(1))+ "seconds.")
    plt.plot(x, y, 'ro')
    plt.plot(x, m_curr*x+b_curr, 'b-')
    plt.grid()
    plt.show()

x = np.array([1,2,3,4,5])
y = np.array([5,7,9,11,13])
gradient_descent(x,y)
```

2 Mini-Batch Descend Method

Mini-batch gradient descent is a popular optimization algorithm used in machine learning for training deep neural networks. It is a variant of the gradient descent algorithm that updates the model weights in small batches rather than updating the weights after every training example.

In mini-batch gradient descent, the training dataset is divided into small batches of fixed size. Each batch contains a subset of the training data, and the algorithm updates the model weights using the gradient of the loss function computed on that batch. The size of the batch is typically chosen based on the available memory resources and the computational efficiency of the algorithm. Common batch sizes are in the range of 32 to 256. The algorithm updates the model weights using the following steps:

1.Initialize the model weights with random values.

2.Divide the training data set into small batches of fixed size.

3.For each batch, compute the gradient of the loss function with respect to the model weights.

4.Update the model weights by subtracting a fraction of the gradient from the current weights. This fraction is called the learning rate and controls the step size of the update.

5.Repeat steps 3 and 4 for a fixed number of iterations or until convergence.

Below is a python code showing how the mini batch gradient descent method works:

Python code

```
import numpy as np
import matplotlib.pyplot as plt
import timeit

def mini_batch_gradient_descent(X, y, alpha=0.01, epochs=100, batch_size=32):
    """
    Implementation of mini-batch gradient descent.
    Parameters:
    X: numpy array of shape (m, n)
       Training examples, where m is the number of examples and n is the number of features
    y: numpy array of shape (m, 1)
       Target values.
    alpha: float
       Learning rate.
    epochs: int
       Number of passes over the training set.
    batch_size: int
       Size of mini-batch.
    Returns:
    theta: numpy array of shape (n, 1)
       Optimal parameters.
    loss: list
       Training loss at each epoch.
    """
    m, n = X.shape
    theta = np.zeros((n, 1))
    loss = []

    for epoch in range(epochs):
        # Shuffle the training set
        permutation = np.random.permutation(m)
        X = X[permutation]
        y = y[permutation]

        for i in range(0, m, batch_size):
            # Get mini-batch
            X_batch = X[i:i+batch_size]
            y_batch = y[i:i+batch_size]

            # Compute gradient
            grad = 1/batch_size * X_batch.T @ (X_batch @ theta - y_batch)

            # Update parameters
            theta = theta - alpha * grad
        # Compute training loss
        loss.append(1/m * np.sum((X @ theta - y)**2))

    return theta, loss

# Generate synthetic data
m = 1000
n = 10
X = np.random.randn(m, n)
y = X @ np.random.randn(n, 1)
# Train model using mini-batch gradient descent
start_time = timeit.default_timer()
theta, loss = mini_batch_gradient_descent(X, y)
end_time = timeit.default_timer()
print(f"Training time: {end_time - start_time:.4f} seconds")

# Plot training loss over time
plt.plot(loss)
plt.xlabel('Epoch')
plt.ylabel('Training Loss')
plt.show()
```

The mini-batch gradient descent function takes in the training examples $\mathbf{x}$, target values $\mathbf{y}$, learning rate α, number of passes over the training set *epochs*, and size of mini-batch $batch_size$. It returns the optimal parameters θ and the training loss at each epoch *loss*.
The training process involves shuffling the training set and processing it in mini-batches. For each mini-batch, the gradient of the cost function is computed and used to update the parameters. The training loss is also computed at each epoch and stored in *loss*.
The script generates synthetic data with $m = 1000$ examples and $n = 10$ features. The model is trained using mini-batch gradient descent and the training time is printed. The training loss over time is then plotted using 'matplotlib'.

3 Stochastic Gradient Descend Method

Stochastic gradient descent is an iterative optimization algorithm that updates the parameters of a model by processing one training example at a time. The algorithm shuffles the training set and then processes each example in a random order, making small incremental updates to the model's parameters after each example is processed. The goal of stochastic gradient descent is to minimize the cost function by finding the optimal set of parameters that produce the smallest error on the training data.

In the provided code, the function sgd takes in the following parameters: the gradient function gradient, the training examples x, the target values y, the starting point for the optimization start, the learning rate learning rate, the size of the mini-batches batch-size, the number of iterations n-iterations, the tolerance level for convergence tolerance, the data type dtype, and the random seed random-state.

Inside the sgd function, the data is converted to NumPy arrays, the learning rate and tolerance are checked to ensure they are valid, and the gradient descent loop is performed using mini-batches of the specified size. The gradient function calculates the gradient of the cost function with respect to the model parameters for a given mini-batch.

During the optimization process, the cost history array is updated with the current values of the model parameters at each iteration. After the optimization is complete, this array contains the values of the cost function for each iteration. These values can be plotted to visualize the convergence of the algorithm.

Python code:

```
#Get gradient by doing the derivative of the cost function
def gradient(x,y,theta):
    m=len(y)
    sum=0
    for i in range(m):
        sum+=(-1)*(1/m*(np.sum(np.square((y[i]-2*x[i]))-theta)))
    return sum
import numpy as np

def sgd(gradient, x, y, start, learn_rate=0.1, batch_size=1, n_iter=50,tolerance=1e-06, dtyp

    # Setting up the data type for NumPy arrays
    dtype_ = np.dtype(dtype)

    #create an array filled with zeros
    cost_history = np.zeros(iterations)

    # Converting x and y to NumPy arrays
    x, y = np.array(x, dtype=dtype_), np.array(y, dtype=dtype_)
    n_obs = x.shape[0]

    if n_obs != y.shape[0]:
        raise ValueError("'x' and 'y' lengths do not match")
    xy = np.c_[x.reshape(n_obs, -1), y.reshape(n_obs, 1)]

    # Initializing the random number generator
    seed = None if random_state is None else int(random_state)
    rng = np.random.default_rng(seed=seed)

    # Initializing the values of the variables
    vector = np.array(start, dtype=dtype_)

    # Setting up and checking the learning rate
    learn_rate = np.array(learn_rate, dtype=dtype_)
    if np.any(learn_rate <= 0):
        raise ValueError("'learn_rate' must be greater than zero")
     # Setting up and checking the size of minibatches
    batch_size = int(batch_size)
    if not 0 < batch_size <= n_obs:
        raise ValueError(
            "'batch_size' must be greater than zero and less than "
            "or equal to the number of observations"
        )

    # Setting up and checking the tolerance
    tolerance = np.array(tolerance, dtype=dtype_)
    if np.any(tolerance <= 0):
        raise ValueError("'tolerance' must be greater than zero")

    # Performing the gradient descent loop
    for _ in range(n_iter):
        # Shuffle x and y
        rng.shuffle(xy)

        # Performing minibatch moves
        for start in range(0, n_obs, batch_size):
            stop = start + batch_size
            x_batch, y_batch = xy[start:stop, :-1], xy[start:stop, -1:]

            # Recalculating the difference
            grad = np.array(gradient(x_batch, y_batch, vector), dtype_)
            diff = -learn_rate * grad

            # Checking if the absolute difference is small enough
            if np.all(np.abs(diff) <= tolerance):
                break

            # Updating the values of the variables
            vector += diff
            cost_history[start]= vector
         return vector if vector.shape else vector.item()

#initialise values
```

```
x=np.random.rand(100,1)
y=2*x+ np.random.rand()
#call the function
sgd(gradient, x, y, start=1,learn_rate=0.1, batch_size=1, n_iter=50,tolerance=1e-06, dtype='

print("Values of cost_function: ",cost_history)
plt.plot(range(10),cost_history)
plt.xlabel('Number of iterations')
plt.ylabel('Cost')
```