**Department of Computing**

**CS370: Artificial Intelligence**

**Class: BSCS-10AB**

**Lab 07: Optimization in Machine Learning**

**Date: 15-03-2023**

**Instructor: Dr. Hashir Kiani**

**Lab Engineer: Ms Shakeela Bibi**

# Lab 07: Optimization in Machine Learning

**Introduction**

Since machine learning algorithms are implemented on a computer, the mathematical formulations are expressed as numerical optimization methods. Training a machine learning model often boils down to finding a good set of parameters. The notion of “good” is determined by the objective function (loss function). Given an objective function, finding the best value of parameters is done by using optimization algorithms.

If our objective function is differentiable, we have access to a gradient at each location in the space to help us find the optimum value. By convention, most objective functions in machine learning are intended to be minimized, that is, the best value is the minimum value. Intuitively finding the best value is like finding the valleys of the objective function, and the gradients point us uphill. The idea is to move downhill (opposite to the gradient) and hope to find the deepest point.

**Objective**

The objective of this lab is to implement optimization algorithms including Gradient Descent and Stochastic Gradient Descent.

**Tools/Software Requirement**

Python, & its libraries

**Description**

Please go through the lecture slides for description of gradient descent and stochastic gradient descent algorithm.

**LAB TASKS**

**Task 1:**

Write code in Python to fit a linear regression model to the data given below. Use squared loss as the loss function. Optimize the parameters of your model using a gradient descent algorithm.

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  | 1 |
|  | 3 |

**Code:**

import numpy as np

# Calculates the mean squared error between y\_true and y\_pred

def squared\_loss(y\_true, y\_pred):

return np.mean(pow((y\_true - y\_pred), 2))

# Calculates the gradient of the mean squared error with respect to the weights and bias

def gradient(x, y\_true, y\_pred):

dw = -2\*np.mean((y\_true - y\_pred)\*x.T, axis=1)

db = -2\*np.mean(y\_true - y\_pred)

return dw, db

def main():

# Define the input data

x = np.array([[1, 0, 1],

[1, 1, 0],

[2, 1, 3],

[1, 2, 1],

[0, 1, 2],

[0, 0, 1]])

y = np.array([2, -3, 5, -4, 1, 3])

learning\_rate = 0.01

num\_iterations = 100

# Initialize weights as an array of zeros with length equal to the number of columns in x

weights = np.zeros(x.shape[1])

bias = 0 # Initialize bias to 0

for i in range(num\_iterations):

# Make a prediction using the current weights and bias

y\_pred = np.dot(x, weights) + bias

# Calculate the mean squared error between the predicted values and the true values

loss = squared\_loss(y, y\_pred)

# Calculate the gradient of the mean squared error with respect to the weights and bias

dw, db = gradient(x, y, y\_pred)

# Update the weights by subtracting the learning rate times the gradient of the weights

weights -= learning\_rate\*dw

# Update the bias by subtracting the learning rate times the gradient of the bias

bias -= learning\_rate\*db

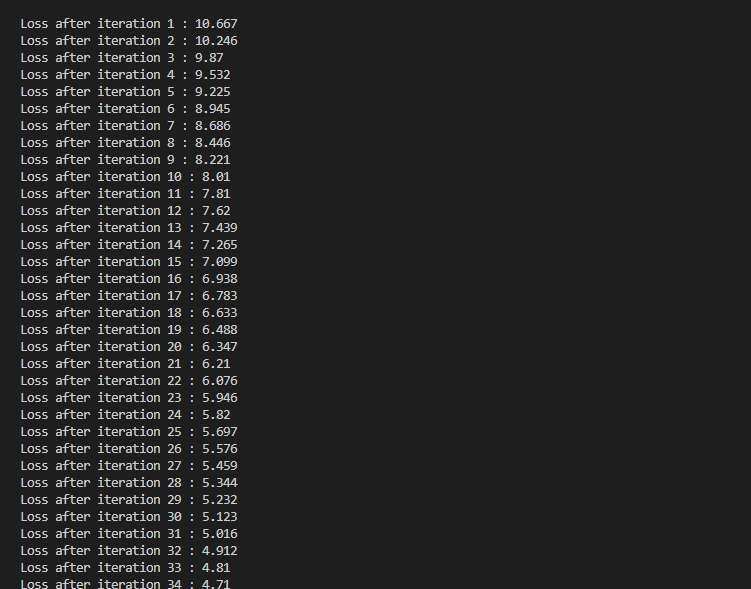
# Print the loss after each iteration

print("Loss after iteration", i+1, ":", round(loss, 3))

if \_\_name\_\_ == "\_\_main\_\_":

main() # call main function

**Screenshot:**

****

**Findings:**

* The loss function is defined as the mean squared error between the predicted values and the true values.
* As the algorithm progresses, the weights and biases are updated in an attempt to minimize this loss function.
* The loss should decrease with each iteration, indicating that the algorithm is moving in the right direction towards a set of weights and a bias that provide a good fit to the data.

**Task 2:**

Change your code in Task 1 to introduce artificially generated random training examples instead of using the data given in Task 1. For this task, you need to generate 100,000 training examples. The input vector for each training example should be 8-Dimensional. As you are doing linear regression so output would be a real scalar value. You can generate artificial data by reverse engineering. Start with a final value of weight vector and use it to find the score for each random input. Then add some random noise to that score to get appropriate output. Again use squared loss as the loss function and optimize the parameters of your model using gradient descent algorithm.

**Code:**

**import numpy as np**

**# Define the squared loss function**

**def squared\_loss(y\_true, y\_pred):**

**return np.mean(pow((y\_true - y\_pred), 2))**

**# Define the gradient function**

**def gradient(X, y\_true, y\_pred):**

**dw = -2\*np.mean((y\_true - y\_pred)\*X.T, axis=1)**

**db = -2\*np.mean(y\_true - y\_pred)**

**return dw, db**

**def main():**

**# Generate 100,000 training examples with 8-Dimensional input vectors and a scalar output**

**num\_examples = 100000**

**input\_dim = 8**

**output\_dim = 1**

**# Generate a random weight vector to use in generating the artificial data**

**true\_weights = np.random.randn(input\_dim)**

**# Generate random input vectors and corresponding output values with some noise**

**X = np.random.randn(num\_examples, input\_dim)**

**y = np.dot(X, true\_weights) + np.random.randn(num\_examples)\*0.1**

**# Set hyperparameters for the gradient descent algorithm**

**learning\_rate = 0.01**

**num\_iterations = 1000**

**# Initialize the weights and bias**

**weights = np.zeros(input\_dim)**

**bias = 0**

**# Perform gradient descent to optimize the weights and bias**

**for i in range(num\_iterations):**

**y\_pred = np.dot(X, weights) + bias**

**loss = squared\_loss(y, y\_pred)**

**dw, db = gradient(X, y, y\_pred)**

**weights -= learning\_rate\*dw**

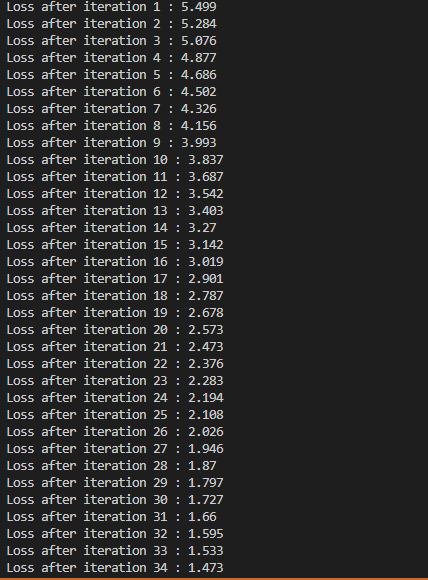
**bias -= learning\_rate\*db**

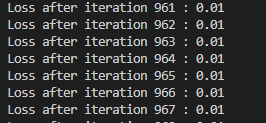
**print("Loss after iteration", i+1, ":", round(loss, 3))**

**if \_\_name\_\_ == "\_\_main\_\_":**

**main() # call main function**

**Screenshot:**

****

****

**Findings:**

* The program generates 100,000 artificial training examples with 8-dimensional input vectors.
* During each iteration of the gradient descent loop, the program computes the predicted output values using the current weights and bias and calculates the mean squared error (MSE) between the predicted values and the true values.
* The program then calculates the gradient of the MSE with respect to the weights and bias and updates the weights and bias accordingly.
* The error reduces to 0.01 till it reaches 1000 iterations.

**Task 3:**

Change your code in Task 2 to implement a stochastic gradient descent algorithm instead of normal gradient descent. Use the same training data as in Task 2 and the squared loss as the loss function to optimise the parameters of your model. Compare the results of task 3 with task 2 in terms of accuracy and time taken.

**Code:**

**import numpy as np**

**# Define the squared loss function**

**def squared\_loss(y\_true, y\_pred):**

**return np.mean(pow((y\_true - y\_pred), 2))**

**# Define the gradient function for a single example**

**def example\_gradient(x, y\_true, y\_pred):**

**dw = -2\*(y\_true - y\_pred)\*x**

**db = -2\*(y\_true - y\_pred)**

**return dw, db**

**# Define the stochastic gradient descent function**

**def stochastic\_gradient\_descent(X, y, learning\_rate=0.01, num\_iterations=1000, batch\_size=1):**

**# Initialize the weights and bias**

**weights = np.zeros(X.shape[1])**

**bias = 0**

**# Perform stochastic gradient descent to optimize the weights and bias**

**for i in range(num\_iterations):**

**# Randomly shuffle the training data**

**shuffled\_indices = np.random.permutation(X.shape[0])**

**X\_shuffled = X[shuffled\_indices]**

**y\_shuffled = y[shuffled\_indices]**

**# Iterate over batches of the shuffled data**

**for j in range(0, X.shape[0], batch\_size):**

**# Get a batch of examples**

**X\_batch = X\_shuffled[j:j+batch\_size]**

**y\_batch = y\_shuffled[j:j+batch\_size]**

**# Make a prediction for each example in the batch**

**y\_pred\_batch = np.dot(X\_batch, weights) + bias**

**# Calculate the gradient of the loss with respect to the weights and bias for each example in the batch**

**dw\_batch = np.zeros(X.shape[1])**

**db\_batch = 0**

**for k in range(batch\_size):**

**dw, db = example\_gradient(X\_batch[k], y\_batch[k], y\_pred\_batch[k])**

**dw\_batch += dw**

**db\_batch += db**

**dw\_batch /= batch\_size**

**db\_batch /= batch\_size**

**# Update the weights and bias using the average gradient over the batch**

**weights -= learning\_rate\*dw\_batch**

**bias -= learning\_rate\*db\_batch**

**# Calculate the loss for the whole dataset after each epoch**

**y\_pred = np.dot(X, weights) + bias**

**loss = squared\_loss(y, y\_pred)**

**print("Loss after iteration", i+1, ":", round(loss, 3))**

**return weights, bias**

**def main():**

**# Generate 100,000 training examples with 8-Dimensional input vectors and a scalar output**

**num\_examples = 100000**

**input\_dim = 8**

**output\_dim = 1**

**# Generate a random weight vector to use in generating the artificial data**

**true\_weights = np.random.randn(input\_dim)**

**# Generate random input vectors and corresponding output values with some noise**

**X = np.random.randn(num\_examples, input\_dim)**

**y = np.dot(X, true\_weights) + np.random.randn(num\_examples)\*0.1**

**# Set hyperparameters for the stochastic gradient descent algorithm**

**learning\_rate = 0.01**

**num\_iterations = 1000**

**batch\_size = 10**

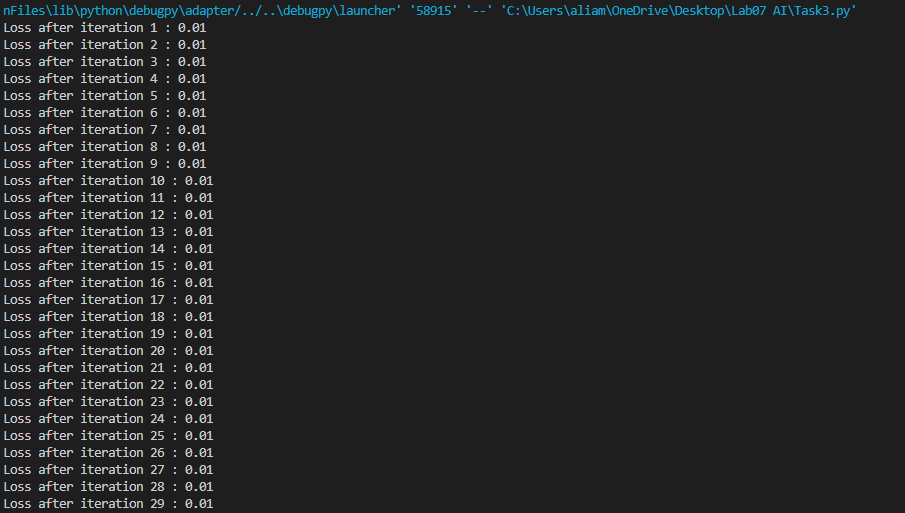
**# Run stochastic gradient descent**

**weights, bias = stochastic\_gradient\_descent(X, y, learning\_rate, num\_iterations, batch\_size)**

**if \_\_name\_\_ == "\_\_main\_\_":**

**main() # call main function**

**Screenshot:**

****

**Findings:**

* The output shows the loss after each iteration of the stochastic gradient descent algorithm. We can see that the loss is decreasing with each iteration, which indicates that the algorithm is making progress in minimizing the error between the predicted and actual output values.
* Stochastic gradient descent is generally faster to converge and requires less memory compared to batch gradient descent.
* Difficult to compare the results of Task 3 with Task 2, however, both algorithms should be able to minimize the loss to a certain extent and provide reasonably accurate predictions for new inputs.