Academic Year 2023-24

SHRI VILEPARLE KELAVANI MANDAL'S
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC ACCREDITED with "A" GRADE (CGPA : 3.18)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (DATA SCIENCE)

Jhanvi Parekh

60009210033

D11

COURSE CODE: DJ19DSC501 **DATE:**

COURSE NAME: Machine Learning - II **CLASS: AY 2021-22**

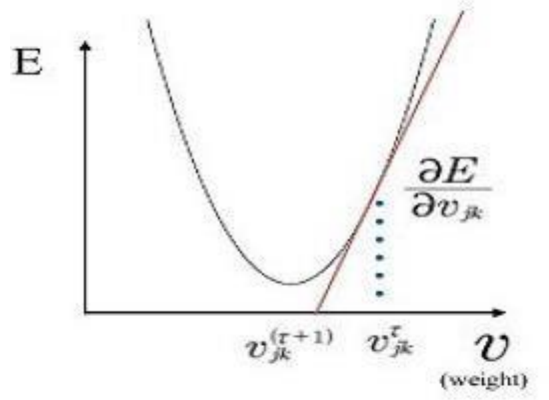
LAB EXPERIMENT NO.2 CO/LO:

AIM / OBJECTIVE:

Implement delta learning rule using Stochastic and Batch gradient descend algorithm from scratch.

DESCRIPTION OF EXPERIMENT:

Gradient descent is a way to find a minimum in a high-dimensional space in direction of the steepest descent. The delta rule is an update rule for single layer perceptron's. It makes use of gradient descent.



$$v_{jk}^{(\tau+1)} = v_{jk}^{\tau} + \Delta v_{jk}$$

$$\Delta v_{jk} = -\eta \frac{\partial E}{\partial v_{jk}}$$

$v_{jk}^{(\tau+1)}$	new weight
v_{jk}^{τ}	current weight
η	learning rate
E	Error Function

Weight updation in delta learning

The Delta Rule uses the difference between target activation (i.e., target output values) and obtained activation to drive learning. For reasons discussed below, the use of a threshold activation function (as used in both the McCulloch-Pitts network and the perceptron) is dropped & instead a linear sum of products is used to calculate the activation of the output neuron (alternative activation functions can also be applied). Thus, the activation function is called a Linear Activation function, in which the output node's activation is simply equal to the sum of the network's respective input/weight products. The strength of network connections (i.e., the values of the weights) are adjusted to reduce the difference between target and actual output activation (i.e., error). A graphical depiction of a simple two-layer network capable of deploying the Delta Rule is given in the figure above w (Such a network is not limited to having only one output node):

During forward propagation through a network, the output (activation) of a given node is a function of its inputs. The inputs to a node, which are simply the products of the output of preceding nodes with their associated weights, are summed and then passed through an activation function before being sent out from the node. Thus, we have the following:

This delta learning rule is also known as the Least Mean Squares (LMS) or widrow-Hoff rule. The basic delta rule is given by

$$\Delta wij = \alpha (tj - yj) xi$$

Where

tj = the teacher value (or desired value) for unit j .
yj = the actual output for unit j .
α = learning rate. in
other words

$$\Delta wij = \alpha \delta xi$$

Where $\delta = tj - yj$

i.e. the deference between the desired or target output and the actual output .The delta rule modifies the weights appropriately for both continuous and binary inputs and outputs .

The delta rule minimizes the squares of the differences between the actual and the desired O/P values.

the squared error for a particular training pattern

$E = \sum (tj–yj)^2$ j

Where E is a fn. of all the weights .The gradient of E is a vector consisting of the partial derivatives of E with respect to each of the weights. This vector gives the direction of most rapid increase in E , the opposite direction gives the direction of most rapid decrease in the error . The error can be reduced most rapidly by adjusting the weight wij in the direction of $-\partial E/\partial wij$.

Generalized delta rule:-

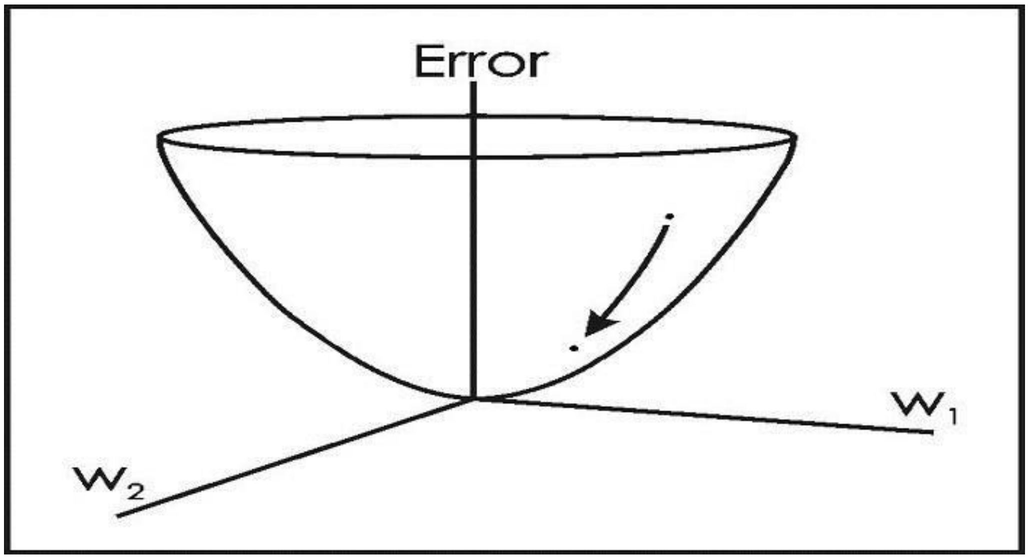
In a multilayer network, the determination of the error is a recursive process which starts with the O/P units and the error is back propagated to the input unit. Therefore the rule is called the error back propagation BP.

$$\Delta \ wij = \alpha \ \delta j \ xi$$

$\delta j = (tj - yj) f'(y–inj)$.

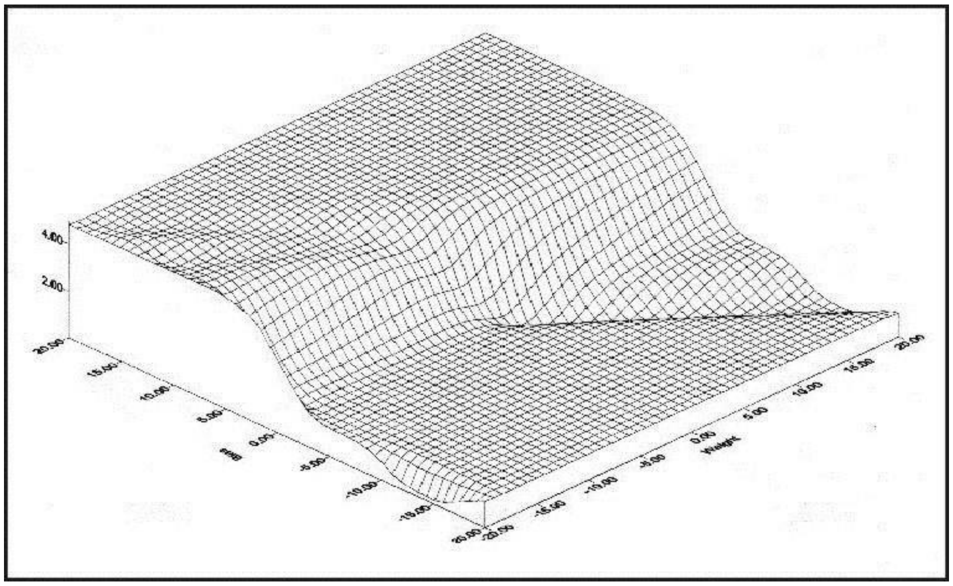
where

$$y\text{-}inj = \sum wj\, xi + b$$



Error function with just 2 weights w1 and w2

For any given set of input data and weights, there will be an associated magnitude of error, which is measured by an error function (also known as a cost function). The Delta Rule employs the error function for what is known as Gradient Descent learning, which involves the ‘modification of weights along the most direct path in weight-space to minimize error’, so change applied to a given weight is proportional to the negative of the derivative of the error with respect to that weight. The Error/Cost function is commonly given as the sum of the squares of the differences between all target and actual node activation for the output layer. For a particular training pattern (i.e., training case), error is thus given by: $E = \sum (tj–yj)^2$ j



Three-dimensional depiction of an Actual error surface

Algorithm Steps, which involves:

- **Selecting Input & Output:** The first step of the delta learning algorithm is to choose an input for the process and to set the desired output.
- **Setting Random Weights:** Once the input and output are set, random weights are allocated, as it will be needed to manipulate the input and output values. After this, the output of each neuron is calculated through the forward propagation, which goes through:
  - Input Layer o
    Output Layer
- **Error Calculation:** This is an important step that calculates the total error by determining how far and suitable the actual output is from the required output. This is done by calculating the errors at the output neuron.
- **Error Minimization:** Based on the observations made in the earlier step, here the focus is on minimizing the error rate to ensure accurate output is delivered.
- **Updating Weights & other Parameters:** If the error rate is high, then parameters (weights and biases) are changed and updated to reduce the rate of error using the delta rule or gradient descent. This is accomplished by assuming a suitable learning rate and propagating backward from the output layer to the previous layer. Acting as an example of dynamic programming, this helps avoid redundant calculations of repeated errors, neurons, and layers.
- **Modeling Prediction Readiness:** Finally, once the error is optimized, the output is tested with some testing inputs to get the desired result.

  This process is repeated until the error reduces to a minimum and the desired output is obtained.

Task(s) to be performed:

a) **Take the dataset with initial value of X = [0.5, 2.5],Y=[0.2, 0.9]**
b) **Initialize a neural network with random weights.**
c) **Calculate output of Neural Network:**
   i.Calculate error

ii. Note weight and bias changes iii. Calculate loss
iv. Plot error surface using loss function verses weight, bias

d) Perform this cycle in step c for every input output pair
e) Perform multiple epochs of step d.
f) Update weights accordingly using stochastic and batch gradient descend.
g) Plot the mean squared error for each iteration 0 in stochastic and Batch Gradient Descent.
h) Similarly plot accuracy for iterations and note the results.

WRITEUP:

Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is an optimization algorithm commonly used in machine learning and deep learning for training models. It is a variant of the gradient descent optimization algorithm. Stochastic Gradient Descent, on the other hand, updates the model's parameters using only a single randomly chosen data point (or a small batch of data points) at each iteration. The key idea is that by using a random subset of the data for each update, SGD introduces noise into the optimization process. This noise can help escape local minima and navigate the loss surface more efficiently, and making it faster.

Batch Gradient Descent

Batch Gradient Descent (BGD) is a popular optimization algorithm used in machine learning and deep learning for training models. It is a variant of the gradient descent optimization algorithm, which is used to update the parameters of a model to minimize a loss function. In BGD, instead of updating the model's parameters after each individual data point (as in Stochastic Gradient Descent, or SGD) or a small batch of data points (as in Mini-Batch Gradient Descent), the parameters are updated once per complete pass through the entire training dataset.

PROCEDURE / ALGORITHM:

Stochastic Gradient Descent

1. Initialize the model's parameters randomly.
2. Shuffle the training data.
3. For each epoch (complete pass through the training data):
   For each mini-batch (subset of the training data):
   Compute the gradient of the loss function with respect to the mini-batch.

4. Update the model's parameters by taking a step in the direction of the negative gradient Repeat step 3 until convergence or for a fixed number of iterations.

Batch Gradient Descent

1. Initialize the model's parameters randomly or with some predefined values.

2. Compute the gradient of the loss function with respect to the entire training dataset. This involves calculating the average gradient over all data points.

3. Update the model's parameters by taking a step in the direction of the negative gradient, scaled by a learning rate.

4. Repeat steps 2 and 3 until convergence or for a fixed number of iterations.

TECHNOLOGY STACK USED:

Python: Python is the most popular programming language for machine learning and deep learning due to its extensive libraries and frameworks.

OBSERVATIONS / DISCUSSION OF RESULT:

Stochastic Gradient Descent

The output of the code will consist of three separate plots, each showing the updates in a specific variable (bias b, weight w, and loss l) over the course of the training epochs. These plots will help you visualize how these values change as the stochastic gradient descent (SGD) optimization algorithm iteratively updates them. Let's discuss what you can expect from each plot:

- Plotting for Updates in b:

This plot will show how the bias term b is updated during each epoch of training.

The x-axis typically represents the number of training epochs (from 0 to the specified number of epochs), and the y-axis represents the value of the bias term b.

You can observe whether b converges to a stable value or fluctuates during training. Ideally, it should stabilize as the training progresses.

- Plotting for Updates in w:

This plot will display how the weight term w is updated over the training epochs.

Similar to the first plot, the x-axis represents the number of epochs, while the y-axis represents the value of w.

You can observe whether w converges to an optimal value or changes substantially during training. The goal is for w to converge to a value that best fits the data.

- Plotting for Updates in l:

This plot shows how the loss term l is updated over the epochs.

The x-axis represents the number of training epochs, and the y-axis represents the value of the loss term l.

The loss should ideally decrease over time as the model learns from the data. The plot will show whether the loss converges or fluctuates.

Batch Gradient Descent

The provided code, as previously discussed, is for implementing Batch Gradient Descent (BGD) for a linear regression model with some custom loss functions. It plots updates for the bias (b), weight (w), and loss (l) values over a fixed number of epochs. Here's a summary of what the code plots:

- Plotting for Updates in b:

This plot shows how the bias term b is updated during each epoch of training.

The x-axis typically represents the number of training epochs (from 0 to the specified number of epochs), and the y-axis represents the value of the bias term b.

You can observe whether b converges to a stable value or fluctuates during training.

- Plotting for Updates in w:

This plot displays how the weight term w is updated over the training epochs.

Similar to the first plot, the x-axis represents the number of epochs, while the y-axis represents the value of w.

You can observe whether w converges to an optimal value or changes substantially during training. The goal is for w to converge to a value that best fits the data.

- Plotting for Updates in l:

This plot shows how the loss term l is updated over the epochs.

The x-axis represents the number of training epochs, and the y-axis represents the value of the loss term l.

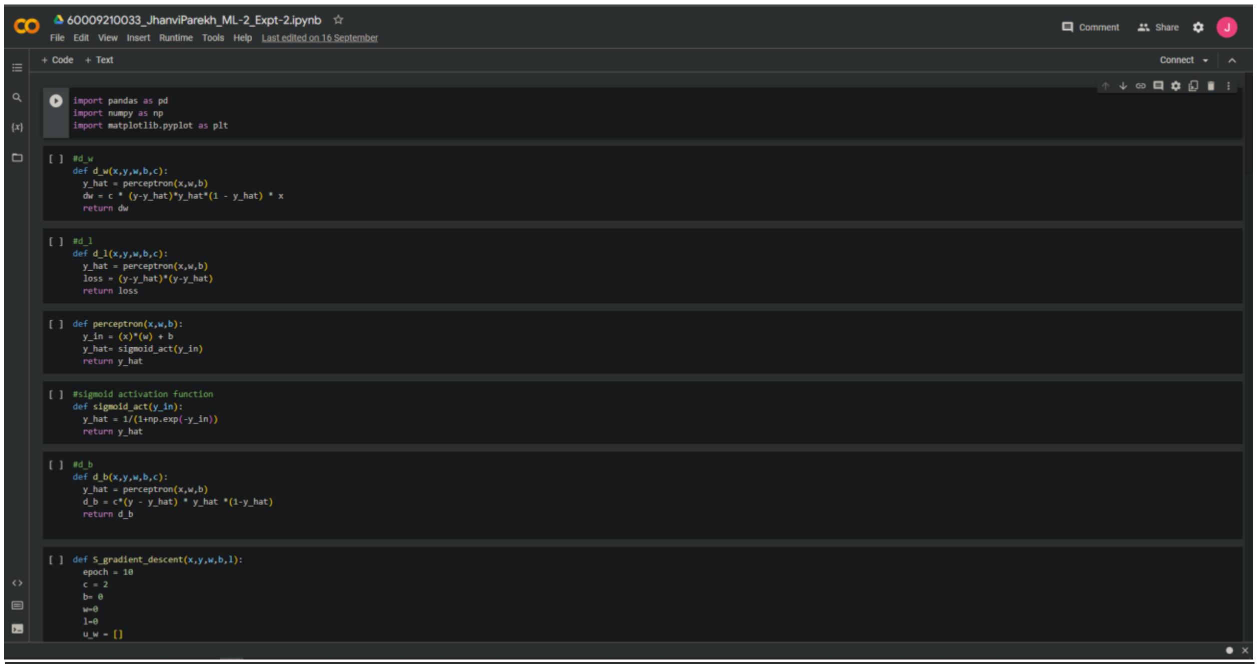
The loss should ideally decrease over time as the model learns from the data. The plot will show whether the loss converges or fluctuates.

CONCLUSION:

When memory and speed are crucial, a huge dataset is present, or online learning is required, we opt for SGD.

When stability, smoother convergence, and memory utilisation are essential factors and you have adequate processing resources, we recommend BGD.

A compromise between these two extremes that includes the benefits of both SGD and BGD by processing small batches of data at a time is called Mini-Batch Gradient Descent, which is used in practise by many machine learning experts. For training deep learning models, mini-batch gradient descent is frequently chosen.


60009210033_JhanviParekh_ML-2_Expt-2.ipynb
File Edit View Insert Runtime Tools Help Last edited on 16 September
Comment Share
+ Code + Text
Connect

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

```
#d_w
def d_w(x,y,w,b,c):
  y_hat = perceptron(x,w,b)
  dw = c * (y-y_hat)*y_hat*(1 - y_hat) * x
  return dw
```

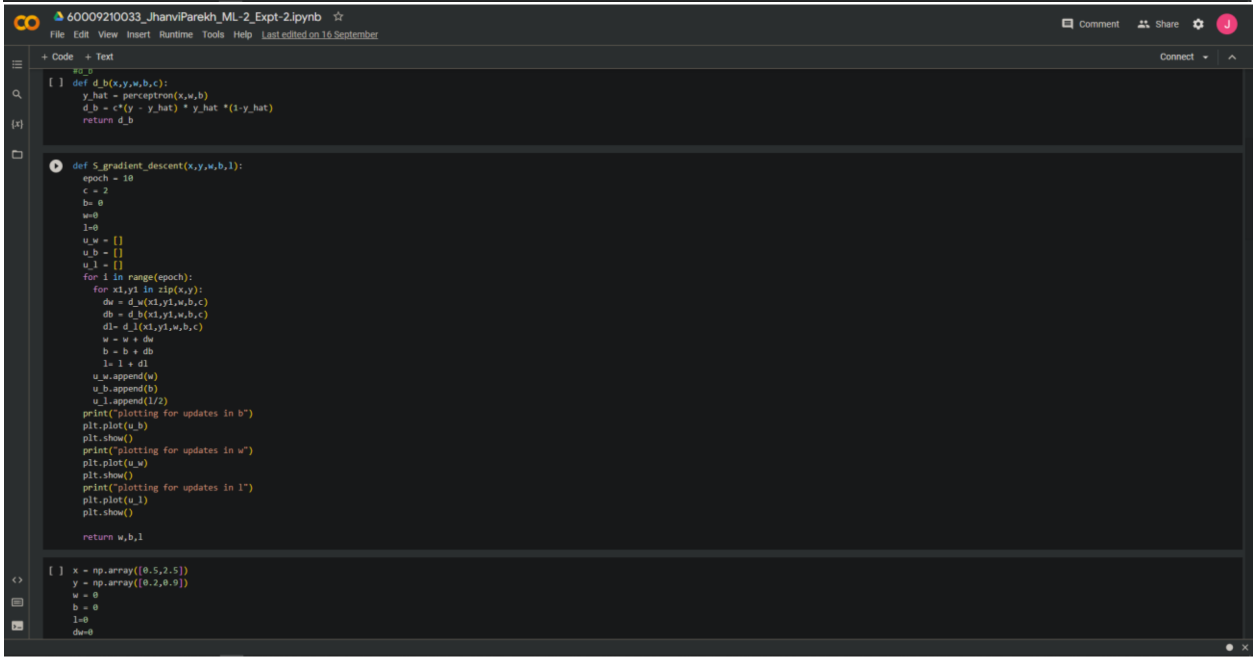
```
#d_l
def d_l(x,y,w,b,c):
  y_hat = perceptron(x,w,b)
  loss = (y-y_hat)*(y-y_hat)
  return loss
```

```
def perceptron(x,w,b):
  y_in = (x)*(w) + b
  y_hat= sigmoid_act(y_in)
  return y_hat
```

```
#sigmoid activation function
def sigmoid_act(y_in):
  y_hat = 1/(1+np.exp(-y_in))
  return y_hat
```

```
#d_b
def d_b(x,y,w,b,c):
  y_hat = perceptron(x,w,b)
  d_b = c*(y - y_hat) * y_hat *(1-y_hat)
  return d_b
```

```
def S_gradient_descent(x,y,w,b,l):
  epoch = 10
  c = 2
  b= 0
  w=0
  l=0
  u_w = []
```

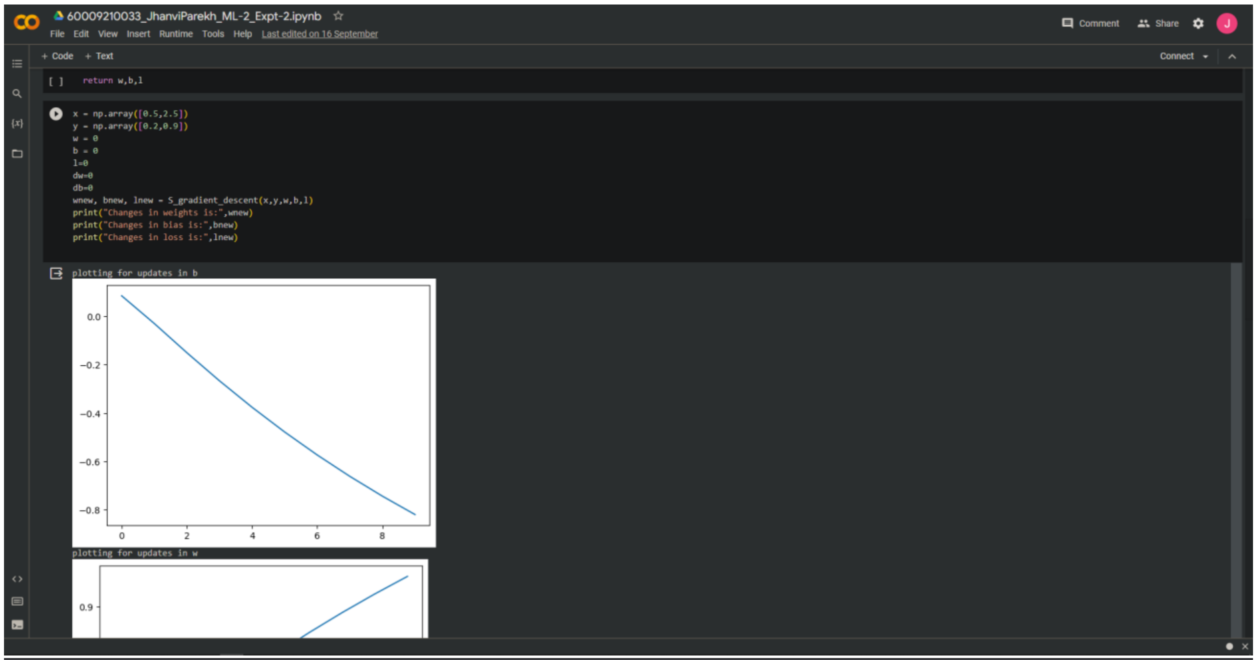

60009210033_JhanviParekh_ML-2_Expt-2.ipynb
File Edit View Insert Runtime Tools Help Last edited on 16 September
Comment Share
+ Code + Text
Connect

```
#d_b
def d_b(x,y,w,b,c):
  y_hat = perceptron(x,w,b)
  d_b = c*(y - y_hat) * y_hat *(1-y_hat)
  return d_b
```

```
def S_gradient_descent(x,y,w,b,l):
  epoch = 10
  c = 2
  b= 0
  w=0
  l=0
  u_w = []
  u_b = []
  u_l = []
  for i in range(epoch):
    for x1,y1 in zip(x,y):
      dw = d_w(x1,y1,w,b,c)
      db = d_b(x1,y1,w,b,c)
      dl= d_l(x1,y1,w,b,c)
      w = w + dw
      b = b + db
      l= l + dl
    u_w.append(w)
    u_b.append(b)
    u_l.append(l/2)
  print("plotting for updates in b")
  plt.plot(u_b)
  plt.show()
  print("plotting for updates in w")
  plt.plot(u_w)
  plt.show()
  print("plotting for updates in l")
  plt.plot(u_l)
  plt.show()

  return w,b,l
```

```
x = np.array([0.5,2.5])
y = np.array([0.2,0.9])
w = 0
b = 0
l=0
dw=0
```

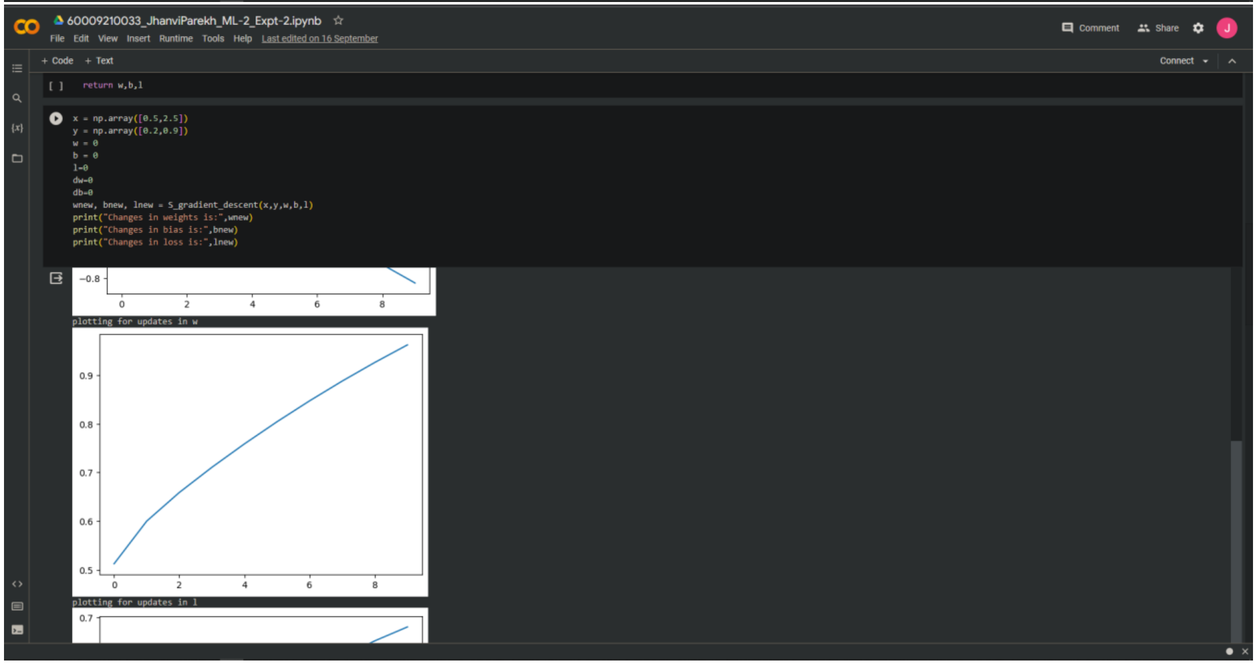

60009210033_JhanviParekh_ML-2_Expt-2.ipynb
File Edit View Insert Runtime Tools Help Last edited on 16 September
Comment Share
+ Code + Text
Connect

```
return w,b,l
```

```
x = np.array([0.5,2.5])
y = np.array([0.2,0.9])
w = 0
b = 0
l=0
dw=0
db=0
wnew, bnew, lnew = S_gradient_descent(x,y,w,b,l)
print("Changes in weights is:",wnew)
print("Changes in bias is:",bnew)
print("Changes in loss is:",lnew)
```

```
plotting for updates in b
```

```
plotting for updates in w
```

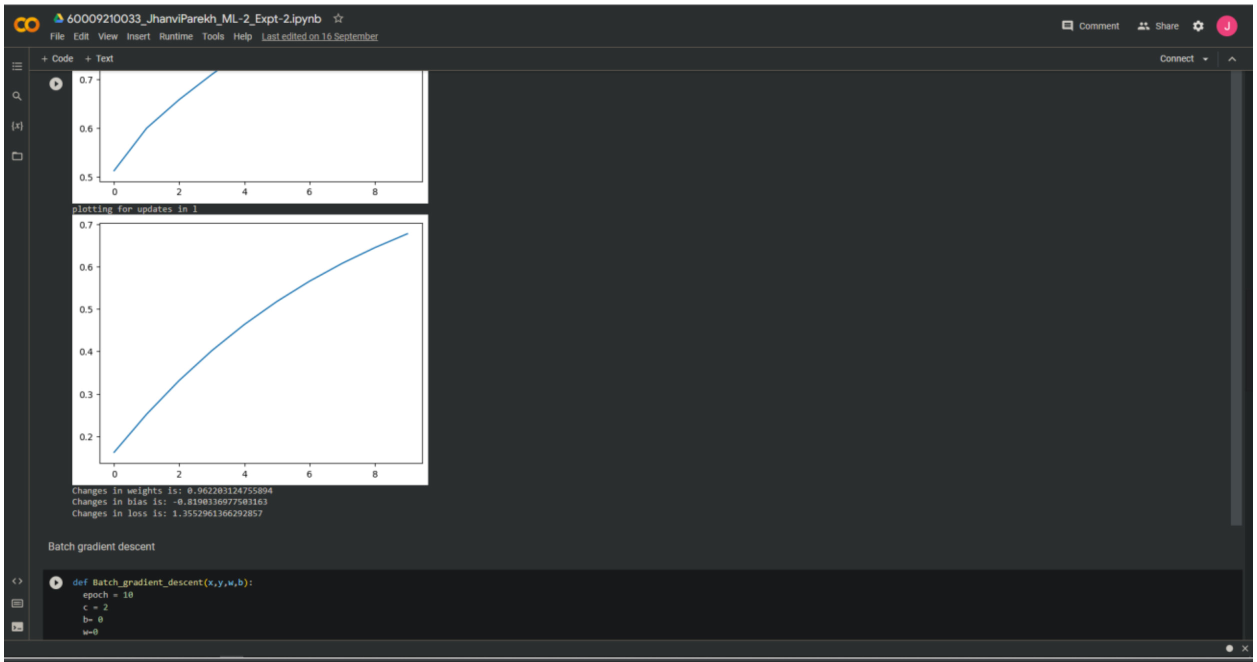


```
return w,b,l
```

```
x = np.array([0.5,2.5])
y = np.array([0.2,0.9])
w = 0
b = 0
l=0
dw=0
db=0
wnew, bnew, lnew = S_gradient_descent(x,y,w,b,l)
print("Changes in weights is:",wnew)
print("Changes in bias is:",bnew)
print("Changes in loss is:",lnew)
```

```
plotting for updates in w
```

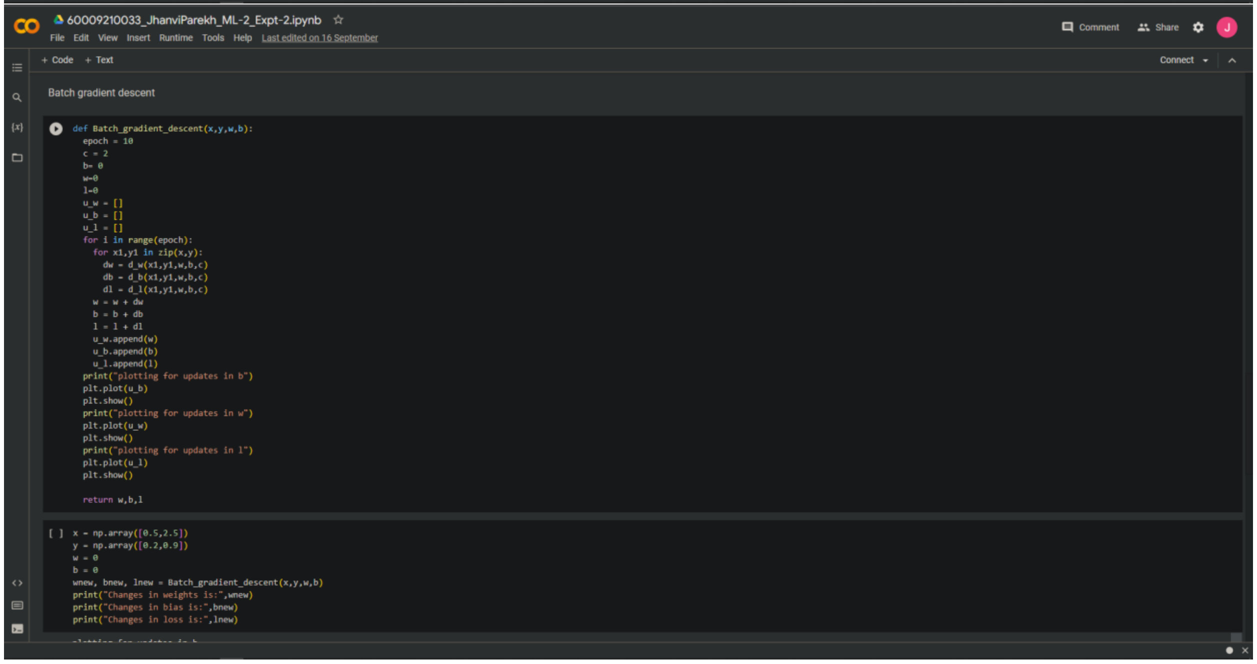
```
plotting for updates in l
```


60009210033_JhanviParekh_ML-2_Expt-2.ipynb
File Edit View Insert Runtime Tools Help Last edited on 16 September
Comment
Share
+ Code + Text
Connect

```
plotting for updates in l
Changes in weights is: 0.962203124755894
Changes in bias is: -0.8190336977503163
Changes in loss is: 1.3552961366292857
```

Batch gradient descent

```
def Batch_gradient_descent(x,y,w,b):
  epoch = 10
  c = 2
  b= 0
  w=0
```


60009210033_JhanviParekh_ML-2_Expt-2.ipynb
File Edit View Insert Runtime Tools Help Last edited on 16 September
Comment
Share
+ Code + Text
Connect

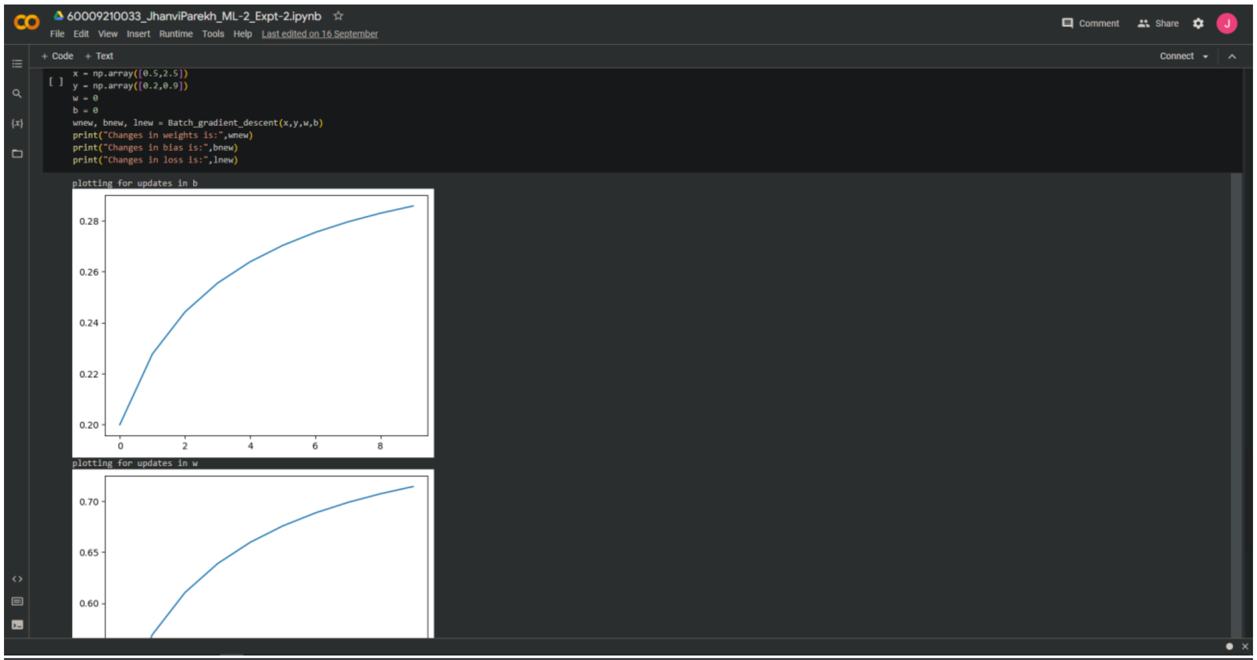
Batch gradient descent

```
def Batch_gradient_descent(x,y,w,b):
  epoch = 10
  c = 2
  b= 0
  w=0
  l=0
  u_w = []
  u_b = []
  u_l = []
  for i in range(epoch):
    for x1,y1 in zip(x,y):
      dw = d_w(x1,y1,w,b,c)
      db = d_b(x1,y1,w,b,c)
      dl = d_l(x1,y1,w,b,c)
    w = w + dw
    b = b + db
    l = l + dl
    u_w.append(w)
    u_b.append(b)
    u_l.append(l)
  print("plotting for updates in b")
  plt.plot(u_b)
  plt.show()
  print("plotting for updates in w")
  plt.plot(u_w)
  plt.show()
  print("plotting for updates in l")
  plt.plot(u_l)
  plt.show()

  return w,b,l
```

```
[ ] x = np.array([0.5,2.5])
    y = np.array([0.2,0.9])
    w = 0
    b = 0
    wnew, bnew, lnew = Batch_gradient_descent(x,y,w,b)
    print("Changes in weights is:",wnew)
    print("Changes in bias is:",bnew)
    print("Changes in loss is:",lnew)
```

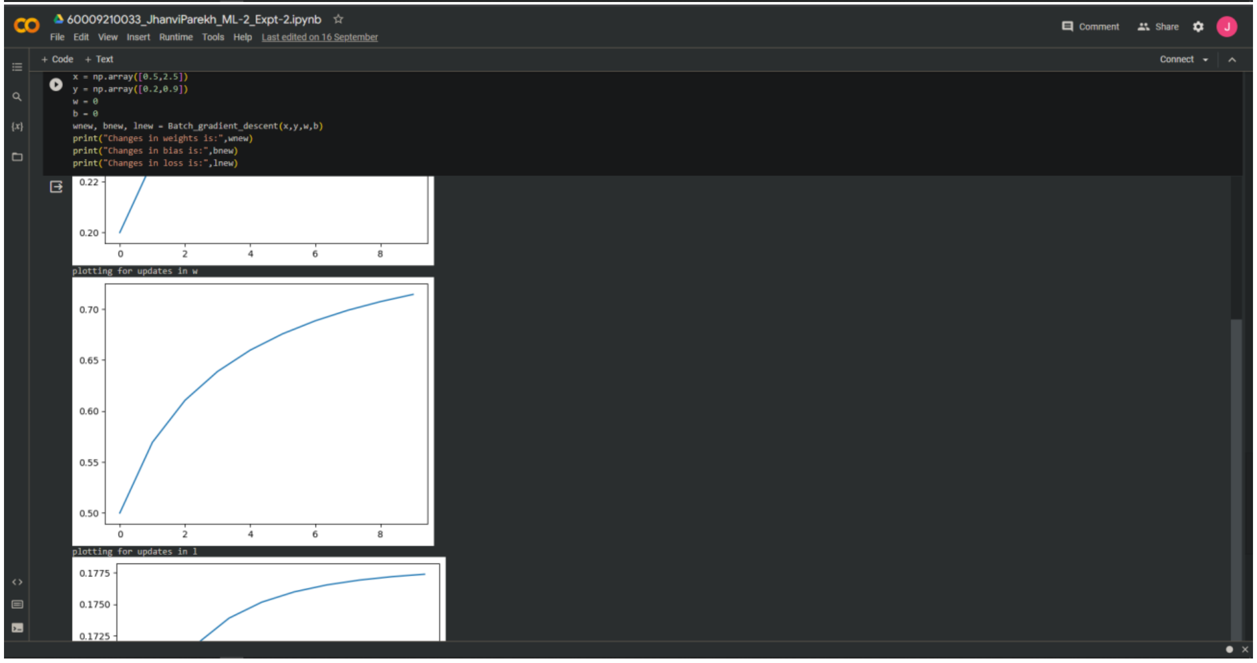
SHRI VILEPARLE KELAVANI MANDAL'S
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC ACCREDITED with "A" GRADE (CGPA : 3.18)


60009210033_JhanviParekh_ML-2_Expt-2.ipynb
File Edit View Insert Runtime Tools Help Last edited on 16 September
Comment Share
+ Code + Text
Connect
0.28
0.26
0.24
0.22
0.20
0 2 4 6 8
0.70
0.65
0.60

```
x = np.array([0.5,2.5])
y = np.array([0.2,0.9])
w = 0
b = 0
wnew, bnew, lnew = Batch_gradient_descent(x,y,w,b)
print("Changes in weights is:",wnew)
print("Changes in bias is:",bnew)
print("Changes in loss is:",lnew)
```

```
plotting for updates in b
```

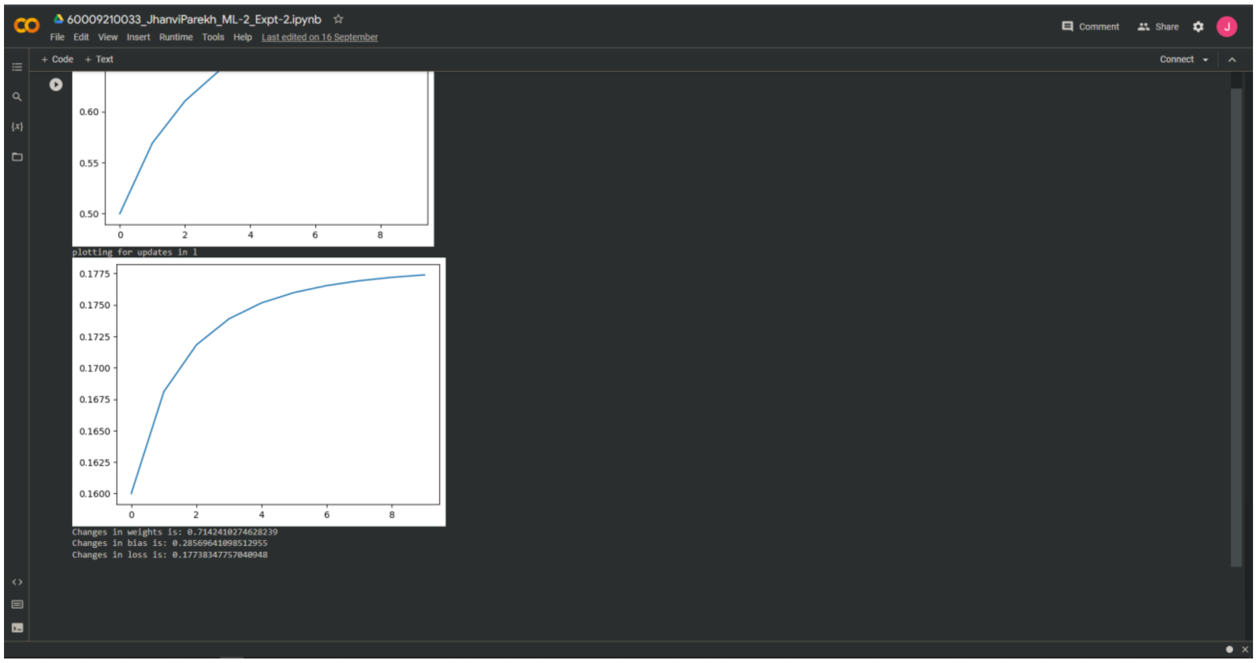
```
plotting for updates in w
```


60009210033_JhanviParekh_ML-2_Expt-2.ipynb
File Edit View Insert Runtime Tools Help Last edited on 16 September
Comment Share
+ Code + Text
Connect
0.22
0.20
0 2 4 6 8
0.70
0.65
0.60
0.55
0.50
0 2 4 6 8
0.1775
0.1750
0.1725

```
x = np.array([0.5,2.5])
y = np.array([0.2,0.9])
w = 0
b = 0
wnew, bnew, lnew = Batch_gradient_descent(x,y,w,b)
print("Changes in weights is:",wnew)
print("Changes in bias is:",bnew)
print("Changes in loss is:",lnew)
```

```
plotting for updates in w
```

```
plotting for updates in l
```


60009210033_JhanviParekh_ML-2_Expt-2.ipynb
File Edit View Insert Runtime Tools Help Last edited on 16 September
Comment Share
+ Code + Text
Connect

```
plotting for updates in l
Changes in weights is: 0.7142410274628239
Changes in bias is: 0.28569641098512955
Changes in loss is: 0.17738347757040948
```

LINK:
https://colab.research.google.com/drive/1oArrmLzOq03qATkuzeHiqgoUNGHWOePB?usp=sharing