```
!git clone https://github.com/SwastikGorai/NN-from-scratch.git
```

```
fatal: destination path 'NN-from-scratch' already exists and is not an
empty directory.
```

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
```

```
dataset = pd.read_csv("dataset/train.csv")
```

```
dataset.head(10)
```

```
   label  pixel0  pixel1  pixel2  pixel3  pixel4  pixel5  pixel6
pixel7  \
0      1       0       0       0       0       0       0       0
0
1      0       0       0       0       0       0       0       0
0
2      1       0       0       0       0       0       0       0
0
3      4       0       0       0       0       0       0       0
0
4      0       0       0       0       0       0       0       0
0
5      0       0       0       0       0       0       0       0
0
6      7       0       0       0       0       0       0       0
0
7      3       0       0       0       0       0       0       0
0
8      5       0       0       0       0       0       0       0
0
9      3       0       0       0       0       0       0       0
0

   pixel8  ...  pixel774  pixel775  pixel776  pixel777  pixel778
pixel779  \
0       0  ...         0         0         0         0         0
0
1       0  ...         0         0         0         0         0
0
2       0  ...         0         0         0         0         0
0
3       0  ...         0         0         0         0         0
0
4       0  ...         0         0         0         0         0
0
5       0  ...         0         0         0         0         0
```

```
0
6        0  ...         0         0         0         0         0
0
7        0  ...         0         0         0         0         0
0
8        0  ...         0         0         0         0         0
0
9        0  ...         0         0         0         0         0
0

   pixel780  pixel781  pixel782  pixel783
0         0         0         0         0
1         0         0         0         0
2         0         0         0         0
3         0         0         0         0
4         0         0         0         0
5         0         0         0         0
6         0         0         0         0
7         0         0         0         0
8         0         0         0         0
9         0         0         0         0

[10 rows x 785 columns]
```

```
data = np.array(dataset)
```

```
data
```

```
array([[1, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [1, 0, 0, ..., 0, 0, 0],
       ...,
       [7, 0, 0, ..., 0, 0, 0],
       [6, 0, 0, ..., 0, 0, 0],
       [9, 0, 0, ..., 0, 0, 0]], dtype=int64)
```

```
row,column = data.shape
print(row,column)
```

```
42000 785
```

```
train, test = train_test_split(data, test_size = 0.2)
```

```
train, test = train.T , test.T
```

```
train_y = train[0]
train_x = train[1:]
train_x = train_x/255.
```

```
test_x = test[0]
test_y = test[1:]
test_y = test_y/255.

train_x[:,0].shape # or train_x.shape[0]

(784,)
```

Create parameters, weights, functions

Initialize the weights and Biases

```
def initialize():


  weight_1 = np.random.rand(10,784) -0.5
  bias_1 = np.random.rand(10,1) -0.5

  weight_2 = np.random.rand(10,10) -0.5
  bias_2 = np.random.rand(10,1) -0.5

  weight_3 = np.random.rand(10,10) -0.5
  bias_3 = np.random.rand(10,1)-0.5

  return weight_1, bias_1, weight_2, bias_2, weight_3, bias_3
```

Define the Rectified Linear Unit (ReLU) function:

$Relu(z) = max(0, z) $

```
def ReLU(x):
  return np.maximum(x,0)

def derivative_ReLU(x):
  return x>0
```

Define the softmax function

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \; for \; i = 1, 2, \ldots, K$$

```
def Softmax(x):
  # x = np.float128(x)
  e = np.exp(x) / sum(np.exp(x))
  return e
```

Define the Forward Propagation function

```
def forward_propagation(w1, b1, w2, b2, w3, b3, layer):
  Z1 = w1.dot(layer) + b1
  L1 = ReLU(Z1)

  Z2 = w2.dot(L1) + b2
  L2 = ReLU(Z2)

  Z3 = w3.dot(L2) + b3
  L3 = Softmax(Z3)

  return Z1, L1, Z2, L2, Z3, L3
```

Define a One-Hot-Encoding Function

```
def one_hot_encode(Y):
  one_hot_Y = np.zeros((Y.size, Y.max() + 1))
  one_hot_Y[np.arange(Y.size), Y] = 1
  one_hot_Y = one_hot_Y.T
  return one_hot_Y
```

Explaination:

1. `one_hot_Y = np.zeros((Y.size, Y.max() + 1))` :

   Creates a matrix of size `Y.size` and `Y.max() + 1`. `Y.max() + 1` means that if there are 3 distinct values, say `Y=  [3,1,2]` then there would be 4 columns and 3 rows created(incl. 0) like `[[0,0,0,0],[0,0,0,0],[0,0,0,0],]`
2. `one_hot_Y[np.arange(Y.size), Y] = 1`
   - `np.arange(Y.size)` creates a range of values starting from 0, For example, if Y is `[3,1,2]` then np.arange would create `[0,1,2,3]`
   - `one_hot_Y[np.arange(Y.size), Y]` will be like:
     1. `one_hot[0,3]` = 1
     2. `one_hot[1,1]` = 1
     3. `one_hot[2,2]` = 1
3. Next, transposing the values, to bring labels to column headers. Here, each row is a record(or value). We need each column to be a record(or value).

Define the Backward Propagation Function

```
def backward_propagation(Z1, L1, W1, Z2, L2, W2, Z3, L3, W3, X, Y):
  one_hot_y = one_hot_encode(Y)
  m = Y.size
  # Find the gradient of loss function wrt to the pre-activated values
(Z2) of the output layer
  dz3 = L3 - one_hot_y # This calculation essentially measures the
```

```
difference between the predicted output and the true target for each
class. It gives us a measure of how much the predictions deviate from
the actual target class probabilities.

  # find the gradient of the loss function wrt to the Weights in the
output layers
  dw3 = 1 / m * dz3.dot(L2.T) # This multiplication effectively
calculates how the gradients of z3 with respect to the weights w3
influence the overall loss. The 1/m rescales the resultant.

  # Find the gradient of the loss function wrt to the biases in the
output layer
  # db3 = 1 / m * np.sum(dz3) # Summing its(dz3) elements gives a
measure of how the loss function changes with respect to changes in
the overall output of the output layer.
  db3 = 1 / m * np.sum(dz3, axis=1).reshape(-1, 1)

  dz2 = W3.T.dot(dz3) * (derivative_ReLU(dz3))
  dw2 = 1 / m * dz3.dot(L1.T)
  # db2 = 1 / m * np.sum(dz2)
  db2 = 1 / m * np.sum(dz2, axis=1).reshape(-1, 1)

  dz1 = W2.T.dot(dz2) * (derivative_ReLU(dz2))
  dw1 = 1 / m * dz2.dot(X.T)
  # db1 = 1 / m * np.sum(dz1)
  db1 = 1 / m * np.sum(dz1, axis=1).reshape(-1, 1)

  return dw3, db3, dw2, db2, dw1, db1
```

Define the function to apply changes to the values

```
def update(W1, W2, W3, B1, B2, B3, dw1, dw2, dw3, db1, db2, db3,
learning_rate):
  W1 = W1 - learning_rate * dw1
  B1 = B1 - learning_rate * db1
  W2 = W2 - learning_rate * dw2
  B2 = B2 - learning_rate * db2
  W3 = W3 - learning_rate * dw3
  B3 = B3 - learning_rate * db3
  return W1, W2, W3, B1, B2 ,B3
```

Define function to get predictions and accuracy

```
def predict(output_L):
  return np.argmax(output_L, axis=0) # axis = 0=> max values along
rows

def accuracy(preds, labels):
```

```
  is_correct = np.sum(preds == labels)
  return is_correct/labels.size
```

Using Gradient descent

```
def gradient_descent(X, Y, epochs, learning_rate):
  W1, b1, W2, b2, W3, b3 = initialize()
  for i in range(epochs):
    Z1, L1, Z2, L2, Z3, L3 = forward_propagation(W1, b1, W2, b2, W3,
b3, X)
    dw3, db3, dw2, db2, dw1, db1 = backward_propagation(Z1, L1, W1,
Z2, L2, W2, Z3, L3, W3, X, Y)
    W1, W2, W3, b1, b2, b3 = update(W1, W2, W3, b1, b2, b3, dw1, dw2,
dw3, db1, db2, db3, learning_rate )
    if i%10 == 0:

print("---------------------------------------------------------------
---")
      print("Epoch ==> ", i)
      preds = predict(L3)
      acc = accuracy(preds, Y)
      print("Accuracy : ", acc )

print("---------------------------------------------------------------
---")
  return W1, b1, W2, b2, W3, b3
```

Training

```
W1, b1, W2, b2, W3, b3 = gradient_descent(train_x, train_y, 100, 0.1)
```

```
------------------------------------------------------------------
Epoch ==>  0
Accuracy :  0.14098214285714286
------------------------------------------------------------------
------------------------------------------------------------------
Epoch ==>  10
Accuracy :  0.15142857142857144
------------------------------------------------------------------
------------------------------------------------------------------
Epoch ==>  20
Accuracy :  0.07639880952380952
------------------------------------------------------------------
------------------------------------------------------------------
Epoch ==>  30
Accuracy :  0.13848214285714286
```

```
------------------------------------------------------------------
------------------------------------------------------------------
Epoch ==>  40
Accuracy :  0.2833333333333333
------------------------------------------------------------------
------------------------------------------------------------------
Epoch ==>  50
Accuracy :  0.37973214285714285
------------------------------------------------------------------
------------------------------------------------------------------
Epoch ==>  60
Accuracy :  0.38916666666666666
------------------------------------------------------------------
------------------------------------------------------------------
Epoch ==>  70
Accuracy :  0.36622023809523807
------------------------------------------------------------------
------------------------------------------------------------------
Epoch ==>  80
Accuracy :  0.4249107142857143
------------------------------------------------------------------
------------------------------------------------------------------
Epoch ==>  90
Accuracy :  0.4616369047619048
------------------------------------------------------------------
```