Sudarshan Pillai

11/2/20

Ali Minai-EECE 6036

HW # 3

1.1 System Description

Number of hidden layers - 1
Number of hidden neurons – 100
Learning rate – 0.01
Momentum – 0.01
Output thresholds – 1 is >= 0.75, 0 is < 0.25
Hidden & Output layer activation - Sigmoid
Rule for choosing initial weights – random values between range of -1 to 1
Epochs - 200
Criterion to stop training – if loss function is less than 0.01

1.2 Results

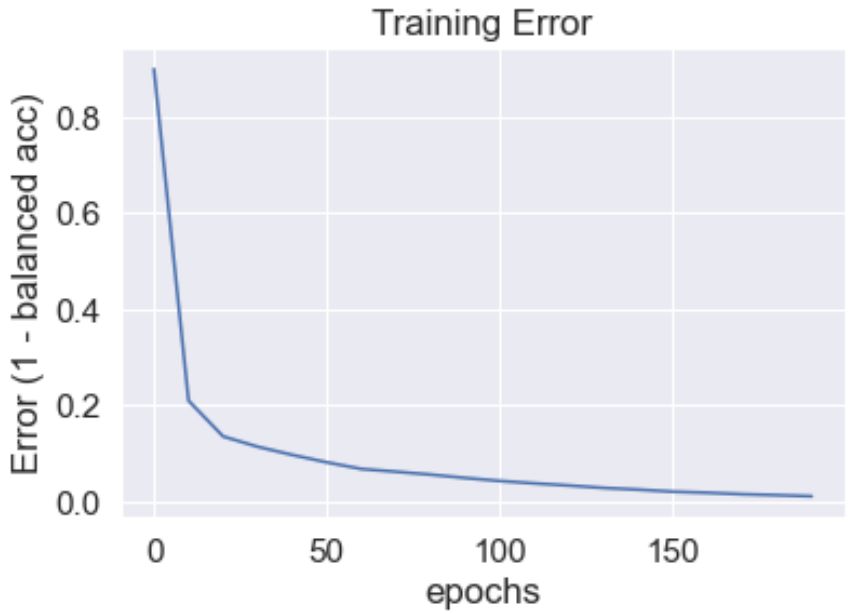


Figure 1.1 – Hit rate error of the training set

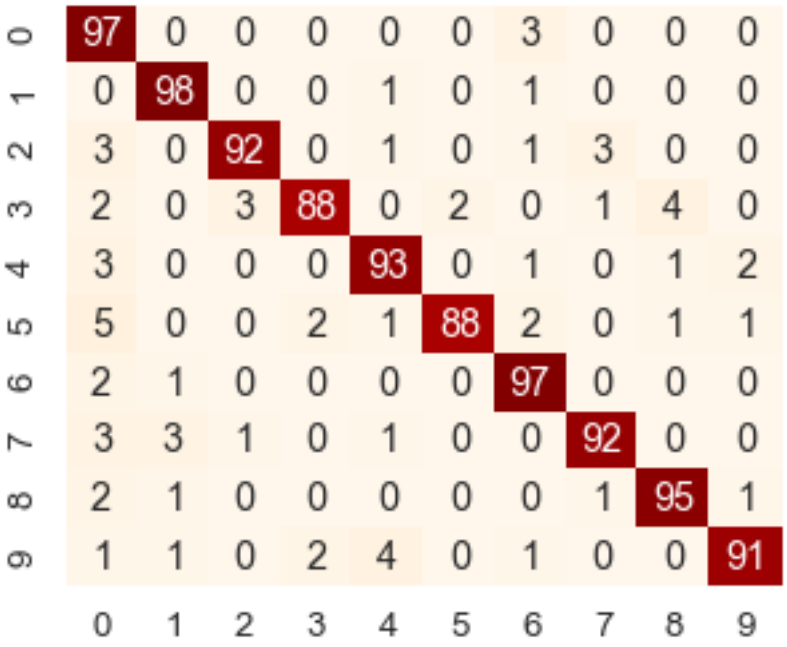


Figure 1.2 – Confusion matrix created from the test set

1.3 Analysis of Results

The learning rate was determined by trial and error after running through all the epochs and the final value which gave the best results for this algorithm was *lr = 0.01*. The momentum used for this algorithm after trial and error was *momentum = 0.*01, and the number of hidden layers neurons used was 100. Figure 1.1 shows the overall performance of the network over 200 epochs, and it can be noticed that the error rate decreases as the training continues and eventually stays constant around the 140th epoch. This was due to the condition that the training stops once it reaches an error less than 1%. From Figure 1.2, we can see that the model did a good job on classifying all the numbers. However, it performed the best on predicting the 1s which is probably due to the fact that the shape of a 1 is just a line.

2.1 System Description

Number of hidden layers - 1
Number of hidden neurons – 100
Learning rate – 0.01
Momentum – 0.01
Output thresholds – 1 is >= 0.75, 0 is < 0.25
Hidden & Output layer activation - Sigmoid
Rule for choosing initial weights – random values between range of -1 to 1
Epochs - 200

2.2 Results

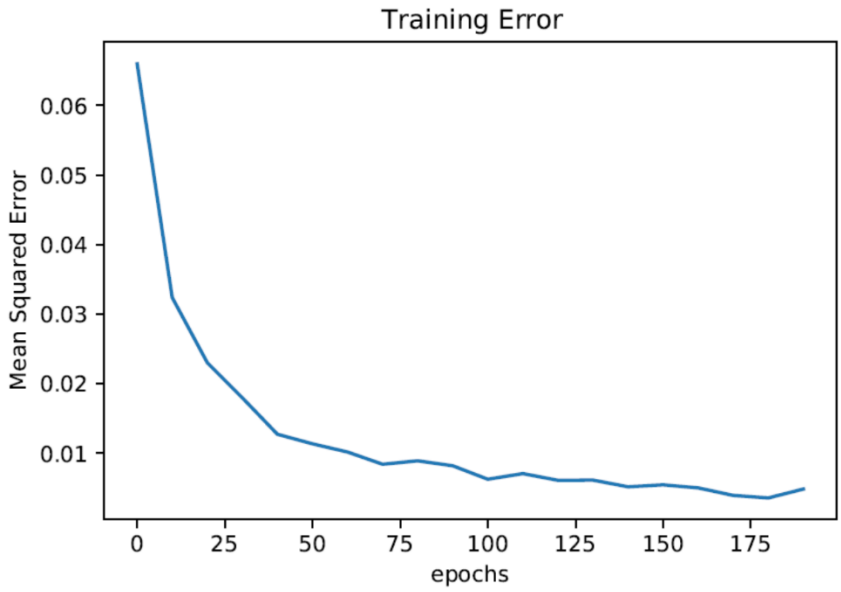
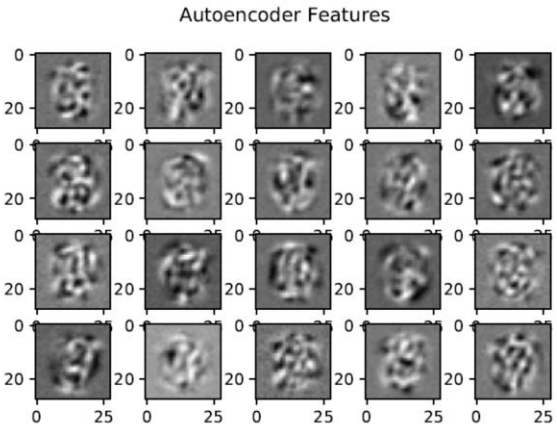


Figure 2.1 – Error on the training set

2.3 Features



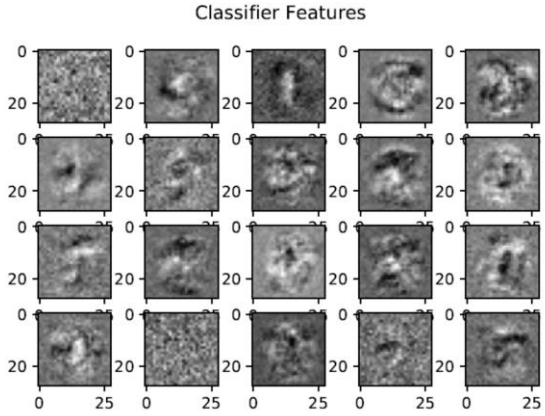


Figure 2.2 – autoencoder features for 20 random neurons from the hidden layer

Figure 2.3 – classifier features for 20 random neurons from the hidden layer

With figure 2.3 and 2.4, we can see that the hidden layer features for both of these algorithms are producing some sort of random pattern which is not interpretable.

2.4 Sample Outputs

Figure 2.4 – Upper image is true values and bottom image is the image produced from autoencoder

2.5 Analysis of Results

The learning rate was determined by trial and error after running through all the epochs and the final value which gave the best results for this algorithm was *lr = 0.01*. The momentum used for this algorithm after trial and error was *momentum = 0.*01, and the number of hidden layers neurons used was 100. Figure 2.1 shows the overall performance of the network over 200 epochs, and it can be noticed that the error rate decreases as the training continues and eventually stays constant around the 140^{th} epoch.

Figure 2.1 gives the error function for training set per epoch (200 epochs total). We can see that the error function is getting better and better as the epoch increases and decreases at a constant rate.

Figure 2.2 and 2.3 gives the feature images captured by the weight of random 20 neurons and for both the classifier and the autoencoder. The features of these algorithms are produced by the weights of the hidden layers. Finally, figure 2.4 shows the 8 random autoencoder outputs reconstructed to show the true image. The autoencoder algorithm is good enough to produce the original image with not much error.

3 Program Appendix

3.1 Q1.py

```
Created on Sun Nov  1 00:30:14 2020

@author: sudarshan19
"""

from numpy.core.fromnumeric import ptp
import pandas as pd
import random
import numpy as np
import csv
import math
import matplotlib.pyplot as plt

#import files in pandas
pred = []
all_errors_list = [0.8995, 0.20924999999999994, 0.13475000000000004, 0.11325000000000007, 0.09625
dataFile = pd.read_csv('MNISTnumImages5000_balanced.txt',sep='\t', header=None)
dataFile1 = pd.read_csv('MNISTnumLabels5000_balanced.txt', header=None)
x = pd.concat([dataFile, dataFile1], axis = 1)
train_data = []
neurons = 0
positive = 0
test_data = []
random_train_data = []
random_test_data = []
error_val = 0
count = 0
```

```
count = 0
l1 = []
l2 = []
value1 = []
difference_avg = []
current_value = np.zeros((784,1))
train_data = []
true_values = np.zeros((10,1))
all_true_values = []
difference = []
differences = []
delta_w = {}
difference_weights = {}
initial_weights = {}
weight1 = 0
weight2 = 0
prev_weight2 = 0
prev_weight1 = 0
all_sig = []
delta = []
lr = 0.01
momentum = 0.01
#Randomly chose 4000 data points for training set
train_data = x[0:400]
for i in range(9):
    train_data = train_data.append(x[(i+1)*500:((i+1)*500) + 400])
    random_train_data = train_data.sample(frac=1)
```

```
#Randomly chose 1000 data points for testing set
test_data = x[400:500]
for j in range(9):
    test_data = test_data.append(x[(j+2)*500-100:((j+2)*500)])
    random_test_data = test_data.sample(frac=1)

all_layers = {}
all_layers_input = {}

epoch = 10


def reset():
    global train_data
    global test_data
    global random_train_data
    global random_test_data
    global error_val
    global l1
    global l2
    global value1
    global difference_avg
    global current_value
    global train_data
    global true_values
    global all_true_values
    global difference
    global differences
```

```
    global delta_w
    global weight1
    global weight2
    global all_sig
    global delta
    global momentum
    global positive

    positive = 0
    error_val = 0
    l1 = []
    l2 = []
    value1 = []
    difference_avg = []
    current_value = np.zeros((784,1))
    train_data = []
    true_values = np.zeros((10,1))
    all_true_values = []
    difference = []
    differences = []
    delta_w = {}
    weight1 = 0
    weight2 = 0
    all_sig = []
    delta = []
    momentum = 0
    #Randomly chose 4000 data points for training set
```

```
    #Randomly chose 4000 data points for training set
    train_data = x[0:400]
    for i in range(9):
        train_data = train_data.append(x[(i+1)*500:((i+1)*500) + 400])
        random_train_data = train_data.sample(frac=1)

    #Randomly chose 1000 data points for testing set
    test_data = x[400:500]
    for j in range(9):
        test_data = test_data.append(x[(j+2)*500-100:((j+2)*500)])
        random_test_data = test_data.sample(frac=1)

def error_func():

    global error_val
    global difference_avg
    sum = 0
    error_val = 0

    for i in range(4000):

        for j in difference[i][0]:

            sum = sum + math.pow(j, 2)

        difference_avg.append(sum/10)
        sum = 0
```

```
    for i in difference_avg:

        error_val = error_val + i

    error_val = error_val/4000


def hit_rate(values, label):

    global positive
    x = 0
    y = 0
    z = 0
    for i in values:
        if(x < i):
            x = i
            y = z
        z = z+1

    if(label == y):
        positive = positive + 1

def add_weights(num, n):

    if(num == -1):
        return [np.random.randn(10,n)*0.1, np.zeros([10,1])]
```

/

```
def add_weights(num, n):

    if(num == -1):
        return [np.random.randn(10,n)*0.1, np.zeros([10,1])]

    if(num == 1):
        return [np.random.randn(n,784)*0.1, np.zeros([n,1])]
    else:
        return [np.random.randn(n,n)*0.1, np.zeros([n,1])]


def add_input(layer_input_values, idx):

    global all_layers_input

    all_layers_input[idx] = layer_input_values

#1 layer of the networks loop through 200 epoch.
# training data set
def update_weights(n, first):

    global all_layers
    global delta_w
    global l1
    global l2

    if(first == True):
        l1 = np.asarray(np.add(np.asarray(all_layers[1][0]).reshape(n, 784) , np.asarray(delta_w[
```

```
    global all_layers
    global delta_w
    global l1
    global l2

    if(first == True):
        l1 = np.asarray(np.add(np.asarray(all_layers[1][0]).reshape(n, 784) , np.asarray(delta_w[1
        l2 = np.asarray(np.add(np.asarray(all_layers[2][0]).reshape(10, n) , np.asarray(delta_w[2]
        all_layers[1] = [l1, np.zeros([n, 1])]
        all_layers[2] = [l2, np.zeros([10,1])]
    else:
        l1 = np.asarray(np.add(np.asarray(np.add(np.asarray(all_layers[1][0]).reshape(n, 784) , np
        l2 = np.asarray(np.add(np.asarray(np.add(np.asarray(all_layers[2][0]).reshape(10, n) , np.
        all_layers[1] = [l1, np.zeros([n, 1])]
        all_layers[2] = [l2, np.zeros([10, 1])]

def calc_sig(curr_input):

    new_curr_values = []
    z = 0
    for i in curr_input:
        new_curr_values.append(1/(1 + np.exp(i*-1)))
        z = z + 1

    return np.asarray(new_curr_values).reshape(z, 1)
```

```
def calc_relu(curr_img_values):

    new_values = []
    z = 0
    for i in curr_img_values:
        new_values.append(np.maximum(0,i))
        z = z + 1

    return np.asarray(new_values).reshape(z, 1)

def calc_derv_relu(curr_img_values):

    new_values = []
    z = 0
    for i in curr_img_values:
        if(i <= 0):
            new_values.append(0)
        else:
            new_values.append(1)
        z = z + 1

    return np.asarray(new_values).reshape(z, 1)

def get_data(val):

    for i in range(20):
        val.append(all_errors_list[i])
```

```
def calc_derv_sig(values):

    new_values = []
    z = 0
    for i in values:
        new_values.append((1/(1 + np.exp(i*-1)))*((np.exp(i*-1)/(1+np.exp(i*-1)))))
        z = z + 1

    return np.asarray(new_values).reshape(z, 1)

def calc_thresh(values):

    thresh_values = np.zeros((10,1))
    z = 0
    for i in values:

        if(i < 0.25):
            thresh_values[z] = 0

        elif(i > 0.75):
            thresh_values[z] = 1

        else:
            thresh_values[z] = i

        z = z + 1
    return thresh_values
```

```
def calculate_delta_weights(layer_num, n, last, i):

    global delta_w
    global weight1
    global weight2
    global delta
    global prev_weight1
    global prev_weight2
    global count

    if(last == True):
        delta = np.multiply(np.asarray(difference[i]).reshape(10,1), np.asarray(calc_derv_sig(all_
        delta = [delta]*n
        delta = np.asarray(delta).transpose().reshape(10,n)
        delta = np.multiply(np.asarray([all_layers_input[i][layer_num-1]]*10).reshape(10,n), delta
        delta = (lr*delta)
        weight2 = np.asarray(delta).reshape(10, n)
        delta_w[layer_num] = weight2
        if(count == 0):
            update_weights(n, True)
            count = 1
        else:
            update_weights(n, False)

        prev_weight2 = weight2
        prev_weight1 = weight1

    else:
```

```
    else:
        delta = np.multiply(np.asarray(difference[i]).reshape(10,1), np.asarray(calc_derv_sig(all_
        delta = [delta]*n
        delta = np.asarray(delta).transpose().reshape(10,n)
        delta = np.multiply(np.asarray(all_layers[layer_num+1][0]).reshape(10,n), delta)
        delta = np.asarray(np.asarray([sum(x) for x in zip(*delta)]).transpose()).reshape(n, 1)
        delta = np.multiply(np.asarray(delta).reshape(n,1), np.asarray(calc_derv_sig(np.asarray(al
        delta = np.asarray([delta]*784).reshape(n, 784)
        delta = np.multiply(delta, np.asarray([np.asarray(all_layers_input[i][layer_num-1]).reshap
        delta = (lr*(-1)*delta)
        weight1 = np.asarray(delta).reshape(n,784)
        delta_w[layer_num] = weight1

def iterate_layers(curr_img_values, num, idx, cl):

    layers_input = []
    before_sig = []
    global all_sig

    for i in all_layers:

        layers_input.append(curr_img_values)
        curr_img_values = np.dot(all_layers[i][0], curr_img_values) + all_layers[i][1]
        if(num >= int(i)):
            before_sig.append(curr_img_values)
            curr_img_values = calc_sig(curr_img_values)
        else:
            before_sig.append(curr_img_values)
```

```
        else:
            before_sig.append(curr_img_values)
            curr_img_values = calc_sig(curr_img_values)
            hit_rate(curr_img_values, cl)
            curr_img_values = calc_thresh(curr_img_values)

    all_sig.append(before_sig)
    add_input(layers_input, idx)
    return curr_img_values

def plot():

    errors =[]
    errors = get_data(errors)
    plt.plot(list(range(0, len(errors) * 10, 10)), errors)
    plt.ylabel('Error (1 - balanced acc)')
    plt.xlabel('epochs')
    plt.title('Training Error')

def iterate_img_values(layer_num, n):

    global new_train_data
    new_train_data = random_train_data.transpose()
    global value1
    global current_value
    global true_vlaues
    global all_true_values
    global difference
```

```
    for train_idx in range(4000):
        current_label = new_train_data.iloc[784, train_idx]
        current_img = new_train_data.iloc[0:784, train_idx]
        current_value = current_img.values.reshape(784,1)
        true_values = np.zeros((10,1))
        true_values[int(current_label)] = 1
        all_true_values.append(true_values)
        value1.append(iterate_layers(current_value, layer_num, train_idx, int(current_label)))
        difference.append((true_values - value1[train_idx]).reshape(1,10))

        for i in range(layer_num+1):

            if(i == layer_num):
                calculate_delta_weights(i+1, n, True, train_idx)

            else:
                calculate_delta_weights(i+1, n, False, train_idx)


    error_func()

def confusion_matrix(true_values, value1):
    from sklearn.metrics import confusion_matrix
    cm = confusion_matrix(np.argmax(true_values, axis=1), np.argmax(value1, axis=1))
    import seaborn as sns
    df_cm = pd.DataFrame(cm, range(10), range(10))
    sns.set(font_scale=1.2)
    sns.heatmap(df_cm, annot=True, cmap="OrRd")
```

```
    plt.show()


def main(number):

    global all_layers
    global lr
    global momentum
    global neurons
    global initial_weights
    global difference_weights

    if (count == 1):

        reset()
        x = 1
    else:
        num_hidden_layers = int(input("Enter the number of hidden layers: "))
        neurons = int(input("Enter the number of neurons: "))
        layers = 1
        for i in range(layers):
            all_layers[i+1] = add_weights(i+1, neurons)
            initial_weights[i+1] = all_layers[i+1]

        all_layers[layers+1] = add_weights(-1, neurons)
        initial_weights[layers+1] = all_layers[layers+1]

    difference_weights[1] = np.subtract(all_layers[1][0], initial_weights[1][0])
```

```
        reset()
        x = 1
    else:
        num_hidden_layers = int(input("Enter the number of hidden layers: "))
        neurons = int(input("Enter the number of neurons: "))
        layers = 1
        for i in range(layers):
            all_layers[i+1] = add_weights(i+1, neurons)
            initial_weights[i+1] = all_layers[i+1]

        all_layers[layers+1] = add_weights(-1, neurons)
        initial_weights[layers+1] = all_layers[layers+1]

    difference_weights[1] = np.subtract(all_layers[1][0], initial_weights[1][0])
    difference_weights[2] = np.subtract(all_layers[2][0], initial_weights[2][0])

    layers = 1
    momentum = 0.01
    iterate_img_values(layers, neurons)


for i in range(1):
    main(i+1)

plot()
```

3.2 Q2.py

```
from numpy.core.fromnumeric import ptp
import pandas as pd
import random
import numpy as np
import csv
import math
import matplotlib.pyplot as plt

#import files in pandas
dataFile = pd.read_csv('MNISTnumImages5000_balanced.txt',sep='\t', header=None)
dataFile1 = pd.read_csv('MNISTnumLabels5000_balanced.txt', header=None)
x = pd.concat([dataFile, dataFile1], axis = 1)
train_data = []
neurons = 0
positive = 0
test_data = []
random_train_data = []
random_test_data = []
error_val = 0
count = 0
l1 = []
l2 = []
value1 = []
difference_avg = []
current_value = np.zeros((784,1))
train_data = []
true_values = np.zeros((784,1))
all_true_values = []
difference = []
differences = []
delta_w = {}
difference_weights = {}
initial_weights = {}
weight1 = 0
weight2 = 0
prev_weight2 = 0
prev_weight1 = 0
all_sig = []
delta = []
lr = 0.01
momentum = 0.01
#Randomly chose 4000 data points for training set
train_data = x[0:400]
```

```
#Randomly chose 4000 data points for training set
train_data = x[0:400]
for i in range(9):
    train_data = train_data.append(x[(i+1)*500:((i+1)*500) + 400])
    random_train_data = train_data.sample(frac=1)

#Randomly chose 1000 data points for testing set
test_data = x[400:500]
for j in range(9):
    test_data = test_data.append(x[(j+2)*500-100:((j+2)*500)])
    random_test_data = test_data.sample(frac=1)

all_layers = {}
all_layers_input = {}

epoch = 10


def reset():
    global train_data
    global test_data
    global random_train_data
    global random_test_data
    global error_val
    global l1
    global l2
    global value1
    global difference_avg
    global current_value
    global train_data
    global true_values
    global all_true_values
    global difference
    global differences
    global delta_w
    global weight1
    global weight2
    global all_sig
    global delta
    global momentum
    global positive

    positive = 0
    error_val = 0
```

```
    positive = 0
    error_val = 0
    l1 = []
    l2 = []
    value1 = []
    difference_avg = []
    current_value = np.zeros((784,1))
    train_data = []
    true_values = np.zeros((784,1))
    all_true_values = []
    difference = []
    differences = []
    delta_w = {}
    weight1 = 0
    weight2 = 0
    all_sig = []
    delta = []
    momentum = 0
    #Randomly chose 4000 data points for training set
    train_data = x[0:400]
    for i in range(9):
        train_data = train_data.append(x[(i+1)*500:((i+1)*500) + 400])
        random_train_data = train_data.sample(frac=1)

    #Randomly chose 1000 data points for testing set
    test_data = x[400:500]
    for j in range(9):
        test_data = test_data.append(x[(j+2)*500-100:((j+2)*500)])
        random_test_data = test_data.sample(frac=1)

def error_func():

    global error_val
    global difference_avg
    sum = 0
    error_val = 0

    for i in range(4000):

        for j in difference[i][0]:

            sum = sum + math.pow(j, 2)

        difference_avg.append(sum/784)
```

```
        difference_avg.append(sum/784)
        sum = 0

    for i in difference_avg:

        error_val = error_val + i

    error_val = error_val/4000


def hit_rate(values, label):

    global positive
    x = 0
    y = 0
    z = 0
    for i in values:
        if(x < i):
            x = i
            y = z
        z = z+1

    if(label == y):
        positive = positive + 1

def add_weights(num, n):

    if(num == -1):
        return [np.random.randn(784,n)*0.1, np.zeros([784,1])]

    if(num == 1):
        return [np.random.randn(n,784)*0.1, np.zeros([n,1])]
    else:
        return [np.random.randn(n,n)*0.1, np.zeros([n,1])]


def add_input(layer_input_values, idx):

    global all_layers_input

    all_layers_input[idx] = layer_input_values
```

```
def update_weights(n, first):

    global all_layers
    global delta_w
    global l1
    global l2

    if(first == True):
        l1 = np.asarray(np.add(np.asarray(all_layers[1][0]).reshape(n, 784) , np.asarray(delta_w[1]).reshape(n, 784))).reshape(n, 784)
        l2 = np.asarray(np.add(np.asarray(all_layers[2][0]).reshape(784, n) , np.asarray(delta_w[2]).reshape(784, n))).reshape(784, n)
        all_layers[1] = [l1, np.zeros([n, 1])]
        all_layers[2] = [l2, np.zeros([784,1])]
    else:
        l1 = np.asarray(np.add(np.asarray(np.add(np.asarray(all_layers[1][0]).reshape(n, 784) , np.asarray(delta_w[1]).reshape(n, 784))).reshape(n, 784), np.asarray(mome
        l2 = np.asarray(np.add(np.asarray(np.add(np.asarray(all_layers[2][0]).reshape(784, n) , np.asarray(delta_w[2]).reshape(784, n))).reshape(784, n), np.asarray(mome
        all_layers[1] = [l1, np.zeros([n, 1])]
        all_layers[2] = [l2, np.zeros([784, 1])]

def calc_sig(curr_input):

    new_curr_values = []
    z = 0
    for i in curr_input:
        new_curr_values.append(1/(1 + np.exp(i*-1)))
        z = z + 1

    return np.asarray(new_curr_values).reshape(z, 1)

def calc_relu(curr_img_values):

    new_values = []
    z = 0
    for i in curr_img_values:
        new_values.append(np.maximum(0,i))
        z = z + 1

    return np.asarray(new_values).reshape(z, 1)
```

```
def calc_derv_relu(curr_img_values):

    new_values = []
    z = 0
    for i in curr_img_values:
        if(i <= 0):
            new_values.append(0)
        else:
            new_values.append(1)
        z = z + 1

    return np.asarray(new_values).reshape(z, 1)

def calc_derv_sig(values):

    new_values = []
    z = 0
    for i in values:
        new_values.append((1/(1 + np.exp(i*-1)))*((np.exp(i*-1)/(1+np.exp(i*-1)))))
        z = z + 1

    return np.asarray(new_values).reshape(z, 1)

def calc_thresh(values):

    thresh_values = np.zeros((784,1))
    z = 0
    for i in values:

        if(i < 0.25):
            thresh_values[z] = 0

        elif(i > 0.75):
            thresh_values[z] = 1

        else:
            thresh_values[z] = i

        z = z + 1
    return thresh_values
```

```
def calculate_delta_weights(layer_num, n, last, i):

    global delta_w
    global weight1
    global weight2
    global delta
    global prev_weight1
    global prev_weight2
    global count

    if(last == True):
        delta = np.multiply(np.asarray(difference[i]).reshape(784,1), np.asarray(calc_derv_sig(all_sig[i][1]).reshape(784,1)))
        delta = [delta]*n
        delta = np.asarray(delta).transpose().reshape(784,n)
        delta = np.multiply(np.asarray([all_layers_input[i][layer_num-1]]*784).reshape(784,n), delta)
        delta = (lr*delta)
        weight2 = np.asarray(delta).reshape(784, n)
        delta_w[layer_num] = weight2
        if(count == 0):
            update_weights(n, True)
            count = 1
        else:
            update_weights(n, False)

        prev_weight2 = weight2
        prev_weight1 = weight1

    else:
        delta = np.multiply(np.asarray(difference[i]).reshape(784,1), np.asarray(calc_derv_sig(all_sig[i][1]).reshape(784,1)))
        delta = [delta]*n
        delta = np.asarray(delta).transpose().reshape(784,n)
        delta = np.multiply(np.asarray(all_layers[layer_num+1][0]).reshape(784,n), delta)
        delta = np.asarray(np.asarray([sum(x) for x in zip(*delta)]).transpose()).reshape(n, 1)
        delta = np.multiply(np.asarray(delta).reshape(n,1), np.asarray(calc_derv_sig(np.asarray(all_sig[i][0]).reshape(n, 1))).reshape(n,1))
        delta = np.asarray([delta]*784).reshape(n, 784)
        delta = np.multiply(delta, np.asarray([np.asarray(all_layers_input[i][layer_num-1]).reshape(784, 1)]*n).reshape(n,784))
        delta = (lr*(-1)*delta)
        weight1 = np.asarray(delta).reshape(n,784)
        delta_w[layer_num] = weight1
```

```
def iterate_layers(curr_img_values, num, idx, cl):

    layers_input = []
    before_sig = []
    global all_sig

    for i in all_layers:

        layers_input.append(curr_img_values)
        curr_img_values = np.dot(all_layers[i][0], curr_img_values) + all_layers[i][1]
        if(num >= int(i)):
            before_sig.append(curr_img_values)
            curr_img_values = calc_sig(curr_img_values)
        else:
            before_sig.append(curr_img_values)
            curr_img_values = calc_sig(curr_img_values)
            hit_rate(curr_img_values, cl)
            curr_img_values = calc_thresh(curr_img_values)

    all_sig.append(before_sig)
    add_input(layers_input, idx)
    return curr_img_values

def iterate_img_values(layer_num, n):

    global new_train_data
    new_train_data = random_train_data.transpose()
    global value1
    global current_value
    global true_vlaues
    global all_true_values
    global difference
    global differences
    for train_idx in range(4000):
        current_label = new_train_data.iloc[784, train_idx]
        current_img = new_train_data.iloc[0:784, train_idx]
        current_value = current_img.values.reshape(784,1)
        true_values = np.zeros((784,1))
        true_values[int(current_label)] = 1
        all_true_values.append(true_values)
        value1.append(iterate_layers(current_value, layer_num, train_idx, int(current_label)))
```

```
def reconstruct_output(all_layers):
    c = np.random.randint(1000, size=8)
    x_data = all_layers[c,:]
    y_data = all_layers[c,:]
    fig1, fig2 = plt.subplots(2, 8)
    for i in range(8):
        fig2[0][i].imshow(x_data[i].reshape(28,28).T, cmap='gray')
        fig2[1][i].imshow(y_data[i].reshape(28,28).T, cmap='gray')

def calc_features(classify_weights, auto_weights):
    idx = np.random.randint(200, size=20)
    class_weights = classify_weights[idx, :]
    autoen_weights = auto_weights[idx, :]

    fig, ax = plt.subplots(4, 5)
    pos = 0
    for i in range(4):
        for j in range(5):
            x = (class_weights[pos] - np.min(class_weights[pos]))/np.ptp(class_weights[pos])
            ax[i][j].imshow(x.reshape(28,28).T, cmap='gray')
            pos+=1
    pos = 0
    for i in range(4):
        for j in range(5):
            x = (autoen_weights[pos] - np.min(autoen_weights[pos]))/np.ptp(autoen_weights[pos])
            ax[i][j].imshow(x.reshape(28,28).T, cmap='gray')
            pos+=1


def main(number):

    global all_layers
    global lr
    global momentum
    global neurons
    global initial_weights
    global difference_weights

    if (count == 1):

        reset()
```

```
def main(number):

    global all_layers
    global lr
    global momentum
    global neurons
    global initial_weights
    global difference_weights

    if (count == 1):

        reset()
        x = 1
    else:
        num_hidden_layers = int(input("Enter the number of hidden layers: "))
        neurons = int(input("Enter the number of neurons: "))
        layers = 1
        for i in range(layers):
            all_layers[i+1] = add_weights(i+1, neurons)
            initial_weights[i+1] = all_layers[i+1]

        all_layers[layers+1] = add_weights(-1, neurons)
        initial_weights[layers+1] = all_layers[layers+1]

    difference_weights[1] = np.subtract(all_layers[1][0], initial_weights[1][0])
    difference_weights[2] = np.subtract(all_layers[2][0], initial_weights[2][0])

    layers = 1
    momentum = 0.01
    iterate_img_values(layers, neurons)


for i in range(1):
    main(i+1)
    print(i, positive, error_val)
```