Nisa Ahsen Öztürk
22001858

EEE 443: Mini Project 1

MNIST Handwritten Digit Classification

1. Introduction

The aim of this project is to classify handwritten digit dataset by using one hidden layer and one output layer. MNIST dataset is used to realize this aim. MNIST dataset has 70,000 handwritten characters which are represented as 28x28 pixel (greyscale) image. 60,000 of them are used as training data and 10,000 of them are used as test dataset.

2. Theory and Algorithm

2.1 Dataset Implementations Matrix Initialization

Firstly, MNIST dataset is loaded to the Jupyter notebook by using load function. Dataset is split into different entries to obtain training and testing samples. 28x28 data were reshaped as 784x1 by using reshape function of NumPy. After that samples are normalized in order to speed up the training process.

To implement the design, size of weight matrices should be found. Notice that parameter N affects the hidden layer weight matrix's size. The size of hidden layer's and output layer's weight matrices are given below. Furthermore, it is asked to initialize weight matrices as random gaussian distribution with zero mean and 0.01 variance. To initialize weight matrices for every case, a initialization function is added to the code.

$$W_h = [\because]_{Nx784}$$

$$W_o = [\because]_{10x\,N}$$

Variable x represents the input matrix. x is a matrix with size 784x1. For the sake of simplicity extended matrices are used in the network. The main aim of using extended matrices is to add and change biases simultaneously. Therefore a -1 term is added to the end of the input matrix. New x_e matrix is in the shape of 785x1. A change in the size of input matrix, will change the size of other layers' matrices. The size of each matrix is given below.

$$x_e = \begin{bmatrix} \vdots \\ -1 \end{bmatrix}_{785x1}$$

$$z_e = \begin{bmatrix} \vdots \\ -1 \end{bmatrix}_{N+1x1}$$

$$W_{oe} = [W_o|\theta]_{10xN+1}$$

$$W_{he} = [W_o|\theta]_{Nx\,785}$$

Matrix notation z represents the input matrix of hidden layer.

2.2 Forward Propagation

For the first case, both activation functions are defined as tanh function. For the second case hidden layer's activation function is ReLu and output layer's activation function is defined as sigmoid function. Activation function is the calculations are symbolized as f(.). The forward propagation calculations are given below.

Let us start with extended input matrix and weight matrix.

$$a_{Nx1} = W_{he} \mathrm{x}\, x_e$$
$$z_{Nx1} = f(a)$$
$$v_{10x1} = W_{oe} \mathrm{x}\, z_e$$
$$o_{10x1} = f(v)$$

The o matrix will be used to label each digit (from 0 to 9) in the code. For assigning those neurons to a digit a labeling function is added to the code.

2.3 Back Propagation

The aim of the backpropagation is to update the weights to minimize the loss in the network. The error function is defined as follows

$$E = 0.5 \sum_{i=1}^{R} (d_i(n) - o_i(n)^2)$$

where $d_i(n)$ is the desired output and $o_i(n)$ is the network's output.

The computation for weight updating part is given below.

$$W \leftarrow W - \eta \frac{\partial E}{\partial W}$$

The partial derivative can be found from the chain rule.

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial e_j} \frac{\partial e_j}{\partial o_j} \frac{\partial o_j}{\partial v_j} \frac{\partial v_j}{\partial w_{ji}} = e_j(n)(-1) f'(v_j(n) z_i$$

For the sake of the simplicity a gradient (δ) function is defined as

$$\delta_j(\mathrm{n}) = e_j(n) f'(v_j(n))$$

Therefore, updated weight matrix computation can be written as follows.

$$w_{ij} \leftarrow w_{ij} + \eta \delta_j(\mathrm{n}) z_i$$

This equation will be used for both hidden layer and output layer weight matrices

3. Results

The success rate, accuracy of this network is tested for different cases with different parameters. Firstly, network is checked for distinct activation functions. In the first case tanh function is used for layers as an activation function. In the second case, ReLu function is used for hidden layer and sigmoid function is used as an activation

function of output layer. Moreover, N is assigned as the number of neurons in the hidden layer. The size of the hidden layer can take distinct values as N= 300, 500, 1000. In addition to parameter N, a new parameter η is generated as the learning rate of the network. The simulation is examined for η=0.01,0.05, 0.09. Therefore, to test all outcomes for different combinations of parameters, 18 (3x3x2) different training is needed. Each training is realized for 50 epochs. The error of the training and test sets are given below.

N, η	Training Error	Testing Error	Accuracy (%)
300,0.01	0.01224	0.0196	98.77
300,0.05	0428	0.463	57.23
300,0.09	0.6323	0.671	37.2
500,0.01	0.0122	0.0208	98.78
500,0.05	0.585	0.63	74.33
500,0.09	0.838	0.973	29.87
1000,0.01	0.014007	0.0194	98.6
1000,0.05	0.368	0.4325	91.491
1000,0.09	0.937	1.064	34.56

Table 1: Accuracy- error table for activation function tanh

N, η	Training Error	Testing Error	Accuracy (%)
300,0.01	0.0372	0.0508	93.45
300,0.05	0.2658	0.4031	87.49
300,0.09	0.273	0.2701	95.92

500,0.01	0.0274	0.03095	97.33
500,0.05	0.0682	0.0709	96.51
500,0.09	0.174	0.202	88.64
1000,0.01	0.368	0.5328	73.55
1000,0.05	0.9026	0.9018	34.96
1000,0.09	1.121	0.807	26.25

Table 2: Accuracy- error table for activation function ReLu and Sigmoid

Note that exact training times cannot be added to the report. However, for 50 epochs, when N=300 and η=0.01 training time was around 2 hours 10 minutes. Similarly, for N=500 and η=0.01 training time was measured as 4 hours. Lastly, for N=1000 and η=0.01 training time was around 8 hours 30 minutes.

From Table 1 and Table 2 it can be seen that the best accuracy occurs when N=500 and η=0.01 with tanh activation function. Therefore, those parameters will be used for batch training process.

Batch Size	Training Error	Test Error	Accuracy (%)
10	0.01352	0.01579	98.85
50	0.04257	0.04368	96.48
100	0.685	0.692	73.3

Table 3: Batch training outcomes.

From Table 3, it can be seen that batch training increases the accuracy of the network. Furthermore, implementation time is decreased due to batch training. Average training time is around 1 hour 15 minutes.

Best case occurs when batch size is 10.

Lastly weight regularization is implemented.

$$E_{reg}(n) = E(n) + \frac{\lambda}{2} \sum_{all\ wiegths} w_{ij}$$

Regularization is applied to the best batch training algorithm which is Batch size= 10 from Table 3. The value of λ selected as λ= 0.01 and λ=0.001

λ	Training error	Test Error	Accuracy (%)
0.01	0.046	0.0614	90.38
0.001	0.04162	0.04351	98.33

Table 4: Weight Regularization results

4. Conclusion

In this project a neural network is designed to classify handwritten digit images from MNIST dataset. The theorical computations are used in network algorithm.

The accuracy of the network is mostly affected by learning rate, and the activation function. It can be understood that smaller learning rates give more accurate results. Also, tanh function gives better outcomes in this network. However, the number of neurons in the hidden layer did not affect the accuracy of the network since the best case occurs when N=500.

Using batch training algorithm increased the accuracy and efficiency of the network since the elapsed time is shorter. It can be seen that the smaller batch sizes give more accurate results.

Lastly, in L_2 regularization part, a crucial change in accuracy is observed when λ decreases.

Appendix

Please see some of the outputs.

Activation Function: Tanh N= 300, η=0.01

```
In [14]: err
Out[14]: 0.012246425224487312

In [15]: np.sum(y != outcomes)
Out[15]: 310

In [16]: np.sum(y != outcomes)/len(y)
Out[16]: 0.005166666666666667

In [ ]:
```

```
    outcomes[i]     = np.argmax
    err = err+ (np.sum(np.squar
err = err/len(X)
```

```
In [20]: np.sum(y != outcomes)/len(y)
Out[20]: 0.0196

In [ ]: class case2:
```

Tanh 500- 0.01

```
In [10]: X = x_train
         y = y_train

In [11]: for i in range(50):
             for i in range(len(X)):
                 W1, W2, W1C, W2C = case1().gradient(X[i], y[i], W1, W2, W1C, W2C,N)

In [12]: case1().forward_propagation(X[0], W1C, W2C)[4]

Out[12]: array([-1.        , -1.        , -1.        , -0.99993387, -1.        ,
                 0.99999534, -1.        , -0.99999997, -1.        , -1.        ])

In [13]: err = 0
         obs = np.zeros((len(y),10))
         outcomes = np.zeros(len(y))

         for i in range(len(X)):
             obs[i] = case1().forward_propagation(X[i], W1C, W2C)[4]
             outcomes[i]     = np.argmax(obs[i])
             err = err+ (np.sum(np.square(obs[i] - case1().labeling1(y[i]))/2, axis = 0))
         err = err/len(X)

In [14]: err
Out[14]: 0.012213910784241302

In [15]: np.sum(y != outcomes)
Out[15]: 309

In [16]: np.sum(y != outcomes)/len(y)
Out[16]: 0.00515
```

In [17]:
```
X = x_test
y = y_test
```

In [18]:
```
err = 0
obs = np.zeros((len(y),10))
outcomes = np.zeros(len(y))

for i in range(len(X)):
    obs[i] = case1().forward_propagation(X[i], W1C, W2C)[4]
    outcomes[i]     = np.argmax(obs[i])
    err = err+ (np.sum(np.square(obs[i] - case1().labeling1(y[i]))/2, axis = 0))
err = err/len(X)
```

In [19]:
```
np.sum(y != outcomes)/len(y)
```

Out[19]: 0.0208

In [ ]:
```
class case2:
    #initialization of parameters
    def initilization_prmt(self,N):
        W1=np.random.normal(loc=0.0, scale=0.01, size=(N,784))
        b1=np.zeros((N,1))
        W2=np.random.normal(loc=0.0, scale=0.01, size=(10,N))
        b2=np.zeros((10,1))
        W1C = np.concatenate((W1, b1), axis = 1)
        W1C = np.concatenate((W2, b2), axis = 1)
        return W1, b1,W1C, W2, b2, W2C

    def activation_relu(self, X):
        return np.maximum(X,0)

    def activation_sigmo(self, X):
        sig=1/(1+np.exp(-X))
        return sig
```

Tanh 1000-0.01

In [13]:
```
err = 0
obs = np.zeros((len(y),10))
outcomes = np.zeros(len(y))

for i in range(len(X)):
    obs[i] = case1().forward_propagation(X[i], W1C, W2C)[4]
    outcomes[i]     = np.argmax(obs[i])
    err = err+ (np.sum(np.square(obs[i] - case1().labeling1(y[i]))/2, axis = 0))
err = err/len(X)
```

In [14]:
```
err
```

Out[14]: 0.015007730922510417

In [15]:
```
np.sum(y != outcomes)
```

Out[15]: 359

In [16]:
```
np.sum(y != outcomes)/len(y)
```

Out[16]: 0.0059833333333333336

In [17]:
```
X = x_test
y = y_test
```

In [18]:
```
err = 0
obs = np.zeros((len(y),10))
outcomes = np.zeros(len(y))

for i in range(len(X)):
    obs[i] = case1().forward_propagation(X[i], W1C, W2C)[4]
    outcomes[i]     = np.argmax(obs[i])
    err = err+ (np.sum(np.square(obs[i] - case1().labeling1(y[i]))/2, axis = 0))
err = err/len(X)
```

In [19]:
```
np.sum(y != outcomes)/len(y)
```

Out[19]: 0.0194

In [ ]:
```
class case2:
    #initialization of parameters
    def initilization_prmt(self,N):
        W1=np.random.normal(loc=0.0, scale=0.01, size=(N,784))
```

Batch Training result

In [26]:
```
print('Case 1 \nMini Batch Size = %.d\nN = %.d\nLearning Rate     = %.2f \nTraining Error    = %.4f \nTraining Accuracy = %.4f \r
```

```
Case 1
Mini Batch Size = 10
N = 500
Learning Rate     = 0.01
Training Error    = 0.0158
Training Accuracy = 98.8500
Completation Time = 1.08 seconds
```