Deep Learning Assignment 1

Name : Dipankar Sarkar (MT20307) Shivam Dalmia (MT20304)

PART I: PERCEPTRON TRAINING ALGORITHM (PTA)

1.

A)

Methodology:

We have to create a perceptron for OR,AND and NOT gates. A perceptron by definition uses threshold as it's activation function. The threshold taken here is '1' and if a result is less than '1' -1 is assigned to it and greater than equal to 1 we assign 1 to it.
The first step is creating OR and AND table. Then we have one function for both OR,AND and NOT tables that is perceptron. The neuron takes 2 inputs 'x1' and 'x2' and one bias that is always '1' so we have 3 weights for the perceptron. We take all the weights as random and send it to the perceptron function. The flow of the function is first it takes the first point of the data and multiply the data with its respective weights and see if the threshold is greater than equal to 1 or less than one according to that the value is given if the value matches the output then we check it for the remaining points and if the given value does not match then we subtract the weights with multiplication of the given value and the respective data value. And check it again with the updated weights. Now if the weights are giving the right result then we check it for the remaining points any point which gives the wrong output or any error we take that point and repeat the same procedure as mentioned above.

Helper Functions or Code Explanation:

i) ther=(W1_*df['x1'][i])+(W2_*df['x2'][i])+(b_*df['bi'][i])
or
ther=(W1*df['x1'][i])+(b*df['bi'][i]):
This is the snippet for attaining the output from the given weights and data points value (W1_, W2_ and b_ are weights and x1 ,x2 and bi are the data points). If ther is greater than equal to 1 then output is 1 otherwise -1.

ii)W1_ -= df['x1'][i] Weights update if the desired values are not attained
W2_ -= df['x2'][i] This step is not included in the NOT gate.
b_ -= df['bi'][i]

Preprocessing :

In preprocessing we just created the table for OR , AND and NOT with a bias column.

Assumptions :

The bias columns value is always 1.

B)

Methodology:

The weight calculated by the perceptron is the vector of a line that is perpendicular to the decision boundary that we want to draw. The property of the two perpendicular lines is that the dot product of the vectors of those lines is equal to 0. Now from the calculated weight vectors are assuming the value of x-axis and ‘c’ in the formula y=mx+c we calculated the value of y with the assumed x and c value.

Helper Functions or Code Explanation:

```
plt.figure()
    X = np.linspace(-0.5,2,10)
    C = np.linspace(-0.5,2,10)
    Y = -(W2_*X+b_*c)/W1_
    plt.plot(X,Y)
    plt.scatter(df[df['y']==0]['x1'],df[df['y']==0]['x2'])
    plt.scatter(df[df['y']==1]['x1'],df[df['y']==1]['x2'])
    plt.show()
```

We use matplotlib figure function to plot the points and the decision boundary.

Assumptions :

Assumption : We assume the value of X and C.

C)

To show that XOR is not possible we run the same function for XOR that we did for OR and AND but then it went on the infinite loop then after force stopping the code when we looked at the different weights of the perceptron we found that the weights were repeating thus confirming an endless loop and stating that one perceptron is unable to converge XOR gate.

2.

A)

Methodology:

We have to create a network using Madeline for the graph given in the question. So first we calculate the number of neurons that we need to solve this problem. Looking at the problem we need 6 decision boundaries horizontally and 6 decision boundaries vertically to separate all the red areas and white areas. So to draw 1 decision boundary we need one neuron so to draw 12

decision boundary we need 12 neurons and we also need 5 neuron for the boxes and to connect all the neurons we need another neuron. So in total we need 18 neurons to solve this problem if we have a two hidden layer. So 12 neurons in the first hidden layer and 5 in the second hidden layer 1 neuron at the output layer, each neuron in the middle layers takes 3 inputs x1,x2 and bias (always 1) and for the second hidden layer we have 12 weights for each neuron. So each middle neuron has three weights and the outer neurons will have 13 inputs one from each neuron and 1 for bias. For the first epoch we took all the weights as random then we took a data point and calculated the output of the network. To do that we calculated the output of each neuron in the middle layer and passed it through our threshold activation function where 0 is taken as the threshold, greater than equal to 0 is given 1 otherwise -1. Now all the output from the neurons are stored in the Output array and then we calculate the output of the output neuron by middle layer output and output weights. The output is then passed through our threshold function and then we produce our final result for that data point. While doing this we calculated the difference between the threshold and output a neuron calculates before passing it to the activation function. So if the generated output matches the desired output then we check all the data points with the weights and if it satisfies all the data points we stop the code. If the generated output does not match the desired output we take the neuron with the minimum difference from our difference array and change the weights associated with that neuron. We change the neuron by
W_new = W_old - a(t-o)x
W_old = weights of the neuron
a = learning rate
t = desired output from that neuron after threshold
o = output from the neuron
x = datapoint
After changing the neuron weights we check the final output again if the output matches then we check for all the data points again and repeat the steps.

Helper Functions or Code Explanation:

output[] : This matrix contains the output of every neuron.
diff[] : This matrix contains the difference between threshold and output from the neuron.
def out(c,n,u,i,diff,mid_w,mid_1_w,out_w,output): This function returns the final value from a data point and updates the difference matrix.
def update(c,n,u,i,diff,mid_w,mid_1_w,out_w,output):This function checks the output from the data point and updates the weights of the neurons, difference matrix if the output is wrong.
def check(n,k, mid_w,mid_1_w,out_w): This function returns the final value for a data point.

Assumptions :

The weights of all the neurons and the data points are taken from the given graph and all the biases are taken as 1.

B) Since f(x1,x2) is a non linear function hence it cannot be computed using two neurons as two neurons will not be able to create decision boundaries that will be required for classification. Moreover the network will not be sufficiently wide and deep with two neuron hence it will not be able to achieve.

PART II: MULTILAYER PERCEPTRON (MLP)

1.
Methodology:

We first take the data as input from the user in the form of CSV files fashion-mnist_train.csv and fashion-mnist_test.csv of the dimension **60K x 785** and **10k x 785**. We then separate the “label” column from the loaded data frames and create two numpy array for data and label for both train and test dataset. We normalize the data across 784 feature as pixel values by dividing it with 255.

For the Neural Network part we take two weights array one for the hidden layer W_H and W_O for the output layer along with biases for the hidden and output layers as B_H and B_O. For the hidden layer we have created our results based on 1**28 neurons** thereby making the weight matrix of size **784 x 128** for hidden layers and **128 x 10** for the output layer. For the biases we have we have **1 x 128** for the hidden layer and **1 x 10** for the output layer.

We then iterate through each epoch and for every epoch we either update the weights batch_size wise as mini batch or we do it for the entire dataset as a batch size. The training is done using both a forward pass and a backward pass. If we use gradient descent with full batch size equal to the training dataset we update the weight after the error generated from the entire dataset and in case of mini batch we do that with the batch size of **64.** We have created activation function from scratch for tanh, reLU and sigmoid and softmax for the output and hidden layers. Before the training we initialize the various weights using random normal functions. After this we go for the forward pass where we calculate the output of the hidden layer neuron by dot multiplication of the weight and data matrix followed by activation of that layer. The activate output is then passed to the output layer which has a activation function as softmax always. The output of the softmax function is in terms of probability of the 10 classes.

For weight update we first calculate the output layer error and then we change the output layer weights based on the derivation of the Loss that is categorical cross entropy loss with respect to weight and than change the weight according as

Wi =Wi - eta * dL/dWi where we fix the eta as 0.01 which here is the learning rate. We then backpass the output layer error to the hidden layer and then we find the change in weight with respect to the hidden layers weight using the same formula.

For output layer weight we have

dL/dWi=(dz(2)/dW_0)*(dE/dz(2))

For the hidden layer wight update we have

dL/dW=(dz(1)/dW_H)*f`(z)*dE/dy(1)

We perform the model suggested above on batch size equal to entire dataset with 0.01 learning rate and 100 epochs for tanh ,sigmoid and relu and analyze the results. **Out of them we decide to go with tanh as the it has nearly equal accuracy as that of relu but the convergence is better in terms of loss for tanh.**

Helper Functions or Code Explanation

```
#Loading the dataset

train_data=pd.read_csv('./dataset_fashion/fashion-mnist_train.csv')
test_data=pd.read_csv('./dataset_fashion/fashion-mnist_test.csv')
train_label=train_data['label']
test_label=test_data['label']
train_data.drop('label',axis='columns',inplace=True)
test_data.drop('label',axis='columns',inplace=True)

X_train=train_data.to_numpy().astype('float64')/255
Y_train=train_label.to_numpy()
X_test=test_data.to_numpy().astype('float64')/255
Y_test=test_label.to_numpy()
```

In the above code we are loading the data files for training and testing .

```
# activation functions
def tanh(x):
    return 2*sigmoid(2*x)-1

def sigmoid(x):
        if x.any()<0:
            return np.exp(x)/(1+np.exp(x))
        else:
            return 1/(1+np.exp(-x))
def relu(x):
    return np.maximum(x,0)

def softmax(output_array):
    return np.exp(output_array)/np.sum(np.exp(output_array), axis = 1, keepdims = True)
```

```
# activation function derivatives
def sigmoid_der(x):
    return sigmoid(x) *(1-sigmoid (x))

def tanh_der(x):
    return (1-tanh(x)**2)

def relu_der(x):
    x[x<0]=0
    return x
```

In the above code we define helper for all the different activation functions along with their derivatives.

A thing to note we use sigmoid in two different forms in order to avoid overflow of the exp function. Moreover since sigmoid can handle the overflows we represent the tanh as a function of sigmoid hence getting the advantage of handling overflow. Resource https://sebastianraschka.com/faq/docs/tanh-sigmoid-relationship.html

```
def NN(number_of_neuron , data , label , epochs , batch_size ,activation_func,learning_rate,optimizer_name,
       momentum,decay_rate,beta1,beta2):
```

This is the helper function for training the model.

```
W_H=np.random.randn(data.shape[1], number_of_neuron)#weight initialization of Hidden layer(784*128)
W_O=np.random.randn(number_of_neuron,label_count)#weight initilization of output layer(128*10)
B_H=np.random.randn(number_of_neuron)#weights for hidden layer bias(1*128)
B_O=np.random.randn(label_count)#weights for output bias (1*10)


# optimizers weights initilization

M_W_H=np.zeros((data.shape[1], number_of_neuron))
M_W_O=np.zeros((number_of_neuron,label_count))
M_B_H=np.zeros(number_of_neuron)
M_B_O=np.zeros(label_count)

M_W_H1=np.zeros((data.shape[1], number_of_neuron))
M_W_O1=np.zeros((number_of_neuron,label_count))
M_B_H1=np.zeros(number_of_neuron)
M_B_O1=np.zeros(label_count)

M_W_H2=np.zeros((data.shape[1], number_of_neuron))
M_W_O2=np.zeros((number_of_neuron,label_count))
M_B_H2=np.zeros(number_of_neuron)
M_B_O2=np.zeros(label_count)
```

Weight initialization for the layers bias and also for the optimizers

```
hidden_layer = np.dot(data[start:end,:],W_H)+B_H # batch_size*no_of_neuron
hidden_layer_activation=activation(hidden_layer,activation_func)# batch_size*no_of_neuron
output_layer=np.dot(hidden_layer_activation,W_O)+B_O# batch_size*labels
ouput_layer_activation=softmax(output_layer)# batch_size*labels probability in terms of classes(10 class
```

Forward pass implemented as above

```
            #Backward Pass

            output_layer_error=(ouput_layer_activation-one_hot_label[start:end,:])/(end-start) #dE/dz(2) for output
            W_O_G=np.dot(hidden_layer_activation.T,output_layer_error) #(dz(2)/dW_0)*(dE/dz(2))
            B_O_G=np.sum(output_layer_error, axis = 0, keepdims = True)

            hidden_layer_error=np.dot(output_layer_error, W_O.T) # (dz(1)/dW_H)
            derivative=activation_derivative(hidden_layer_error,activation_func)# f`(z)

            W_H_G = np.dot(data[start:end,:].T, derivative*hidden_layer_error)# (dz(1)/dW_H)*f`(z)*dE/dy(1)
            B_H_G = np.sum(hidden_layer_error*derivative, axis = 0, keepdims = True)

            #Weight updates
```

Backward pass implemented as above where we update the middle and outlier weights based on batch and optimizers.

```
def predict(model,data,label):
    hidden_layer = np.dot(data,model[1])+model[3] # batch_size*no_of_neuron
    hidden_layer_activation=activation(hidden_layer,model[5])# batch_size*no_of_neuron
    output_layer=np.dot(hidden_layer_activation,model[0])+model[2]# batch_size*labels
    ouput_layer_activation=softmax(output_layer)
    predicted_label=np.argmax(ouput_layer_activation,axis=1)
    print(label.shape)
    count=0
    for i in range(data.shape[0]):
        if label[i]==predicted_label[i]:
            count+=1
    print("Accuracy of the Network is ",((count/10000)*100))
```

The above function is used to predict the test datas with the trained model to get accuracy

The final model with tanh had an accuracy of 42.91 but the loss convergence was better than relu so we proceed with that.

Preprocessing :

We have normalized the 784 feature of the training and testing dataset.

Assumptions :

Our network is fully customizable but for now we have assumed that it can have only one hidden layer all other parameters configuration wise as well as optimization wise are user dependent.

2. From the above we choose to go continue with the model having 128 hidden layer neurons with tanh and output layer with 10 neuron and softmax activation. We continue with the same learning rate 0.01 and mini batch as 64. We use various optimizers

a) Gradient Descent with Momentum: Every thing remains the same we have to update the weights using momentum. The equation is

Mi=momentum*Mi+(1-momentum)*dWi
Wi=Wi-learning_rate*Mi

```
M_W_O=momentum*M_W_O+(1-momentum)*W_O_G
W_O=W_O-learning_rate*M_W_O

M_B_O=momentum*M_B_O+(1-momentum)*B_O_G
B_O=B_O-learning_rate*M_B_O

M_W_H=momentum*M_W_H+(1-momentum)*W_H_G
W_H=W_H-learning_rate*M_W_H

M_B_H=momentum*M_B_H+(1-momentum)*B_H_G
B_H=B_H-learning_rate*M_B_H
```

The above code is used for implementing the momentum. We see that with momentum and mini_batch we get an accuracy of 82.28 and also converges good. But generally with momentum we run the risk of missing the missing the local minima because of high momentum hence we need to reduce the momentum and can expect to get a better performance in terms of accuracy.

b) Nesterov's Accelerated Gradient: We here use the NAG to keep a check on the momentum with an expectation that we will have better loss and hance better result.

Wi=Wi+(momentum*momentum)*Mi-(1+momentum)*learning_rate*dWi
Mi=momentum*Mi-learning_rate*dWi

```
f optimizer_name=="nag":

W_O=W_O+(momentum*momentum)*M_W_O-(1+momentum)*learning_rate*W_O_G
M_W_O=momentum*M_W_O-learning_rate*W_O_G

B_O=B_O+(momentum*momentum)*M_B_O-(1+momentum)*learning_rate*B_O_G
M_B_O=momentum*M_B_O-learning_rate*B_O_G

W_H=W_H+(momentum*momentum)*M_W_H-(1+momentum)*learning_rate*W_H_G
M_W_H=momentum*M_W_H-learning_rate*W_H_G

B_H=B_H+(momentum*momentum)*M_B_H-(1+momentum)*learning_rate*B_H_G
M_B_H=momentum*M_B_H-learning_rate*B_H_G
```

The above is the implementation of NAG

What we observe is that since we have used momentum initially to reach the global minima faster it overshot the global minima as we use mini batch and for this even though it is faster in reaching the global minima still because of its momentum it tends to miss the global minima where as NAG was able to control the velocity and thus controlled the oscillation and we achieved better results with accuracy of 85.21 and and loss of 0.04 better than momentum above.

c) AdaGrad optimizer: In order to control the gradient lengths we use adagrad optimizer

Mi=Mi+(dWi)**2
Wi=Wi-(learning_rate*dWi)/(np.sqrt(Mi)+epsilon)

```
elif optimizer_name=="adagrad":

    M_W_O=M_W_O+(W_O_G)**2
    W_O=W_O-(learning_rate*W_O_G)/(np.sqrt(M_W_O)+1e-7)

    M_B_O=M_B_O+(B_O_G)**2
    B_O=B_O-(learning_rate*B_O_G)/(np.sqrt(M_B_O)+1e-7)

    M_W_H=M_W_H+(W_H_G)**2
    W_H=W_H-(learning_rate*W_H_G)/(np.sqrt(M_W_H)+1e-7)

    M_B_H=M_B_H+(B_H_G)**2
    B_H=B_H-(learning_rate*B_H_G)/(np.sqrt(M_B_H)+1e-7)
```

The above is the implementation of Adagrad optimizer with their weights

We observe that adagrad uses sum of squares of gradients of previous steps to decide the learning rate thus keeping a check on the weight change. Thus with a paricular combination of weights their is a possibility that the denominator becomes big and as a result the learning can get slow. Thus because of this we can find the convergence got effected with a loss of 0.1 at the end of 100 epoch and also because of slow learning the accuracy got to 78.17

d) RMSProp optimizer: As stated above to overcome the difficulty we can use rmsprop as it keeps a check on the denominator and does not allow the learning rate to drastically reduce like that in adagrad. We took decay rate as 0.99 to balance the denominator.

Mi =(decay_rate * Mi) + ((1-decay_rate) * (dWi **2))
Wi =Wi-((learning_rate * dWi)/(np.sqrt(Mi)+ epsilon))

```
elif optimizer_name=="rmsprop":

    M_W_O =(decay_rate * M_W_O) + ((1-decay_rate) * (W_O_G **2))
    W_O =W_O-((learning_rate * W_O_G)/(np.sqrt(M_W_O)+ 1e-7))

    M_B_O =(decay_rate * M_B_O) + ((1-decay_rate) * (B_O_G **2))
    B_O =B_O-((learning_rate * B_O_G)/(np.sqrt(M_B_O)+ 1e-7))

    M_W_H =(decay_rate * M_W_H) + ((1-decay_rate) * (W_H_G **2))
    W_H =W_H-((learning_rate * W_H_G)/(np.sqrt(M_W_H)+ 1e-7))

    M_B_H =(decay_rate * M_B_H) + ((1-decay_rate) * (B_H_G **2))
    B_H =B_H-((learning_rate * B_H_G)/(np.sqrt(M_B_H)+ 1e-7))
```

The above is the implementation of RMsprop optimizer with their weights

We observed that with rmsprop optimizer we were able to overcome the issues we faced with adagrad and hence the loss convergence got better as well the accuracy 81.33 better than adagrad.

e) Adam optimizer: We found that rmsprop was able to control the learning rate pretty well but it was not having the speed to reach the global minima with the same epoch of 100 so we implemented adam optimizer that has the power of both rmsprop and momentum inorder to have both control and speed for the model. We took beta1 and beta2 as 0.1 & 0.2

M1i= (beta1 * M1i +(1-beta1) * (dWi))
M2i = (beta2 * M2i +(1-beta2) * (dWi**2))
M1i=M1i/(1-beta1**itr)
M2i=M2i/(1-beta2**itr)
Wi=Wi-(learning_rate * M1i)/(np.sqrt(M2i) + epsilon)

```
M_W_H1 = (beta1 * M_W_H1 + (1-beta1) * (W_H_G))
M_W_H2 = (beta2 * M_W_H2 + (1-beta2) * (W_H_G**2))
M_W_H1=M_W_H1/(1-beta1**itr)
M_W_H2=M_W_H2/(1-beta2**itr)
W_H=W_H-(learning_rate * M_W_H1)/(np.sqrt(M_W_H2) + 1e-8)

M_B_H1 = (beta1 * M_B_H1 + (1-beta1) * (B_H_G))
M_B_H2 = (beta2 * M_B_H2 + (1-beta2) * (B_H_G**2))
M_B_H1=M_B_H1/(1-beta1**itr)
M_B_H2=M_B_H2/(1-beta2**itr)
B_H=B_H-(learning_rate * M_B_H1)/(np.sqrt(M_B_H2) + 1e-8)
```

The above is the implementation of Adam optimizer with their weights

We observed that with adam we have both the speed and control over the gradient length and hence we got a better convergence and accuracy also improved to 83.28 which is better than rmsprop and momentum.

Assumptions :

Our assumption about the epsilon was we used a fixed epsilon of 1e-7 for all optimizer other than adam where we had 1e-8 and all optimizer had tha same configuration and for optimization parameters we will have 0.01 learning rate.