Homework 1 – EEL6814

Q1. Create a letter A (see figure below) with a two hidden layer MLP

with signum units (threshold nonlinearity).

State the smallest number of hidden units you will need in each layer and

explain their role in creating the mask. Assume that black is -1 and white is 1.

Select by hand the parameters of the network (assume that the center of the figure is (0,0)).

Can you achieve the same goal with a single hidden layer network? Black=0, white=1.

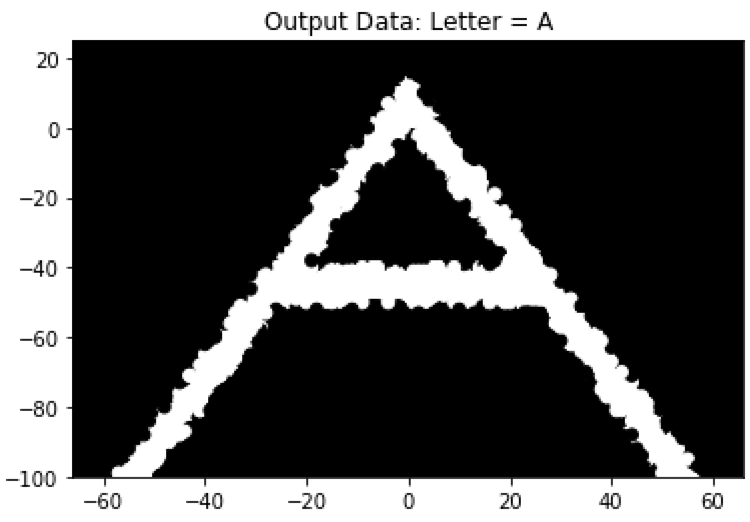
Justify your answer.

Code:

```
import numpy as np

import matplotlib.pyplot as plt



x=np.random.randint(-100,100, size=(100000,2))


w1 = np.array([[2,-1],[2,-1],[-2,-1],[-2,-1],[0,1],[0,1]])

b1 = np.array([15,1,15,1,40,35]).reshape(6,1)

v1 = (np.dot(w1, x.T) + b1)

a1 = np.sign(v1.T)


w2 = np.array([[1,0,1,0,0,0],[0,1,0,1,-1,1]])

b2 = np.array([[-1],[-1]])

v2= np.dot(w2, a1.T) + b2

a2 = np.sign(v2.T)


w3 = np.array([1,-1]).reshape(1,2)

b3 = np.array([-0.8]).reshape(1,1)
```

```
v3 = np.dot(w3, a2.T) + b3

a3 = np.sign(v3.T)


data=np.zeros((100000,3))

data[:,0:2]=x

data[:,2]=a3[:,0]


plt.figure()

plt.title('Output Data: Letter = A')

col= np.array(['w','w','k'])

plt.xlim(-66,66)

plt.ylim(-100,25)

plt.scatter(x[:,0],x[:,1], c=col[a3[:,0].astype(int)])

plt.show()
```

Explanation- We have implemented this solution and the result is below.



So for this implementation we’ve had a six neuron input layer, a two neuron hidden layer and a single output neuron. For the first hidden layer, we must distinguish the regions around the six edges, two for left edges, two for the right edges and two edges for the portion in between. The first layers set out these edges and adjust their positions. The next layer assigns value black for the triangle and trapezium in the figure, which is in the center of the image. Finally, the output layer sets the regions between these segments as white.

Q2. Code the backpropagation algorithm and test it in the following 2 class problem called the Star problem. Use a single hidden layer MLP and specify the size of the hidden layer and tell why you select that number of hidden PEs.

x1	**x2**	**d**
1	**0**	**1**
0	**1**	**1**
-1	**0**	**1**
0	**-1**	**1**
0.5	**0**	
-.5	**0.5**	**0**
-.5	**0**	
-.5	**-.5**	**0**

I expect that the system learns this pattern exactly. You can think that points close to the x/y axes belong to one class and all the others belong to another class (hence the star). See how well your solution performs in points that do not belong to the training set, by selecting other points that follow the pattern (keep the points within the square of size 2, center at the origin). How could you improve the generalization accuracy of the solution? Show experimentally how your suggestion works.

Code:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon Sep 28 20:51:04 2020
```

```
@author: Abhishek Jha

HMW1 - Problem 2

Code the backpropagation algorithm and test it in the following 2 class problem
called the Star problem. Use a single hidden layer MLP and specify the size of the
hidden layer and tell why you select that number of hidden PEs.

x1   x2    d
1    0     1
0    1     1
-1   0     1
0   -1     1
0.5   0
-.5  0.5   0
-.5   0
-.5  -.5   0

I expect that the system learns this pattern exactly. You can think that points close
to the x/y axes belong to one class and all the others belong to another class (hence the star).
See how well your solution performs in points that do not belong to the training set,
by selecting other points that follow the pattern (keep the points within the square
of size 2, center at the origin). How could you improve the generalization accuracy of
the solution? Show experimentally how your suggestion works.


"""

import numpy as np
```

```
class MLP:

    def __init__(self, ip=2, hidden=[8], op=1):

        self.ip = ip
        self.hidden = hidden
        self.op = op

        layers = [self.ip] + hidden + [self.op]

        self.weights = []
        for i in range(len(layers) - 1):
            w = np.random.rand(layers[i], layers[i+1])
            self.weights.append(w)

        activations = []
        for i in range(len(layers)):
            a=np.zeros(layers[i])
            activations.append(a)
        self.activations = activations

        derivatives = []
        for i in range(len(layers) - 1):
            d=np.zeros((layers[i], layers[i+1]))
            derivatives.append(d)
        self.derivatives = derivatives
```

```
def forward_propogation(self, inputs):

    activations = inputs
    self.activations[0] = inputs

    for i, w in enumerate(self.weights):
        net_inputs = np.dot(activations, w)
        activations = self._sigmoid(net_inputs)
        self.activations[i+1] = activations
    return activations

def back_propagate(self, error, flag=False):

     for i in reversed(range(len(self.derivatives))):
         activations = self.activations[i+1]
         delta = error * self._sigmoid_derivative(activations)
         delta_reshaped = delta.reshape(delta.shape[0], -1).T
         current_activations = self.activations[i]
         current_activations_reshaped = current_activations.reshape(current_activations.shape[0], -1)
         self.derivatives[i] = np.dot(current_activations_reshaped, delta_reshaped)
         error = np.dot(delta, self.weights[i].T)

         if flag:
             print("Derivatives for W{} : {}".format(i, self.derivatives[i]))

     return error

def steepest_descent(self, learning_rate):
        for i in range(len(self.weights)):
            weights = self.weights[i]
#           print("Original W{} {}".format(i, weights))
            derivatives = self.derivatives[i]
            weights += derivatives * learning_rate
#           print("Updated W{} {}".format(i, weights))



    def _sigmoid(self, x):
        return 1 / (1 + np.exp(-x))


    def _sigmoid_derivative(self, x):
        return (x * (1.0 - x))


    def _mse(self, target, output):
        return np.average((target - output)**2)



    def train(self, inputs, targets, epochs, learning_rate):


        for i in range(epochs):
            sum_error = 0
            for inp, target in (zip(inputs, targets)):


                output = self.forward_propogation(inp)


                error = target - output


                self.back_propagate(error)
            self.steepest_descent(learning_rate)

            sum_error += self._mse(target, output)

        av_error = sum_error/len(inputs)
        print("Average Error : {} at epoch {}".format(av_error, i))


if __name__ == "__main__":


    #Create a MLP and train the model
    # Use this section to modify the Learning rate
    mlp = MLP()
    learning_rate=0.15 #hyper-parameter
    epochs=10000
    inputs = np.array([[1,0],[0,1],[-1,0],[0,-1],[5,0.5],[-0.5,0.5],[5,-0.5],[-0.5,-0.5]])
    d = np.array([[1],[1],[1],[1],[0],[0],[0],[0]])
    mlp.train(inputs, d, epochs, learning_rate)

    # Test The Model

    input_test = np.array([0,1])
    t = np.array([1])
    output = mlp.forward_propogation(input_test)
    print("Input, Expected, Actual")
    print(input_test, t, output)
```

Explanation:

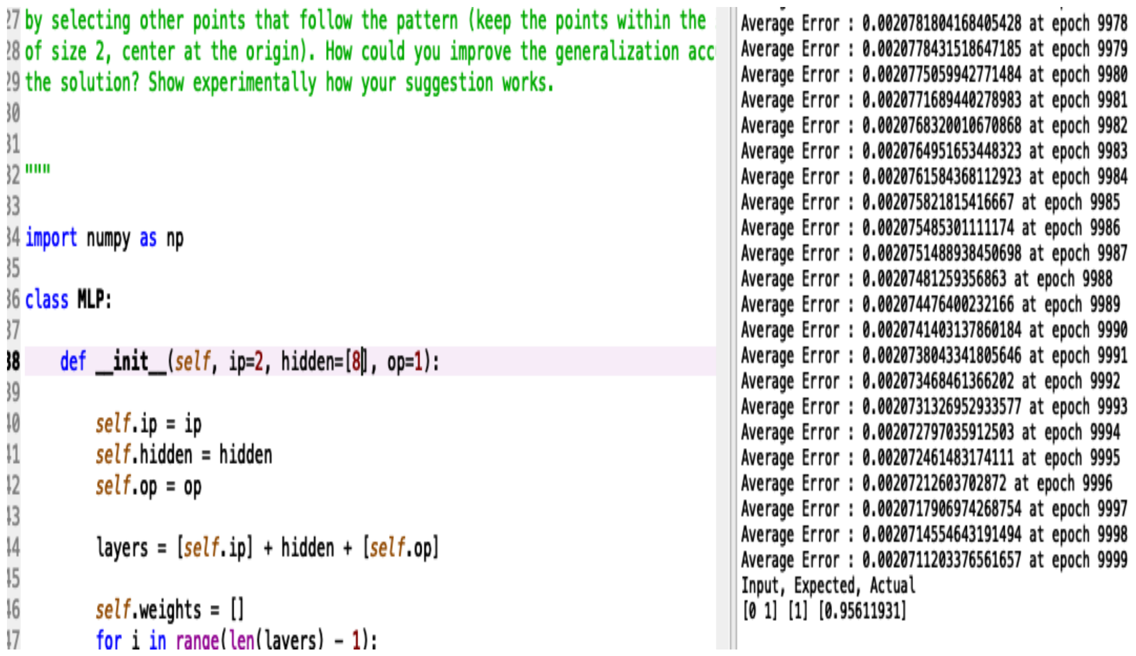
For this implementation we have python object Multi-Layer Perceptron, a class which has encapsulated functions such as forward propagation, back propagation, the steepest descent learning algorithm and the training model which continues to perform back-propagation to adjust randomly generated weights. The training method loops through the number of epochs we wish to train the model. The weights are updated depending on the hyper-parameter learning rate. Also, the main method triggers the number of epochs the model wants to continue back-propagation.

For this implementation we have chosen the default learning rate of the order of 0.1, and since we are generating the weight matrix with random integers the results vary sometimes when training the data. However, the number of epochs can be adjusted to get a better generalization accuracy over time. For this model we used the hyper-parameter adjusted to 0.11, trained for 100000 epochs and the results are below.

```
Average Error: 0.06971669028977288 at epoch 99994

Average Error: 0.06971670729284702 at epoch 99995

Average Error: 0.06971672429741958 at epoch 99996

Average Error: 0.06971167413034890 at epoch 99997

Average Error: 0.06971167583110537 at epoch 99998

Average Error: 0.06971167753201122 at epoch 99999


Testing the model -

Input, Expected, Actual

[0 1] [1] [0.94277595]
```

The model achieved accuracy of 94% over time. Here we calculate the updated average error per iteration. If we train this model for 1/10 of the epoch size, we can achieve roughly with 18% average error. This algorithm however does not perform well for samples outside the range and the results are inconsistent. Another encapsulation of an optimizer perhaps may solve this problem such as Adam or RMSProp.

To better improve training in shorter time we can increase the number of hidden layers, for this setup we have 2 input neurons, 8 neurons in hidden layer and 1 output neuron. The results are as below



```
by selecting other points that follow the pattern (keep the points within the
of size 2, center at the origin). How could you improve the generalization acc
the solution? Show experimentally how your suggestion works.


"""

import numpy as np

class MLP:

    def __init__(self, ip=2, hidden=[8], op=1):

        self.ip = ip
        self.hidden = hidden
        self.op = op

        layers = [self.ip] + hidden + [self.op]

        self.weights = []
        for i in range(len(layers) - 1):
```

```
Average Error : 0.0020781804168405428 at epoch 9978
Average Error : 0.0020778431518647185 at epoch 9979
Average Error : 0.0020775059942771484 at epoch 9980
Average Error : 0.0020771689440278983 at epoch 9981
Average Error : 0.0020768320010670868 at epoch 9982
Average Error : 0.0020764951653448323 at epoch 9983
Average Error : 0.0020761584368112923 at epoch 9984
Average Error : 0.002075821815416667 at epoch 9985
Average Error : 0.0020754853011111174 at epoch 9986
Average Error : 0.0020751488938450698 at epoch 9987
Average Error : 0.00207481259356863 at epoch 9988
Average Error : 0.0020744764002321 66 at epoch 9989
Average Error : 0.0020741403137860184 at epoch 9990
Average Error : 0.0020738043341805646 at epoch 9991
Average Error : 0.002073468461366202 at epoch 9992
Average Error : 0.0020731326952933577 at epoch 9993
Average Error : 0.002072797035912503 at epoch 9994
Average Error : 0.0020724614831 74111 at epoch 9995
Average Error : 0.00207212603702872 at epoch 9996
Average Error : 0.0020717906974268754 at epoch 9997
Average Error : 0.0020714554643191494 at epoch 9998
Average Error : 0.0020711203376561657 at epoch 9999
Input, Expected, Actual
[0 1] [1] [0.95611931]
```

This case we have reduced the average error to 0.00207% and predicted a 0.956 for 1 which is a decent approximation considering we started with random weights.