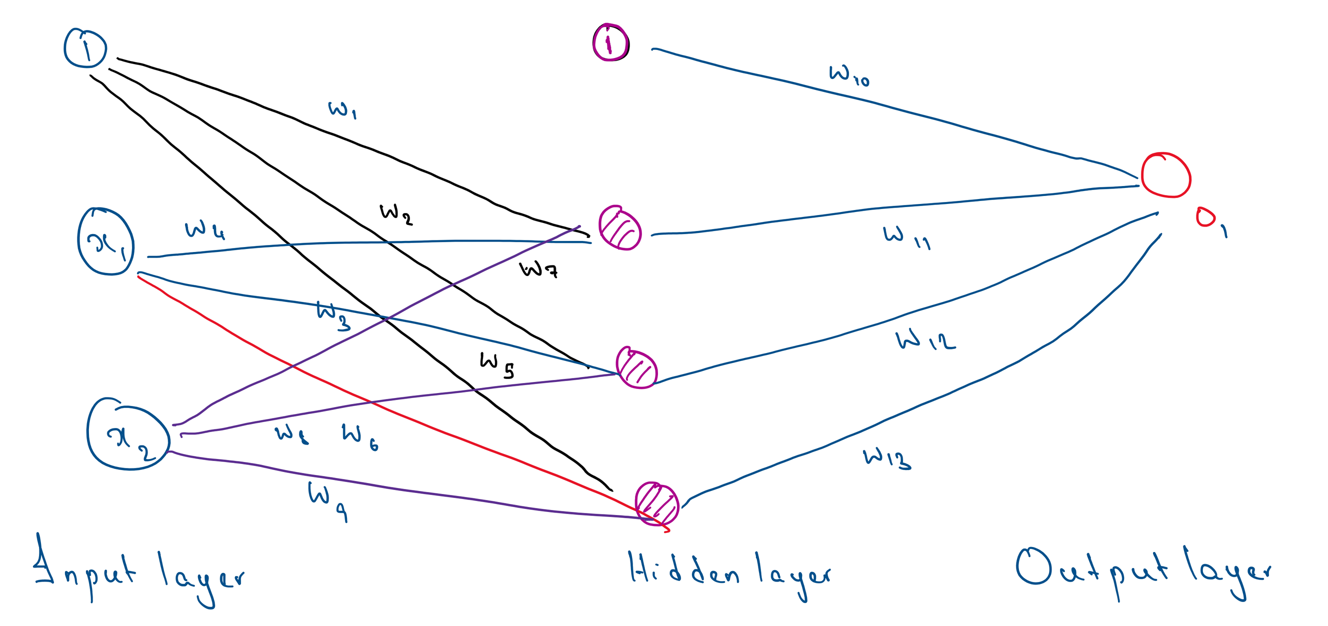
Data Analysis and Machine Learning: Problem Set 5

Tuesday, 29 March 2022   8:04 pm

Q1)



- Choose a sigmoid activation function

$$S(x) = \frac{1}{1+e^{-x}}$$

- Denote α to be the input of each neuron, H = hidden layer neurons and O = output layer neuron
  β = output of each neuron.

$$\alpha_{H_1} = w_1 + x_1 w_4 + x_2 w_7 \qquad \beta_{H_1} = f(\alpha_{H_1})$$

$$\alpha_{H_2} = w_2 + x_1 w_5 + x_2 w_8 \qquad \beta_{H_2} = f(\alpha_{H_2})$$

$$\alpha_{H_3} = w_3 + x_1 w_6 + x_2 w_9 \qquad \beta_{H_3} = f(\alpha_{H_3})$$

- Output layer:

$$\underbrace{\beta_{O_1}}_{\text{predicted output value.}} = f(\alpha_{O_1}) = w_{10} + \beta_{H_1} w_{11} + \beta_{H_2} w_{12} + \beta_{H_3} w_{13}$$

- Define the initial loss function:

$$E = \frac{1}{2}\sum_i^N (y - y_{pred})^2 = \frac{1}{2}(y - \beta_{O_1})^2$$

- Suppose one wishes to compute the derivative of the loss function wrt w_{10}:

$$\frac{\partial E}{\partial w_{10}} = \frac{\partial E}{\partial \beta_{O_1}} \frac{\partial \beta_{O_1}}{\partial \alpha_{O_1}} \frac{\partial \alpha_{O_1}}{\partial w_{10}}$$

$$\therefore \frac{\partial E}{\partial \beta_{O_1}} = -(y - \beta_{O_1}) \cdot \frac{\partial \beta_{O_1}}{\partial \alpha_{O_1}} = \beta_{O_1}(1-\beta_{O_1}) \quad \frac{\partial \alpha_{O_1}}{\partial w_{10}} = 1$$

$$\therefore \frac{\partial E}{\partial w_{10}} = -\beta_{O_1}(y - \beta_{O_1})(1-\beta_{O_1})$$

- One can update the estimate of w_{10} as follows:

$$w_{10}^{(t+1)} = w_{10}^{(t)} - \underbrace{\eta}_{\text{learning rate}} \frac{\partial E}{\partial w_{10}}$$

$$= w_{10}^{(t)} + \eta \beta_{O_1}(y-\beta_{O_1})(1-\beta_{O_1})$$

$$\frac{\partial E}{\partial w_1} = \frac{\partial E}{\partial \beta_{O_1}} \frac{\partial \beta_{O_1}}{\partial \alpha_{O_1}} \cdot \frac{\partial \alpha_{O_1}}{\partial w_1}$$
$$= -(y-\beta_{O_1})(1-\beta_{O_1})\beta_{O_1} w_{11} \beta_1 (1-\beta_{H_1})$$

$$\frac{\partial E}{\partial w_2} = \frac{\partial E}{\partial \beta_{O_1}} \frac{\partial \beta_{O_1}}{\partial \alpha_{O_1}} \frac{\partial \alpha_{O_1}}{\partial w_2}$$
$$= -(y-\beta_{O_1})(1-\beta_{O_1})\beta_{O_1} w_{12} \beta_{H_2}(1-\beta_{H_2})$$

$$\frac{\partial E}{\partial w_3} = \frac{\partial E}{\partial \beta_{O_1}} \frac{\partial \beta_{O_1}}{\partial \alpha_{O_1}} \frac{\partial \alpha_{O_1}}{\partial w_3}$$
$$= -(y-\beta_{O_1})(1-\beta_{O_1})\beta_{O_1} w_{13} \beta_{H_3}(1-\beta_{H_3})$$

$$\frac{\partial E}{\partial w_4} = \frac{\partial E}{\partial \beta_{O_1}} \frac{\partial \beta_{O_1}}{\partial \alpha_{O_1}} \frac{\partial \alpha_{O_1}}{\partial w_4}$$
$$-(y-\beta_{O_1})(1-\beta_{O_1})\beta_{O_1} w_{11} \beta_{H_1}(1-\beta_{H_1}) x_1$$

$$\frac{\partial E}{\partial w_5} = \frac{\partial E}{\partial \beta_{O_1}} \frac{\partial \beta_{O_1}}{\partial \alpha} \frac{\partial \alpha_{O_1}}{\partial w_5}$$
$$= -(y-\beta_{O_1})(1-\beta_{O_1})\beta_{O_1} \beta_{H_2} w_{12}(1-\beta_{H_2}) x_1$$

$$\frac{\partial E}{\partial w_6} = -(y-\beta_{O_1})(1-\beta_{O_1})\beta_{O_1}\beta_{H_3} w_{13}(1-\beta_{H_3}) x_1$$

$$\frac{\partial E}{\partial w_7} = -(y-\beta_{O_1})(1-\beta_{O_1})\beta_{O_1}\beta_{H_1} w_{11}(1-\beta_{H_1}) x_2$$

$$\frac{\partial E}{\partial w_8} = -(y-\beta_{O_1})(1-\beta_{O_1})\beta_{O_1}\beta_{H_2} w_{12}(1-\beta_{H_2}) x_2$$

$$\frac{\partial E}{\partial w_9} = -(y-\beta_{O_1})(1-\beta_{O_1})\beta_{O_1}\beta_{H_3} w_{13}(1-\beta_{H_3}) x_2$$

$$\frac{\partial E}{\partial w_{10}} \quad -(y-\beta_{O_1})(1-\beta_{O_1})$$

$$\frac{\partial E}{\partial w_{11}} = -(y-\beta_{O_1})(1-\beta_{O_1})\beta_{H_1}$$

$$\frac{\partial E}{\partial w_{12}} = -(y-\beta_{O_1})(1-\beta_{O_1})\beta_{H_2}$$

$$\frac{\partial E}{\partial w_{13}} = -(y-\beta_{O_1})(1-\beta_{O_1})\beta_{H_3}$$

- The general procedure for updating weights is given by:

$$w_n^{(t+1)} = w_n^{(t)} - \frac{\partial E}{\partial w_n}$$

$$\therefore w_1^{(t+1)} = w_1 + (y-\beta_{O_1})(1-\beta_{O_1})\beta_{O_1} w_{11} \beta_{H_1}(1-\beta_{H_1})$$

$$w_2^{(t+1)} = w_2 + (y-\beta_{O_1})(1-\beta_{O_1})\beta_{O_1} w_{12}\beta_{H_2}(1-\beta_{H_2})$$

$$w_3^{(t+1)} = w_3 + (y-\beta_{O_1})(1-\beta_{O_1})\beta_{O_1} w_{13}\beta_{H_3}(1-\beta_{H_3})$$

$$w_4^{(t+1)} = w_4 + (y-\beta_{O_1})(1-\beta_{O_1})\beta_{O_1} w_{11}\beta_{H_1}(1-\beta_{H_1}) x_1$$

$$w_5^{(t+1)} = w_5 + (y-\beta_{O_1})(1-\beta_{O_1})\beta_{O_1}\beta_{H_2} w_{12}(1-\beta_{H_2}) x_1$$

$$w_6^{(t+1)} = w_6 + (y-\beta_{O_1})(1-\beta_{O_1})\beta_{O_1}\beta_{H_3} w_{13}(1-\beta_{H_3}) x_1$$

$$w_7^{(t+1)} = w_7 + (y-\beta_{O_1})(1-\beta_{O_1})\beta_{O_1}\beta_{H_1} w_{11}(1-\beta_{H_1}) x_2$$

$$w_8^{(t+1)} = w_8 + (y-\beta_{O_1})(1-\beta_{O_1})\beta_{O_1}\beta_{H_2} w_{12}(1-\beta_{H_2}) x_2$$

$$w_9^{(t+1)} = w_9 + (y-\beta_{O_1})(1-\beta_{O_1})\beta_{O_1}\beta_{H_3} w_{13}(1-\beta_{H_3}) x_2$$

$$w_{10}^{(t+1)} = w_{10} + (y-\beta_{O_1})(1-\beta_{O_1})$$

$$w_{11}^{(t+1)} = w_{11} + (y-\beta_{O_1})(1-\beta_{O_1})\beta_{H_1}$$

$$w_{12}^{(t+1)} = w_{12} + (y-\beta_{O_1})(1-\beta_{O_1})\beta_{H_1}$$

$$w_{13}^{(t+1)} = w_{13} + (y-\beta_{O_1})(1-\beta_{O_1})\beta_{H_3}$$

##Question 1

(Total 10 marks available)

In this file you will find a set of 100 observations. Each observation has some features x_1 and x_2 , and some classification of 0 or 1.

Plot the data x_1 and x_2 , coloured by the classification c . Your task is to build a neural network that will predict some classification given x_1 and x_2. Your network should have one hidden layer, with three neurons in the hidden layer. Chose an activation function. Draw the network, labelling inputs and weights. Derive the updated estimates for the weights by finding the derivatives of the loss function with respect to the weights.

```
import matplotlib.pyplot as plt
from google.colab import files
import pandas as pd
files.upload()
```

```
<IPython.core.display.HTML object>
```

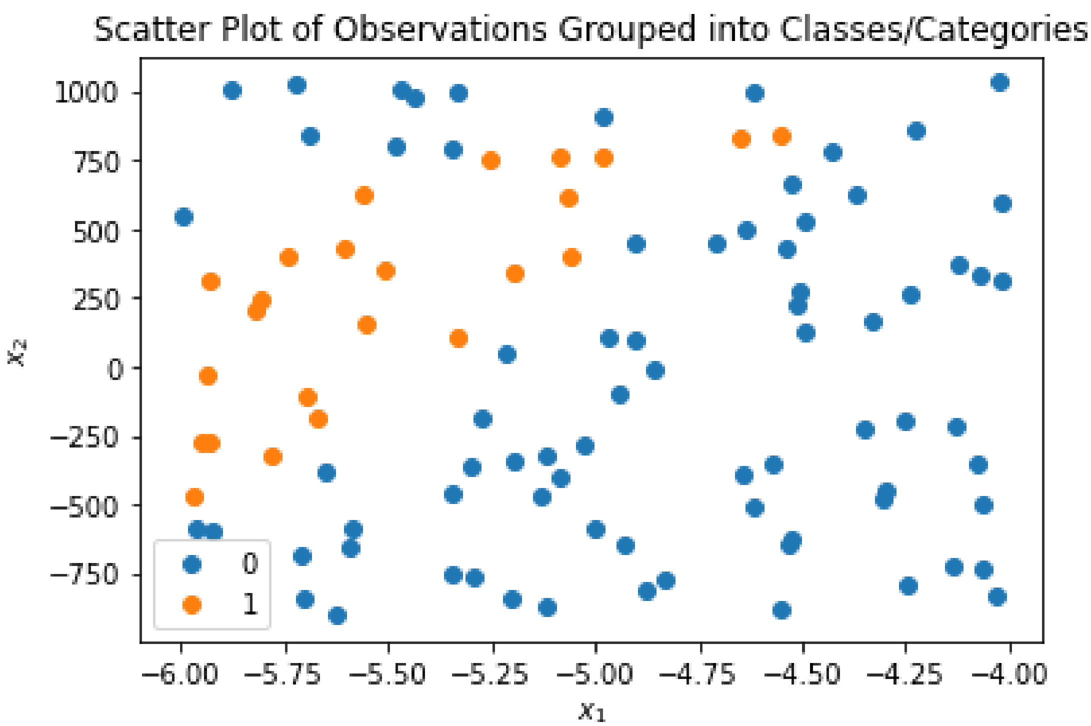
```
Saving ps5_data.csv to ps5_data.csv
```


```
{'ps5_data.csv': b'x_1,x_2,classification\n-4.031616298914846,-
828.0353067721127,0\n-5.333175467970693,999.0512341950695,0\n-
4.652596750661725,830.3911585113359,1\n-
5.607219165570951,430.65544812024,1\n-5.291107070717953,-
760.8323600496894,0\n-4.373268093635879,629.9343390641365,0\n-
5.5043008929774775,353.97154807605784,1\n-5.082777349707572,-
401.78144321647505,0\n-4.245397157005825,-793.9726658540217,0\n-
5.216683077665184,45.933618011998874,0\n-5.34773315929938,-
463.4353132718821,0\n-5.8198097795124575,206.2535808518968,1\n-
4.331564500152332,171.09637807853102,0\n-
5.807576790115939,248.7304426385823,1\n-
4.70490983580544,451.4507119834112,0\n-
4.900992023265309,447.05701718034004,0\n-4.300518916218776,-
453.04235857936993,0\n-5.469365363551335,1003.6911308161824,0\n-
5.1182633580009504,-319.010540763798,0\n-
4.238229235010239,259.8725896950924,0\n-4.5272845900935295,-
627.847386902609 4,0\n-4.574719925196659,-347.94226675209467,0\n-
5.928656062124076,313.0814864616373,1\n-5.960537542114975,-
583.9140479532721,0\n-5.344406257511974,788.2285196181317,0\n-
4.873966733270089,-809.8403199846927,0\n-
4.508565905749209,278.11543995487864,0\n-
4.540864224360201,430.566914908115,0\n-
4.019617760635869,591.9021918856932,0\n-
4.073032522873158,329.95988611759736,0\n-
4.899609330108733,98.98432088509287,0\n-
4.023476657565113,1032.2194586698774,0\n-
4.9814907682835825,908.2190670073888,0\n-5.300455125311976,-
365.69099836260426,0\n-4.1373605684340955,-719.0022716873108,0\n-
```

5.591635689354609,-656.8490054101835,0\n-
5.197666999325878,345.24827237588846,1\n-
5.055606749415959,404.15298708276976,1\n-
5.72434525486958,1022.1112679738076,0\n-
4.549594560776578,843.6149499514382,1\n-4.828526527169773,-
773.8264821438584,0\n-5.686948567956602,842.965899805774,0\n-
4.427847943565688,782.501479749105,0\n-5.669323640829482,-
185.49030739741704,1\n-4.511324872074113,228.09462270079646,0\n-
5.481492473270342,798.6741602949023,0\n-
4.49551380203754,131.05254611026913,0\n-
5.742151646659287,397.9028044031309,1\n-5.949158057130328,-
274.06298902809107,1\n-5.646837689970238,-380.6237148454568,0\n-
4.065074287589714,-494.04006457811784,0\n-
5.435372449722608,981.9325855988017,0\n-4.347398617808287,-
227.25902259426658,0\n-5.255686292175658,750.4781710264425,1\n-
5.062037099433177,620.8124474361982,1\n-5.623223043316544,-
899.8473817008104,0\n-5.694528856663204,-107.83769694404202,1\n-
5.996715127490145,552.2846933323619,0\n-
5.082758396766727,767.115519102008,1\n-5.92237406364193,-
592.8917874495801,0\n-4.533346484641491,-641.2148793932151,0\n-
4.641392234246339,-388.74615185656097,0\n-5.707813060626834,-
685.183256008237,0\n-4.9294533550525195,-648.2285214397616,0\n-
4.075654063438028,-349.5394955960529,0\n-
4.524411856220931,667.561876683346,0\n-4.126861363755881,-
218.61471702742023,0\n-5.345212908440115,-756.8817166455153,0\n-
4.302587850581708,-481.09582274309355,0\n-4.253457225024153,-
195.4401183908723,0\n-5.698531277366167,-839.846959802538,0\n-
5.9651727463791175,-468.9884916457048,1\n-5.130375005369687,-
466.3337971911666,0\n-5.193564727910921,-338.4593833641916,0\n-
5.332387487103945,110.88853974251053,1\n-5.9286742268792825,-
276.38365088440025,1\n-5.555927666167554,629.9922354296785,1\n-
5.271261054557892,-187.02422024754455,0\n-4.944696089822219,-
101.4285092368892,0\n-4.492284537516854,529.4643347889455,0\n-
4.615850836654439,-505.23278989490893,0\n-5.779586665531261,-
317.98448451823697,1\n-4.022400293538347,313.5486291033915,0\n-
4.123599101719589,375.6983937122693,0\n-4.9988799391651595,-
581.8675894636488,0\n-4.634345373514211,501.92205447295544,0\n-
5.025135604934466,-281.6738092274396,0\n-
5.553739320377479,155.32600427097518,1\n-5.202835458821014,-
844.6119144607047,0\n-4.981619495315229,761.4408868613293,1\n-
4.067089823904108,-737.0215581475882,0\n-5.586426567879273,-
586.6308953975544,0\n-4.968783992268251,107.45979888592024,0\n-
4.8554479387775995,-10.35936577604572,0\n-
4.2255810764744375,859.7583391149608,0\n-
4.619218489387048,996.5017716475588,0\n-5.116134966799282,-
866.280574312063,0\n-4.552493550495021,-878.0070720837334,0\n-
5.877481680665834,1002.7137605781413,0\n-5.933995059425942,-
25.408751028821044,1\n'}

```
data = pd.read_csv('ps5_data.csv')
data.head()
groups = data.groupby('classification')

for name, group in groups:
  plt.scatter(group['x_1'],group['x_2'], label = name)


plt.xlabel(r'$x_{1}$')
plt.ylabel(r'$x_{2}$')
plt.title("Scatter Plot of Observations Grouped into
Classes/Categories")
plt.legend(loc="best")
```

```
<matplotlib.legend.Legend at 0x7fe040f9cb50>
```



##Question 2

(Total 10 marks available)

Implement the neural network from Question 1 using numpy. Train the neural network using an appropriate portion of the data, and plot the training and test loss as a function of epoch. (You will need to make appropriate decisions regarding learning rates, initialisations, data segmentation, and number of epochs. Be sure to comment on these decisions.)

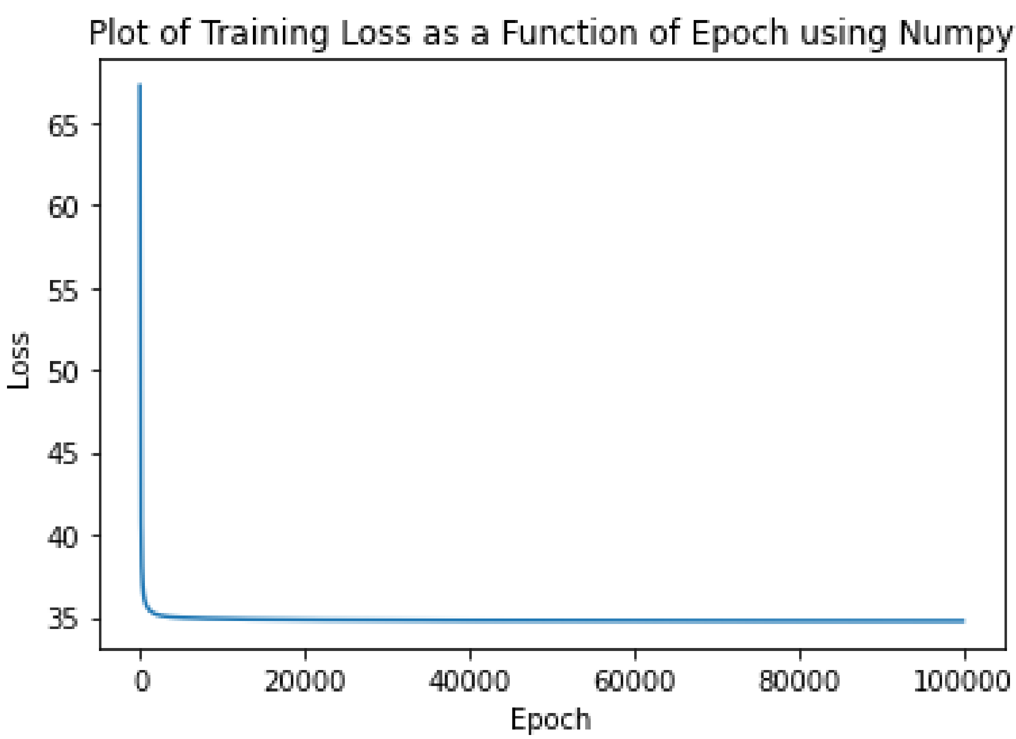
```
import numpy as np
from sklearn.model_selection import train_test_split
```

```
import matplotlib.pyplot as plt

X = np.array([data['x_1'], data['x_2']]).T

Y = np.array(data['classification']).reshape((-1, 1))

#Normalise x and y
X_mean, X_std = (np.mean(X, axis=0), np.std(X, axis=0))
x = (X - X_mean) / X_std

Y_mean, Y_std = (np.mean(Y), np.std(Y))
y = (Y - Y_mean) / Y_std

# Split data into 75-25 training-test sets using sklearn (this is the
default train_test_split ratio)
x_train, x_test, y_train, y_test = train_test_split(x, y)
#Define variables for number of inputs and outputs
N_train, D_train_in = x_train.shape
N_train, D_train_out = y_train.shape
N_test, D_test_in = x_test.shape
N_test, D_test_out = y_test.shape

H = 3 # Number of hidden layers

# Define the activation function.
sigmoid = lambda x: 1/(1 + np.exp(-x))

#Initialise weights

#Bias terms in hidden layer
w1, w2, w3 = np.random.randn(H)
# Weights for x1 to all neurons.
w4, w5, w6 = np.random.randn(H)
#Weights for x2 to all neurons.
w7, w8, w9 = np.random.randn(H)
#Weights for hidden layer outputs to output neuron
w10, w11, w12, w13 = np.random.randn(H+1)

#Training set
num_epochs = 100000
losses = np.empty(num_epochs)

eta = 1e-3 #Learning rate is initialised to attempt to b
for epoch in range(num_epochs):
  training_hidden_layer_inputs = np.hstack([
    x_train, np.ones((int(N_train), 1))
])
  training_hidden_layer_weights = np.array([
    [ w1,  w2,  w3],
```

```
    [ w4,  w5,  w6],
    [w7, w8, w9]
])

  alpha_h = training_hidden_layer_inputs @
training_hidden_layer_weights
  beta_h = sigmoid(alpha_h)

# Output layer.
  training_output_layer_inputs = np.hstack([
    beta_h,
    np.ones((N_train, 1))
])
  training_output_layer_weights = np.array([
    [w10, w11, w12, w13]
]).T

  alpha_o = training_output_layer_inputs @
training_output_layer_weights
  beta_o = sigmoid(alpha_o)
  y_pred = beta_o
# Calculate our loss function: the total error in our predictions
# compared to the target.

  loss = 0.5 * np.sum((y_pred - y_train)**2)

  losses[epoch] = loss


  # Calculate gradients
  s = (beta_o - y_train) * beta_o * (1 - beta_o)

  #Compute gradients
  dE_dw13 = s * beta_h[:, [2]]
  dE_dw12 = s * beta_h[:, [1]]
  dE_dw11 = s * beta_h[:, [0]]
  dE_dw10 = s
  dE_dw9  = s * w13 * beta_h[:, [2]] * (1 - beta_h[:, [2]]) *
x_train[:, [1]]
  dE_dw8  = s * w12 * beta_h[:, [1]] * (1 - beta_h[:, [1]]) *
x_train[:, [1]]
  dE_dw7  = s * w11 * beta_h[:, [0]] * (1 - beta_h[:, [0]]) *
x_train[:, [1]]
  dE_dw6  = s * w13 * beta_h[:, [2]] * (1 - beta_h[:, [2]])*
x_train[:, [0]]
  dE_dw5  = s * w12 * beta_h[:, [1]] * (1 - beta_h[:, [1]])*
x_train[:, [0]]
  dE_dw4  = s * w11 * beta_h[:, [0]] * (1 - beta_h[:, [0]])*
x_train[:, [0]]
```

```
  dE_dw3  = s * w13 * beta_h[:, [2]] * (1 - beta_h[:, [2]])
  dE_dw2  = s * w12 * beta_h[:, [1]] * (1 - beta_h[:, [1]])
  dE_dw1  = s * w11 * beta_h[:, [0]] * (1 - beta_h[:, [0]])

# Now update the weights using stochastic gradient descent.
  w1 = w1 - eta * np.sum(dE_dw1)
  w2 = w2 - eta * np.sum(dE_dw2)
  w3 = w3 - eta * np.sum(dE_dw3)
  w4 = w4 - eta * np.sum(dE_dw4)
  w5 = w5 - eta * np.sum(dE_dw5)
  w6 = w6 - eta * np.sum(dE_dw6)
  w7 = w7 - eta * np.sum(dE_dw7)
  w8 = w8 - eta * np.sum(dE_dw8)
  w9 = w9 - eta * np.sum(dE_dw9)
  w10 = w10 - eta * np.sum(dE_dw10)
  w11 = w11 - eta * np.sum(dE_dw11)
  w12 = w12 - eta * np.sum(dE_dw12)
  w13 = w13 - eta * np.sum(dE_dw13)

plt.figure()
plt.plot(losses)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Plot of Training Loss as a Function of Epoch using Numpy')
```

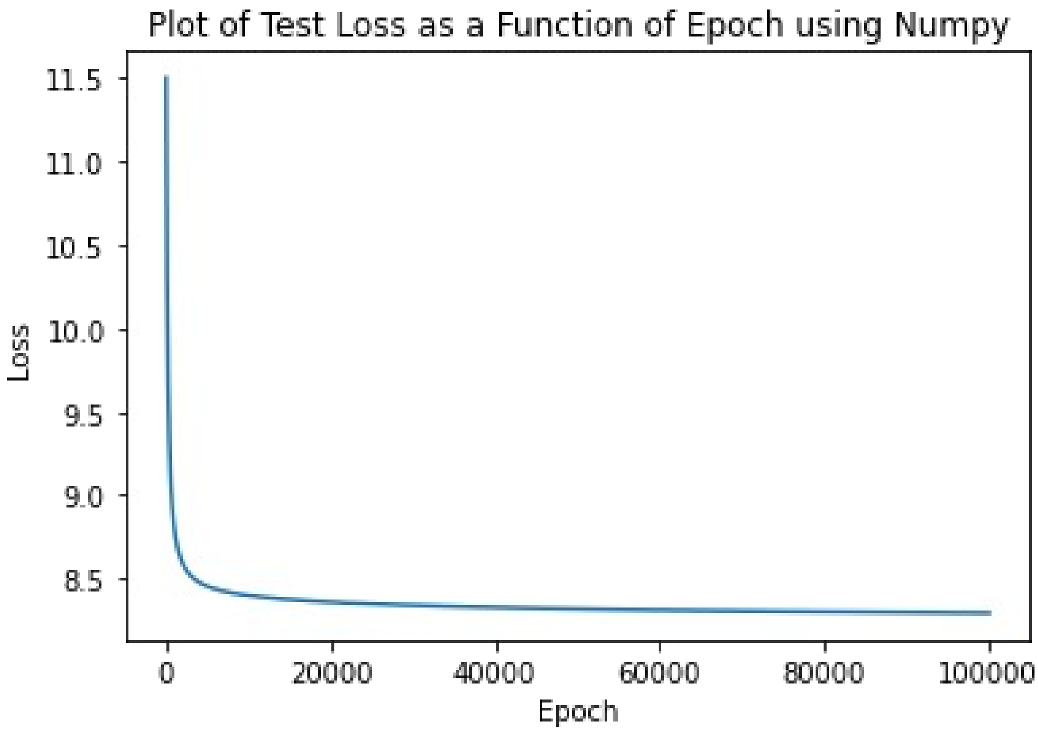
```
Text(0.5, 1.0, 'Plot of Training Loss as a Function of Epoch using
Numpy')
```



```
#Test Loss
num_epochs = 100000
losses = np.empty(num_epochs)
#Initialise weights

#Bias terms in hidden layer
w1, w2, w3 = np.random.randn(H)
# Weights for x1 to all neurons.
w4, w5, w6 = np.random.randn(H)
#Weights for x2 to all neurons.
w7, w8, w9 = np.random.randn(H)
#Weights for hidden layer outputs to output neuron
w10, w11, w12, w13 = np.random.randn(H+1)

eta = 1e-3
for epoch in range(num_epochs):
  test_hidden_layer_inputs = np.hstack([
    x_test, np.ones((int(N_test), 1))
])
  test_hidden_layer_weights = np.array([
    [ w1,  w2,  w3],
    [ w4,  w5,  w6],
    [w7, w8, w9]
])

  alpha_h = test_hidden_layer_inputs @ test_hidden_layer_weights
  beta_h = sigmoid(alpha_h)

# Output layer.
  test_output_layer_inputs = np.hstack([
    beta_h,
    np.ones((N_test, 1))
])
  test_output_layer_weights = np.array([
    [w10, w11, w12, w13]
]).T

  alpha_o = test_output_layer_inputs @ test_output_layer_weights
  beta_o = sigmoid(alpha_o)
  y_pred = beta_o
# Calculate our loss function: the total error in our predictions
# compared to the target.

  loss = 0.5 * np.sum((y_pred - y_test)**2)

  losses[epoch] = loss


  # Calculate gradients
```

```
  s = (beta_o - y_test) * beta_o * (1 - beta_o)

  #Compute gradients
  dE_dw13 = s * beta_h[:, [2]]
  dE_dw12 = s * beta_h[:, [1]]
  dE_dw11 = s * beta_h[:, [0]]
  dE_dw10 = s
  dE_dw9  = s * w13 * beta_h[:, [2]] * (1 - beta_h[:, [2]]) *
x_test[:, [1]]
  dE_dw8  = s * w12 * beta_h[:, [1]] * (1 - beta_h[:, [1]]) *
x_test[:, [1]]
  dE_dw7  = s * w11 * beta_h[:, [0]] * (1 - beta_h[:, [0]]) *
x_test[:, [1]]
  dE_dw6  = s * w13 * beta_h[:, [2]] * (1 - beta_h[:, [2]])* x_test[:,
[0]]
  dE_dw5  = s * w12 * beta_h[:, [1]] * (1 - beta_h[:, [1]])* x_test[:,
[0]]
  dE_dw4  = s * w11 * beta_h[:, [0]] * (1 - beta_h[:, [0]])* x_test[:,
[0]]
  dE_dw3  = s * w13 * beta_h[:, [2]] * (1 - beta_h[:, [2]])
  dE_dw2  = s * w12 * beta_h[:, [1]] * (1 - beta_h[:, [1]])
  dE_dw1  = s * w11 * beta_h[:, [0]] * (1 - beta_h[:, [0]])

# Now update the weights using stochastic gradient descent.
  w1 = w1 - eta * np.sum(dE_dw1)
  w2 = w2 - eta * np.sum(dE_dw2)
  w3 = w3 - eta * np.sum(dE_dw3)
  w4 = w4 - eta * np.sum(dE_dw4)
  w5 = w5 - eta * np.sum(dE_dw5)
  w6 = w6 - eta * np.sum(dE_dw6)
  w7 = w7 - eta * np.sum(dE_dw7)
  w8 = w8 - eta * np.sum(dE_dw8)
  w9 = w9 - eta * np.sum(dE_dw9)
  w10 = w10 - eta * np.sum(dE_dw10)
  w11 = w11 - eta * np.sum(dE_dw11)
  w12 = w12 - eta * np.sum(dE_dw12)
  w13 = w13 - eta * np.sum(dE_dw13)

plt.figure()
plt.plot(losses)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Plot of Test Loss as a Function of Epoch using Numpy')
```

```
Text(0.5, 1.0, 'Plot of Test Loss as a Function of Epoch using Numpy')
```



Question 3

(Total 10 marks available)

Implement the neural network from Question 1 using a neural network packaage of your choice (e.g., keras/TensorFlow, PyTorch). Make the same plot as you did for Question 2.

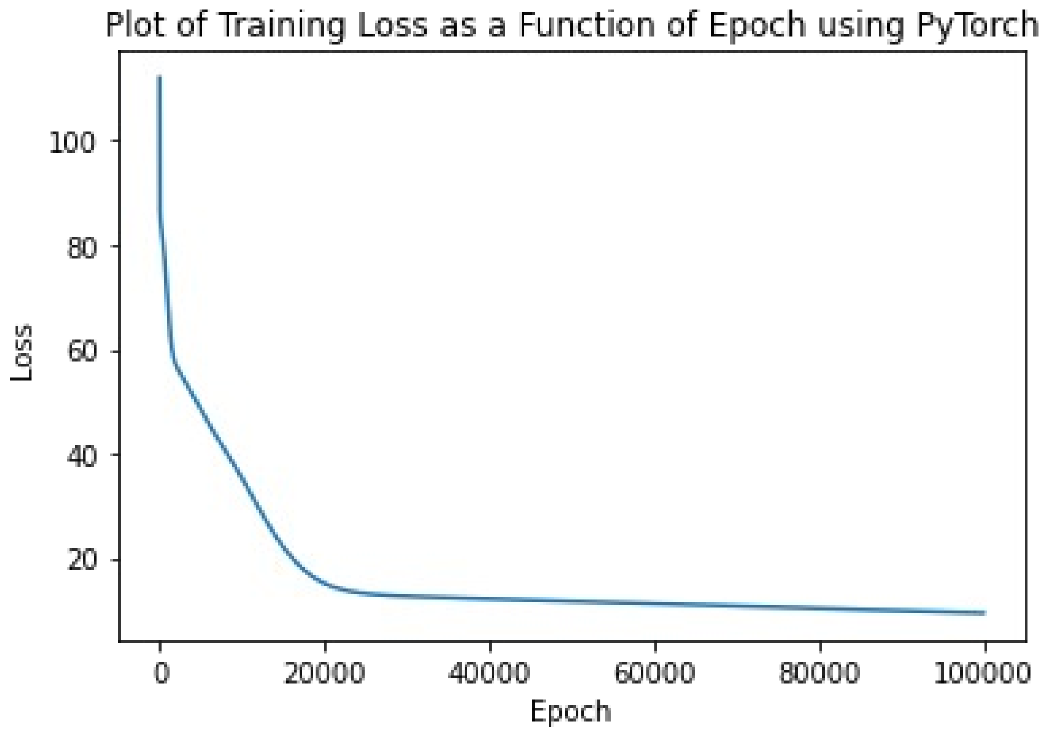
```
import torch
Y = np.atleast_2d(Y)
print(Y.shape)

Y_mean, Y_std = (np.mean(Y), np.std(Y))
y = (Y - Y_mean) / Y_std

# Normalise X -- we will explain why we do this in later classes.
X_mean, X_std = (np.mean(X, axis=0), np.std(X, axis=0))
x = (X - X_mean) / X_std

N_train, D_train_in = x_train.shape
N, D_train_out = y_train.shape

H = 3

x_train = torch.tensor(x_train, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.float32)

train_model = torch.nn.Sequential(
```

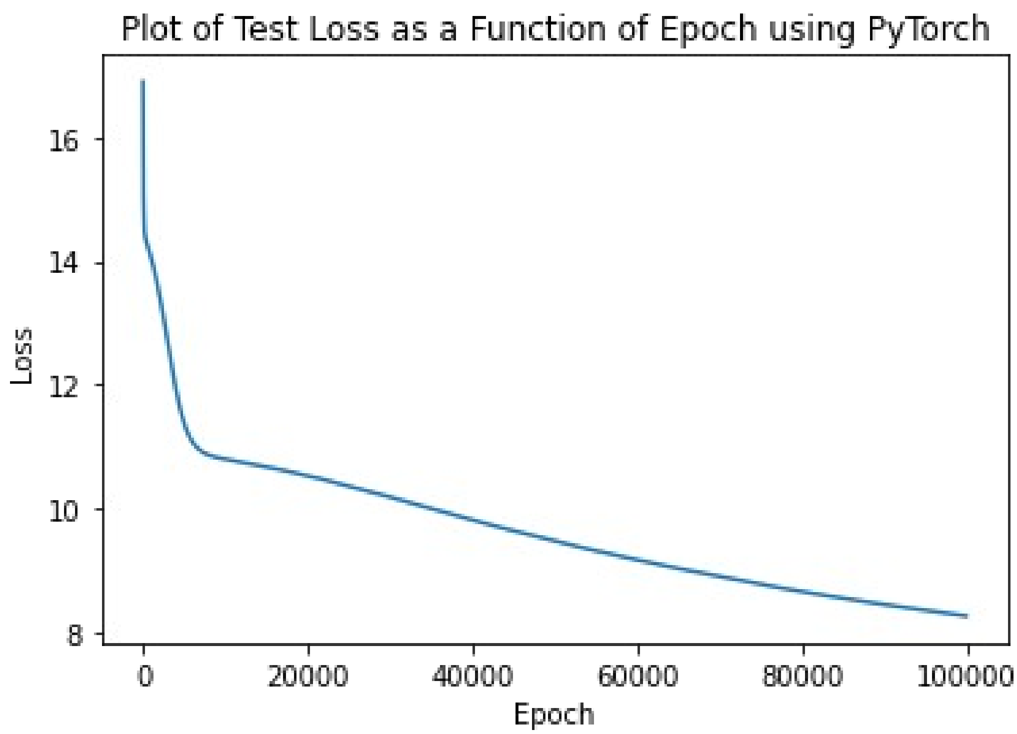
```
    torch.nn.Linear(D_train_in, H),
    torch.nn.Sigmoid(),
    torch.nn.Linear(H, D_train_out),
)

loss_fn = torch.nn.MSELoss(reduction='sum')

epochs = 100000
learning_rate = 1e-4

losses = np.empty(epochs)
for t in range(epochs):
    # Forward pass.
    y_pred = train_model(x_train)

    # Compute loss.
    loss_val = loss_fn(y_pred, y_train)
    losses[t] = loss_val.item()

    # Zero the gradients before running the backward pass.
    train_model.zero_grad()

    # Backward pass.
    loss_val.backward()

    # Update the weights using gradient descent.
    with torch.no_grad():
        for param in train_model.parameters():
            param -= learning_rate * param.grad



plt.figure()
plt.plot(losses)
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Plot of Training Loss as a Function of Epoch using
PyTorch")


# Test loss
x_test = torch.tensor(x_test, dtype=torch.float32)
y_test = torch.tensor(y_test, dtype=torch.float32)

test_model = torch.nn.Sequential(
    torch.nn.Linear(D_test_in, H),
    torch.nn.Sigmoid(),
    torch.nn.Linear(H, D_test_out),
)
```

```python
loss_fn = torch.nn.MSELoss(reduction='sum')

epochs = 100000
learning_rate = 1e-4

test_losses = np.empty(epochs)
for t in range(epochs):
    # Forward pass.
    y_pred = test_model(x_test)

    # Compute loss.
    loss_val = loss_fn(y_pred, y_test)

    test_losses[t] = loss_val.item()

    # Zero the gradients before running the backward pass.
    test_model.zero_grad()

    # Backward pass.
    loss_val.backward()

    # Update the weights using gradient descent.
    with torch.no_grad():
        for new_param in test_model.parameters():
            new_param -= learning_rate * new_param.grad


plt.figure()
plt.plot(test_losses)
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Plot of Test Loss as a Function of Epoch using PyTorch")
```

```
(100, 1)
```

```
Text(0.5, 1.0, 'Plot of Test Loss as a Function of Epoch using
PyTorch')
```





##Question 4

(Total 10 marks available)

A colleague is trying to train a neural network to make some classifications of objects. They have tried many different choices of initialisation, data splits, and network architecture, and they have encountered different problems for each choice they have made. For each of the situations below, the colleague has asked your opinion on what might be causing the problem, and what to do about it.

1. The training loss is decreasing with epoch but the test loss is unchanged, maybe even increasing.
2. The training loss is decreasing, but so slowly that it is going to take a lifetime to train!
3. The training and test loss is increasing!
4. I've run for 1,000 epochs and the training loss is exactly the same as it was in the first epoch.
5. The training loss has decreased, and the test loss has decreased, but the test loss has not decreased as much as the training loss. Is there something I could do to improve the network predictions in a generalised way so it performs better on unseen data?

The following measures can be adopted to address the aforementioned issues:

1. Since the training loss is decreasing with epoch but the test loss is unchanged, it is evident that the model performs well on the training data, but is not as effective in classifying data from the test set. This implies that the model has overfit the data. Three possible measures that can be adopted to address the issue are:

- Appropriate data pre-processing (i.e. standardising and normalising the data prior to implementing the model).
- Lowering the learning rate
- Introducing dropout to reduce the model complexity if required (this can ensure that the gradient and the outputs can be noisier)

1. The likely cause of slow decrease in the training loss is the incorrect initialisation of the weights. If the weights are initialised to be too high, this can result in the saturation of neurons at deeper layers in the network and can therefore significantly increase computational time. On the other hand, if the weights are initialised to be too low, this can cause the weights of the deeper layers of the network to become vanishingly small. This effect can flow back through the network and influence all of the neurons in the network. In both cases, the weights are unable to be updated due to the lack of sufficiently large gradients. In order to address this issue, one can ensure that for small networks that are zero-centred and have unit variance, the weights are initialised as

$$w \sim N(0,1)$$

2. In the case of deeper networks, one can scale the width of the normal distribution relative to the number of inputs for each neuron. For a layer of neurons with W_{in} inputs to the layer, one can initialise the weights as:

$$w \sim N\left(0, \frac{1}{\sqrt{W_{in}}}\right)$$

3. The training and validation losses increasing could either be a result of the incorrect method employed to update the weights of the network or the incorrect computation of the derivatives of the loss function with respect to the weights, which contribute directly to incorrect updated values for the weights. In order to mitigate this issue, one can implement tools like check_grad to ensure that the derivatives have been computed correctly. Where possible, one must refrain from computing these derivatives manually, and must resort to computational methods instead

4. The constant nature of the training loss after 1000 epochs could be a result of the negligible contributions made to the update of the weights of each neuron, which could be caused by a low value of the learning rate. A possible solution to mitigate the effect of this error is to increase the learning rate such that the updates to the weights are more significant

5. The higher value of the test loss, compared to the training loss could be a result of overfitting the model to the data. This could have been caused by the presence and absence of dropout in the training and validation/test sets respectively. It could also be the result of a smaller validation set, in which case one can increase the size by altering the training-test set ratio. One could also introduce regularisation in the form of dropout to the validation set to address this issue