ECE457B - Assignment 1

Sheen Thusoo | 20728766

Question 1

a.

Using the chain differentiation rule, the weight update formula can be derived as follows:

$$\Delta w_i = -\eta \frac{\partial E(w)}{\partial w_i}$$

$$= -\eta \frac{1}{2}(2)(-1)\left(t^{(k)} - s\left(\Sigma w_i x_i^{(k)}\right)\right) \cdot \frac{\partial}{\partial(\Sigma w_i x_i^{(k)})} s\left(\Sigma w_i x_i^{(k)}\right) \cdot \frac{\partial}{\partial w_i}\left(\Sigma w_i x_i^{(k)}\right)$$

Find $\frac{ds}{d(net)}$

$$\frac{ds}{d(net)} = (-1)\left(1 + e^{-net}\right)^{-2}\left(e^{-net}\right)(-1)$$

$$\frac{\partial s}{\partial net} = \frac{e^{-net}}{\left(1 + e^{-net}\right)^2}$$

$$\frac{ds}{d(net)} = s^2(net) \cdot e^{-net}$$

Substitute $net = \Sigma w_i x_i^{(k)}$

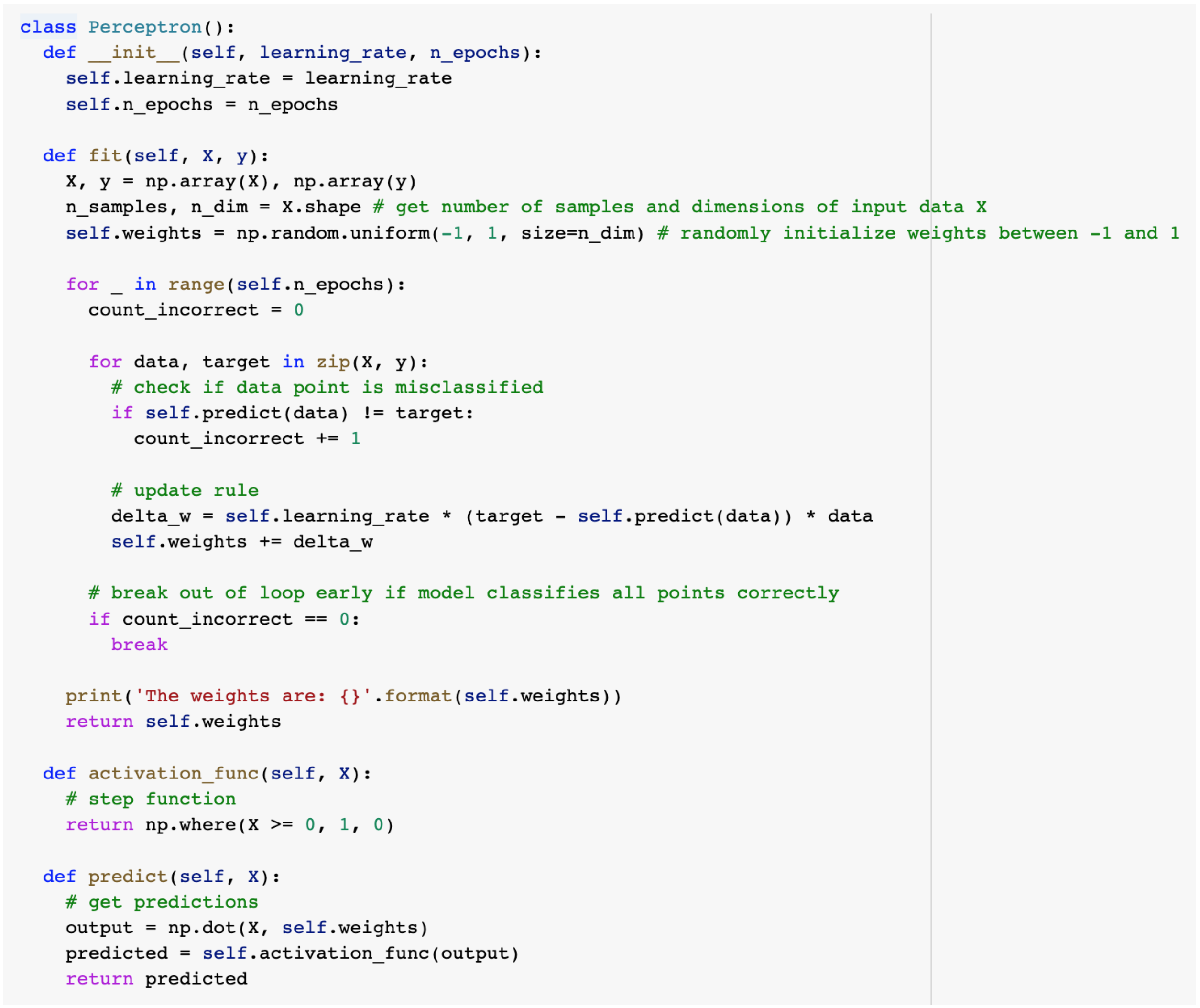
$$\Delta w_i = \eta\left(t^{(k)} - s\left(\Sigma w_i x_i^{(k)}\right)\right) \cdot \left(s^2\left(\Sigma w_i x_i^{(k)}\right) \cdot e^{-\left(\Sigma w_i x_i^{(k)}\right)}\right) \cdot x_i^{(k)}$$

Therefore,

$$\Delta w_i = \eta(t^{(k)} - s)\left(s^2 \cdot e^{-\left(\Sigma w_i x_i^{(k)}\right)}\right) x_i^{(k)}$$

b.

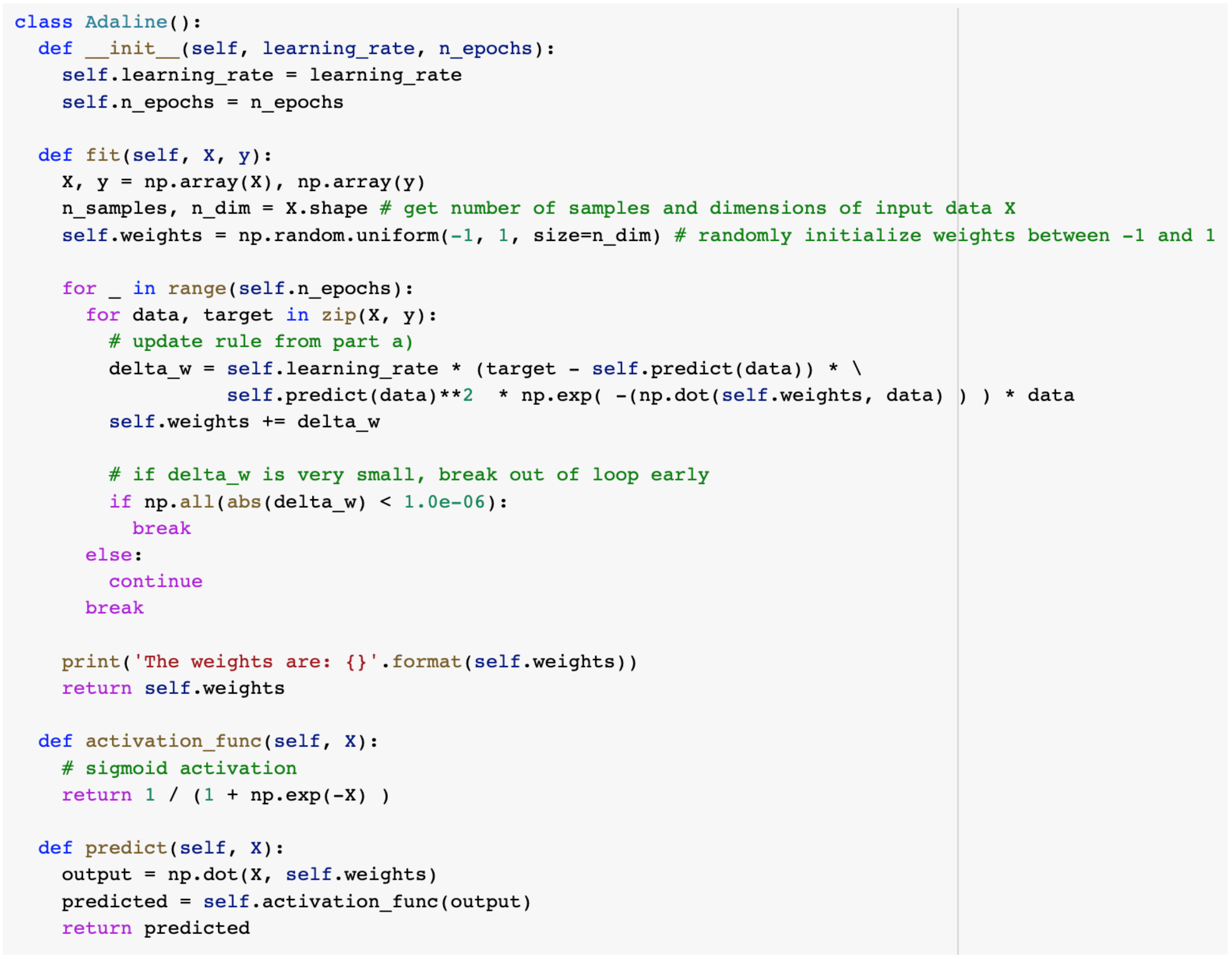
The Perceptron code is shown in Figure 1. The weights were initialized between -1 and 1 and the weight update rule is implemented in the *fit* method. Here, the weights are updated until the Perceptron correctly classifies all inputs; if it is unable to classify all points correctly, it runs for a certain number of epochs as specified by the user by the *n_epochs* parameter. The activation function used by the Perceptron is the step function which is implemented in the *activation_func* method.



```python
class Perceptron():
  def __init__(self, learning_rate, n_epochs):
    self.learning_rate = learning_rate
    self.n_epochs = n_epochs

  def fit(self, X, y):
    X, y = np.array(X), np.array(y)
    n_samples, n_dim = X.shape # get number of samples and dimensions of input data X
    self.weights = np.random.uniform(-1, 1, size=n_dim) # randomly initialize weights between -1 and 1

    for _ in range(self.n_epochs):
      count_incorrect = 0

      for data, target in zip(X, y):
        # check if data point is misclassified
        if self.predict(data) != target:
          count_incorrect += 1

        # update rule
        delta_w = self.learning_rate * (target - self.predict(data)) * data
        self.weights += delta_w

      # break out of loop early if model classifies all points correctly
      if count_incorrect == 0:
        break

    print('The weights are: {}'.format(self.weights))
    return self.weights

  def activation_func(self, X):
    # step function
    return np.where(X >= 0, 1, 0)

  def predict(self, X):
    # get predictions
    output = np.dot(X, self.weights)
    predicted = self.activation_func(output)
    return predicted
```

Figure 1: Perceptron Model Code

The Adaline code is shown in Figure 2. The weights were initialized between -1 and 1 and the weight update rule is implemented in the *fit* method. The update rule that is implemented is the one that was derived in part a) of this question. Here, the weights are updated until the change in weights is less than a threshold amount (set as 1^{-6}); it runs for a certain number of maximum epochs as specified by the user with the *n_epochs* parameter. The activation function used by the Perceptron is the sigmoid function which is implemented in the *activation_func* method.



```python
class Adaline():
  def __init__(self, learning_rate, n_epochs):
    self.learning_rate = learning_rate
    self.n_epochs = n_epochs

  def fit(self, X, y):
    X, y = np.array(X), np.array(y)
    n_samples, n_dim = X.shape # get number of samples and dimensions of input data X
    self.weights = np.random.uniform(-1, 1, size=n_dim) # randomly initialize weights between -1 and 1

    for _ in range(self.n_epochs):
      for data, target in zip(X, y):
        # update rule from part a)
        delta_w = self.learning_rate * (target - self.predict(data)) * \
                  self.predict(data)**2  * np.exp( -(np.dot(self.weights, data) ) ) * data
        self.weights += delta_w

        # if delta_w is very small, break out of loop early
        if np.all(abs(delta_w) < 1.0e-06):
          break
      else:
        continue
      break

    print('The weights are: {}'.format(self.weights))
    return self.weights

  def activation_func(self, X):
    # sigmoid activation
    return 1 / (1 + np.exp(-X) )

  def predict(self, X):
    output = np.dot(X, self.weights)
    predicted = self.activation_func(output)
    return predicted
```

Figure 2: Adaline Model Code

c.

Using a learning rate of 0.6, Figure 3 and 4 below show the data points and the boundaries created by the Perceptron and Adaline models, respectively.
The equation of the Perceptron boundary is

$$z = \frac{1.102 - 0.642x + 1.402y}{0.242}$$

The equation of the Adaline boundary is

$$z = \frac{2.525 - 1.991x + 3.244y}{-0.304}$$

The numbers in these equations have been rounded to 3 decimal places.

```
The equation of the boundary for Perceptron is:
 (1.1022451374001094 - 0.6420446834270908*x - (-1.4018374949412433)*y) / 0.24155140753283533
```

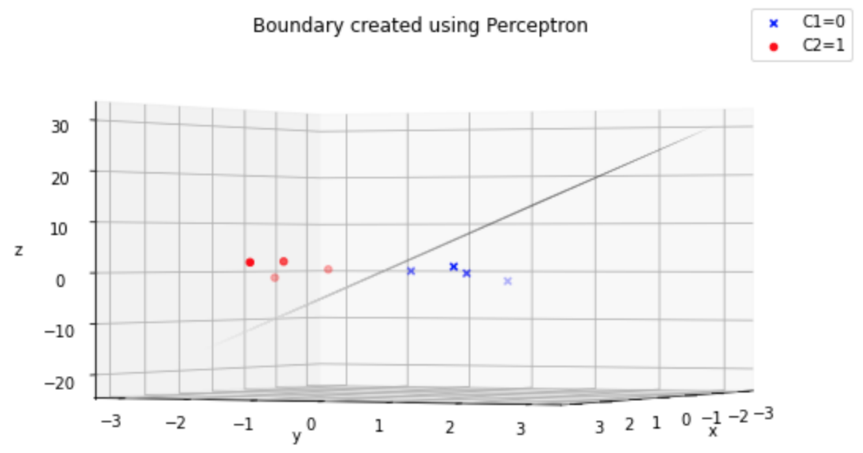


Figure 3: Perceptron Boundary and its equation

```
The equation of the boundary for Adaline is:
 (2.5246887268795954 - 1.9912202870684885*x - (-3.24367603276392)*y) / -0.3038148849291844
```

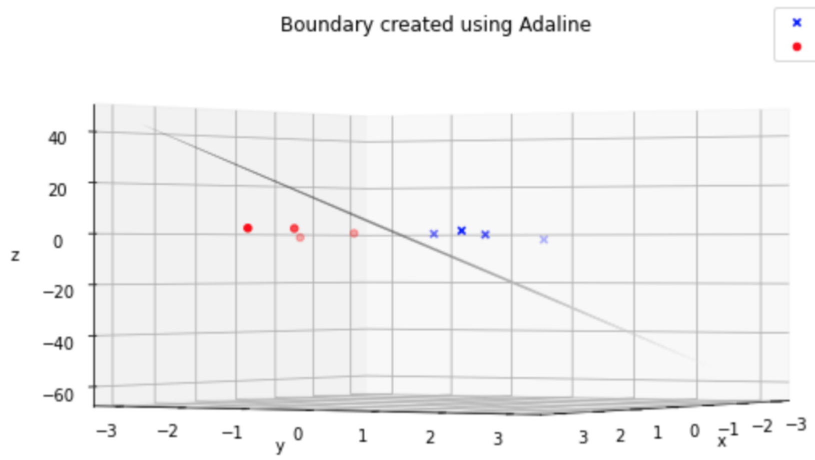


Figure 4: Adaline Boundary and its equation

d.

The new data point along with the boundaries for Perceptron and Adaline are shown in Figure 5 and 6, respectively.

```
The equation of the boundary for Perceptron is:
 (1.1022451374001094 - 0.6420446834270908*x - (-1.4018374949412433)*y) / 0.24155140753283533
```

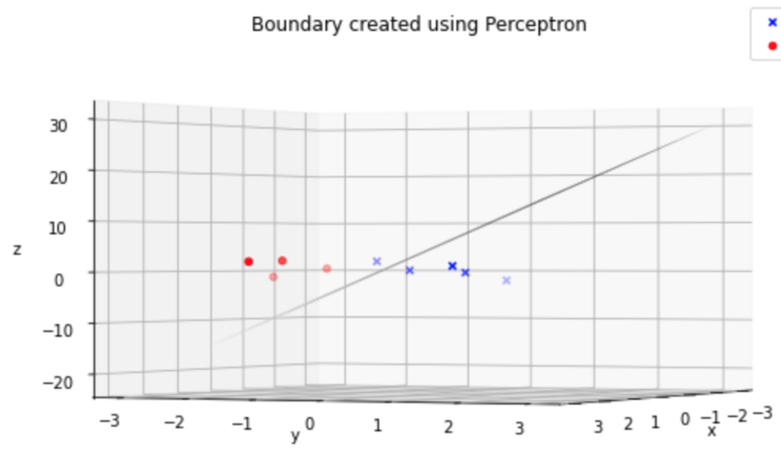


Figure 5: Perceptron Boundary with New Data Point

```
The equation of the boundary for Adaline is:
 (2.5246887268795954 - 1.9912202870684885*x - (-3.24367603276392)*y) / -0.3038148849291844
```

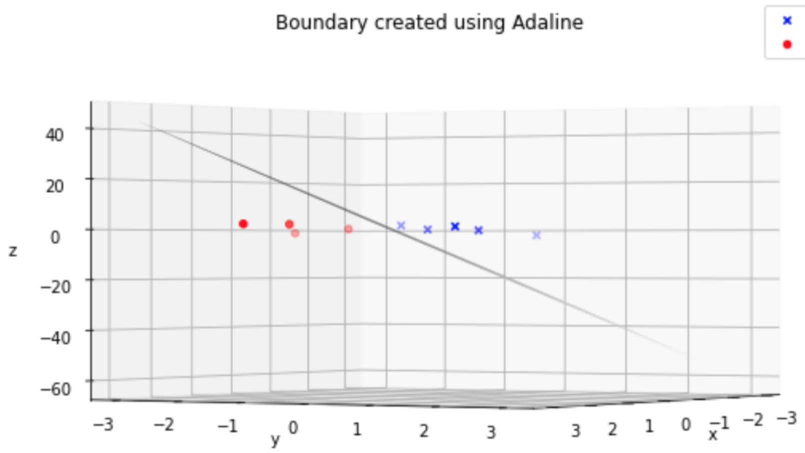


Figure 6: Adaline Boundary with New Data Point

From the above figures, it is clear that the Perceptron boundary is not valid for the new data point; however, the Adaline boundary is valid for the new data point.

e.

The Adaline structure with LMS learning has better learning capabilities than the Perceptron with Hebbian learning. This is due to the fact that Adaline is more suitable for generalization which is a missing feature for the Perceptron. Adaline uses a continuous activation function to optimize weights; for example, the sigmoid function. On the other hand, the Perceptron uses a step function which outputs discrete values. The perceptron lacks generalization as once it is trained, it cannot adapt its weights on a new set of data. Since the Adaline structure uses continuous predicted values, it is more powerful in weight optimization since these values tell us more information about the amount we were correct or incorrect in classifying the data.

Question 2

Table 1: XNOR Table

x_1	x_2	y
-1	-1	1
-1	1	-1
1	-1	-1
1	1	1

The dividing line for a neuron is:

$$x_2 = -\frac{w_1}{w_2}x_1 - \frac{w_0}{w_2} \tag{1}$$

The madaline structure given in the lectures was updated in order to solve the XNOR problem.

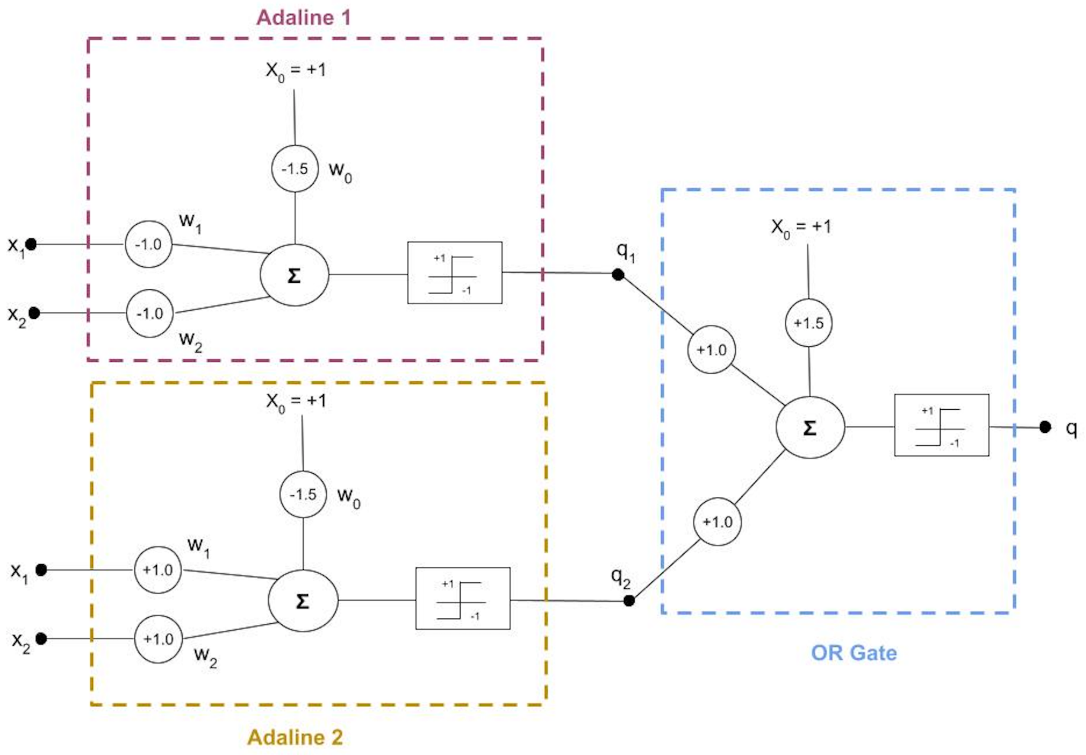


Figure 7: Madaline Structure for XNOR

The structure shown in Figure 7 uses two Adaline structures connected by an OR gate in order to solve the XNOR problem.

For Adaline 1,

$$w_0 = -1.5 \qquad w_1 = +1 \qquad w_2 = +1$$

For Adaline 2,

$$w_0 = -1.5 \qquad w_1 = -1 \qquad w_2 = -1$$

Once the weights were determined, the compounded boundaries for each Adaline structure were computed according to Equation (1) described above.

$$\text{Adaline 1:} \qquad x_2 = -\frac{-1}{-1}x_1 - \frac{-1.5}{-1}$$

$$x_2 = -x_1 - 1.5$$

$$\text{Adaline 2:} \qquad x_2 = -\frac{1}{1}x_1 - \frac{-1.5}{1}$$

$$x_2 = -x_1 + 1.5$$

These compounded boundaries are shown in Figure 8.

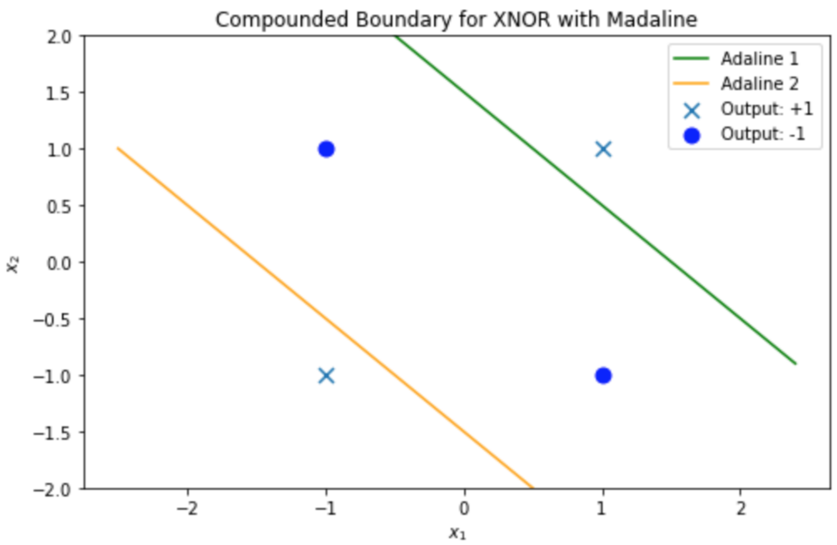


Figure 8: Boundaries for XNOR Madaline Structure

```python
from matplotlib import pyplot as plt
x = np.arange(-2.5, 2.5, 0.1)
x1_class1 = [1, -1]
x2_class1 = [1, -1]
x1_class2 = [1, -1]
x2_class2 = [-1, 1]
fig, ax = plt.subplots(figsize=(8,5))
ax.scatter(x1_class1, x2_class1, s=80, marker='x', label='Output: +1')
ax.scatter(x1_class2, x2_class2, s=80, marker="o", c='b', label='Output: -1')
ax.plot(x, -x + 1.5, color="green", label='Adaline 1')
ax.plot(x, -x - 1.5, color="orange", label='Adaline 2')
ax.set_title('Compounded Boundary for XNOR with Madaline')
ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')
ax.set_ylim(-2, 2)
ax.legend(loc='best')
```

Figure 9: Code for Question 2

Question 3

a.

Tables 2 and 3 were generated by varying the number of data points and number of nodes given to the model. A logistic activation function, specifically the sigmoid function, was used to investigate the output of the neural networks for the f_1 and f_2 function mappings. The tables below were generated by using 10-fold cross validation and monitoring both the training and validation losses for each model.

Table 2: Training and Validation Errors For Function 1

		Number of Data Points							
		10		**40**		**80**		**200**	
		Training Error	**Validation Error**	**Training Error**	**Validation Error**	**Training Error**	**Validation Error**	**Training Error**	**Validation Error**
Number of Nodes	**2**	0.11213658838532865	0.06183519973822058	0.10542855612933637	0.06130300795659423	0.0738526163250208	0.05759875166229904	0.057600513249635686	0.057650212459266194
	10	0.15307581112720073	0.09457300616741122	0.10079435274004936	0.07133217110764235	0.05977086406201124	0.06120596615597605	0.05680844899266958	0.0559660119190812
	40	0.04771740288706496	0.06235041379121355	0.07926708873361349	0.07985121527686716	0.03913203554227949	0.04011564844753594	0.0272062228154391	0.02969920517876744
	100	0.0634592538466677	0.14551531430097384	0.06138932347297669	0.06908168079331517	0.029309335525613277	0.03727777183288709	0.005285104999784381	0.007516138569044414

Table 3: Training and Validation Errors For Function 2

		Number of Data Points							
		10		**40**		**80**		**200**	
		Training Error	**Validation Error**	**Training Error**	**Validation Error**	**Training Error**	**Validation Error**	**Training Error**	**Validation Error**
Number of Nodes	**2**	0.09596340414136648	0.05410206258184334	0.05854585926979781	0.03868237202172168	0.02358982866629958	0.02399927708320319	0.031265896186232565	0.03007534757256508
	10	0.1154511916305637	0.02692908646135379	0.04241482459008693	0.034872349770303124	0.04439171344041824	0.04285637675784528	0.02413396706804633	0.02298824353143573
	40	0.05324907876392899	0.09436564803268084	0.04000010330229998	0.04763300196267664	0.022749114078469575	0.02702860444551334	0.008739127105800434	0.010384711789665744
	100	0.05854585926979781	0.03868237202172168	0.024826127556152645	0.03312191363152124	0.014181707855896095	0.01892680615419522	0.002177379974309588	0.0021014462801394983

From Table 2, it was found that the model with the lowest training and validation losses for function 1 used 200 data points and 100 hidden nodes. This model had a training loss of approximately 0.0053 and a validation loss of approximately 0.0075.

From Table 3, it was found that the model with the lowest training and validation losses for function 2 used 200 data points and 100 hidden nodes as well. This model had a training loss of about 0.0022 and validation loss of about 0.0021.

b.

The best model for each function must balance bias and variance; it must be complex enough to capture the function, but also avoid overfitting to the training data. The best model for both function 1 and function 2 was found to be the model with 200 data points and 100 hidden nodes. This model best approximated the functions as it had the lowest training/validation losses as found in part a). Thus, it was able to capture the function well. Further, this model did not overfit the training data. From the tables in part a), it is seen that the training and validation losses are very similar. If overfitting were to occur, the training loss would be low and the validation loss would be much higher. After applying the full training data to this model, the original function and the best model's predictions on the testing data were plotted. These results for function 1 are displayed in Figure 10 and for function 2 are displayed in Figure 11. The red circles represent the model's predictions on testing data. As seen in these graphs, the model is able to capture the complexity of both the functions very closely.

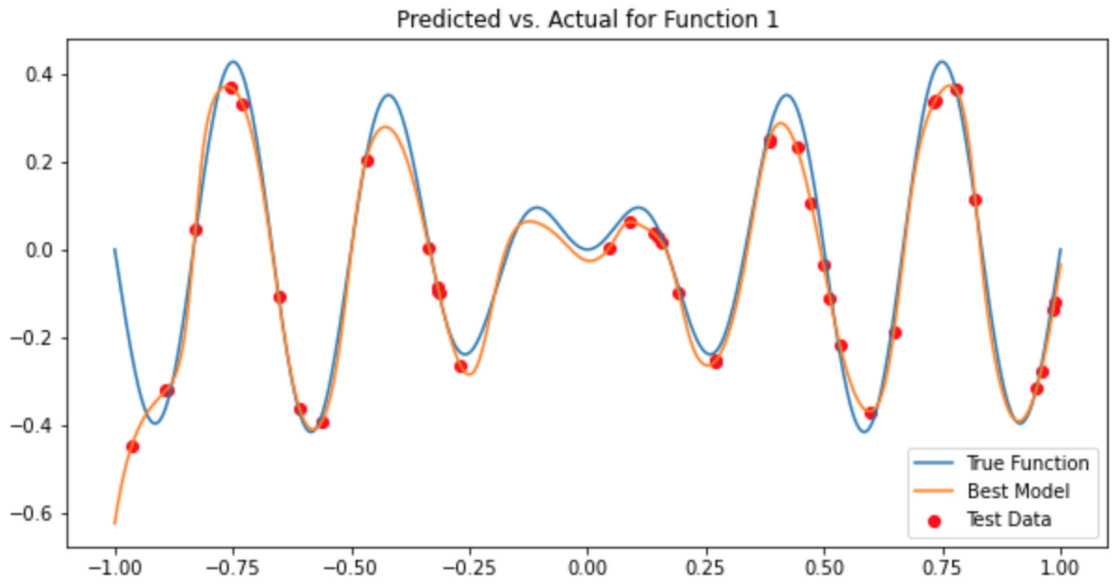


Figure 10: True Function and Model Predictions for Function 1

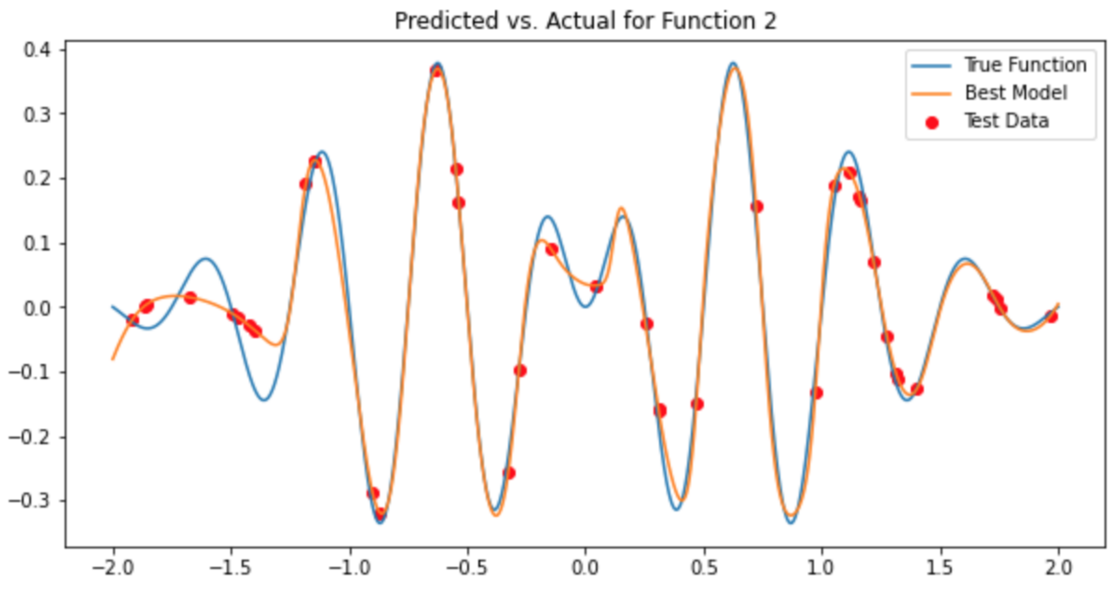


Figure 11: True Function and Model Predictions for Function 2

c.

Referencing Appendix A, it is clear that in order for a model to generalize well, it must account for a bias-variance tradeoff. This allows the model to maintain a good balance where it does not underfit or overfit the data. Underfitting occurs when both training and test data have high prediction errors; in this case, the model has a high bias and low variance. Overfitting occurs when the training data has low prediction error but the test data has a high prediction error. This indicates that the model has a high variance and low bias.

The model loss for function 1 is plotted in Figure 12 and for function 2 in Figure 13. From these graphs, it is clear that both the training and validation/testing losses decrease equally as the model runs over its epochs. If overfitting occurs, the validation/testing loss would be much higher than the training loss. This does not occur in Figure 12 or Figure 13, thus it is clear that the model does not overfit the data. From Figure 10 and 11 in part b), it can be seen that the model is able to predict the functions very closely. The predicted functions have the similar peaks and valleys as the true functions. This shows that the model is complex enough to capture the complexity of the functions; the model does not have a high bias. However, the model does not exactly fit each function, this proves that overfitting does not occur and thus it does not have a high variance. Therefore, this model maintains a good balance of bias and variance.

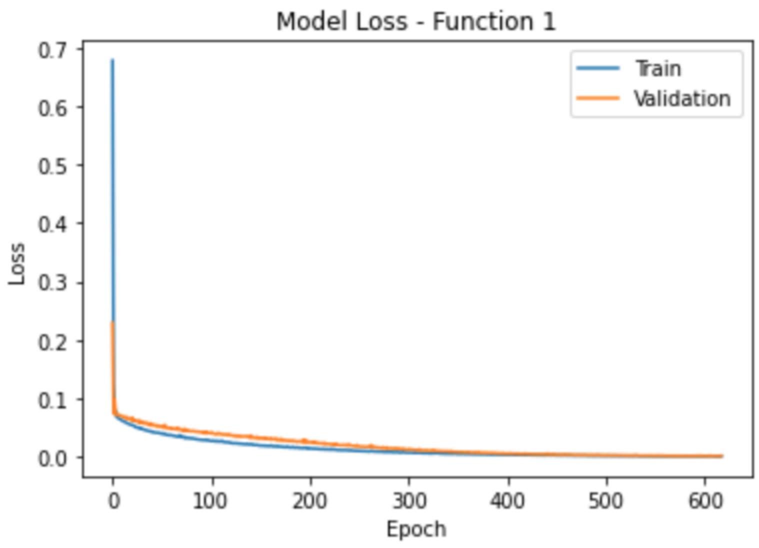


Figure 12: Model Loss vs. Epochs for Function 1

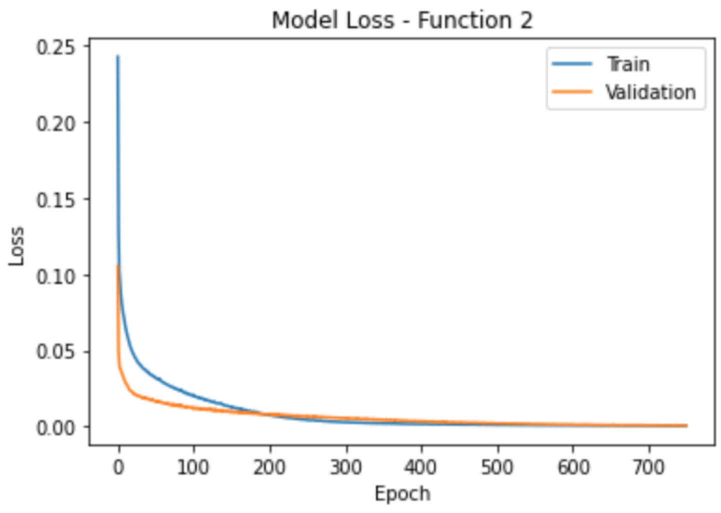


Figure 13: Model Loss vs. Epochs for Function 2

Question 4

4.1

Using Keras and Tensorflow, a multilayer neural network was built to classify the products. The data was first read from the text file and converted to a Dataframe using the pandas library. The constituents were normalized between 0 and 1 using scikit-learn's MinMaxScaler. The data was then split so that 75% would be training data and 25% would be test data. Lastly, one-hot encoding was performed on the class labels.

Different parameters were varied in order to evaluate the best model. Specifically, the number of hidden layers was varied between 1 and 5 and the number of nodes in each hidden layer was varied between 4 and 15. These models were trained on the training data and evaluated against the test data. To compare each model, the test accuracy was used. This is because the model has not seen the test data, so this accuracy would allow us to evaluate how well the model generalizes. From Table 4 below, it is clear that the model with 2 hidden layers and 10 nodes in each hidden layer performed the best. Its test accuracy was 93.33%.

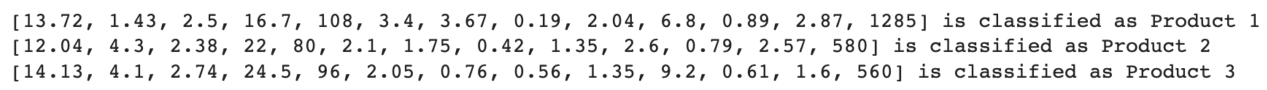
Table 4: Test Accuracies (%) of Models with Varying Hidden Layers and Number of Nodes

		Number of Hidden Layers				
		1	**2**	**3**	**4**	**5**
Number of Nodes in Each Hidden Layer	**4**	73.33 %	68.89 %	44.44 %	62.22 %	44.44 %
	5	88.89 %	77.78 %	53.33 %	66.67 %	44.44 %
	10	84.44 %	93.33 %	73.33 %	64.44 %	66.67 %
	12	88.89 %	86.67 %	66.67 %	77.78 %	68.89 %
	15	84.44 %	91.11 %	84.44 %	68.89 %	62.22 %

Given the results, it is clear that models that were too complex, having too many or too little hidden layers or nodes, did not perform well on the data. For instance, the models having only 4 nodes in each layer had low test accuracies, between about 40 to 70%. When increasing the number of nodes to 5, the model with 1 hidden layer performed better with an accuracy of 88.89%; however, when the layers increased, the model performed poorly. For example, with 2 hidden layers the accuracy was 77.78% and this number kept decreasing until at 5 hidden layers, the accuracy was 44.44%. This is due to the fact that the model became too complex and its variance was too high. This same pattern can be observed when 10 nodes are in each layer. The test accuracy initially increases when the hidden layers are increased from 1 to 2. However, having more than 2 hidden layers causes the model to perform poorly. Increasing the number of nodes from 10 to 12 or 15 did not increase test accuracy of the model. This is due to the fact that the model became too complex to capture the features in the data. Thus, the best model which was able to generalize well was one with 2 hidden layers and 10 nodes per hidden layer.

4.2

Using the model with 2 hidden layers and 10 nodes per hidden layer, the data points given in the question were classified. The results are shown in Figure 14.



```
[13.72, 1.43, 2.5, 16.7, 108, 3.4, 3.67, 0.19, 2.04, 6.8, 0.89, 2.87, 1285] is classified as Product 1
[12.04, 4.3, 2.38, 22, 80, 2.1, 1.75, 0.42, 1.35, 2.6, 0.79, 2.57, 580] is classified as Product 2
[14.13, 4.1, 2.74, 24.5, 96, 2.05, 0.76, 0.56, 1.35, 9.2, 0.61, 1.6, 560] is classified as Product 3
```

Figure 14: Product Classifications

ECE457B_Assignment1_20728766

February 13, 2022

0.1 Question 1

b) Write two small programs implementing the weight update rule for the perceptron and the Adaline for an arbitrary number of input (dimension of input x) and arbitrary training patterns. Initial values for the weights could be selected randomly in the interval [-1 1].

[ ]:
```
import numpy as np
np.random.seed(300)
```

[ ]:
```
class Perceptron():
  def __init__(self, learning_rate, n_epochs):
    self.learning_rate = learning_rate
    self.n_epochs = n_epochs

  def fit(self, X, y):
    X, y = np.array(X), np.array(y)
    n_samples, n_dim = X.shape # get number of samples and dimensions of input␣
↪data X
    self.weights = np.random.uniform(-1, 1, size=n_dim) # randomly initialize␣
↪weights between -1 and 1

    for _ in range(self.n_epochs):
      count_incorrect = 0

      for data, target in zip(X, y):
        # check if data point is misclassified
        if self.predict(data) != target:
          count_incorrect += 1

        # update rule
        delta_w = self.learning_rate * (target - self.predict(data)) * data
        self.weights += delta_w

      # break out of loop early if model classifies all points correctly
      if count_incorrect == 0:
        break

    print('The weights are: {}'.format(self.weights))
    return self.weights

  def activation_func(self, X):
    # step function
    return np.where(X >= 0, 1, 0)

  def predict(self, X):
    # get predictions
    output = np.dot(X, self.weights)
    predicted = self.activation_func(output)
    return predicted
```

[ ]:
```
class Adaline():
  def __init__(self, learning_rate, n_epochs):
    self.learning_rate = learning_rate
    self.n_epochs = n_epochs

  def fit(self, X, y):
    X, y = np.array(X), np.array(y)
    n_samples, n_dim = X.shape # get number of samples and dimensions of input␣
↪data X
    self.weights = np.random.uniform(-1, 1, size=n_dim) # randomly initialize␣
↪weights between -1 and 1

    for _ in range(self.n_epochs):
      for data, target in zip(X, y):
        # update rule from part a)
        delta_w = self.learning_rate * (target - self.predict(data)) * \
                  self.predict(data)**2  * np.exp( -(np.dot(self.weights, data)␣
↪) ) * data
        self.weights += delta_w

        # if delta_w is very small, break out of loop early
        if np.all(abs(delta_w) < 1.0e-06):
          break
      else:
        continue
      break

    print('The weights are: {}'.format(self.weights))
    return self.weights

  def activation_func(self, X):
    # sigmoid activation
    return 1 / (1 + np.exp(-X) )

  def predict(self, X):
    output = np.dot(X, self.weights)
    predicted = self.activation_func(output)
    return predicted
```

c) We need to classify the following patterns using boundaries obtained from the perceptron and the Adaline. These data points are located in 3D space and the first component x0 of each vector is the artificial input associated with w0, which plays the role of the bias term (-q). The remaining entries of the vector are the actual location of the data point in 3D space.

Class "C1=0" (x1=[-1, 0.8, 0.7, 1.2] , x2=[-1, -0.8,- 0.7, 0.2], x3=[-1, -0.5,0.3,- 0.2], x4=[-1, -2.8, -0.1, -2])
Class "C2=1" (y1=[-1, 1.2,- 1.7, 2.2] , y2=[-1, -0.8,-2, 0.5], y3=[-1, -0.5,-2.7,- 1.2], y4=[-1, 2.8, -1.4, 2.1])
If you are able to separate these classes, provide the equation of the boundaries in the case of the perceptron and the case of the Adaline. Draw, the data points of the two classes, and the corresponding boundaries in 3D Cartesian space (remember the first entry of each vector, is not part of the coordinate of the data point in 3D). Use the value of $\eta = 0.6$

[ ]:
```
c1 = np.array([ [-1, 0.8, 0.7, 1.2], [-1, -0.8,- 0.7, 0.2], [-1, -0.5,0.3,- 0.
↪2], [-1, -2.8, -0.1, -2] ])
c2 = np.array([ [-1, 1.2,- 1.7, 2.2], [-1, -0.8,-2, 0.5], [-1, -0.5,-2.7,- 1.
↪2], [-1, 2.8, -1.4, 2.1] ])
X = np.concatenate([c1, c2])
y = np.array([0 for x in range(4)] + [1 for x in range(4)]) # target values
```

[ ]:
```
# Perceptron
perceptron = Perceptron(learning_rate=0.6, n_epochs=300)
perceptron_weights = perceptron.fit(X, y)
perceptron_predictions = perceptron.predict(X)
print('The predictions are: {}'.format(perceptron_predictions))
```

```
The weights are: [ 1.10224514  0.64204468 -1.40183749  0.24155141]
The predictions are: [0 0 0 0 1 1 1 1]
```

[ ]:
```
# Adaline
adaline = Adaline(learning_rate=0.6, n_epochs=300)
adaline_weights = adaline.fit(X, y)
adaline_predictions = adaline.predict(X)
print('The predictions are: {}'.format(adaline_predictions))
```

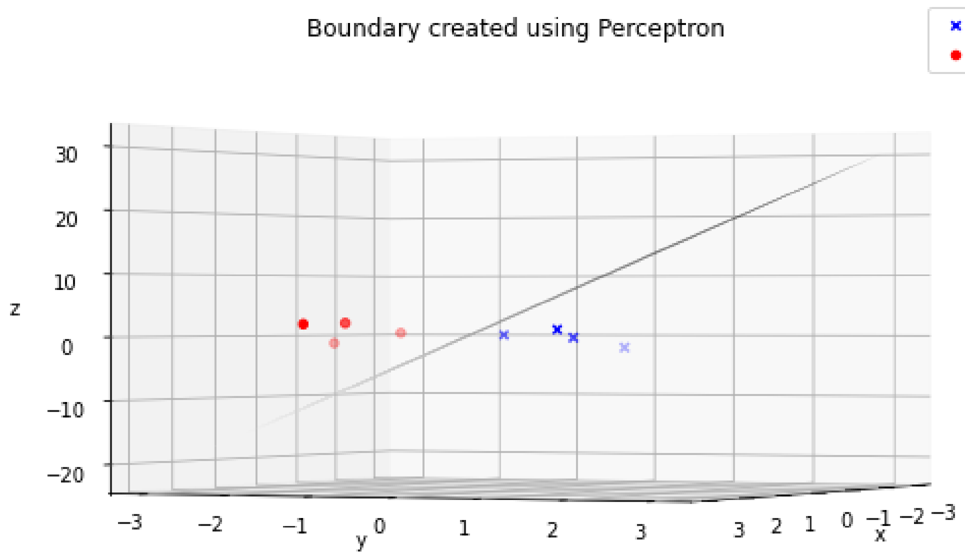
```
The weights are: [ 2.52468873  1.99122029 -3.24367603 -0.30381488]
The predictions are: [2.74684730e-02 1.29222056e-01 1.17435427e-02
7.70185903e-04
 9.91081174e-01 9.01834808e-01 9.96323842e-01 9.99045972e-01]
```

[ ]:
```
from mpl_toolkits import mplot3d
from matplotlib import pyplot as plt
```

```python
def plot_boundary(method, weights, c1, c2, view=25):
  print('The equation of the boundary for {} is: \n ({} - {}*x - ({})*y) / {}'.
 ↪format(method, weights[0], weights[1], weights[2], weights[3]))
  fig = plt.figure(figsize=(10,6))
  ax = plt.axes(projection='3d')
  x = np.linspace(-3, 3, 100)
  y = np.linspace(-3, 3, 100)

  xv, yv = np.meshgrid(x, y)
  boundary = (weights[0] - weights[1] * xv - weights[2] * yv) / weights[3]
  ax.plot_surface(xv,yv, boundary, cmap=plt.get_cmap('Greys'))

  ax.scatter(c1[:,1], c1[:,2], c1[:, 3], c='b', marker='x', label='C1=0')
  ax.scatter(c2[:,1], c2[:,2], c2[:, 3], c='r', marker='o', label='C2=1')
  ax.legend(loc='best')

  ax.set_xlabel('x')
  ax.set_ylabel('y')
  ax.set_zlabel('z')
  ax.set_title('Boundary created using {}'.format(method))
  ax.view_init(0, view)
```

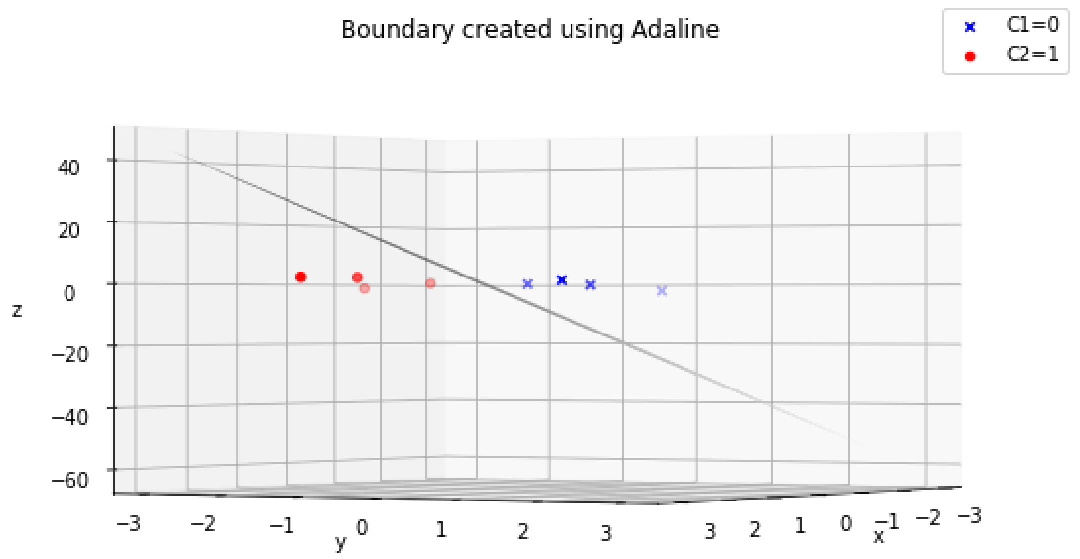
```python
[ ]: plot_boundary('Perceptron', perceptron_weights, c1, c2)
```

```
The equation of the boundary for Perceptron is:
 (1.1022451374001094 - 0.6420446834270908*x - (-1.4018374949412433)*y) /
0.24155140753283533
```



[ ]:
```
plot_boundary('Adaline', adaline_weights, c1, c2, view=31)
```

```
The equation of the boundary for Adaline is:
 (2.5246887268795954 - 1.9912202870684885*x - (-3.24367603276392)*y) /
-0.3038148849291844
```

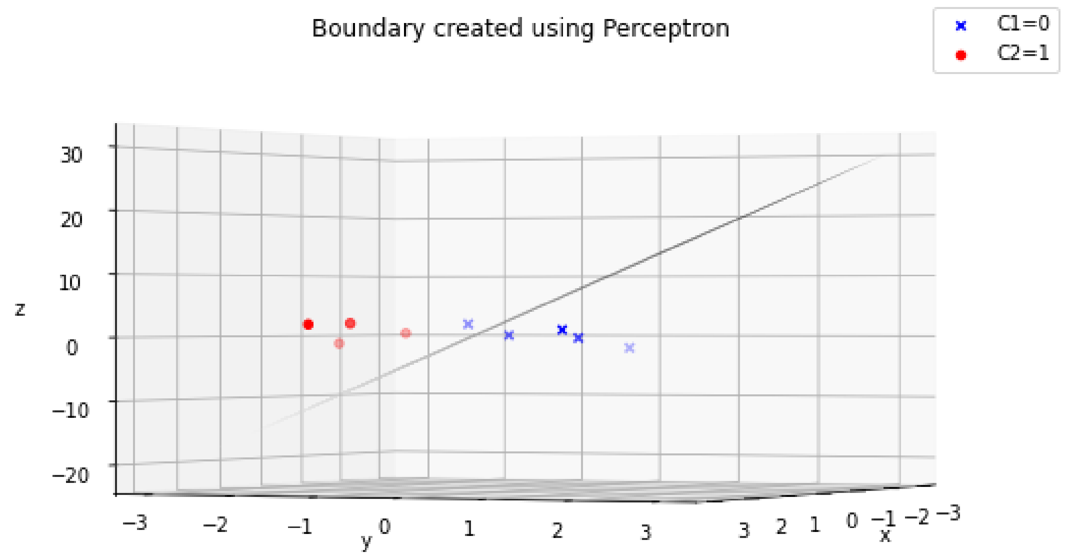


d) (2 marks) We need to place a new data point belonging to "C1" in the location x5=[-1.4,- 1.5, 2]. Is the classifier boundary still valid (for both perceptron and Adaline)

[ ]:
```
x5 = np.array([[-1, -1.4, -1.5, 2]])
percep_prediction = perceptron.predict(x5)
adaline_prediction = adaline.predict(x5)
c1 = np.append(c1, x5, axis=0)
```

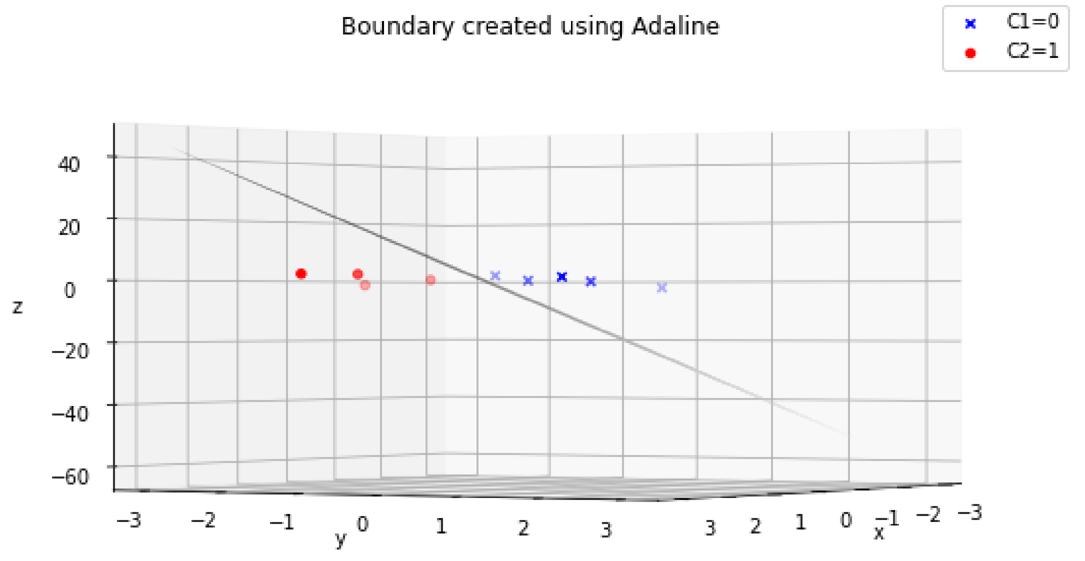
[ ]:
```
print('The Perceptron classified this point as: {}'.␣
 ↪format(percep_prediction[0]))
plot_boundary('Perceptron', perceptron_weights, c1, c2)
```

```
The Perceptron classified this point as: 1
The equation of the boundary for Perceptron is:
 (1.1022451374001094 - 0.6420446834270908*x - (-1.4018374949412433)*y) /
0.24155140753283533
```



[ ]:
```
print('The Adaline classified this point as: {}'. format(adaline_prediction[0]))
plot_boundary('Adaline', adaline_weights, c1, c2, view=31)
```

```
The Adaline classified this point as: 0.25835945191449106
The equation of the boundary for Adaline is:
 (2.5246887268795954 - 1.9912202870684885*x - (-3.24367603276392)*y) /
-0.30381488849291844
```



Therefore, the Perceptron boundary is not valid for this new point, however, the Adaline boundary is valid for this new point.

0.2 Question 2

Build a Madaline structure that is able to provide a compounded boundary (composed of two elementary boundaries) for the Exclusive Nor logic (XNOR) gate with two inputs. Draw the boundary in a 2D cartesian plot and show that the obtained compounded boundary (composed of two boundaries) is able to separate the two output classes (1 and -1).

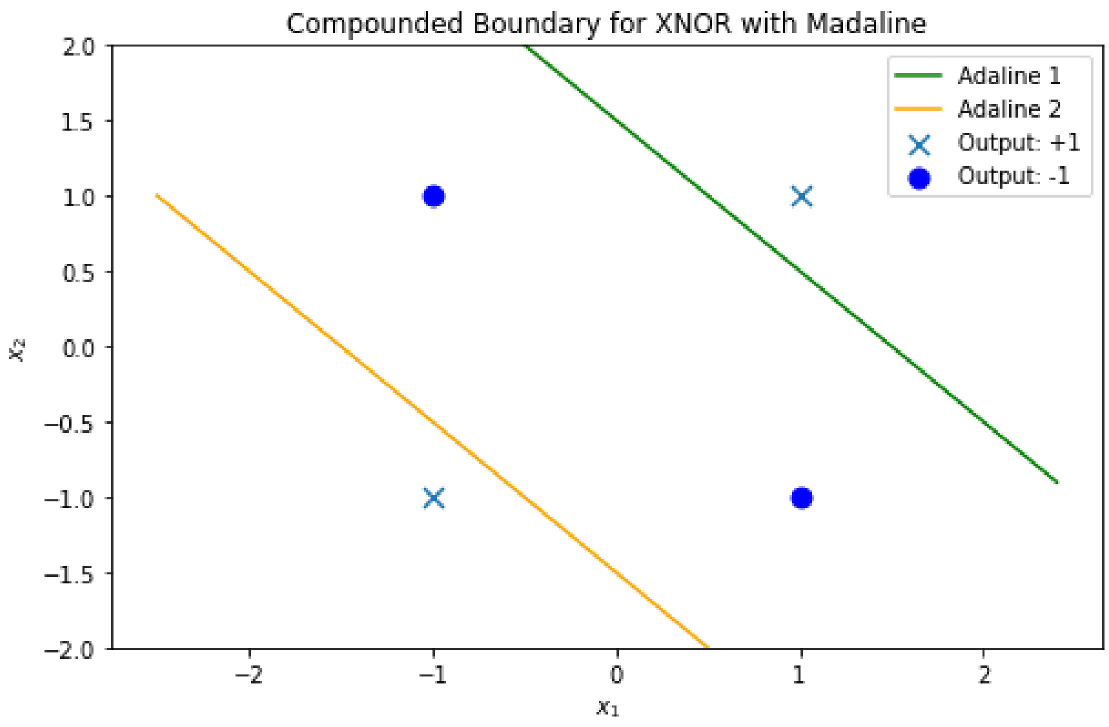
```
[ ]: import numpy as np
     def activation_func(x):
       if x > 0:
         return 1
       else:
         return -1

     def madaline(input, w):
       adaline_1 = input[0] * w[1] + input[1] * w[2] + w[0]
       q_1 = activation_func(adaline_1)

       adaline_2 = input[0] * w[3] + input[1] * w[4] + w[0]
       q_2 = activation_func(adaline_2)

       q_3 = (q_1 * 1) + (q_2 * 1) - w[0]

       output = activation_func(q_3)
       return output
```

```
[ ]: weights = np.array([-1.5, -1.0, -1.0, 1.0, 1.0])
     input = np.array([-1.0, -1.0])
     print("Input {}: {}".format(input, madaline(input, weights)))
```

```
Input [-1. -1.]: 1
```

```
[ ]: from matplotlib import pyplot as plt
     x = np.arange(-2.5, 2.5, 0.1)
     x1_class1 = [1, -1]
     x2_class1 = [1, -1]
     x1_class2 = [1, -1]
     x2_class2 = [-1, 1]
     fig, ax = plt.subplots(figsize=(8,5))
     ax.scatter(x1_class1, x2_class1, s=80, marker='x', label='Output: +1')
     ax.scatter(x1_class2, x2_class2, s=80, marker="o", c='b', label='Output: -1')
     ax.plot(x, -x + 1.5, color="green", label='Adaline 1')
     ax.plot(x, -x - 1.5, color="orange", label='Adaline 2')
     ax.set_title('Compounded Boundary for XNOR with Madaline')
     ax.set_xlabel('$x_1$')
     ax.set_ylabel('$x_2$')
     ax.set_ylim(-2, 2)
     ax.legend(loc='best')
```

```
[ ]: <matplotlib.legend.Legend at 0x7fac1bba1590>
```



0.3 Question 3

Using a feedforward back-propagation neural network that contains a single hidden layer (with a variable number of hidden nodes each having an activation function of the logistic form), investigate the outcome of the neural network for the following mappings:

$f_1(x) = x * sin(6x) * exp(-x^2)$ where $x \in [-1, 1]$

$f_2(x) = exp(-x^2) * arctan(x) * sin(4x)$ where $x \in [-2, 2]$.

For each function, create three sets of input/output data, one for training and validation and one for testing. These will be random values within the interval of the variable x). You can choose the ratio as 80% of the data for training (including validation) and 20% of the data for testing. We will use 10- fold cross validation approach, which means that the training set will be divided into 90% training and 10% for validation.

a) We need to create a 4 by 4 grid of 16 models assessments for each function.

```
[ ]: np.random.seed(457)
```

[ ]:
```
# Function Mappings
def f1(x):
  return x * np.sin(6*np.pi*x) * np.exp(-np.square(x))

def f2(x):
  return np.exp(-np.square(x)) * np.arctan(x) * np.sin(4*np.pi*x)
```

[ ]:
```
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from tensorflow import keras
from tensorflow.keras.layers import Dense, Flatten
from keras import initializers

def find_best_model(function, lower, upper):
  i = [10, 40, 80, 200] # number of data points
  j = [2, 10, 40, 100] # number of hidden nodes
  best_val_loss, best_model = float('inf'), 0

  for i_, num_points in enumerate(i):
    for j_, num_nodes in enumerate(j):
      # generate random data points
      X = np.random.uniform(lower, upper, num_points)
      y = function(X)

      # train/test split
      X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=1)

      # 10-fold cross validation
      kfold_val_avg, kfold_train_avg = [], []

      # repeat process 5 times
      for _ in range(5):
        if num_points == 10:
          kf = KFold(n_splits=8, shuffle=True)
        else:
          kf = KFold(n_splits=10, shuffle=True)

        val_scores, train_scores = [], []
        for train_index, val_index in kf.split(X_train):
          X_train_2, X_val = X_train[train_index], X_train[val_index]
          y_train_2, y_val = y_train[train_index], y_train[val_index]

          # model
          model = keras.models.Sequential( [ Dense(num_nodes,␣
 ↪activation='sigmoid', input_shape=(1,),
                                                            ␣
↪kernel_initializer=initializers.RandomNormal(mean=0, stddev=25),
                                                             bias_initializer=initializers.
↪RandomNormal(mean=0, stddev=24) ),
                                                      Dense(1, activation='linear')
                                                      ])
         model.compile(loss='mean_squared_error', optimizer='adam',␣
↪metrics=['accuracy'])
         callbacks = [keras.callbacks.EarlyStopping(monitor="val_loss",␣
↪min_delta=0.0001, patience=26, verbose=0)]
         # re-train model
         h = model.fit(X_train_2, y_train_2, epochs=750, batch_size=10,␣
↪validation_data=(X_val, y_val), callbacks=callbacks, verbose=0)
         val_scores.append(h.history['val_loss'][-1])
         train_scores.append(h.history['loss'][-1])

       kfold_val_avg.append(np.mean(val_scores))
       kfold_train_avg.append(np.mean(train_scores))

     avg_validation_loss = np.mean(kfold_val_avg)
     avg_training_loss = np.mean(kfold_train_avg)

     if avg_validation_loss < best_val_loss:
       best_model = [num_points, num_nodes]
       best_val_loss = avg_validation_loss

     print('Model with {} points and {} nodes has {} training loss and {}␣
↪validation loss.\n'.format(num_points, num_nodes, avg_training_loss,␣
↪avg_validation_loss))
 print('The Best Model has {} data points and {} nodes --> Validation Loss =␣
↪{}'.format(best_model[0], best_model[1], best_val_loss))
```

```
[ ]: find_best_model(f1, -1, 1)
```

```
Model with 10 points and 2 nodes has 0.11213658838532865 training loss and
0.06183519973822058 validation loss.

Model with 10 points and 10 nodes has 0.15307581112720073 training loss and
0.09457300616741122 validation loss.

Model with 10 points and 40 nodes has 0.04771740288706496 training loss and
0.06235041379121355 validation loss.

Model with 10 points and 100 nodes has 0.0634592538466677 training loss and
0.14551531430097384 validation loss.

Model with 40 points and 2 nodes has 0.10542855612933637 training loss and
```

```
0.06130300795659423 validation loss.

Model with 40 points and 10 nodes has 0.10079435274004936 training loss and
0.07133217110764235 validation loss.

Model with 40 points and 40 nodes has 0.07926708873361349 training loss and
0.07985121527686716 validation loss.

Model with 40 points and 100 nodes has 0.06138932347297669 training loss and
0.06908168079331517 validation loss.

Model with 80 points and 2 nodes has 0.0738526163250208 training loss and
0.05759875166229904 validation loss.

Model with 80 points and 10 nodes has 0.05977086406201124 training loss and
0.06120596615597605 validation loss.

Model with 80 points and 40 nodes has 0.03913203554227949 training loss and
0.040115648447535936 validation loss.

Model with 80 points and 100 nodes has 0.029309335525613277 training loss and
0.03727777183288709 validation loss.

Model with 200 points and 2 nodes has 0.057600513249635686 training loss and
0.057650212459266194 validation loss.

Model with 200 points and 10 nodes has 0.05680844899266958 training loss and
0.0559660119190812 validation loss.

Model with 200 points and 40 nodes has 0.0272062228154391 training loss and
0.02969920517876744 validation loss.

Model with 200 points and 100 nodes has 0.005285104999784381 training loss and
0.007516138569044414 validation loss.

The Best Model has 200 data points and 100 nodes --> Validation Loss =
0.007516138569044414
```

[ ]:
```
find_best_model(f2, -2, 2)
```

```
Model with 10 points and 2 nodes has 0.09596340414136648 training loss and
0.05410206258184334 validation loss.

Model with 10 points and 10 nodes has 0.1154511916305637 training loss and
0.02692908646135379 validation loss.

Model with 10 points and 40 nodes has 0.11398811839385417 training loss and
```

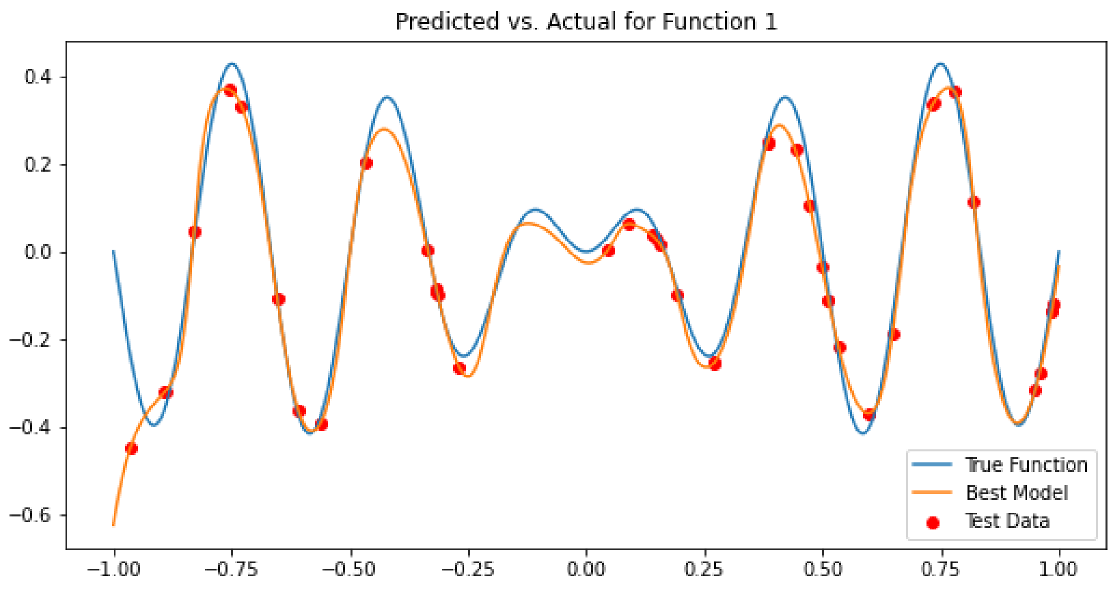
```
0.12699661110959823 validation loss.

Model with 10 points and 100 nodes has 0.053249078763928995 training loss and
0.09436564803268084 validation loss.

Model with 40 points and 2 nodes has 0.05854585926979781 training loss and
0.03868237202172168 validation loss.

Model with 40 points and 10 nodes has 0.04241482459008693 training loss and
0.034872349770303124 validation loss.

Model with 40 points and 40 nodes has 0.04000010330229998 training loss and
0.04763300196267664 validation loss.

Model with 40 points and 100 nodes has 0.024826127556152645 training loss and
0.03312191363152124 validation loss.

Model with 80 points and 2 nodes has 0.02358982866629958 training loss and
0.023999277083203197 validation loss.

Model with 80 points and 10 nodes has 0.04439171344041824 training loss and
0.04285637675784528 validation loss.

Model with 80 points and 40 nodes has 0.022749114078469575 training loss and
0.02702860444551334 validation loss.

Model with 80 points and 100 nodes has 0.014181707855896095 training loss and
0.01892680615419522 validation loss.

Model with 200 points and 2 nodes has 0.031265896186232565 training loss and
0.03007534757256508 validation loss.

Model with 200 points and 10 nodes has 0.02413396706804633 training loss and
0.02298824353143573 validation loss.

Model with 200 points and 40 nodes has 0.008739127105800434 training loss and
0.010384711789665744 validation loss.

Model with 200 points and 100 nodes has 0.002177379974309588 training loss and
0.0021014462801394983 validation loss.

The Best Model has 200 data points and 100 nodes --> Validation Loss =
0.0021014462801394983
```

b) Make qualitative and quantitative deductions in light of these simulations and find the best model that has acceptable bias (complex enough model) and an acceptable variance (no overfitting). Apply the full training data set to this model, and then apply the testing data (which the system never saw and draw the original functions and the best model obtained through neural network prediction.

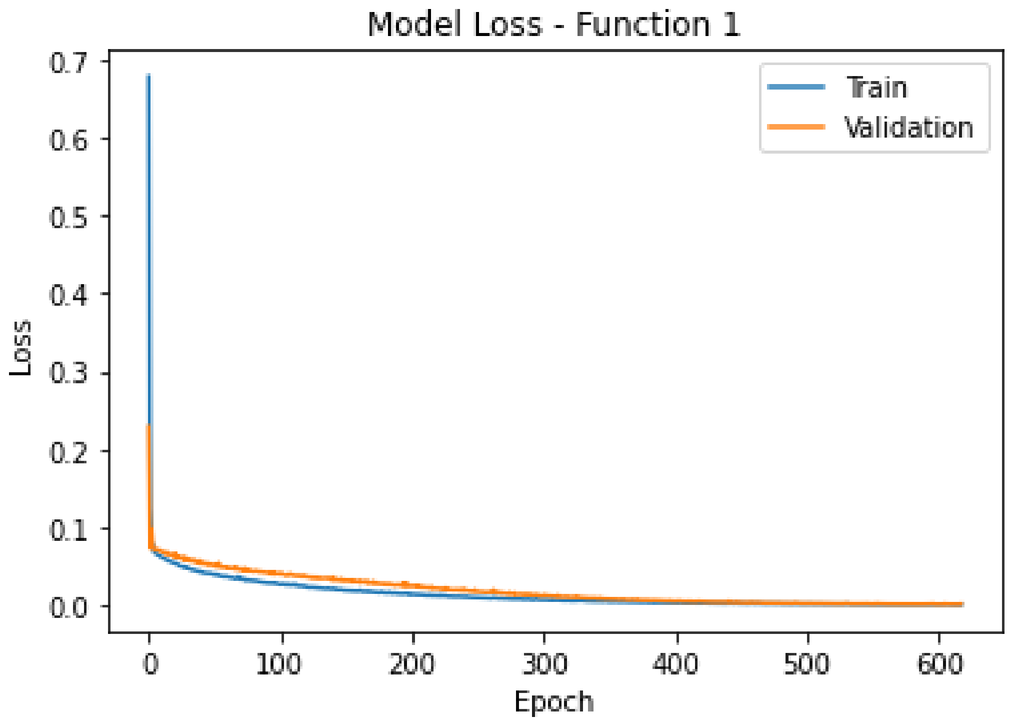
```
[ ]: X = np.random.uniform(-1, 1, 200)
     y = f1(X)
     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
     # model
     f1_model = keras.models.Sequential( [ Dense(100, activation='sigmoid',␣
      ↪input_shape=(1,),
                                             kernel_initializer=initializers.
      ↪RandomNormal(mean=0, stddev=25),
                                             bias_initializer=initializers.
      ↪RandomNormal(mean=0, stddev=24) ),
                                      Dense(1, activation='linear')
                                      ])
     f1_model.compile(loss='mean_squared_error', optimizer='adam',␣
      ↪metrics=['accuracy'])
     callbacks = [keras.callbacks.EarlyStopping(monitor="loss", min_delta=0.0001,␣
      ↪patience=26, verbose=0)]
     h = f1_model.fit(X_train, y_train, epochs=750, validation_split = 0.2,␣
      ↪batch_size=10, callbacks=callbacks, verbose=0)
```

```
[ ]: x = np.linspace(-1, 1, 1000)
     y_predicted = f1_model.predict(x)
     xtest_predicted = f1_model.predict(X_test)

     fig, ax = plt.subplots(figsize=(10,5))
     ax.plot(x, f1(x), label='True Function')
     ax.plot(x, y_predicted, label='Best Model')
     ax.scatter(X_test, xtest_predicted, c='r', label='Test Data')
     ax.legend(loc='best')
     ax.set_title('Predicted vs. Actual for Function 1')
```

```
[ ]: Text(0.5, 1.0, 'Predicted vs. Actual for Function 1')
```



[ ]:
```
plt.plot(h.history['loss'], label='Train')
plt.plot(h.history['val_loss'], label='Validation')
plt.title('Model Loss - Function 1')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(loc='best')
plt.show()
```



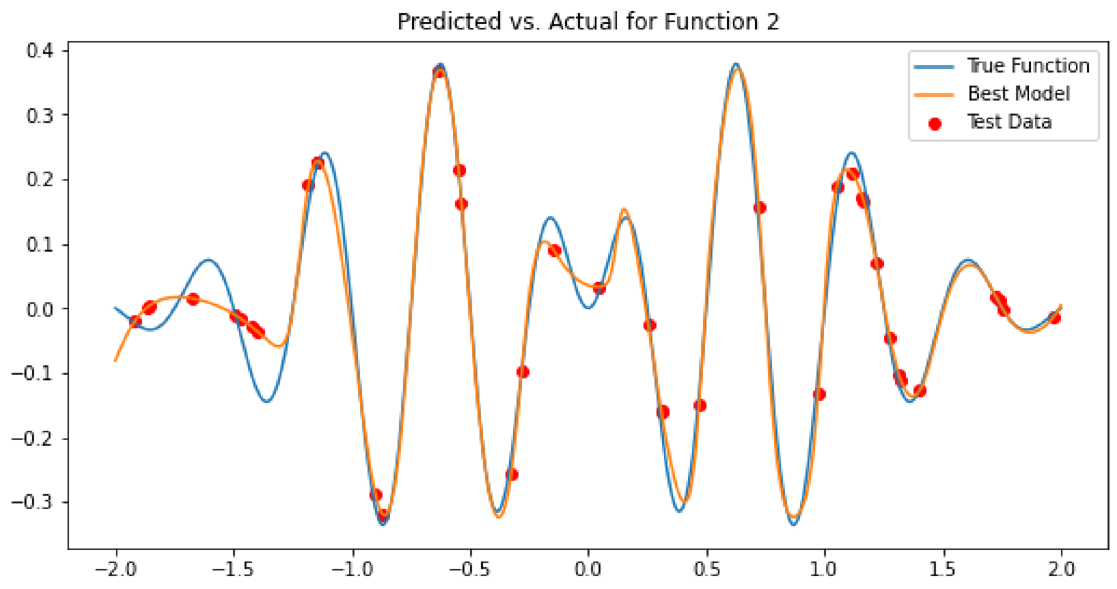
```
[ ]: print('Test loss: %.2f %%'%(100*f1_model.evaluate(X_test,y_test, verbose=0)[0]))
```

```
Test loss: 0.22 %
```

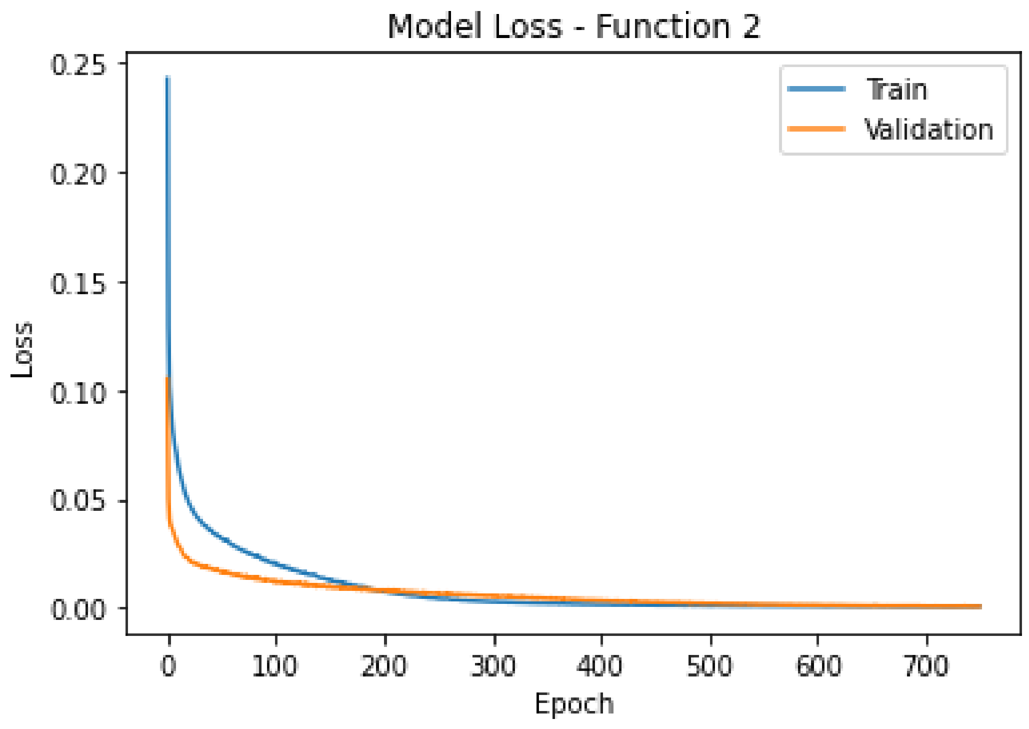
```
[ ]: X = np.random.uniform(-2, 2, 200)
     y = f2(X)
     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
     # model
     f2_model = keras.models.Sequential( [ Dense(100, activation='sigmoid',␣
      ↪input_shape=(1,),
                                                      kernel_initializer=initializers.
      ↪RandomNormal(mean=0, stddev=25),
                                                      bias_initializer=initializers.
      ↪RandomNormal(mean=0, stddev=24) ),
                                          Dense(1, activation='linear')
                                          ])
     f2_model.compile(loss='mean_squared_error', optimizer='adam',␣
      ↪metrics=['accuracy'])
     callbacks = [keras.callbacks.EarlyStopping(monitor="loss", min_delta=0.00001,␣
      ↪patience=26, verbose=0)]
     h = f2_model.fit(X_train, y_train, epochs=750, validation_split = 0.2,␣
      ↪batch_size=10, callbacks=callbacks, verbose=0)
```

```
[ ]: x = np.linspace(-2, 2, 2000)
     y_predicted = f2_model.predict(x)
     xtest_predicted = f2_model.predict(X_test)

     fig, ax = plt.subplots(figsize=(10,5))
     ax.plot(x, f2(x), label='True Function')
     ax.plot(x, y_predicted, label='Best Model')
     ax.scatter(X_test, xtest_predicted, c='r', label='Test Data')
     ax.legend(loc='best')
     ax.set_title('Predicted vs. Actual for Function 2')
```

```
[ ]: Text(0.5, 1.0, 'Predicted vs. Actual for Function 2')
```



[ ]:
```
plt.plot(h.history['loss'], label='Train')
plt.plot(h.history['val_loss'], label='Validation')
plt.title('Model Loss - Function 2')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(loc='best')
plt.show()
```



[ ]:
```
print('Test loss: %.2f %%'%(100*f2_model.evaluate(X_test,y_test, verbose=0)[0]))
```

```
Test loss: 0.08 %
```

0.4 Question 4

We need to develop a neural network based classifier for three various but related products. The collected data are the results of a chemical analysis of liquid products grown in the same region in Italy but derived from three different cultivars. The analysis determined the quantities of 13 constituents found in each of the three types of products. The data file provided is in text format and has fourteen dimensions, the first of which determines the class of products (product '1', product '2', product '3'), which should serve as the output of the neural network classifier. The remaining ones determine the input of the classifier and has 13 constituents as:
1. Ethanol 2. Malic acid
3. Ash
4. Alcalinity of ash
5. Magnesium
6. Total phenols
7. Flavanoids
8. Nonflavanoid phenols
9. Proanthocyanins
10. Color intensity
11. Hue
12. OD280/OD315 of diluted liquid
13. Proline

0.5 4.1

Build a classifier (multilayer neural network), vary its parameters (number of hidden layers and number of nodes in each layer) and try to find the best possible classification performance (a table illustrating various results as parameters are varied would be preferred). Please discuss.

[ ]:
```
import numpy as np
np.random.seed(500)
```

[ ]:
```
# Read data
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

data = pd.read_csv('/content/drive/MyDrive/ECE457B - Assignments/
↪randomized_data.txt', sep=",", header=None)
data.rename(columns={0:'label'}, inplace=True)
```

[ ]:
```
# Data Preprocessing - Normalizing constituents between 0 and 1
from sklearn.preprocessing import MinMaxScaler
y = data.iloc[:, 0]
```

```
scaler = MinMaxScaler()
arr_scaled = scaler.fit_transform(data.loc[:, data.columns!='label'])
data = data.loc[:, data.columns!='label']
X = pd.DataFrame(arr_scaled, columns=data.columns,index=data.index)
```

[ ]:
```
# Test-Train Split (75/25)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,␣
 ↪random_state=457)
```

[ ]:
```
# One-hot Encoding
import tensorflow.keras as keras
from tensorflow.keras.layers import Dense, Flatten
from sklearn.preprocessing import OneHotEncoder
y_train = [[item] for item in y_train]
y_test = [[item] for item in y_test]

enc = OneHotEncoder(sparse=False)
y_train = enc.fit_transform(y_train)
y_test = enc.fit_transform(y_test)
```

[ ]:
```
def create_mlp(num_nodes, num_layers, activation_func, X_train, y_train,␣
 ↪X_test, y_test, title):
  mlp = keras.models.Sequential()
  # input layer
  mlp.add(Dense(num_nodes, activation=activation_func, input_shape=(13,) ) )
  for num in range(num_layers):
    # additional layers
    mlp.add(Dense(num_nodes, activation=activation_func))

  mlp.add(Dense(3, activation='softmax')) # output layer
  mlp.compile(loss='categorical_crossentropy', optimizer='adam',␣
 ↪metrics=['accuracy'])

  h = mlp.fit(X_train, y_train, epochs=100, batch_size=10, verbose=0)

  test_accuracy = 100* mlp.evaluate(X_test, y_test, verbose=0)[1]
  print('Test accuracy: %.2f %%'%(test_accuracy))
  return test_accuracy
```

[ ]:
```
layers = [1, 2, 3, 4, 5]
nodes = [4, 5, 10, 12, 15]

best_accuracy = 0
for num_layers in layers:
  for num_nodes in nodes:
    title = ''
    accuracy = create_mlp(num_nodes, num_layers, 'sigmoid', X_train, y_train,␣
 ↪X_test, y_test, '')
```

```
    if accuracy > best_accuracy:
      best_accuracy = accuracy
      best_model = [num_nodes, num_layers]

print('The best model has {} nodes and {} hidden layers --> test accuracy = {}'.
 ↪format(best_model[0],best_model[1], best_accuracy))
```

```
Test accuracy: 73.33 %
Test accuracy: 88.89 %
Test accuracy: 84.44 %
Test accuracy: 88.89 %
Test accuracy: 84.44 %
Test accuracy: 68.89 %
Test accuracy: 77.78 %
Test accuracy: 93.33 %
Test accuracy: 86.67 %
Test accuracy: 91.11 %
Test accuracy: 44.44 %
Test accuracy: 53.33 %
Test accuracy: 73.33 %
Test accuracy: 66.67 %
Test accuracy: 84.44 %
Test accuracy: 62.22 %
Test accuracy: 66.67 %
Test accuracy: 64.44 %
Test accuracy: 77.78 %
Test accuracy: 68.89 %
Test accuracy: 44.44 %
Test accuracy: 44.44 %
Test accuracy: 66.67 %
Test accuracy: 68.89 %
Test accuracy: 62.22 %
The best model has 10 nodes and 2 hidden layers --> test accuracy =
93.33333373069763
```

0.6 4.2

Once this is done, classify (determine to which product they belong) the following entries each of which has 13 attributes:
a) 13.72; 1.43; 2.5; 16.7; 108; 3.4; 3.67; 0.19; 2.04; 6.8; 0.89; 2.87; 1285
b) 12.04; 4.3; 2.38; 22; 80; 2.1; 1.75; 0.42; 1.35; 2.6; 0.79; 2.57; 580
c) 14.13; 4.1; 2.74; 24.5; 96; 2.05; 0.76; 0.56; 1.35; 9.2; 0.61; 1.6; 560

[ ]:
```
# Normalize data
test_entries = [
              [13.72, 1.43, 2.5, 16.7, 108, 3.4, 3.67, 0.19, 2.04, 6.8, 0.89, 2.
 ↪87, 1285],
            [12.04, 4.3, 2.38, 22, 80, 2.1, 1.75, 0.42, 1.35, 2.6, 0.79, 2.57,␣
 ↪580],
            [14.13, 4.1, 2.74, 24.5, 96, 2.05, 0.76, 0.56, 1.35, 9.2, 0.61, 1.
 ↪6, 560]
]
scaled = scaler.transform(test_entries)
```

[ ]:
```
model = keras.models.Sequential()
model.add(Dense(best_model[0], activation='sigmoid', input_shape=(13,) ) )
for num in range(best_model[1]):
    # additional layers
    model.add(Dense(best_model[0], activation='sigmoid'))
model.add(Dense(3, activation='softmax')) # output layer
model.compile(loss='categorical_crossentropy', optimizer='adam',␣
 ↪metrics=['accuracy'])
h = model.fit(X_train, y_train, epochs=100, batch_size=10, verbose=0)

predict = model.predict(scaled)
classes=np.argmax(predict,axis=1)
```

[ ]:
```
labels = ['Product 1', 'Product 2', 'Product 3']
```

[ ]:
```
for i in range(3):
  print('{} is classified as {}'.format( test_entries[i], labels[classes[i]] ))
```

```
[13.72, 1.43, 2.5, 16.7, 108, 3.4, 3.67, 0.19, 2.04, 6.8, 0.89, 2.87, 1285] is
classified as Product 1
[12.04, 4.3, 2.38, 22, 80, 2.1, 1.75, 0.42, 1.35, 2.6, 0.79, 2.57, 580] is
classified as Product 2
[14.13, 4.1, 2.74, 24.5, 96, 2.05, 0.76, 0.56, 1.35, 9.2, 0.61, 1.6, 560] is
classified as Product 3
```

[ ]:
```
### END OF NOTEBOOK ###
```