ECE 457B Assignment 2

Tzu-Chun Chou
20718966
t6chou@uwaterloo.ca

Q1

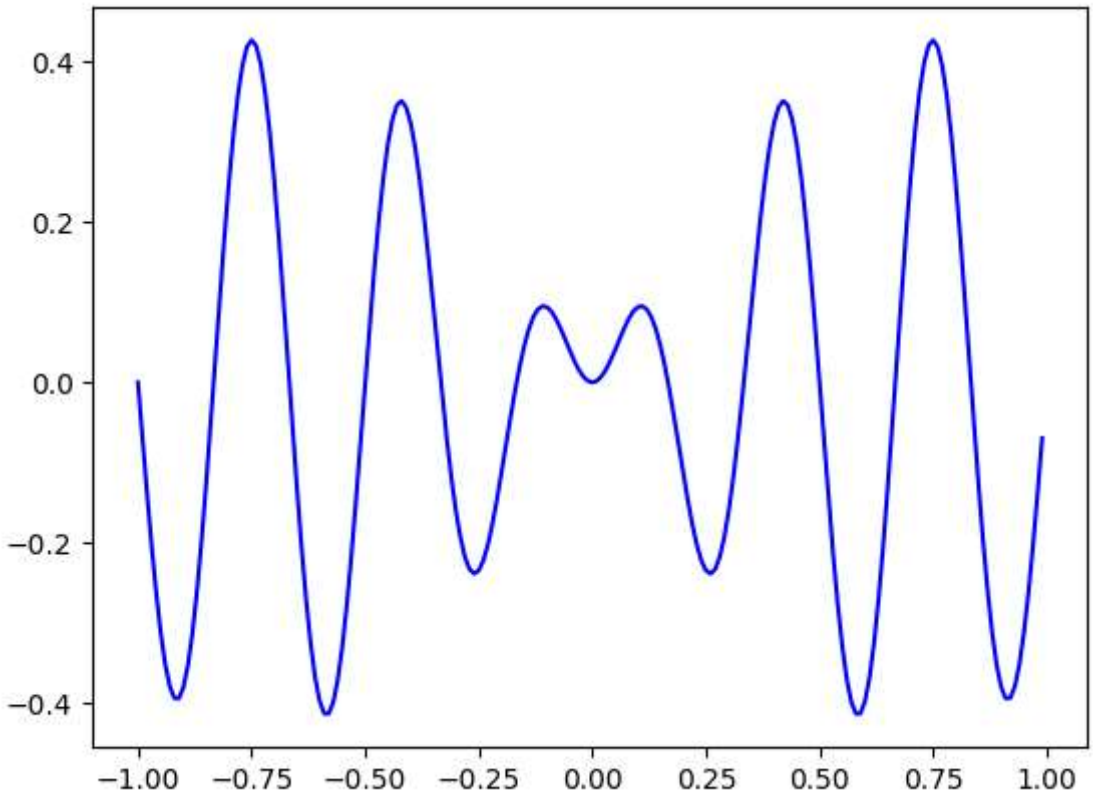
Setup

In [ ]:
```
# import required packages
import numpy as np
import matplotlib.pyplot as plt
import tensorflow.keras as keras
# make it easier to understand by importing the required libraries within keras
from tensorflow.keras.layers import Dense, Flatten
from sklearn.model_selection import KFold
import random as random
import pandas as pd
from sklearn.model_selection import train_test_split
```

f1(x) = x * sin(6πx) * exp(−x^2) *where* $x \in [-1, 1]$

In [ ]:
```
def f1(x):
    return x*np.sin(6*np.pi*x)*np.exp(-(x**2))

x = np.arange(-1, 1, 0.01)
y = f1(x)
plt.plot(x, y, 'b')
```

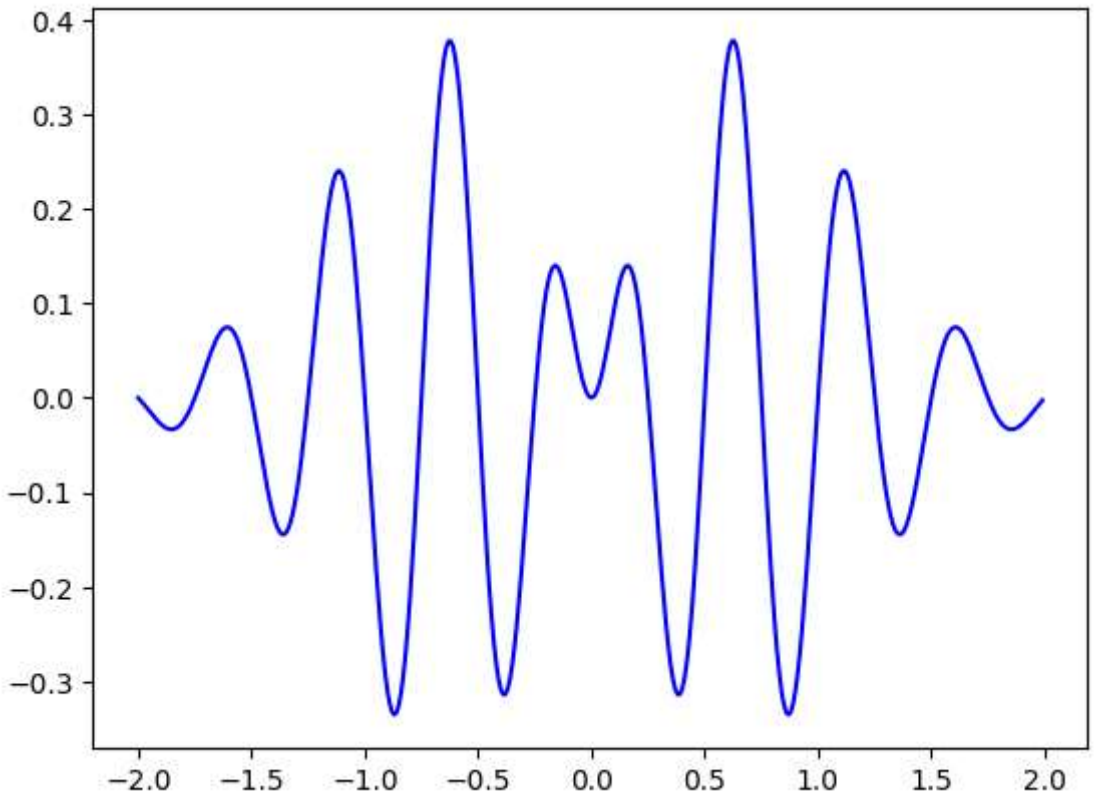
Out[ ]:
```
[<matplotlib.lines.Line2D at 0x1e594fd89a0>]
```



f2(x) = exp(−x^2) * arctan(x) * sin(4πx) *where* $x \in [-2, 2]$

In [ ]:
```
def f2(x):
    return np.exp(-(x**2))*np.arctan(x)*np.sin(4*np.pi*x)

x = np.arange(-2, 2, 0.01)
y = f2(x)
plt.plot(x, y, 'b')
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x1e5970ddba0>]
```

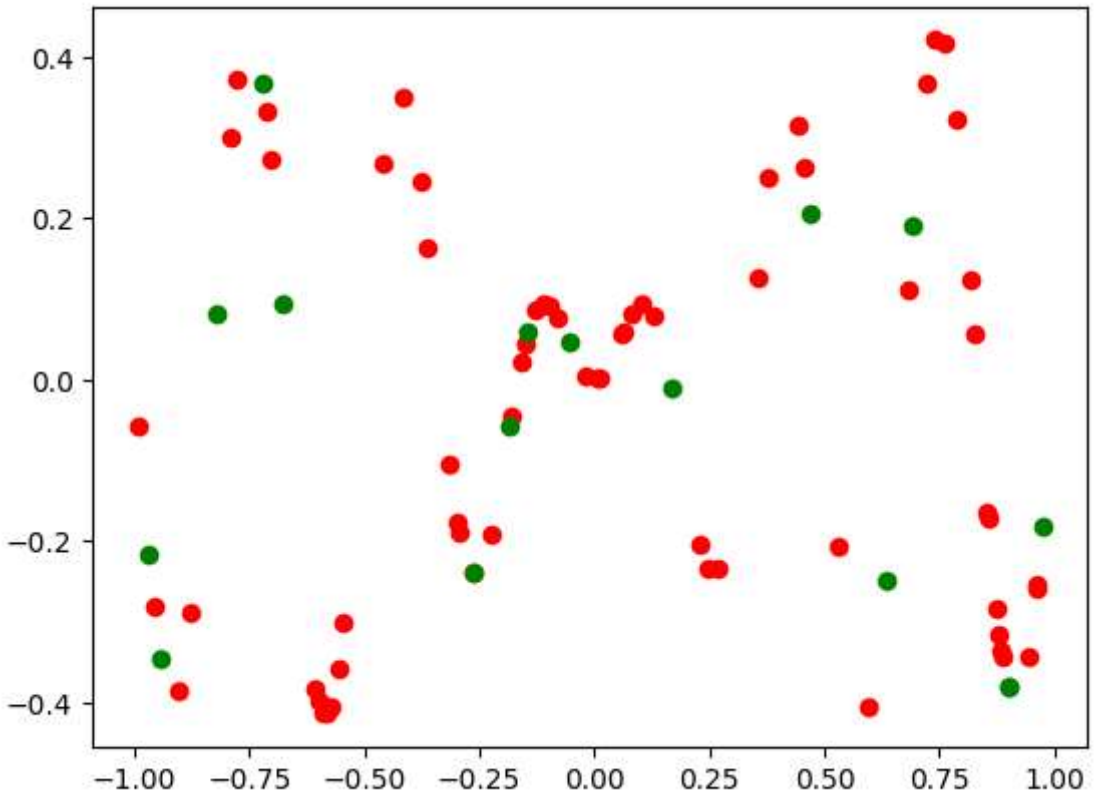


Data Generation for function 1, where data where initially seperated into 80% training and 20% testing

In [ ]:
```
def generatef1(n):
    x = np.empty(n)
    y = np.empty(n)

    for i in range(n):
        x[i] = random.uniform(-1, 1)
        y[i] = f1(x[i])

    # 80% for training, 20% for testing
    x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)
    return x_train, x_test, y_train, y_test

x_train, x_test, y_train, y_test = generatef1(80)
```

```
plt.scatter(x_train, y_train, c="r")
plt.scatter(x_test, y_test, c="g")
```

Out[ ]: <matplotlib.collections.PathCollection at 0x1e59714fa90>



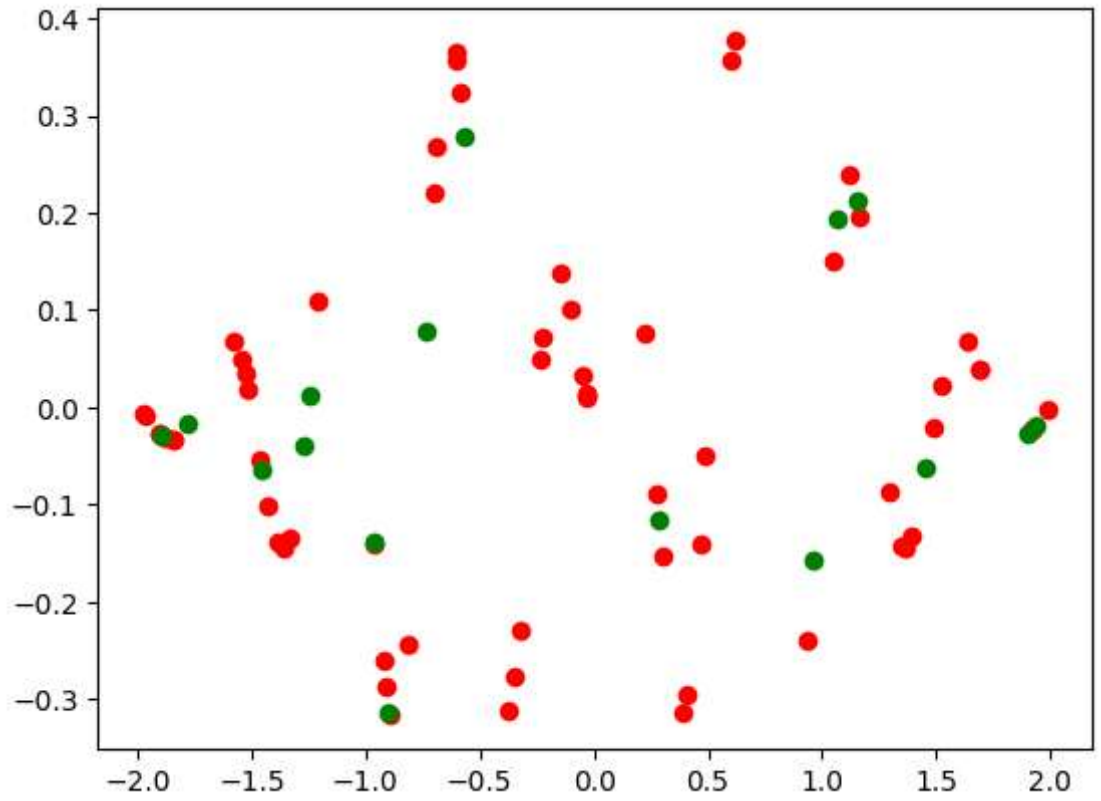
Function for comparing the output of regression model against function 1 using line approximation.

In [ ]:
```
def evaluatef1(model):
    # make prediction using new dataset
    x_train, x_test, y_train, y_test = generatef1(100)
    prediction = model.predict(x_test)

    # generate best fit approximation line using predicted value
    coefficients = np.polyfit(x_test, prediction.flatten(), 6)
    line = np.poly1d(coefficients)
    x_line = np.linspace(min(x_test), max(x_test), 100)
    y_line = line(x_line)

    # plot the data points and the line approximation
    plt.scatter(x_test, prediction.flatten(), c="r")
    plt.plot(x_line, y_line, '-g', label='Line approximation')

    # plot the original function
    x = np.arange(-1, 1, 0.01)
    y = f1(x)
    plt.plot(x, y, 'b')
```

Data Generaton for function 2, where data where initially seperated into 80% training and 20% testing

In [ ]:
```
def generatef2(n):
    x = np.empty(n)
    y = np.empty(n)

    for i in range(n):
        x[i] = random.uniform(-2, 2)
        y[i] = f2(x[i])

    # 80% for training, 20% for testing
    x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)
    return x_train, x_test, y_train, y_test

x_train, x_test, y_train, y_test = generatef2(80)

plt.scatter(x_train, y_train, c="r")
plt.scatter(x_test, y_test, c="g")
```

Out[ ]:
```
<matplotlib.collections.PathCollection at 0x1e5981d7a00>
```



Function for comparing the output of regression model against function 2 using line approximation.

In [ ]:
```
def evaluatef2(model):
    # make prediction using new dataset
    x_train, x_test, y_train, y_test  = generatef2(100)
    prediction = model.predict(x_test)
```

```
    # best fit curve
    coefficients = np.polyfit(x_test, prediction.flatten(), 6)
    line = np.poly1d(coefficients)
    x_line = np.linspace(min(x_test), max(x_test), 100)
    y_line = line(x_line)

    # plot the data points and the line approximation
    plt.scatter(x_test, prediction.flatten(), c="r")
    plt.plot(x_line, y_line, '-g', label='Line approximation')

    # plot the original function
    x = np.arange(-2, 2, 0.01)
    y = f2(x)
    plt.plot(x, y, 'b')
```

Function for training the model, since this is a regression model we use mean square error as the loss function to penalize large errors more than small errors. Here I used 50 epochs and random shuffling to make sure the model is not underfitting. Furthermore, a 10-fold cross validation is also implemented to compute the validation loss, this is used by the early stopping callback function to prevent overfitting.

In [ ]:
```
def fit(model, x_train, y_train, x_val, y_val, x_test, y_test):
    # early stopping mechanism
    callback = keras.callbacks.EarlyStopping(monitor='val_loss', patience=3, verbos

    # mean square error for loss function, rest is default
    model.compile(loss='mse', optimizer='adam')

    h = model.fit(x_train, y_train, epochs=20, batch_size=1, callbacks=[callback],

    # print('Test accuracy: %.2f %%'%(100*model.evaluate(x_test,y_test, verbose=0)[

    # return the average training error and validation error for analysis
    return np.array(h.history['loss']).mean(), np.array(h.history['val_loss']).mean
```

Function for generating a 4x4 grid of training error and validation error comparing different combination of sample size and number of hidden nodes

In [ ]:
```
def generateGrid(data):
    # create the dataframe
    df = pd.DataFrame(data, columns=["10", "40", "80", "200"],
                      index=["2", "10", "40", "100"])

    # print the dataframe
    print(df)
```

Q1.a

Regression model for function 1. After testing differnt combination functions I decided to use relu as activation function since it returns the lowest averge training error. As per

instruction a 10-fold validation is implemented and repeated 5 times to find the smallest average trainig and validation error.

In [ ]:
```
n_samples = [10, 40, 80, 200]
n_neurons = [2, 10, 40, 100]

M1 = np.empty((len(n_samples), len(n_neurons)), dtype=object)
error1 = np.empty((len(n_samples), len(n_neurons)), dtype=tuple)

for i in range(len(n_samples)):
    for j in range(len(n_neurons)):
        # generat dataset
        x_train, x_test, y_train, y_test = generatef1(n_samples[i])
        # model initialization
        M1[i,j] = keras.models.Sequential()
        # input layer + hidden layer
        M1[i,j].add(Dense(n_neurons[j], activation='relu', input_shape=(1,)))
        # output layer
        M1[i,j].add(Dense(1, activation='linear'))

        min_error = 100
        for k in range(5):
            # 10 fold cross validation
            n_split = 10
            if len(x_train) < n_split:
                n_split = len(x_train)
            kf = KFold(n_splits=n_split)
            kf.get_n_splits(x_train)

            # for recording average loss
            training_loss = np.empty(n_split)
            validation_loss = np.empty(n_split)

            counter = 0
            # iterate through all folds
            for train_index, val_index in kf.split(x_train):
                X_train, X_val = x_train[train_index], x_train[val_index]
                Y_train, Y_val = y_train[train_index], y_train[val_index]
                # train
                training_loss[counter], validation_loss[counter] = fit(M1[i,j], X_t
                counter += 1

            # find minimum training error and validation error
            if np.mean(training_loss) + np.mean(validation_loss) < min_error:
                min_error = np.mean(training_loss) + np.mean(validation_loss)
                error1[i,j] = (np.mean(training_loss), np.mean(validation_loss))

generateGrid(error1)
```

```
                                                    10  \
2    (0.05414935135437797, 0.05385130065284708)
10   (0.05199830967932939, 0.05152399278362282)
40   (0.048563607162130734, 0.0485745525306889)
100  (0.05987063880477632, 0.06107437070459128)

                                                    40  \
2      (0.0601191902366866, 0.0626970036282728)
10    (0.06907132840404909, 0.07125054872284334)
40   (0.034031126926697436, 0.03537064360942514)
100   (0.05729163628692428, 0.05764846197639902)

                                                    80  \
2    (0.034630283928247144, 0.05086789566746683)
10    (0.03364432331795494, 0.03771761179125558)
40     (0.0579301312891906, 0.06266143695278267)
100  (0.05169885905991708, 0.051704399492591625)

                                                   100
2    (0.028570994129820195, 0.039252011152100696)
10    (0.058264324383898856, 0.06484172552424881)
40      (0.0448900416703333, 0.04657066136022054)
100    (0.04579001915330688, 0.04600728619843721)
```

Regression model for function 2. After testing differnt combination functions I decided to use relu as activation function since it returns the lowest averge training error. As per instruction a 10-fold validation is implemented and repeated 5 times to find the smallest average trainig and validation error.

In [ ]:
```
n_samples = [10, 40, 80, 200]
n_neurons = [2, 10, 40, 100]

M2 = np.empty((len(n_samples), len(n_neurons)), dtype=object)
error2 = np.empty((len(n_samples), len(n_neurons)), dtype=tuple)

for i in range(len(n_samples)):
    for j in range(len(n_neurons)):
        # generate dataset
        x_train, x_test, y_train, y_test = generatef2(n_samples[i])
        # model initialization
        M2[i,j] = keras.models.Sequential()
        # input layer + hidden layer
        M2[i,j].add(Dense(n_neurons[j], activation='relu', input_shape=(1,)))
        #output layer
        M2[i,j].add(Dense(1, activation='linear'))

        min_error = 100
        for k in range(5):
            # 10 fold cross validation
            n_split = 10
            if len(x_train) < n_split:
                n_split = len(x_train)
            kf = KFold(n_splits=n_split)
            kf.get_n_splits(x_train)
```

```
            # for recording averge loss
            training_loss = np.empty(n_split)
            validation_loss = np.empty(n_split)

            counter = 0
            # iterate through all folds
            for train_index, val_index in kf.split(x_train):
                X_train, X_val = x_train[train_index], x_train[val_index]
                Y_train, Y_val = y_train[train_index], y_train[val_index]
                # train
                training_loss[counter], validation_loss[counter] = fit(M2[i,j], X_t
                counter += 1

            # find minimum training error and validation error
            if np.mean(training_loss) + np.mean(validation_loss) < min_error:
                min_error = np.mean(training_loss) + np.mean(validation_loss)
                error2[i,j] = (np.mean(training_loss), np.mean(validation_loss))

generateGrid(error2)
```

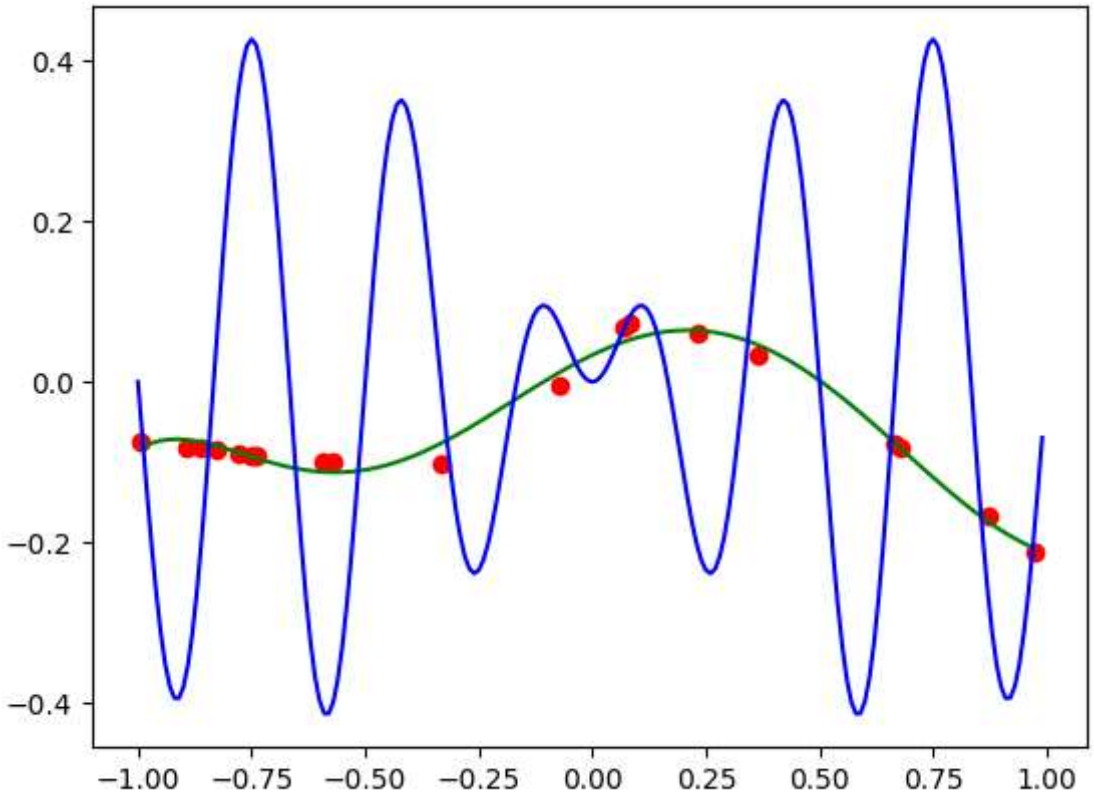
```
                                                    10  \
2      (0.03652091434923932, 0.03689127443385587)
10     (0.0207467647105278, 0.021054603296908593)
40   (0.019851228820958307, 0.020560766014248312)
100  (0.029449138414735593, 0.029418667775933587)

                                                    40  \
2     (0.02780278818681836, 0.029446039169385374)
10    (0.025636663091856804, 0.02817399887612001)
40    (0.02685172565615712, 0.027949892098848256)
100  (0.028464588460822903, 0.028657422671094536)

                                                    80  \
2      (0.032317559472567425, 0.0411827959420126)
10    (0.02205761242391808, 0.023197676583326287)
40   (0.030110226035640054, 0.029710253649851782)
100    (0.02052885327785186, 0.02109490025421661)

                                                   100
2    (0.0013753031136614496, 0.0018347554425671394)
10       (0.02870144869755244, 0.03174450980340009)
40     (0.030333381737582387, 0.031330515050018826)
100    (0.029690192632424905, 0.029913975007625094)
```

Q1.b

For function 1, the model with 10 hidden nodes and 80 samples shows the best average training and validation error of 0.0336 while having a complex enough model. The following chart shows its prediction using 20 fresh data, compared to the original function

In [ ]:
```
print(error1[1 , 2])
evaluatef1(M1[1 , 2])
```

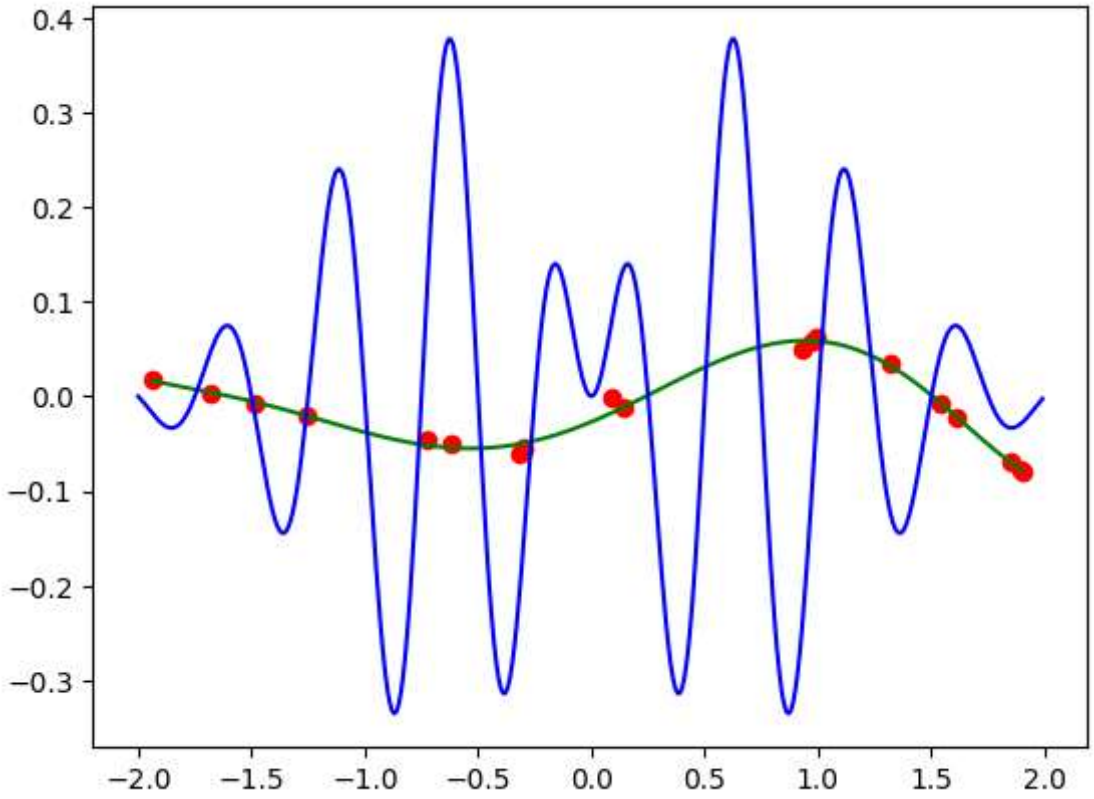
```
(0.03364432331795494, 0.03771761179125558)
1/1 [==============================] - 0s 17ms/step
```



For function 2, the model with 100 hidden nodes and 80 samples shows the best average training and validation error of 0.0207 while having a complex enought model. The following chart shows its prediction using fresh data, compared to the original function

In [ ]:
```
print(error2[3, 2])
evaluatef2(M2[3, 2])
```

```
(0.02052885327785186, 0.02109490025421661)
1/1 [==============================] - 0s 20ms/step
```



Q1.c

From both models and it average error gird, we can see that the average error for the model with small samples and neurons are higher than the rest, indicating that it is underfit since it does not have enough data or node to sufficiently train on. The same can be said for the model on the opposite end, indicating that it is overfit because it has memorized the pattern instead of learning it. The best model of both function lies in the middle, since it has the best balance between variance and bias.

Q2

Setup

In [ ]:
```
# import required packages
import numpy as np
import matplotlib.pyplot as plt
import tensorflow.keras as keras
# make it easier to understand by importing the required libraries within keras
from tensorflow.keras.layers import Dense, Flatten
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
```

Q2.a

Data file IO

In [ ]:
```
df = pd.read_csv("randomized_data.csv")
print(df.shape)
```

```
(177, 14)
```

Load and normalize traing and test data

In [ ]:
```
# column 1 = label, column 2 -> 14 = input features
(x, y) = df.iloc[:, 1:14], df.iloc[:, 0]

# normalization
n_x = MinMaxScaler().fit_transform(x)

# 75% for training, 25% for testing
x_train, x_test, y_train, y_test = train_test_split(n_x, y, test_size=0.25)

print('Training set shape: {}'.format(x_train.shape))
print('Training labels shape: {}'.format(y_train.shape))
print('Test set shape: {}'.format(x_test.shape))
print('Test label shape: {}'.format(y_test.shape))
```

```
Training set shape: (132, 13)
Training labels shape: (132,)
Test set shape: (45, 13)
Test label shape: (45,)
```

Create and convert labels to categorical

In [ ]:
```
labels = set(y_train)

if len(y_train.shape) == 1:
    y_train = keras.utils.to_categorical(y_train-1, num_classes=3)
    y_test = keras.utils.to_categorical(y_test-1, num_classes=3)
```

```
print('The classes are: {}'.format(labels))
print('The values in the input data range from {} to {}'.format(x_train.min(),x_tra
```

```
The classes are: {1, 2, 3}
The values in the input data range from 0.0 to 1.0
```

Function for training and testing model, here I used categorical crossentropy as loss function since there are more than 2 label classes. Epoch is set to 100 to prevent underfitting. Finally, I used the test accuracy as metric for the model. Since the classification is simple enough, cross validation is not implemented here

In [ ]:
```
def fit(model, x_train, y_train, x_test, y_test):
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accu

    h = model.fit(x_train, y_train, epochs=100, batch_size=1, verbose=0)

    # return test accurancy
    return model.evaluate(x_test,y_test, verbose=0)[1]
```

Function for generating a 4x4 grid of accuracy values comparing different combination of neurons and hidden layers

In [ ]:
```
def generateGrid(data):
    # create the dataframe
    df = pd.DataFrame(data, columns=["4", "16", "32", "64"],
                      index=["1", "2", "3", "4"])

    # print the dataframe
    print(df)
```

Classification Model. Here I tested 16 different combinations of number of nodes and number of layers to find the best possible accuracy performance and used sigmoid as activation function.

In [ ]:
```
n_neurons = [4, 16, 32, 64]
n_layers = [1, 2, 3, 4]

M = np.empty((len(n_neurons), len(n_layers)), dtype=object)
accuracy = np.empty((len(n_neurons), len(n_layers)), dtype=float)

for i in range(len(n_neurons)):
    for j in range(len(n_layers)):
        # model initialization
        M[i][j] = keras.models.Sequential()
        # input layer + 1st hidden layer
        M[i][j].add(Dense(n_neurons[i], activation='sigmoid', input_shape=(13,)))
        # hidden layers
        for layer in range(n_layers[j]-1):
            M[i][j].add(Dense(n_neurons[i], activation='sigmoid'))
        # output layer
        M[i][j].add(Dense(3, activation='softmax'))
```

```
        # train
        accuracy[i,j] = fit(M[i, j], x_train, y_train, x_test, y_test)

generateGrid(accuracy)
```

```
          4        16        32        64
1  0.977778  0.955556  0.933333  0.600000
2  0.977778  0.955556  0.977778  0.977778
3  0.977778  0.955556  0.977778  0.977778
4  0.977778  0.977778  1.000000  1.000000
```

According to the table, the best possible classification performance for accuracy is 100% for both 32 neurons X 4 layers and 64 neurons X 4 layers, since both models are complex enough to ensure that the model is not underfitted. Interestingly the model is not overfitted as well, this could be because the model is still sufficiently simple due to the small layer count.

Q2.b

Here I used the model with 64 neurons X 4 layers to predict the classification of the 3 products

In [ ]:
```
test = np.array([[13.72, 1.43, 2.5, 16.7, 108, 3.4, 3.67, 0.19, 2.04, 6.8, 0.89, 2.
                 [12.04, 4.3, 2.38, 22, 80, 2.1, 1.75, 0.42, 1.35, 2.6, 0.79, 2.5
                 [14.13, 4.1, 2.74, 24.5, 96, 2.05, 0.76, 0.56, 1.35, 9.2, 0.61,

n_test = MinMaxScaler().fit_transform(test)

prediction = M[3,3].predict(n_test)

print(list(map(lambda x: list(map(int, map(round, x))), prediction)))
```

```
1/1 [==============================] - 0s 59ms/step
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
```

The classification result of the 3 products are: [0,1,2]