The purpose of this assignment is to generate a neural network capable of performing a Fourier Decomposition of an arbitrary periodic waveform. The Fourier analysis, of course, is intended to break a periodic waveform into its basic harmonics. So, the waveform is defined by a series of points over a cycle of a fundamental (or, from symmetry, over a quarter or a half of the waveform). For this assignment, assumption that all the harmonics are in phase was made and the dc level was removed, hence, the waveform is expressed by a set of harmonics:

$$f(x) = \sum_{n=1}^{\infty} \left(b_n \sin \frac{n\pi x}{L} \right)$$

To achieve the goal of this assignment, Tensorflow was used and the following steps were taken:

1. Importation of libraries and tf.keras:
   - **NumPy** which allows you to work with high-performance arrays and matrices.
   - **Pandas** — to load the data file as a Pandas data frame and analyze the data.
   - From **Sklearn**, I imported the *datasets* module, so I can load a sample dataset, and the *linear_model*, so I can run a linear regression
   - From **Sklearn,** sub-library **model_selection**, I imported the *train_test_split* so I can, well, split to training and test sets
   - From **Matplotlib,** *pyplot was imported* in order to plot graphs of the data

   To get started, I imported tf.keras as part of your TensorFlow program setup which makes TensorFlow easier to use without sacrificing flexibility and performance.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import tensorflow as tf
from tensorflow.python.keras.layers import Dense
from tensorflow.python.keras import Sequential
from sklearn.model_selection import train_test_split
```

2. Generation of a set of random, but periodic waveforms:

   The code below was written to generate random periodic waveforms using 10 Fourier magnitudes. It serves as an output of *b* array of 10 random magnitudes and *f(x)* array of size 120 corresponding waveforms between $0 \leq f\ (x\ ) \leq \pi$

```
def waveform_gen():
  b = np.random.rand(10)
  x = np.linspace(0,2*np.pi,120)
  fx = np.zeros_like(x)
  n = 1
  L = 2*np.pi
  for i in b:
    fx += i*np.sin(n*np.pi*x/L)
    n+=1
  return b,fx
```

3. Generation of a large dataset to train, test and validate the network which provide a set of inputs for the network to be trained. The code below is used to generate 10,000 random waveforms which is then stored in a *.data* file, with the last 10 corresponding to the Fourier magnitudes.

```
coef,f = waveform_gen()
c = np.copy(coef)
vect = np.concatenate((f,coef))
i = 10000
data = np.zeros((10000,np.size(vect)))
data[0,:] = vect
for j in range (1,i):
  coef,f = waveform_gen()
  data[j,:] = np.concatenate((f,coef))
np.savetxt("datafile.data", data, delimiter=",")
```

4. Building the model and Training the dataset:

   Firstly, the dataset was split into the training and test set. Sometimes when you train on data you can get models that works great for the exact data you have trained on, but not other data. The purpose of doing this is basically to see if the model made is "sound". The algorithm will try to minimize the error within the training set. When training is done, the algorithm has seen all the data in training set. However, I wanted to know if the algorithm can be used for further forecasting, I want to know how the algorithm works when given the unseen data.

   Next is feature scaling which is a technique to standardize the independent features present in the data in a fixed range. It is performed during the data pre-processing to handle highly varying magnitudes or values or units. If feature scaling was not done, then the machine learning algorithm tends to weigh greater values, higher and consider smaller values as the lower values, regardless of the unit of the values. To achieve this, the *sklearn.preprocessing* package was used. The preprocessing module further provides a utility class *StandardScaler* that implements the Transformer API to compute the mean and standard deviation on a training set so as to be able to later reapply the same transformation on the testing set.

```
# Importing the dataset
dataset = np.genfromtxt("datafile.data", delimiter=',')
X = dataset[:, :-10]
y = dataset[:, -10:]

# Splitting the dataset into the Training set and Test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.08, random_s
tate = 0)

# Feature Scaling
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

```
# Initialising the ANN
model = Sequential()

# Adding the input layer and the first hidden layer
model.add(Dense(32, activation = 'relu', input_dim = 120))

# Adding the second hidden layer
model.add(Dense(units = 32, activation = 'relu'))

# Adding the third hidden layer
model.add(Dense(units = 32, activation = 'relu'))

# Adding the output layer

model.add(Dense(units = 10))

#model.add(Dense(1))
# Compiling the ANN
model.compile(optimizer = 'adam', loss = 'mean_squared_error' ,  metrics=['acc'])

# Fitting the ANN to the Training set
model.fit(X_train, y_train, batch_size = 100, epochs = 100)

y_pred = model.predict(X_test)

plt.plot(y_test[2,:], color = 'blue', label = 'Real data')
plt.plot(y_pred[2,:], color = 'red', label = 'Predicted data')
plt.title('Prediction')
plt.legend()
plt.ylabel('Magnitude')
plt.xlabel('Harmonics')
plt.show()
```

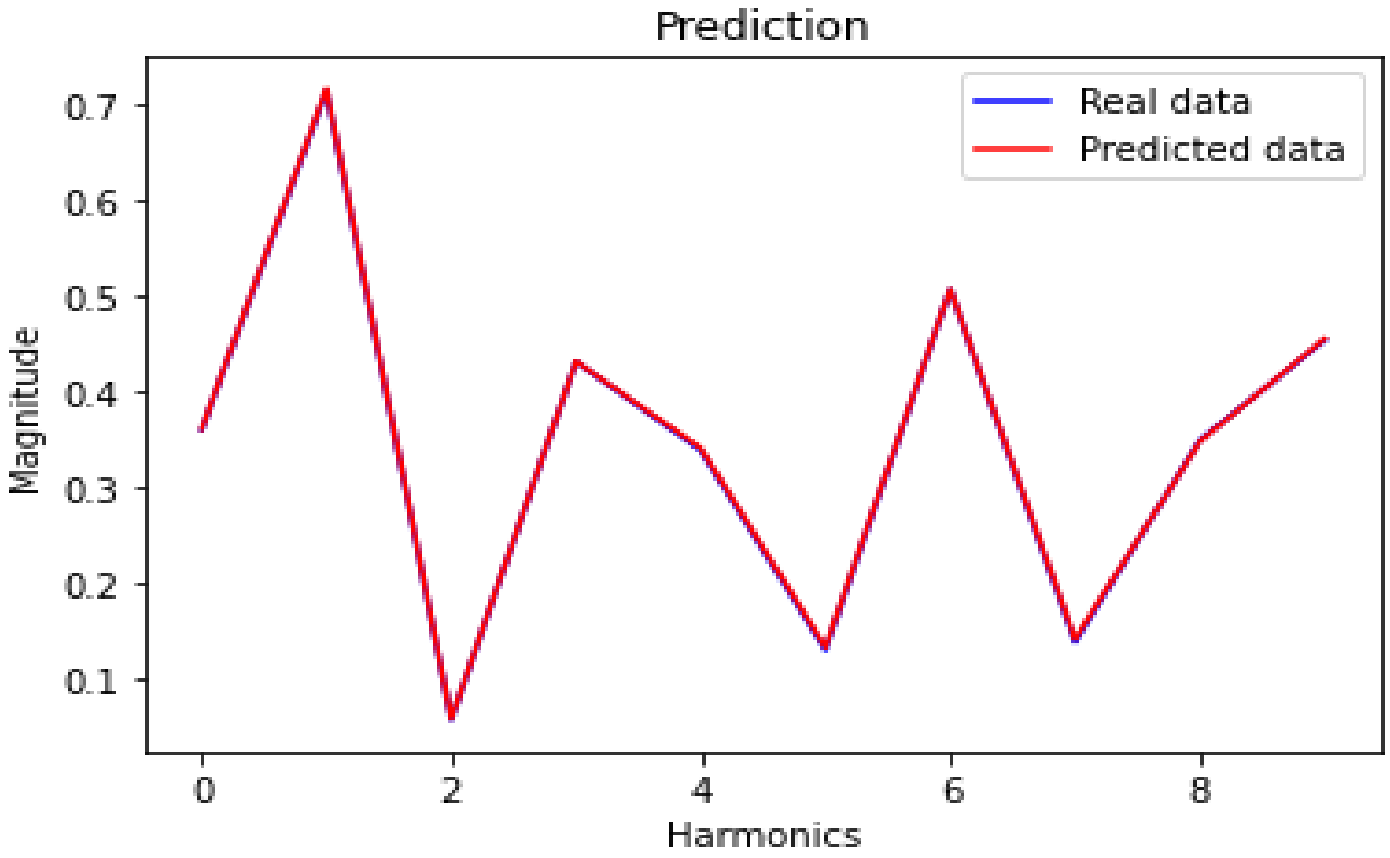
For this network, Keras Sequential model with the architecture shown above was used with an input layer with multiple inputs, three hidden layers, all with 32 neurons fully connected (dense) layer with a ReLU activation, followed by a 0.001-dropout. For a given node, the inputs are multiplied by the weights in a node and summed together and transformed using an activation function that defines the specific output of the node. Activation functions are the gates between one step/layer and another. It gets the output of one layer and feed it as an input to the next layer. ReLU was chose as they are good for networks with fewer layers and for avoid the vanishing gradient problem.

Before training the model, the learning process needs to be configured which was done via the *compile* method. Compiling the model will create a Python object which will build the ANN. This is done by building the computation graph in the correct format based on the Keras backend that was used. The compilation steps prompt one to define the loss function (objective that the model will try to minimize), kind of optimizer I want

to use as well as a list of metrics. For any feed forward neural network, the choice of loss function, optimizer, and regularization function is very important.

Then, the model was trained so that the parameters get tuned to provide the correct outputs for a given input. This was done by feeding inputs at the input layer and then getting an output. This will **fit** the model parameters to the data. Keras models are trained on NumPy arrays of input data and labels. For training a model, *fit* function is typically used. The model was trained by slicing the data into "batches" of size "batch_size", and repeatedly iterating over the entire dataset for a given number of "*epochs*" which was fixed at 100.

The model architecture and the computationally expensive part of deep learning have been completed. We can now take our model and use feed-forward passes and predict inputs. With a batch size and epochs fixed at 100 each, it took approximately **18*secs*** to train the network with an accuracy of **100%.** The output perfectly matches the original spectrum results provided in the output datasets as shown in the plot below.



APPENDIX

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import tensorflow as tf
from tensorflow.python.keras.layers import Dense
from tensorflow.python.keras import Sequential
from sklearn.model_selection import train_test_split


'''
  The function generates random periodic waveforms using 10 Fourier coefficie
nts
  Output: b array of 6 random coefficients
          fx array of size 120 for corresponding waveform between 0 and pi
'''
def waveform_gen():
  b = np.random.rand(10)
  print(b)
  x = np.linspace(0,2*np.pi,120)
  fx = np.zeros_like(x)
  n = 1
  L = 2*np.pi
  for i in b:
    fx += i*np.sin(n*np.pi*x/L)
    n+=1
  return b,fx



  '''
Generate data for training, testing and validation. Generate 1000 random wave
forms,
data is stored in a .data file, with first 120 entries of each row the wavefo
rm data
and the last 6 the corresponding Fourier coefficients
'''
coef,f = waveform_gen()
c = np.copy(coef)
vect = np.concatenate((f,coef))
i = 10000
data = np.zeros((10000,np.size(vect)))
data[0,:] = vect
```

```
for j in range (1,i):
  coef,f = waveform_gen()
  data[j,:] = np.concatenate((f,coef))
np.savetxt("datafile.data", data, delimiter=",")


import matplotlib.pyplot as plt
# Importing the dataset
dataset = np.genfromtxt("datafile.data", delimiter=',')
X = dataset[:, :-10]
y = dataset[:, -10:]

# Splitting the dataset into the Training set and Test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.08, r
andom_state = 0)

# Feature Scaling
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

# Initialising the ANN
model = Sequential()

# Adding the input layer and the first hidden layer
model.add(Dense(32, input_dim = 120))

# Adding the second hidden layer
model.add(Dense(units = 32))

# Adding the third hidden layer
model.add(Dense(units = 32))

# Adding the output layer

model.add(Dense(units = 10))

#model.add(Dense(1))
# Compiling the ANN
model.compile(optimizer = 'adam', loss = 'mean_squared_error' ,  metrics=['ac
c'])

# Fitting the ANN to the Training set
model.fit(X_train, y_train, batch_size = 100, epochs = 100)
```

```
y_pred = model.predict(X_test)

plt.plot(y_test[2,:], color = 'blue', label = 'Real data')
plt.plot(y_pred[2,:], color = 'red', label = 'Predicted data')
plt.title('Prediction')
plt.legend()
plt.ylabel('Magnitude')
plt.xlabel('Harmonics')
plt.show()
```