# RNN Fuel Price

December 5, 2020

## 0.1 RNN Fuel Price

Let's start our Python script by importing our libraries:

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from datetime import datetime as dt
```

We will initially import the data set as a pandas DataFrame using the read_csv method. However, since the keras module of TensorFlow only accepts NumPy arrays as parameters, the data structure will need to be transformed post-import. Let's start by importing the entire .csv file as a DataFrame:

[2]:
```
fueldata = pd.read_csv('data/fuel-prices.csv')
```

[3]:
```
fueldata.head()
```

[3]:
```
        time  price
0  960177600  1.583
1  960782400  1.607
2  961387200  1.653
3  961992000  1.664
4  962596800  1.684
```

I need to change the timestamp in unix time to a datetime format and also reome the hours, minutes and seconds from it

[4]:
```
fueldata["time"] = pd.to_datetime(fueldata['time'],unit='s').dt.date
```

[5]:
```
fueldata.head()
```

[5]:
```
         time  price
0  2000-06-05  1.583
1  2000-06-12  1.607
2  2000-06-19  1.653
3  2000-06-26  1.664
4  2000-07-03  1.684
```

[6]:
```
fueldata.shape
```

```
[6]: (749, 2)
```

```
[7]: training_data = fueldata.iloc[:, 1].values
     training_data = training_data[:-4]
     testing_data = training_data[-4:len(training_data)]
```

```
[8]: training_data.shape
```

```
[8]: (745,)
```

```
[9]: testing_data.shape
```

```
[9]: (4,)
```

```
[10]: training_data = training_data.reshape(-1, 1)
      testing_data = testing_data.reshape(-1, 1)
```

The next thing we need to do is to specify our number of timesteps. Timesteps specify how many previous observations should be considered when the recurrent neural network makes a prediction about the current observation. We will use 4 timesteps in this tutorial. This means that for every week that the neural network predicts, it will consider the previous 4 weeks of fuel prices to determine its output.

```
[11]: x_training_data = []

      y_training_data =[]
```

Now we will use a for loop to populate the actual data into each of these Python lists. Here's the code (with further explanation of the code after the code block):

```
[12]: for i in range(4, len(training_data)):
          x_training_data.append(training_data[i-4:i, 0])
          y_training_data.append(training_data[i, 0])
```

Let's unpack the components of this code block: 1. The range(4, len(training_data)) function causes the for loop to iterate from 4 to the last index of the training data. 2. The x_training_data.append(training_data[i-4:i, 0]) line causes the loop to append the 4 preceding fuel prices to x_training_data with each iteration of the loop. 3. Similarly, the y_training_data.append(training_data[i, 0]) causes the loop to append the next week's stock price to y_training_data with each iteration of the loop.

TensorFlow is designed to work primarily with NumPy arrays. Because of this, the last thing we need to do is transform the two Python lists we just created into NumPy arrays. Fortunately, this is simple. You simply need to wrap the Python lists in the np.array function. Here's the code:

```
[13]: x_training_data = np.array(x_training_data)

      y_training_data = np.array(y_training_data)
```

One important way that you can make sure your script is running as intended is to verify the shape of both NumPy arrays.

The x__training__data array should be a two-directional NumPy array with one dimension being 4 (the number of timesteps) and the second dimension being len(training__data) - 4, which evaluates to 745 in our case.

Similarly, the y__training__data object should be a one-dimensional NumPy array of length 745 (which, again, is len(training__data) - 4).

You can verify the shape of the arrays by printing their shape attribute, like this:

[14]:
```
print(x_training_data.shape)

print(y_training_data.shape)
```

```
(741, 4)
(741,)
```

Both arrays have the dimensions you'd expect.

However, we need to reshape our x__training__data object one more time before proceeding to build our recurrent neural network.

The reason for this is that the recurrent neural network layer available in TensorFlow only accepts data in a very specific format.

To reshape the x__training__data object, I will use the np.reshape method. Here's the code to do this:

[15]:
```
x_training_data = np.reshape(x_training_data, (x_training_data.shape[0],

                                               x_training_data.shape[1],

                                               1))
```

[16]:
```
print(x_training_data.shape)
```

```
(741, 4, 1)
```

Before we can begin building our recurrent neural network, we'll need to import a number of classes from TensorFlow. Here are the statements you should run before proceeding:

[17]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import LSTM
from tensorflow.keras.layers import Dropout
```

It's now time to build our recurrent neural network.

The first thing that needs to be done is initializing an object from TensorFlow's Sequential class. As its name implies, the Sequential class is designed to build neural networks by adding sequences of layers over time.

Here's the code to initialize our recurrent neural network:

```
[18]: rnn = Sequential()
```

The first layer that we will add is an LSTM layer. To do this, pass an invocation of the LSTM class (that we just imported) into the add method.

The LSTM class accepts several parameters. More precisely, we will specify three arguments: 1. The number of LSTM neurons that you'd like to include in this layer. Increasing the number of neurons is one method for increasing the dimensionality of your recurrent neural network. In our case, we will specify units = 10. 2. return_sequences = True - this must always be specified if you plan on including another LSTM layer after the one you're adding. You should specify return_sequences = False for the last LSTM layer in your recurrent neural network. 3. input_shape: the number of timesteps and the number of predictors in our training data. In our case, we are using 40 timesteps and only 1 predictor (stock price), so we will add

```
[19]: rnn.add(LSTM(units = 10, return_sequences = True, input_shape =␣
       ↪(x_training_data.shape[1], 1)))
```

Note that I used x_training_data.shape[1] instead of the hardcoded value in case we decide to train the recurrent neural network on a larger model at a later date.

Dropout regularization is a technique used to avoid overfitting when training neural networks.

It involves randomly excluding - or "dropping out" - certain layer outputs during the training stage.

TensorFlow makes it easy to implement dropout regularization using the Dropout class that we imported earlier in our Python script. The Dropout class accepts a single parameter: the dropout rate.

The dropout rate indicates how many neurons should be dropped in a specific layer of the neural network. It is common to use a dropout rate of 20%. We will follow this convention in our recurrent neural network.

Here's how you can instruct TensorFlow to drop 20% of the LSTM layer's neuron during each iteration of the training stage:

```
[20]: rnn.add(Dropout(0.2))
```

We will now add three more LSTM layers (with dropout regularization) to our recurrent neural network. You will see that after specifying the first LSTM layer, adding more is trivial.

To add more layers, all that needs to be done is copying the first two add methods with one small change. Namely, we should remove the input_shape argument from the LSTM class.

We will keep the number of neurons (or units) and the dropout rate the same in each of the LSTM class invocations. Since the third LSTM layer we're adding in this section will be our last LSTM layer, we can remove the return_sequences = True parameter as mentioned earlier. Removing the parameter sets return_sequences to its default value of False.

```
[21]: rnn.add(LSTM(units = 10, return_sequences = True))

      rnn.add(Dropout(0.2))
      rnn.add(LSTM(units = 10, return_sequences = True))

      rnn.add(Dropout(0.2))

      rnn.add(LSTM(units = 10))

      rnn.add(Dropout(0.2))
```

[ ]:

```

```

Let's finish architecting our recurrent neural network by adding our output layer.

The output layer will be an instance of the Dense class, which is the same class we used to create the full connection layer of our convolutional neural network earlier in this course.

The only parameter we need to specify is units , which is the desired number of dimensions that the output layer should generate. Since we want to output the next day's stock price (a single value), we'll specify units = 1.

[22]:

```
rnn.add(Dense(units = 1))
```

TensorFlow allows us to compile a neural network using the aptly-named compile method. It accepts two arguments: optimizer and loss.

We now need to specify the optimizer and loss parameters.

Let's start by discussing the optimizer parameter. Recurrent neural networks typically use the RMSProp optimizer in their compilation stage. With that said, we will use the Adam optimizer (as before). The Adam optimizer is a workhorse optimizer that is useful in a wide variety of neural network architectures.

The loss parameter is fairly simple. Since we're predicting a continuous variable, we can use mean squared error - just like you would when measuring the performance of a linear regression machine learning model. This means we can specify loss = mean_squared_error.

[23]:

```
rnn.compile(optimizer = 'adam', loss = 'mean_squared_error')
```

It's now time to train our recurrent network on our training data.

To do this, we use the fit method. The fit method accepts four arguments in this case: 1. The training data: in our case, this will be x_training_data and y_training_data 2. Epochs: the number of iterations you'd like the recurrent neural network to be trained on. We will specify epochs = 100 in this case. 3. The batch size: the size of batches that the network will be trained in through each epoch.

Here is the code to train this recurrent neural network according to our specifications:

[24]:

```
rnn.fit(x_training_data, y_training_data, epochs = 75, batch_size = 1)
```

```
Epoch 1/75
741/741 [==============================] - 2s 3ms/step - loss: 1.1496
Epoch 2/75
```

```
741/741 [==============================] - 2s 3ms/step - loss: 0.2798
Epoch 3/75
741/741 [==============================] - 2s 3ms/step - loss: 0.2537
Epoch 4/75
741/741 [==============================] - 2s 3ms/step - loss: 0.2228
Epoch 5/75
741/741 [==============================] - 2s 3ms/step - loss: 0.2030
Epoch 6/75
741/741 [==============================] - 2s 3ms/step - loss: 0.1848
Epoch 7/75
741/741 [==============================] - 2s 3ms/step - loss: 0.1808
Epoch 8/75
741/741 [==============================] - 2s 3ms/step - loss: 0.1546
Epoch 9/75
741/741 [==============================] - 2s 3ms/step - loss: 0.1574
Epoch 10/75
741/741 [==============================] - 2s 3ms/step - loss: 0.1439
Epoch 11/75
741/741 [==============================] - 2s 3ms/step - loss: 0.1472
Epoch 12/75
741/741 [==============================] - 2s 3ms/step - loss: 0.1293
Epoch 13/75
741/741 [==============================] - 2s 3ms/step - loss: 0.1227
Epoch 14/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0976
Epoch 15/75
741/741 [==============================] - 2s 3ms/step - loss: 0.1040
Epoch 16/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0984
Epoch 17/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0970
Epoch 18/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0956
Epoch 19/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0889
Epoch 20/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0832
Epoch 21/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0809
Epoch 22/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0803
Epoch 23/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0802
Epoch 24/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0721
Epoch 25/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0750
Epoch 26/75
```

```
741/741 [==============================] - 2s 3ms/step - loss: 0.0670
Epoch 27/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0702
Epoch 28/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0656
Epoch 29/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0505
Epoch 30/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0590
Epoch 31/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0583
Epoch 32/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0538
Epoch 33/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0587
Epoch 34/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0521
Epoch 35/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0452
Epoch 36/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0439
Epoch 37/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0441
Epoch 38/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0430
Epoch 39/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0482
Epoch 40/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0405
Epoch 41/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0436
Epoch 42/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0428
Epoch 43/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0389
Epoch 44/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0372
Epoch 45/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0405
Epoch 46/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0418
Epoch 47/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0429
Epoch 48/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0365
Epoch 49/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0395
Epoch 50/75
```

```
741/741 [==============================] - 2s 3ms/step - loss: 0.0399
Epoch 51/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0384
Epoch 52/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0360
Epoch 53/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0349
Epoch 54/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0353
Epoch 55/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0394
Epoch 56/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0341
Epoch 57/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0373
Epoch 58/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0395
Epoch 59/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0377
Epoch 60/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0361
Epoch 61/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0400
Epoch 62/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0339
Epoch 63/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0386
Epoch 64/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0388
Epoch 65/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0333
Epoch 66/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0375
Epoch 67/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0352
Epoch 68/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0356
Epoch 69/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0352
Epoch 70/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0368
Epoch 71/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0327
Epoch 72/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0340
Epoch 73/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0338
Epoch 74/75
```

```
741/741 [==============================] - 2s 3ms/step - loss: 0.0357
Epoch 75/75
741/741 [==============================] - 2s 3ms/step - loss: 0.0315
```

```
[24]: <tensorflow.python.keras.callbacks.History at 0x2a5c3123c70>
```

You should see the loss function's value slowly decline as the recurrent neural network is fitted to the training data over time. In my case, the loss function's value declined from 1.2363 in the first iteration to 0.0333 in the last iteration.

We have built our recurrent neural network and trained it on data of the fuel price over the last 5 years. It's now time to make some predictions!

```
[25]: testing_data.shape
```

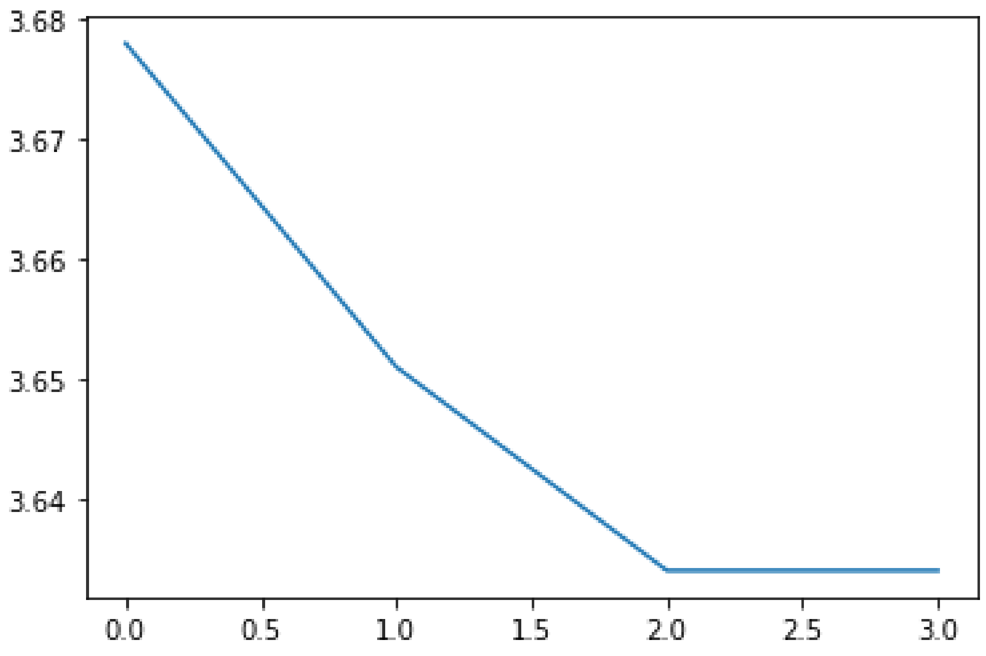
```
[25]: (4, 1)
```

If you run the statement print(test_data.shape), it will return (4,). This shows that our test data is a one-dimensional NumPy array with 4 entries - which means there were 4 weeks .

You can also generate a quick plot of the data using plt.plot(test_data).

```
[26]: plt.plot(testing_data)
```

```
[26]: [<matplotlib.lines.Line2D at 0x2a5cec422e0>]
```



Before we can actually make predictions for the fuel price , we first need to make some changes to our data set.

The reason for this is that to predict each of the 4 test observations, we will need the 4 previous weeks. Some of these weeks will come from the test set while the remainder will come from the training set. Because of this, some concatenation is necessary.

Now we can concatenate together the Open column from each DataFrame with the following statement:

```
[27]: fueldata.shape
```

```
[27]: (749, 2)
```

```
[28]: x_test_data = fueldata[len(fueldata) - 8:]
      x_test_data = x_test_data.iloc[:, 1].values
      x_test_data.shape
```

```
[28]: (8,)
```

You can check whether or not this object has been created as desired by printing len(x_test_data), which has a value of 8. This makes sense - it should contain the 4 values for testing as well as the 4 values prior.

```
[29]: len(x_test_data)
```

```
[29]: 8
```

The last step of this section is to quickly reshape our NumPy array to make it suitable for the predict method:

```
[30]: x_test_data = np.reshape(x_test_data, (-1, 1))
```

The last thing we need to do is group our test data into 4 arrays of size 4. Said differently, we'll now create an array where each entry corresponds to and contains the fuel prices of the 4 previous weeks.

```
[31]: final_x_test_data = []

      for i in range(4, len(x_test_data)):

          final_x_test_data.append(x_test_data[i-4:i, 0])

      final_x_test_data = np.array(final_x_test_data)
```

Lastly, we need to reshape the final_x_test_data variable to meet TensorFlow standards. We saw this previously, so the code should need no explanation:

```
[32]: final_x_test_data = np.reshape(final_x_test_data, (final_x_test_data.shape[0],␣
       ↪final_x_test_data.shape[1],1))
```

After an absurd amount of data reprocessing, we are now ready to make predictions using our test data!

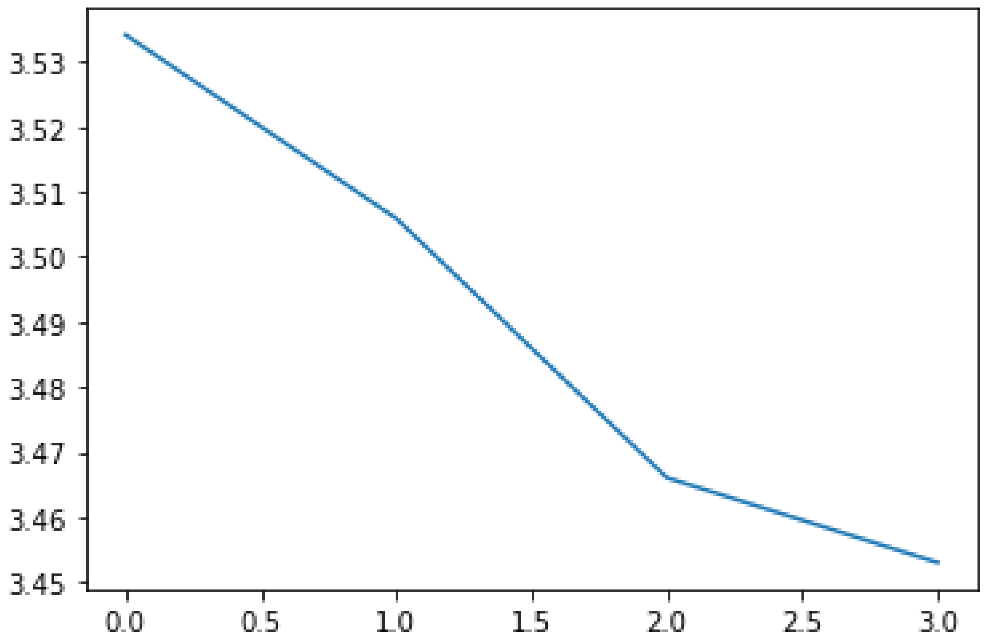
This step is simple. Simply pass in our final_x_test_data object into the predict method called on the rnn object. As an example, here is how you could generate these predictions and store them in an aptly-named variable called predictions:

[33]:
```
predictions = rnn.predict(final_x_test_data)
```

Let's plot these predictions by running plt.plot(predictions)

[34]:
```
plt.plot(predictions)
```

[34]:
```
[<matplotlib.lines.Line2D at 0x2a5d18400a0>]
```



Let's generate plot that compares our fuel prices with the actual fuel price:

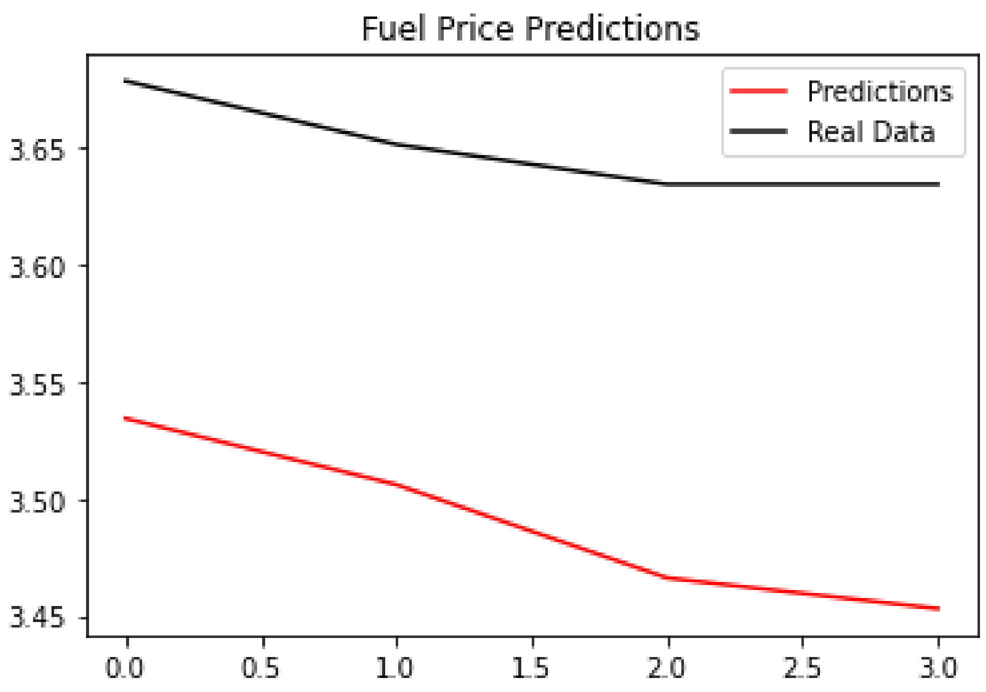
[35]:
```
plt.plot(predictions, color = 'red', label = "Predictions")

plt.plot(testing_data, color = 'black', label = "Real Data")

plt.title('Fuel Price Predictions')

plt.legend()
```

[35]:
```
<matplotlib.legend.Legend at 0x2a5d18784f0>
```



```
[36]: print("The first fuel price prediction is: ",predictions[0])
      print("The real first fuel price is: ",testing_data[0])
```

```
The first fuel price prediction is:  [3.5341415]
The real first fuel price is:  [3.678]
```

```
[37]: print("The last fuel price prediction is: ",predictions[3])
      print("The real fuel price0 is: ",testing_data[3])
```

```
The last fuel price prediction is:  [3.4530792]
The real fuel price0 is:  [3.634]
```

```
[ ]:
```