Natural Gas Price Forecasting Using Deep Learning

Kanishk Kumar

4th August, 2022

Table of Contents

Objective

This project will be focused on **prediction** using a Long Short-Term Memory (LSTM) model. The main objectives of this project are as follows:

- To apply data preprocessing and preparation techniques in order to obtain clean and stationary data.
- To build at least three variations of an LSTM model with patterns of different types and number of layers to improve its performance and accuracy in forecasting gas prices.
- To analyze and compare performance of each model variation in order to choose the best hyper-parameters and layer patterns.

Data Description

We will be using Natural Gas Price Data retrieved from Kaggle.com. It contains gas price in dollars per million British thermal unit, starting from 7th January, 1997 to 1st March, 2022.

```
data = pd.read_csv("data/ngpf_data.csv")
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6321 entries, 0 to 6320
Data columns (total 2 columns):
 #   Column                         Non-Null Count  Dtype
---  ------                         --------------  -----
 0   Day                            6321 non-null   object
 1   Price in Dollars per Million Btu  6320 non-null   float64
dtypes: float64(1), object(1)
memory usage: 98.9+ KB
```

Data Preprocessing & EDA

First, I imported the necessary libraries as shown in the figure on the right.

```
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import plotly.express as px
import tensorflow as tf
import statsmodels.api as sm
import plotly.graph_objects as go
from tensorflow.keras import layers, models
from sklearn.metrics import mean_squared_error, r2_score
from statsmodels.tsa.stattools import adfuller
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from plotly.subplots import make_subplots
```

Image below shows our data frame looks like:

```
data = pd.read_csv("data/ngpf_data.csv")
data.head()
```

	Day	Price in Dollars per Million Btu
0	1/3/2022	4.36
1	28/2/2022	4.46
2	25/2/2022	4.63
3	24/2/2022	4.78
4	23/2/2022	4.59

Then I converted the 'Day' column into a datetime format.

```
data['Day'] = pd.to_datetime(data['Day'],
                             format = "%d/%m/%Y")
data
```

	Day	Price in Dollars per Million Btu
0	2022-03-01	4.36
1	2022-02-28	4.46
2	2022-02-25	4.63
3	2022-02-24	4.78
4	2022-02-23	4.59
...	...	...
6316	1997-01-13	4.00
6317	1997-01-10	3.92
6318	1997-01-09	3.61
6319	1997-01-08	3.80
6320	1997-01-07	3.82

As can be seen, there are 6321 rows × 2 columns, data is from 7th January, 1997 to 1st March, 2022 with 6321 records.

I renamed the columns and set dates as index to make the next steps easier:

```
data = data.rename({'Day': 'date',
                    'Price in Dollars per Million Btu': 'gas_price'},
                   axis = 1)
data = data.set_index('date')
```

⇨

```
data.tail(3)
```

date	gas_price
1997-01-09	3.61
1997-01-08	3.80
1997-01-07	3.82

Made sure our data doesn't have any null values:

```
print(data.isnull().sum())
```

```
gas_price    1
dtype: int64
```

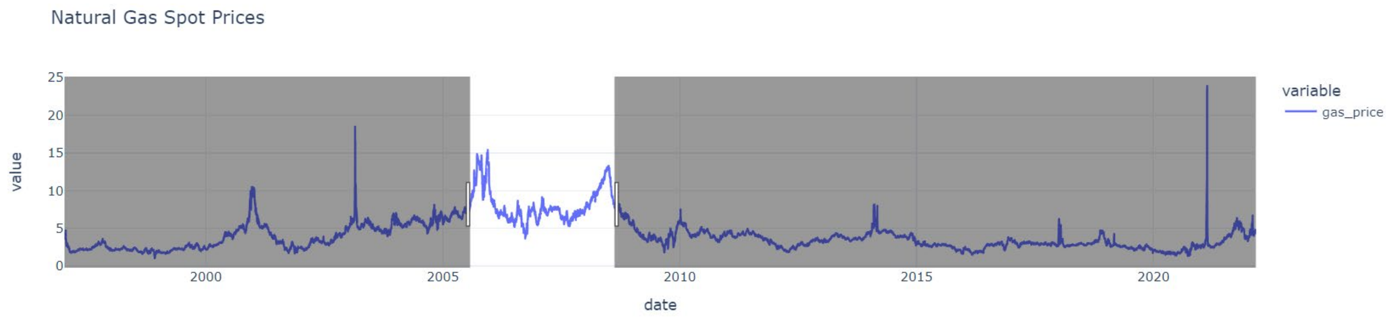
Looks like there was one missing value in the data, so I filled it with previous day's price:

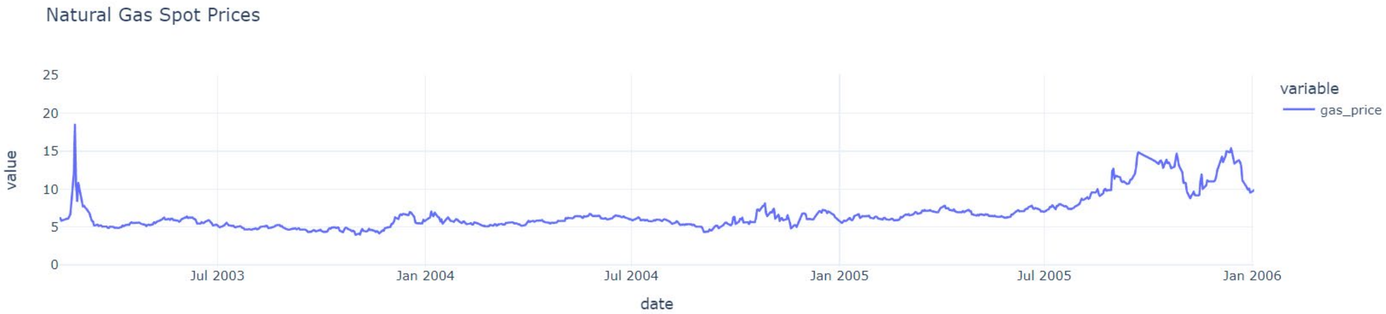
```
data = data.fillna(method = 'pad')
print(data.isnull().sum())
```

```
gas_price    0
dtype: int64
```

Then ploted an interactive line plot to explore our data further:

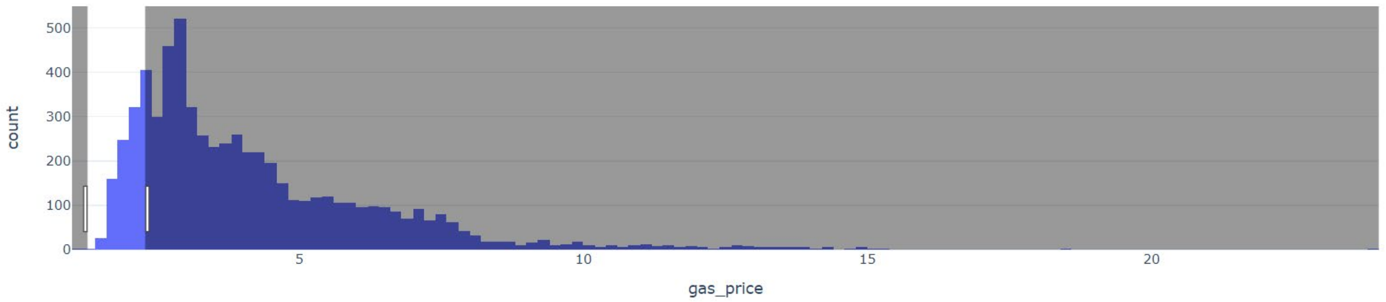
```
fig = px.line(data, title = 'Natural Gas Spot Prices', template = 'plotly_white')
fig.show()
```





An interactive histogram to see the most frequent prices throughout those years:

```
fig = px.histogram(data, x = "gas_price",
                   template = 'plotly_white')
fig.show()
```



Then I checked if the timeseries is stationary or not. A timeseries is said to be stationary if its statistical properties such as mean and variance remain constant over time. I used two methods to see if it is stationary or not: Rolling Stats & Dickey-Fuller Test.

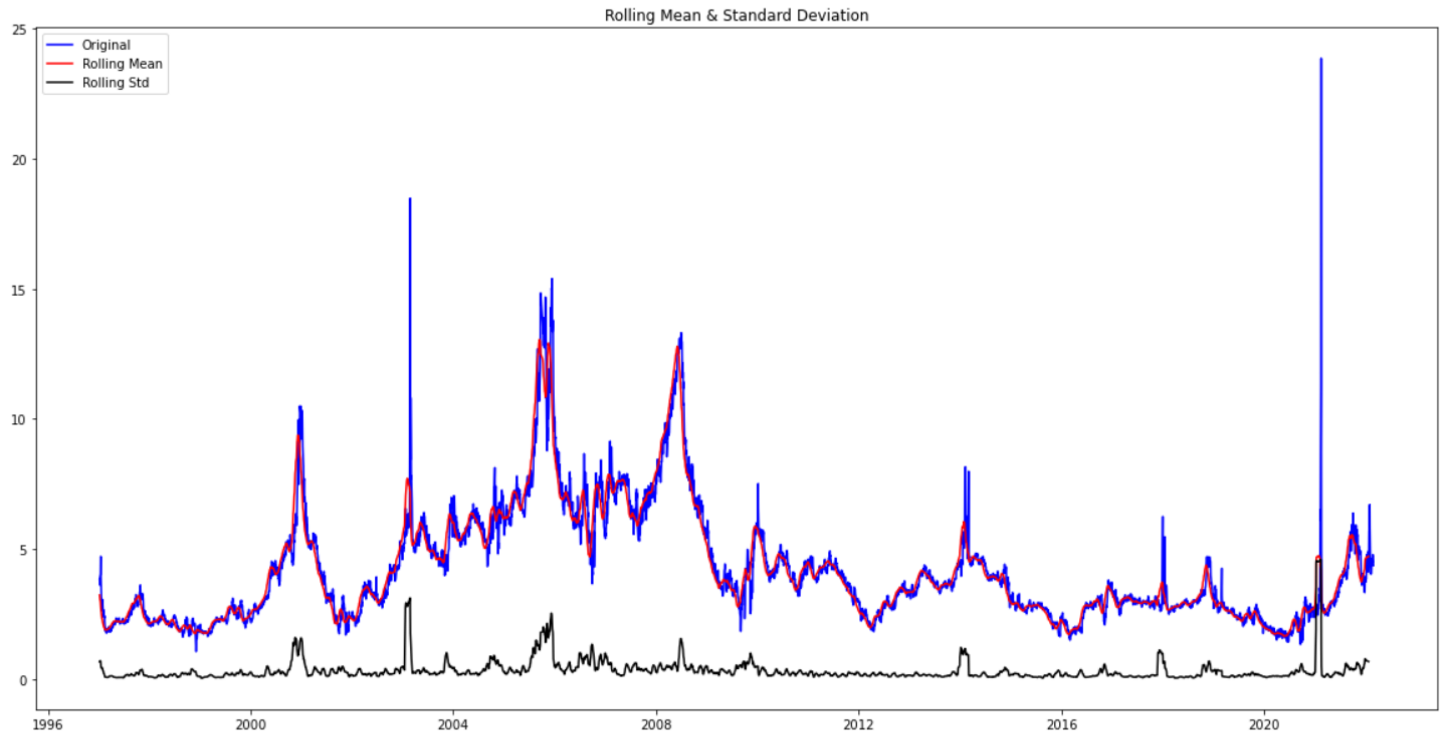
Null Hypothesis: Timeseries is non-Stationary.

Checking Stationarity

Then I used the code on the right to plot the Rolling Stats and do the Dickey-Fuller Test.

```
def test_stationarity(timeseries):

    # Determing rolling statistics
    rolmean = timeseries.rolling(25).mean()
    rolstd = timeseries.rolling(25).std()

    # Plot rolling statistics:
    plt.figure(figsize = (20,10))
    orig = plt.plot(timeseries, color='blue',label='Original')
    mean = plt.plot(rolmean, color='red', label='Rolling Mean')
    std = plt.plot(rolstd, color='black', label = 'Rolling Std')
    plt.legend(loc='best')
    plt.title('Rolling Mean & Standard Deviation')
    plt.show(block=False)

    # Perform Dickey-Fuller test:
    print('Results of Dickey-Fuller Test:')
    dftest = adfuller(timeseries, autolag = 'AIC')
    dfoutput = pd.Series(dftest[0:4],
                         index = ['Test Statistic',
                                  'p-value',
                                  '#Lags Used',
                                  'Number of Observations Used'])
    for key,value in dftest[4].items():
        dfoutput['Critical Value (%s)'%key] = value

    print(dfoutput)
```

```
Results of Dickey-Fuller Test:
Test Statistic                  -3.867920
p-value                          0.002284
#Lags Used                       8.000000
Number of Observations Used   6312.000000
Critical Value (1%)             -3.431386
Critical Value (5%)             -2.861998
Critical Value (10%)            -2.567014
dtype: float64
```



Test Statistics shows how closely our observed data matches the distribution expected under the null hypothesis of that statistical test. Since our Test Statistic is less than Critical Value & p-value is less than 0.05, we'll reject the Null Hypothesis. Our data is Stationary.

However, there are 2 major reasons behind non-stationarity of a timeseries:
Trend – Varying mean over time.
Seasonality – Variations at specific time-frames.

Since our timeseries is non-stationary, I'll estimate the trend and seasonality in it and remove it. Then I'll convert the forecasted values back into the original scale by applying trend and seasonality constraints back. Higher test statistic value would result in more trend in pattern.

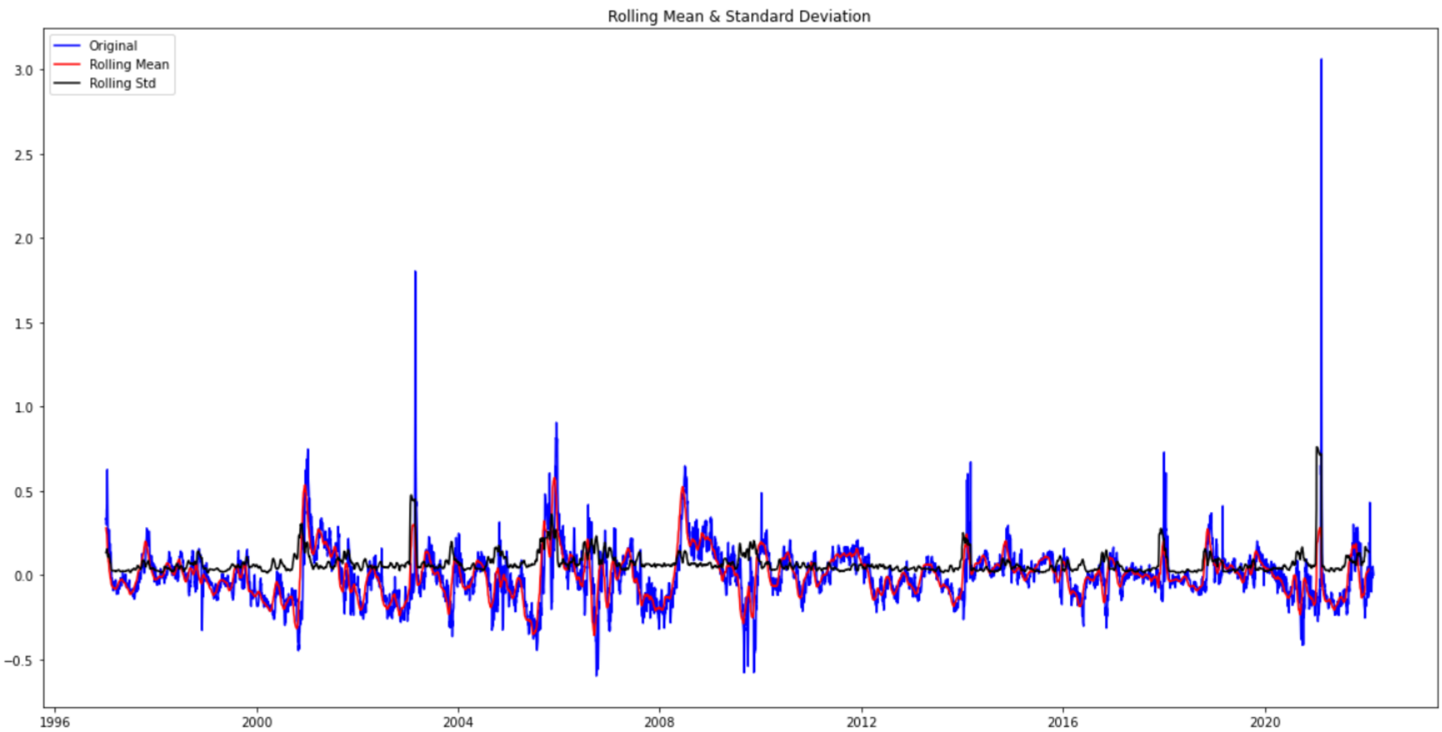
Estimating and Eliminating Trends

Simple Trend Reduction Techniques are:

- Moving Average
- Exponential Weighted Moving Average.

I applied only the second one using the code on the right. You can see the results below:

```
ts_sqrt = np.sqrt(data)
expwighted_avg = ts_sqrt.ewm(halflife = 25).mean()

ts_sqrt_ewma_diff = ts_sqrt - expwighted_avg
test_stationarity(ts_sqrt_ewma_diff)
```



Test Statistic was still pretty close to the Critical Value.

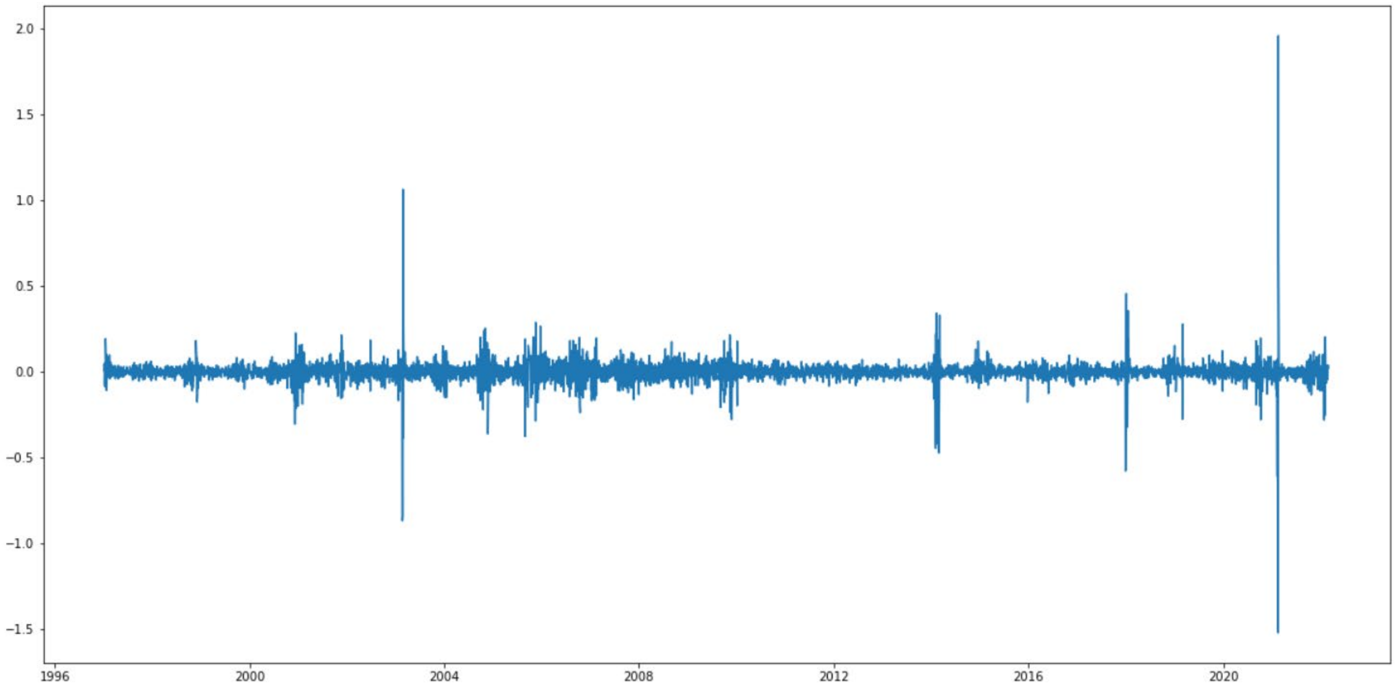
```
Results of Dickey-Fuller Test:
Test Statistic                -1.125358e+01
p-value                        1.688509e-20
#Lags Used                     1.000000e+01
Number of Observations Used    6.310000e+03
Critical Value (1%)           -3.431387e+00
Critical Value (5%)           -2.861998e+00
Critical Value (10%)          -2.567014e+00
dtype: float64
```

The simple trend reduction techniques don't work with high seasonality. So I can further used the following two methods:

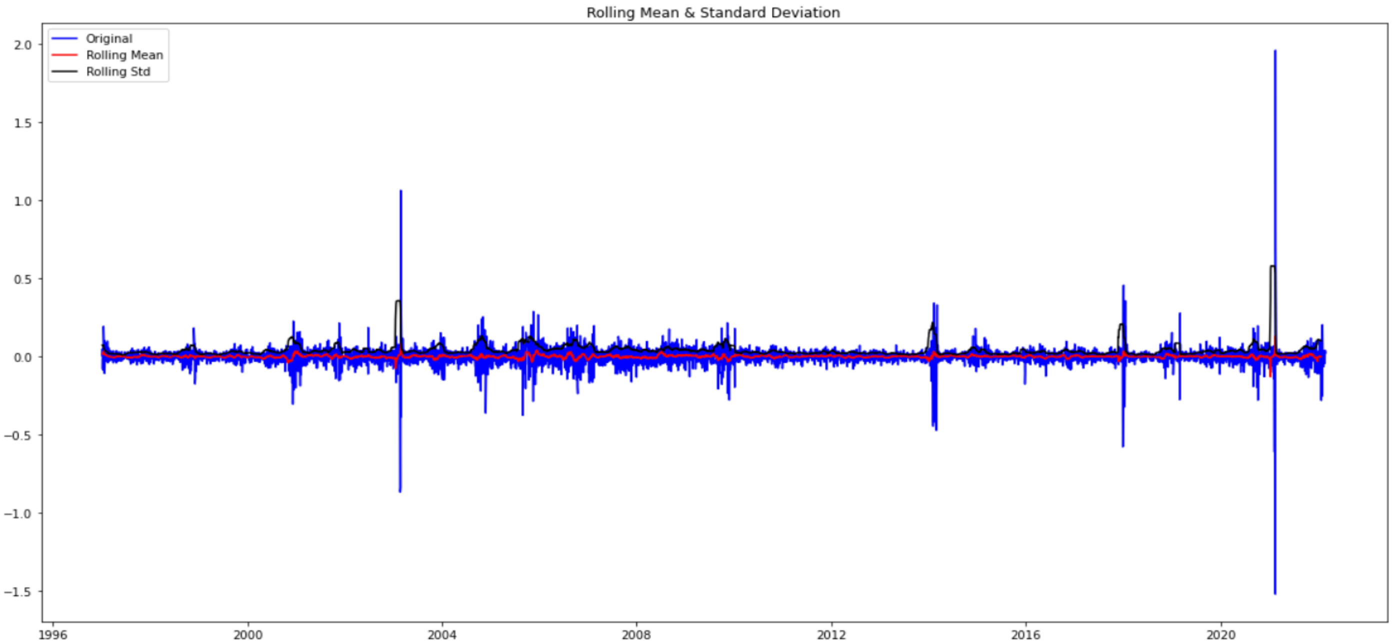
- Differencing – Taking the difference with a particular Time Lag. Subtracting Current observation from previous.
- Decomposition – Additive in this case. Modeling both trend and seasonality and removing them from the model.

```
ts_sqrt_diff = ts_sqrt - ts_sqrt.shift()
ts_sqrt_diff.dropna(inplace = True)

plt.figure(figsize = (20,10))
plt.plot(ts_sqrt_diff)
plt.show()
```

I applied only the first one using the code on the right. The resulting plot is shown below:



Then I tested stationarity again: `test_stationarity(ts_sqrt_diff)`



The Test Statistic got significantly lower than the Critical Value and also there was less diversion in mean and standard deviation. This is a perfect stationary timeseries.

```
Results of Dickey-Fuller Test:
Test Statistic                  -25.917790
p-value                           0.000000
#Lags Used                       11.000000
Number of Observations Used    6308.000000
Critical Value (1%)              -3.431387
Critical Value (5%)              -2.861998
Critical Value (10%)             -2.567014
dtype: float64
```

Data Modeling

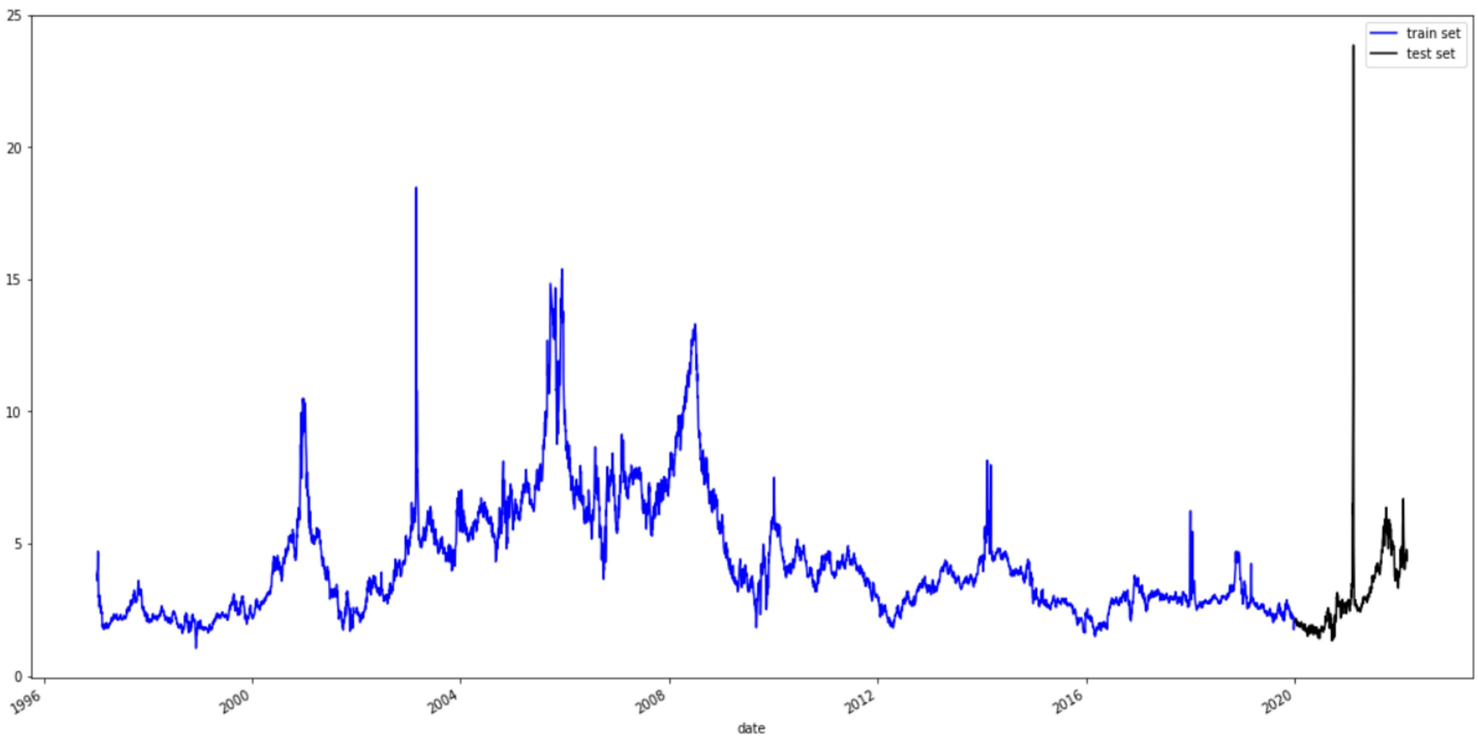
Then I split our data into training and test sets. Training set included data from 7th of January, 1997 to 6th of January, 2020 and test set included from 7th of January, 2021 to 1st of March, 2022.

```
data = data.sort_values(by = 'date')

train = data['1997-01-06': '2020-01-06']
test = data['2020-01-07': '2022-03-01']

print("Length of Train Data: ", len(train))
print("Length of Test Data: ", len(test))
```

```
Length of Train Data:  5783
Length of Test Data:  538
```

Length of training set is 5783 and that of test set is 538. I used the code on the right to plot them. Result is shown below:

```
ax = train.plot(figsize = (20, 10), color = 'b')
test.plot(ax = ax, color = 'black')
plt.legend(['train set', 'test set'])
plt.show()
```



I used the code on the right to make a sample gap (slot) of 15 step size between start of train sequences.

Then reshaped the data so that our models can understand it.

```
slot = 15
x_train = []
y_train = []
for i in range(slot, len(train)):
    x_train.append(train.iloc[i-slot:i, 0])
    y_train.append(train.iloc[i, 0])
x_train, y_train = np.array(x_train), np.array(y_train)
x_train = np.reshape(x_train,
                     (x_train.shape[0],
                      x_train.shape[1], 1))
print(x_train.shape, y_train.shape)
```

Model 1: A Simple LSTM

I made a very simplistic LSTM deep learning model on our data. For all our models I've used Tensorflow library with Keras interface.

```
lstm_model = tf.keras.Sequential()
lstm_model.add(tf.keras.layers.LSTM(10, input_shape = (slot, 1)))
lstm_model.add(tf.keras.layers.Dense(1))

lstm_model.compile(loss = 'mean_squared_error',
                   optimizer = 'adam', metrics = ["mse"])

lstm_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 10)                480

 dense (Dense)               (None, 1)                 11

=================================================================
Total params: 491
Trainable params: 491
Non-trainable params: 0
_________________________________________________________________
```

I chose one LSTM layer with 10 units dimensionality of the output space and a plain dense layer which has output shape as 1.

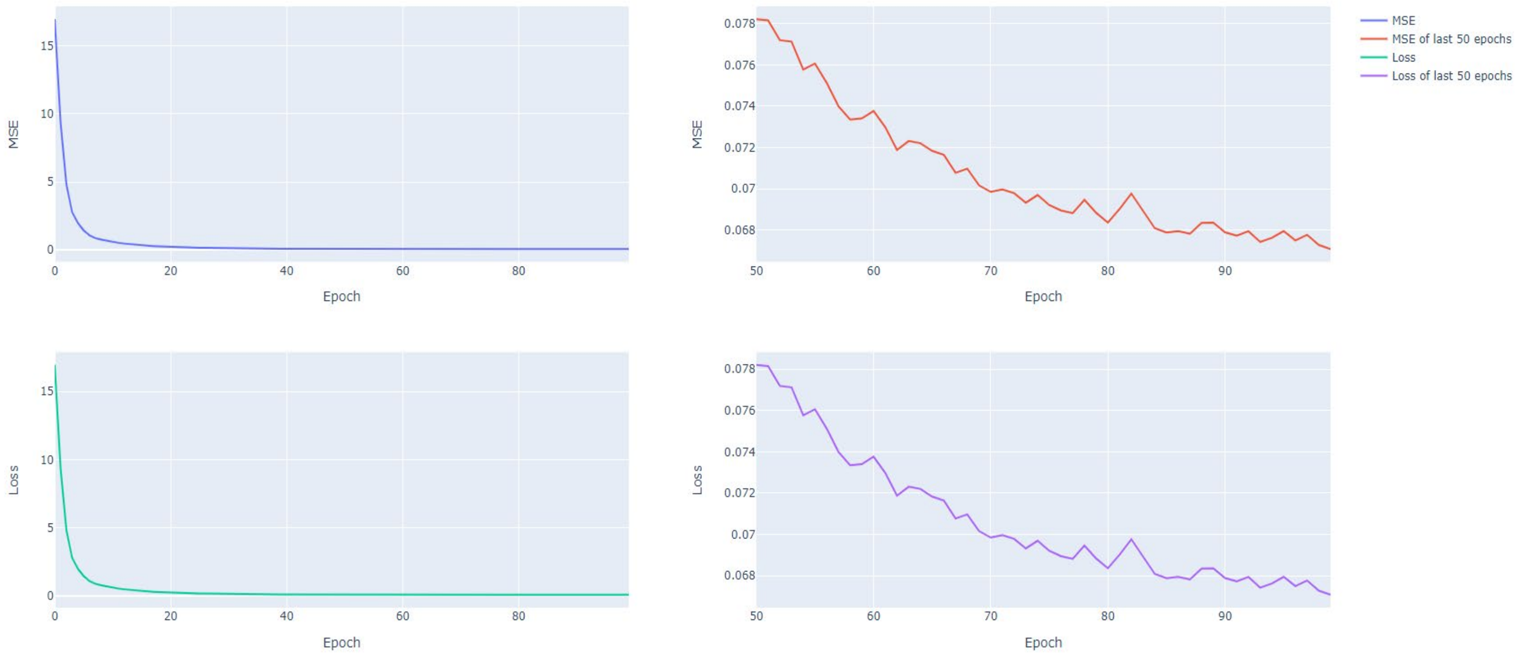
As can be seen it is quite a simple model with a total of 491 parameters.

While training, I also used Keras's EarlyStopping class to stop training when a monitored metric has stopped improving, 'Model Loss' in our case.

```
early_stopping = tf.keras.callbacks.EarlyStopping(monitor = 'loss',
                                                  patience = 10)
epochs = 100
history = lstm_model.fit(x_train, y_train, epochs = epochs,
                         batch_size = 64,
                         verbose = 1,
                         callbacks = [early_stopping])
```

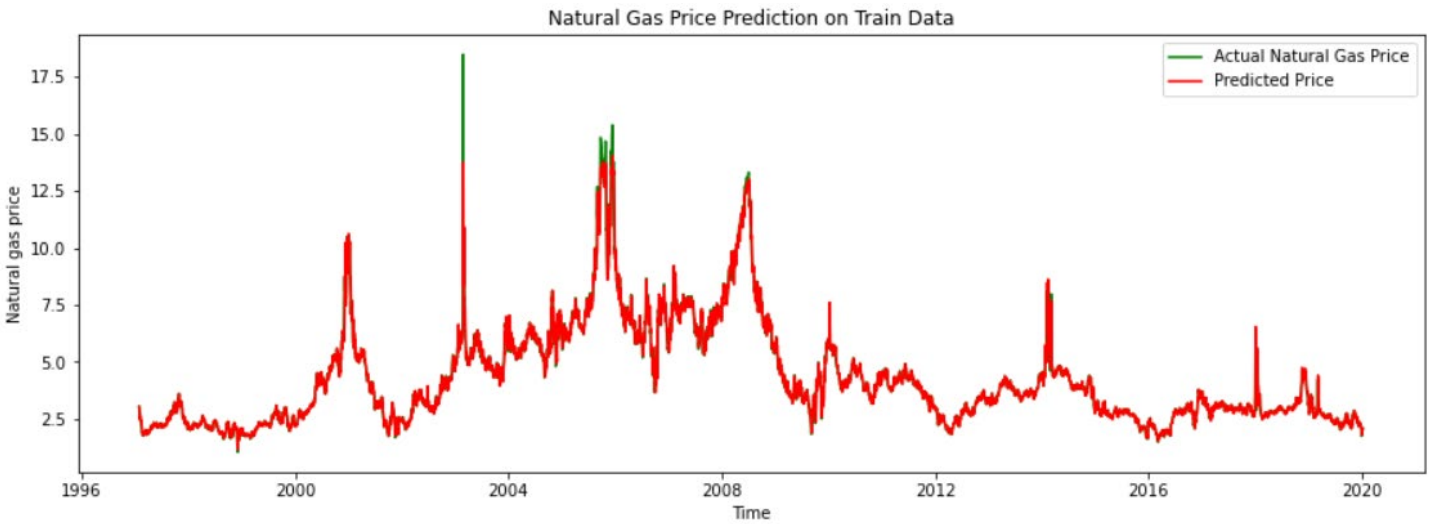
Then I plotted Training Curve for MSE and Model Loss to see how our training went.

Training Curve for MSE and Model Loss



Evaluated our model on the training data using the code on the right here and plotted it. Result shown below.

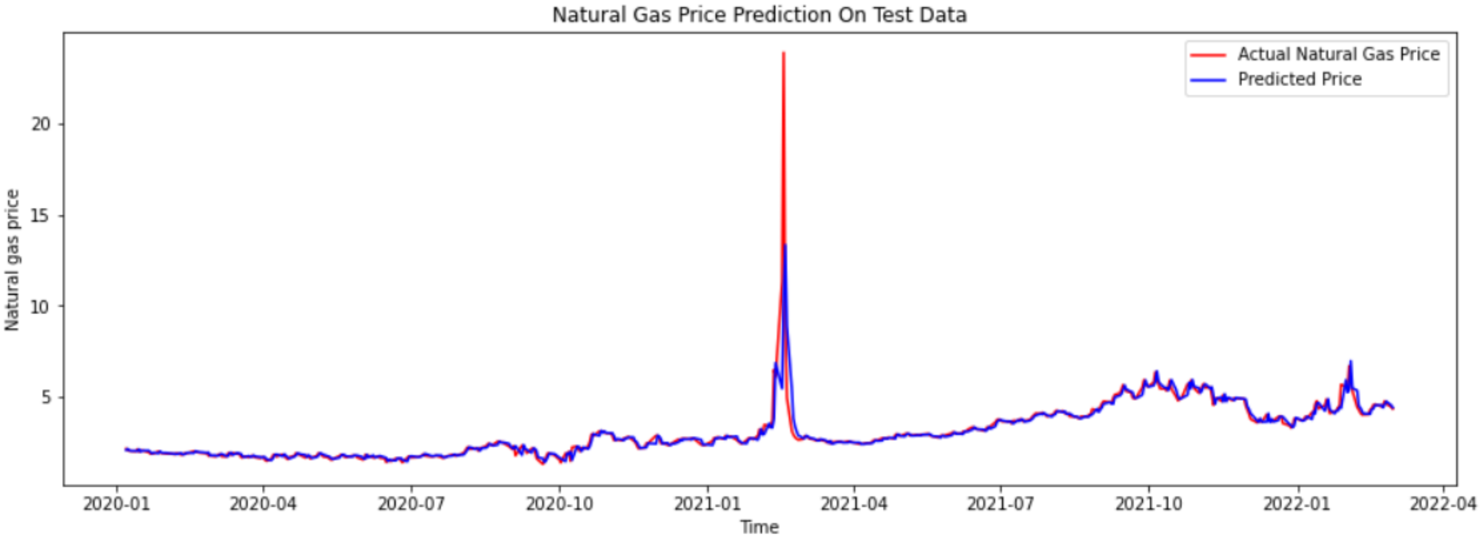
```python
yp_train = lstm_model.predict(x_train)
a = pd.DataFrame(yp_train)
a.rename(columns = {0: 'gp_pred'},
          inplace = True);
a.index = train.iloc[slot:].index
train_compare = pd.concat([train.iloc[slot:], a],
                          axis = 1)
```



Evaluated on the test data using the code below and plotted. Results shown on the next page:

```python
dataset_total = pd.concat((train, test), axis = 0)
inputs = dataset_total[len(dataset_total) - len(test)- slot:].values
inputs = inputs.reshape(-1, 1)

x_test = []
y_test = []
for i in range (slot, len(test)+slot): #Test+15
    x_test.append(inputs[i-slot:i, 0])
    y_test.append(train.iloc[i, 0])

x_test, y_test = np.array(x_test), np.array(y_test)

x_test = np.reshape(x_test, (x_test.shape[0], x_test.shape[1], 1))
pred_price = lstm_model.predict(x_test, verbose = 0)
```

```python
b = pd.DataFrame(pred_price)
b.rename(columns = {0: 'gp_pred'},
          inplace = True);
b.index = test.index
test_compare = pd.concat([test, b],
                         axis = 1)
```



I got a mean squared error of 0.53 on our test data and R Square of 0.79.

Not bad for such a simplistic model.

```
Train Data:
MSE: 0.07
R Square: 0.99

Test Data:
MSE: 0.53
R Square: 0.79
```

```
mse_train = mean_squared_error(train_compare['gas_price'],
                               train_compare['gp_pred'])
mse_test = mean_squared_error(test_compare['gas_price'],
                              test_compare['gp_pred'])

r2_train = r2_score(train_compare['gas_price'],
                    train_compare['gp_pred'])
r2_test = r2_score(test_compare['gas_price'],
                   test_compare['gp_pred'])

print("Train Data:\nMSE: {}\nR Square: {}".format(round(mse_train, 2),
                                                  round(r2_train, 2)))
print("\nTest Data:\nMSE: {}\nR Square: {}".format(round(mse_test, 2),
                                                   round(r2_test, 2)))
```

Model 2: Adding More Layers and Tuning

Then I tried to improve our model by adding three more LSTM layers with 50 units dimensionality of the output space and a plain dense layer which has output shape as 1.

```
lstm_model = tf.keras.Sequential()
lstm_model.add(tf.keras.layers.LSTM(units = 50, input_shape = (slot, 1), return_sequences = True, activation = 'tanh'))
lstm_model.add(tf.keras.layers.LSTM(units = 50, activation = 'tanh', return_sequences = True))
lstm_model.add(tf.keras.layers.LSTM(units = 50, return_sequences = True))
lstm_model.add(tf.keras.layers.LSTM(units = 50, return_sequences = False))
lstm_model.add(tf.keras.layers.Dense(units = 1))

lstm_model.compile(loss = 'mean_squared_error', optimizer = 'adam', metrics = ["mse"])

lstm_model.summary()
```

For the first three layers I selected ‘return_sequences’ as True to return the full sequence for the next LSTM layer.

Also, I went with 'tanh' activation because then I can use the CUDA cores of my GPU.

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_1 (LSTM)               (None, 15, 50)            10400

 lstm_2 (LSTM)               (None, 15, 50)            20200

 lstm_3 (LSTM)               (None, 15, 50)            20200

 lstm_4 (LSTM)               (None, 50)                20200

 dense_1 (Dense)             (None, 1)                 51

=================================================================
Total params: 71,051
Trainable params: 71,051
Non-trainable params: 0
_________________________________________________________________
```

As can be seen it is now a very complex model with a total of 71,051 parameters.

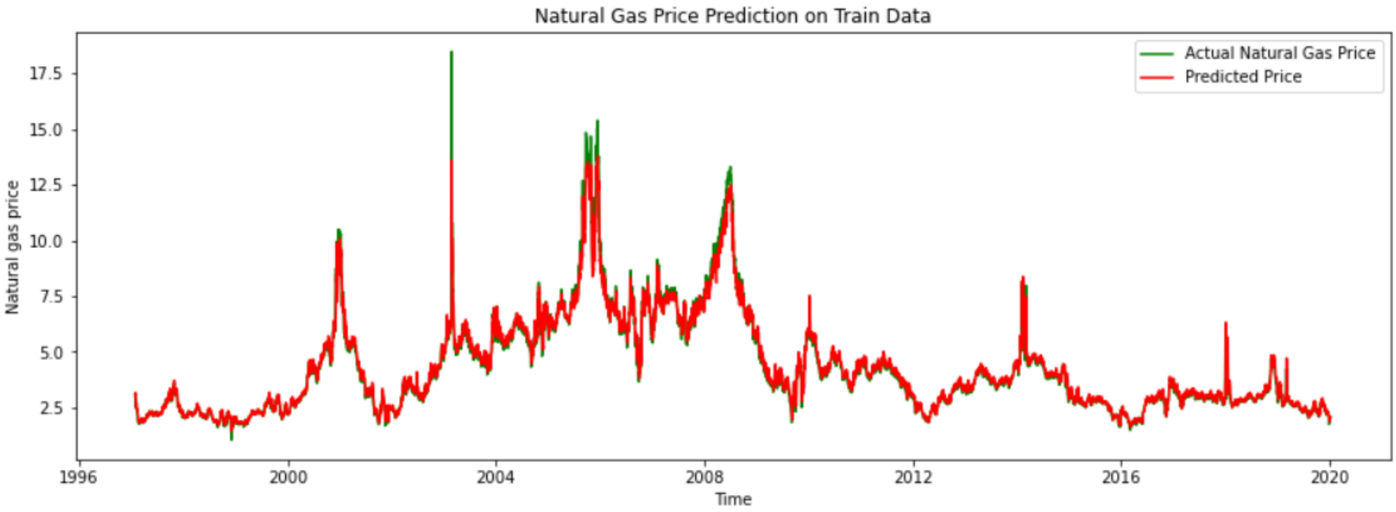
Like before, while training I also used Keras's EarlyStopping class to stop training when a monitored metric has stopped improving, 'Model Loss' in our case.

```
early_stopping = tf.keras.callbacks.EarlyStopping(monitor = 'loss',
                                                  patience = 10)
epochs = 100
history = lstm_model.fit(x_train, y_train, epochs = epochs,
                         batch_size = 64,
                         verbose = 1,
                         callbacks = [early_stopping])
```

Then I plotted Training Curve for MSE and Model Loss to see how our training went.

Evaluated our model on the training data using using the code on the right here and plotted. Results shown on the next page.

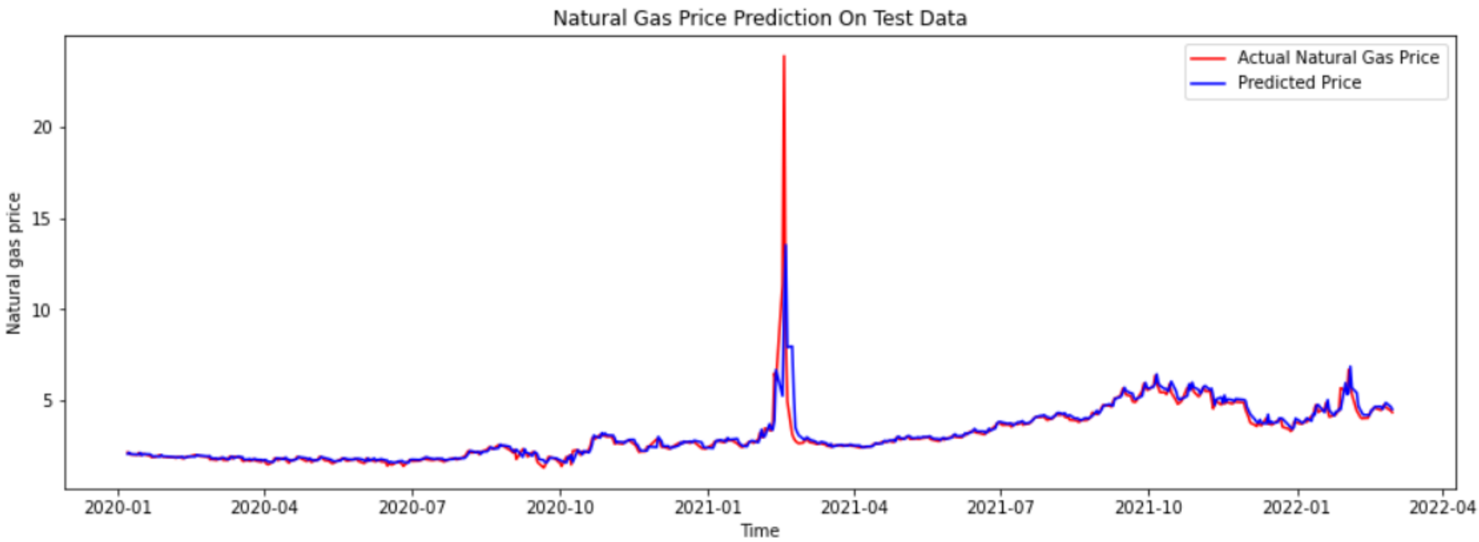
```
yp_train = lstm_model.predict(x_train)
a = pd.DataFrame(yp_train)
a.rename(columns = {0: 'gp_pred'},
         inplace = True);
a.index = train.iloc[slot:].index
train_compare = pd.concat([train.iloc[slot:], a],
                          axis = 1)
```



Evaluated on the test data using the codes presented here and plotted. Results shown below:

```
dataset_total = pd.concat((train, test), axis = 0)
inputs = dataset_total[len(dataset_total) - len(test)- slot:].values
inputs = inputs.reshape(-1, 1)

x_test = []
y_test = []
for i in range (slot, len(test)+slot): #Test+15
    x_test.append(inputs[i-slot:i, 0])
    y_test.append(train.iloc[i, 0])

x_test, y_test = np.array(x_test), np.array(y_test)

x_test = np.reshape(x_test, (x_test.shape[0], x_test.shape[1], 1))
pred_price = lstm_model.predict(x_test, verbose = 0)
```

```
b = pd.DataFrame(pred_price)
b.rename(columns = {0: 'gp_pred'},
         inplace = True);
b.index = test.index
test_compare = pd.concat([test, b],
                         axis = 1)
```



I got a mean squared error of 0.58 on our test data and R Square of 0.76. Looks like I have over fitted this model.

```
Train Data:
MSE: 0.08
R Square: 0.98

Test Data:
MSE: 0.58
R Square: 0.76
```

```
mse_train = mean_squared_error(train_compare['gas_price'],
                               train_compare['gp_pred'])
mse_test = mean_squared_error(test_compare['gas_price'],
                              test_compare['gp_pred'])

r2_train = r2_score(train_compare['gas_price'],
                    train_compare['gp_pred'])
r2_test = r2_score(test_compare['gas_price'],
                   test_compare['gp_pred'])

print("Train Data:\nMSE: {}\nR Square: {}".format(round(mse_train, 2),
                                                   round(r2_train, 2)))
print("\nTest Data:\nMSE: {}\nR Square: {}".format(round(mse_test, 2),
                                                    round(r2_test, 2)))
```

Model 3: Reducing LSTM Layers and Adding Dropout Layers

I tried to further improve our model by reducing LSTM layers down to 2 with 50 units dimensionality of the output space and added a dropout layer after each with 0.1 fraction of the input units to drop. Dropout is a regularization method where input and recurrent

```
lstm_model = tf.keras.Sequential()
lstm_model.add(tf.keras.layers.LSTM(units = 50, input_shape = (slot, 1), return_sequences = True, activation = 'tanh'))
lstm_model.add(tf.keras.layers.Dropout(0.1))
lstm_model.add(tf.keras.layers.LSTM(units = 50, activation = 'tanh', return_sequences = False))
lstm_model.add(tf.keras.layers.Dropout(0.1))
lstm_model.add(tf.keras.layers.Dense(units = 1))

lstm_model.compile(loss = 'mean_squared_error', optimizer = 'adam', metrics = ["mse"])

lstm_model.summary()
```

connections to LSTM units are probabilistically excluded from activation and weight updates while training a network. This has the effect of reducing overfitting and improving model performance.

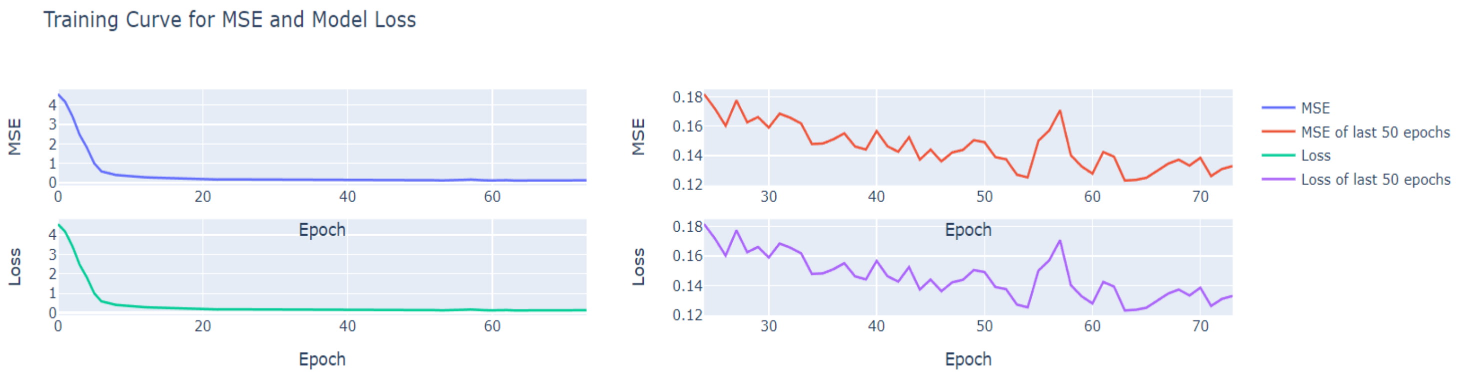
```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_5 (LSTM)               (None, 15, 50)            10400

 dropout (Dropout)           (None, 15, 50)            0

 lstm_6 (LSTM)               (None, 50)                20200

 dropout_1 (Dropout)         (None, 50)                0

 dense_2 (Dense)             (None, 1)                 51

=================================================================
Total params: 30,651
Trainable params: 30,651
Non-trainable params: 0
_________________________________________________________________
```

And of course, added a plain dense layer which has output shape as 1. I've kept the activation function as 'tanh' to utilize my GPU. As can be seen it is a significantly less complex model than the previous one with less than half the number of Total Parameters at 30,651.

Again, while training I also used Keras's EarlyStopping class to stop training when a monitored metric has stopped improving, 'Model Loss' in our case.

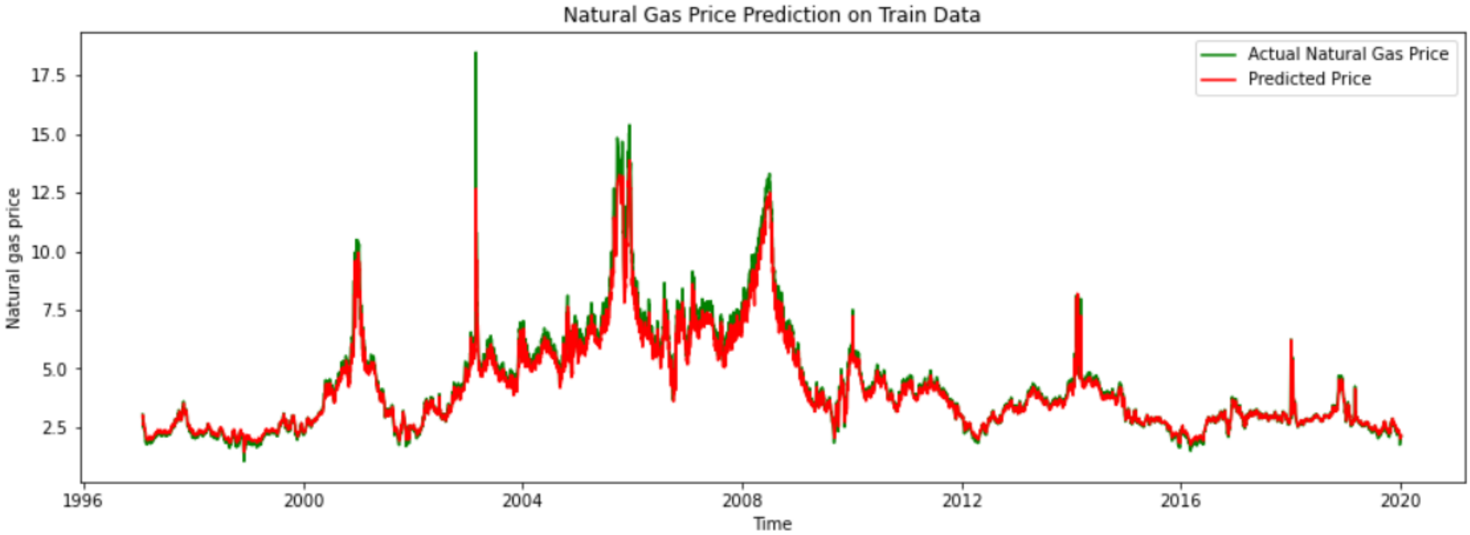
```
early_stopping = tf.keras.callbacks.EarlyStopping(monitor = 'loss',
                                                  patience = 10)
epochs = 100
history = lstm_model.fit(x_train, y_train, epochs = epochs,
                         batch_size = 64,
                         verbose = 1,
                         callbacks = [early_stopping])
```

You can see below how our training went:



Evaluated our model on the training data using the code on the right here and plotted. Results shown below.

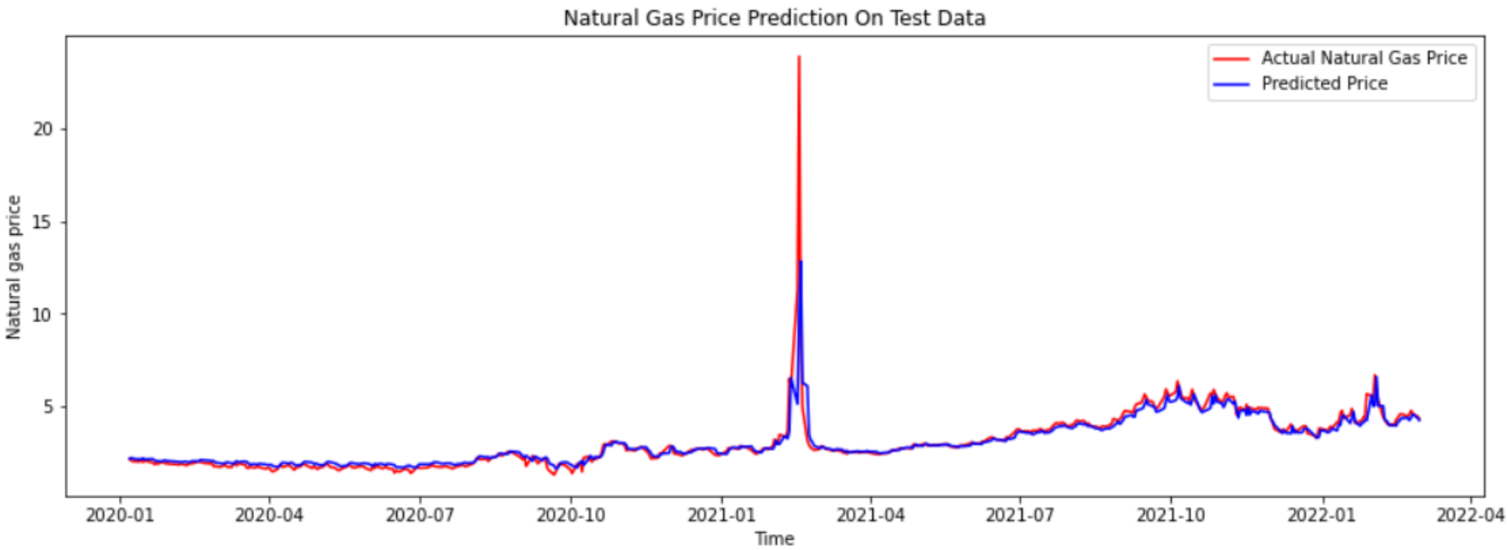
```
yp_train = lstm_model.predict(x_train)
a = pd.DataFrame(yp_train)
a.rename(columns = {0: 'gp_pred'},
         inplace = True);
a.index = train.iloc[slot:].index
train_compare = pd.concat([train.iloc[slot:], a],
                          axis = 1)
```



Evaluated on the test data using the codes presented here and plotted. Results shown below:

```
dataset_total = pd.concat((train, test), axis = 0)
inputs = dataset_total[len(dataset_total) - len(test)- slot:].values
inputs = inputs.reshape(-1, 1)

x_test = []
y_test = []
for i in range (slot, len(test)+slot): #Test+15
    x_test.append(inputs[i-slot:i, 0])
    y_test.append(train.iloc[i, 0])

x_test, y_test = np.array(x_test), np.array(y_test)

x_test = np.reshape(x_test, (x_test.shape[0], x_test.shape[1], 1))
pred_price = lstm_model.predict(x_test, verbose = 0)
```

```
b = pd.DataFrame(pred_price)
b.rename(columns = {0: 'gp_pred'},
         inplace = True);
b.index = test.index
test_compare = pd.concat([test, b],
                         axis = 1)
```



I got a mean squared error of 0.53 on our test data and R Square of 0.78. Better than last and just a little better than my first model. From my current understanding, this is the best I can do right now.

```
Train Data:
MSE: 0.13
R Square: 0.97

Test Data:
MSE: 0.53
R Square: 0.78
```

```
mse_train = mean_squared_error(train_compare['gas_price'],
                               train_compare['gp_pred'])
mse_test = mean_squared_error(test_compare['gas_price'],
                              test_compare['gp_pred'])

r2_train = r2_score(train_compare['gas_price'],
                    train_compare['gp_pred'])
r2_test = r2_score(test_compare['gas_price'],
                   test_compare['gp_pred'])

print("Train Data:\nMSE: {}\nR Square: {}".format(round(mse_train, 2),
                                                  round(r2_train, 2)))
print("\nTest Data:\nMSE: {}\nR Square: {}".format(round(mse_test, 2),
                                                   round(r2_test, 2)))
```

Forecasting

To conclude, let's use our third and final model to forecast gas prices for the next 15 days after the test data ends, i.e., from 2nd to 17th of March, 2022 using the code on the right:

```
forecast = pd.DataFrame({'date': pd.date_range(start = '3/2/2022',
                                                end = '3/17/2022')})
inputs = test[len(test) - slot:].values

for i in range(slot, len(forecast)):
    inputs = inputs.T
    inputs = np.reshape(inputs,
                        (inputs.shape[0],
                         inputs.shape[1], 1))
    pred_price = lstm_model.predict(inputs[:,i-slot:i],
                                    verbose=0)
    inputs = np.append(inputs, pred_price)
    inputs = np.reshape(inputs, (inputs.shape[0], 1))

forecast['gp_pred'] = inputs
forecast = forecast.set_index('date')

forecast.reset_index(inplace = True)

fig = px.line(forecast, x = "date", y = "gp_pred",
              title = 'Natural Gas Price Forecasting',
              template = 'plotly_white')
fig.show()
```

Seems our model is working fine.

Key Findings

As we can clearly see in our first model, if given enough stationary data, a very simplistic LSTM model can do forecasting as good as nearly possible. In our second model, we observed how prone to overfitting our model can get when adding more LSTM layers. We countered overfitting by reducing the number of LSTM layers and adding a dropout layer after each to regularize the data stream in our third model. Even then we only slightly improved our MSE score compared to the first model, again, showing how powerful just a single LSTM layer can be, given enough stationary data.

Advanced Steps

I don't think our final model needs any further improvement unless there is some change in the dataset. It'd be interesting to run these models with more data-points. I noticed while forecasting for more than 20 days, the forecast just becomes a straight horizontal line, I'd like to further understand what limiting factors are causing it and how to overcome. I'd also like to try other activation functions and understand why only the 'tanh' can utilize GPUs.