lab1

May 23, 2022

0.1 Importing the needed libraries

[ ]:
```python
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import tensorflow as tf
```

0.1.1 Set seed

1. For comparable results set the random seed in both numpy and tensorflow to (123). You can use this code:

[ ]:
```python
np.random.seed(123)
tf.random.set_seed(123)
```

```
2. If you don’t work in google colab, import the appropriate metric that will be used for opti

3. To download the csv file from github use !wget -nc https://lazyprogrammer.me/course_files/a
```

0.2 Importing the dataset

4. Create your data frame. To ensure that all works fine index the ‘Month’ column and parse the dates.

[ ]:
```python
#Because time series are ordered sequences we need to indicate the column␣
↪according to wich the rows will be indexed
#parse_date will put the column with dates in the year/month/day format
df = pd.read_csv('airline_passengers.csv', index_col='Month', parse_dates=True)
```

Check how the dataset looks

[ ]:
```python
df.head()
```

[ ]:
```
            Passengers
Month
1949-01-01         112
1949-02-01         118
1949-03-01         132
1949-04-01         129
```

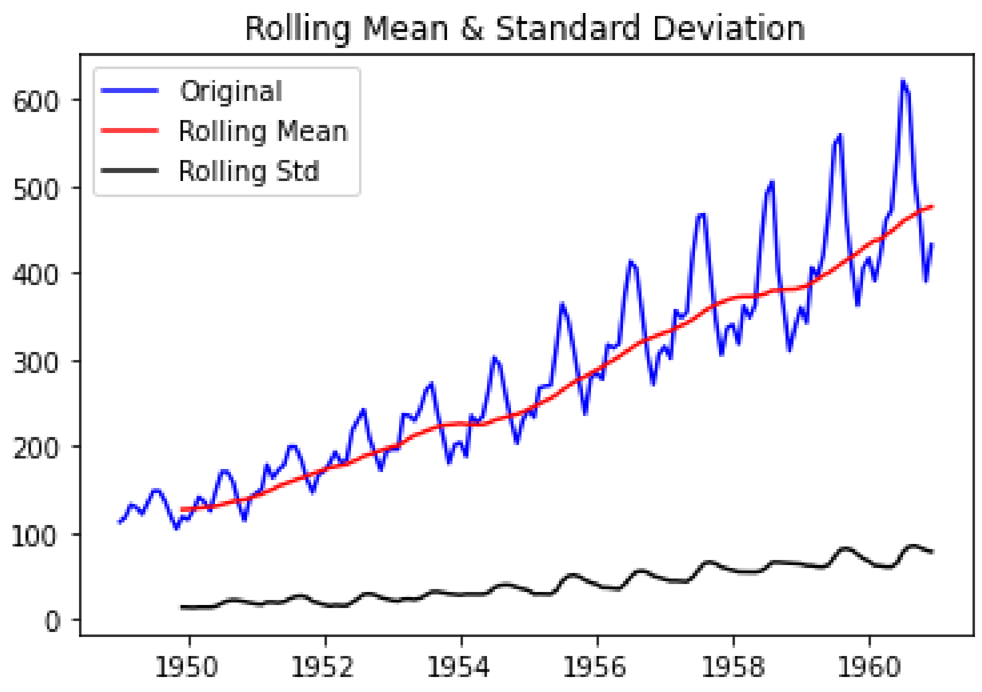
```
1949-05-01          121
```

5. Plot the data and check for stationarity. You can use the Dickey Fuller test to do this. Or if you would rather rely just on visual inspection, argument your conclusion on whether data are stationary or not. Remember, for stationarity you will want a constant variance (i.e. variance is not increasing/decreasing with time) and no trend in your data. If data proved to be non-stationary correct for this problem. (to correct for variance issues you can log transform the data, for trend, you can take the first difference).

0.2.1 Dickey Fuller test

[ ]:
```
# Dickey Fuller test
# The null hypothesis of the Dickey Fuller test is that there is a unit root in␣
↪the time series.
# The alternative hypothesis is that there is no unit root.
# The test statistic is the largest eigenvalue of the covariance matrix of the␣
↪series.
# The p-value is the probability of obtaining a test statistic as large as the␣
↪one that was actually observed,
# assuming that the true distribution of the test statistic is normal.
# The test statistic is approximately normally distributed if the p-value is␣
↪small.
```

[ ]:
```
from statsmodels.tsa.stattools import adfuller


def test_stationarity(timeseries):
    # rolling statistics
    rolmean = timeseries.rolling(window=12).mean()
    rolstd = timeseries.rolling(window=12).std()

    # plot rolling statistics:
    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
    return dfoutput

test_result = test_stationarity(df['Passengers'])
print(test_result)

if test_result['p-value'] < 0.05:
    print('The data is stationary')
else:
    print('The data is not stationary')
```



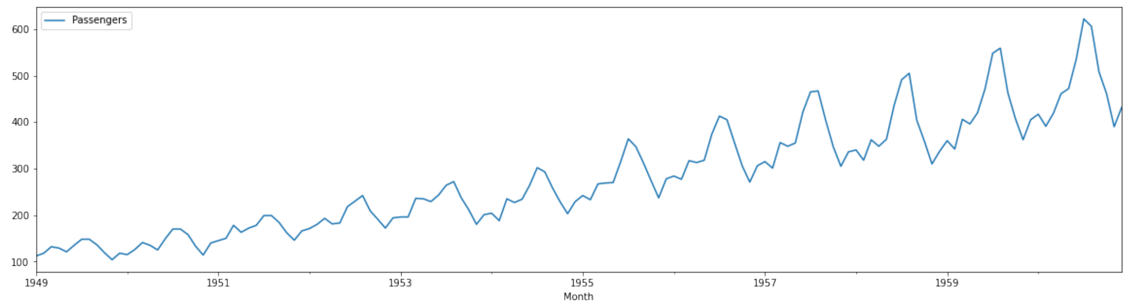
```
Results of Dickey-Fuller Test:
Test Statistic                   0.815369
p-value                          0.991880
#Lags Used                      13.000000
Number of Observations Used    130.000000
Critical Value (1%)             -3.481682
Critical Value (5%)             -2.884042
Critical Value (10%)            -2.578770
dtype: float64
The data is not stationary
```

As we can see, the data are not stationary becasue p-value is greater than 0.05.

0.2.2 Viasualizing the data to detect non-stationarity, variance and trend

```
[ ]: df.plot(figsize=(20, 5))
```

```
[ ]: <AxesSubplot:xlabel='Month'>
```

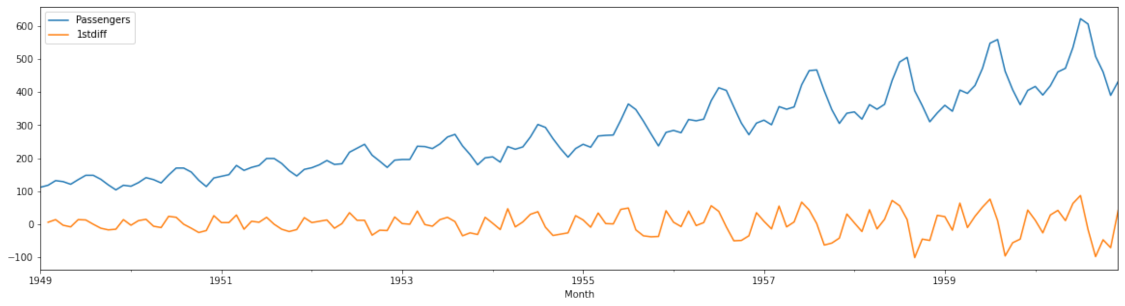


From the graph we can see that the data is not stationary it also has a positive trend. There is also seasonality and variance. The variance is increasing in time which which is called heteroskedasticity.

Fixing the data We will first take the first difference to see if it fixes any of the problems mentioned above.

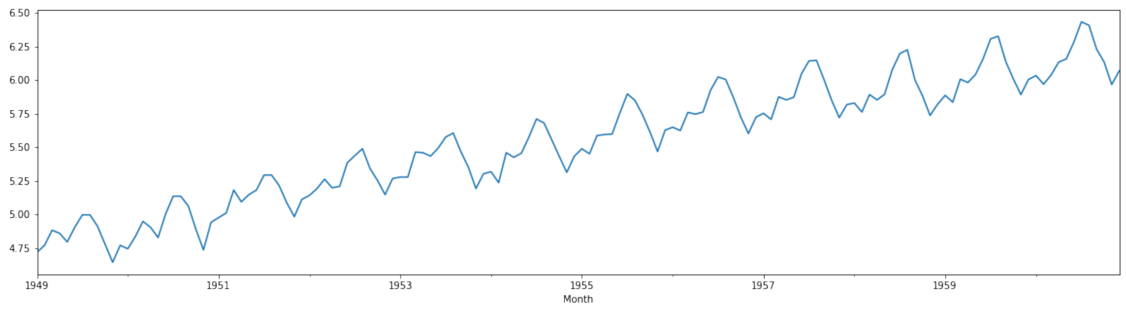
```
[ ]: df['1stdiff'] = df['Passengers'].diff()
```

```
[ ]: df.plot(figsize=(20, 5));
```



After taking the first difference we can see that the trend was removed. But there is still some seasonality and a variance that increases with time. Since it is common to use log function together with the first difference in order to solved the non-stationarity issues, next will be used the log transformation and the results will be analized.

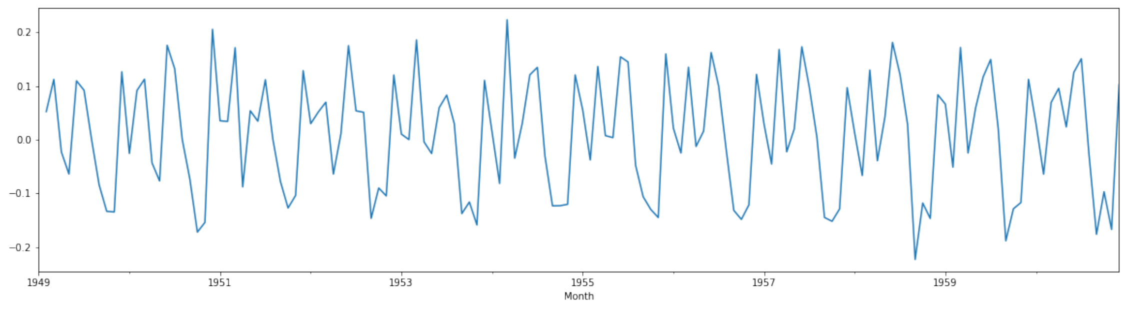
```
[ ]: df['LogPassengers'] = np.log(df['Passengers'])
     df['LogPassengers'].plot(figsize=(20, 5));
```



We can see that the log function helped to remove the variance with time. The log transformation did not remove the trend, but the differenciation did, because of this will will use the two function combined in the next step. The log transformation did not solve the problem with seasonality and heteroskedasticity.

Use Log transformation in combination with the first difference

```
[ ]: df['Log1stDiff'] = df['LogPassengers'].diff()
     df['Log1stDiff'].plot(figsize=(20, 5));
```



This time we have got a much better result. The result is not perfect because there is some cyclicality, with this should work for our model.

See if there are NaN values for Log1stDiff

```
[ ]: print("Number of NaNs:", df['Log1stDiff'].isna().sum())
     df
```

```
Number of NaNs: 1
```

```
[ ]:             Passengers  1stdiff  LogPassengers  Log1stDiff
     Month
     1949-01-01         112      NaN       4.718499         NaN
     1949-02-01         118      6.0       4.770685    0.052186
     1949-03-01         132     14.0       4.882802    0.112117
     1949-04-01         129     -3.0       4.859812   -0.022990
```

```
1949-05-01          121      -8.0       4.795791   -0.064022
…                 …      …            …         …
1960-08-01          606     -16.0       6.406880   -0.026060
1960-09-01          508     -98.0       6.230481   -0.176399
1960-10-01          461     -47.0       6.133398   -0.097083
1960-11-01          390     -71.0       5.966147   -0.167251
1960-12-01          432      42.0       6.068426    0.102279

[144 rows x 4 columns]
```

There is only one NaN and it is at index 0. We can substitute this value with the first non NaN value of the series.

```
[ ]: # substitute value at index 0 with value at index 1
     df['Log1stDiff'].iloc[0] = df['Log1stDiff'].iloc[1]
```

```
/tmp/ipykernel_93230/1787326268.py:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df['Log1stDiff'].iloc[0] = df['Log1stDiff'].iloc[1]
```

0.2.3 Create the train and test sets

Because time series are sequences, we'll take the y values for the last 12 months and assign them to the test set, and the rest will go into the training set.

```
[ ]: df.index.freq = 'MS'

     Ntest = 24
     train = df.iloc[:-Ntest]
     test = df.iloc[-Ntest:]
```

7. Create the index arrays for both train and test datasets. If you don't remember how to do this from our ML class here is one example: train_idx = df.index <= train.index[-1], so just do the same for the test set.

```
[ ]: train_idx = df.index <= train.index[-1]
     test_idx = df.index > train.index[-1]
```

```
[ ]: # taking a look at the dataset
     print(df)
```

```
            Passengers  1stdiff  LogPassengers  Log1stDiff
Month
1949-01-01         112      NaN       4.718499    0.052186
1949-02-01         118      6.0       4.770685    0.052186
1949-03-01         132     14.0       4.882802    0.112117
```

```
1949-04-01          129     -3.0       4.859812   -0.022990
1949-05-01          121     -8.0       4.795791   -0.064022
…                 …       …           …          …
1960-08-01          606    -16.0       6.406880   -0.026060
1960-09-01          508    -98.0       6.230481   -0.176399
1960-10-01          461    -47.0       6.133398   -0.097083
1960-11-01          390    -71.0       5.966147   -0.167251
1960-12-01          432     42.0       6.068426    0.102279

[144 rows x 4 columns]
```

8. Create the appropriate data structure for a time series analysis where the past 10 datapoints will be used to make predictions of the next 1 datapoint.

[ ]:
```python
# prepare dataset for RNN
def prepare_dataset(df, look_back=1):
    X, Y = [], []
    for i in range(len(df) - look_back):
        X.append(df[i:(i+look_back)])
        Y.append(df[i+look_back])
    return np.array(X), np.array(Y)

look_back = 10
n_features = 1

# using Log1stDiff
Xtrain, Ytrain = prepare_dataset(df['Log1stDiff'].loc[train_idx], look_back)
Xtest, Ytest = prepare_dataset(df['Log1stDiff'].loc[test_idx], look_back)
```

9. Reshape your created dataframe into the 3D format that RNN expects.

10. Create the X_train, y_train, X_test and y_test.

[ ]:
```python
# reshape input to be [samples, time steps, features]
Xtrain = np.reshape(Xtrain, (Xtrain.shape[0], Xtrain.shape[1], n_features))
Xtest = np.reshape(Xtest, (Xtest.shape[0], Xtest.shape[1], n_features))

print(Xtrain.shape, Ytrain.shape)
print(Xtest.shape, Ytest.shape)
```

```
(110, 10, 1) (110,)
(14, 10, 1) (14,)
```

11. Create a one-layer RNN with 25 units (neurons). Train it for 100 epochs with the default mini batch size of 32

12. Compile and fit the model. When compiling, specify the 'sgd' optimizer, and choose an accuracy metric (e.g. mse, mae, mape, etc.) to judge the goodness of your predictions. To compare further on the prediction results with the true values in the test set, specify inside the fit function your validation data: validation_data = (X_test, y_test). Save the history

of the loss and the accuracy results in each epoch as the model is being trained, so that you can later on plot them and compare.

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

model = Sequential()
model.add(LSTM(25, activation='relu', input_shape=(look_back, n_features)))
model.add(Dropout(0.2))
model.add(Dense(1))
model.compile(loss='mse', optimizer='sgd', metrics='accuracy')
history = model.fit(Xtrain, Ytrain, epochs=100, batch_size=32,␣
 ↪validation_split=0.1, verbose=2, shuffle=False, validation_data=(Xtest,␣
 ↪Ytest))
```

```
Epoch 1/100
4/4 - 1s - loss: 0.0109 - accuracy: 0.0303 - val_loss: 0.0148 - val_accuracy:
0.0000e+00 - 777ms/epoch - 194ms/step
Epoch 2/100
4/4 - 0s - loss: 0.0108 - accuracy: 0.0303 - val_loss: 0.0148 - val_accuracy:
0.0000e+00 - 32ms/epoch - 8ms/step
Epoch 3/100
4/4 - 0s - loss: 0.0106 - accuracy: 0.0303 - val_loss: 0.0148 - val_accuracy:
0.0000e+00 - 24ms/epoch - 6ms/step
Epoch 4/100
4/4 - 0s - loss: 0.0109 - accuracy: 0.0303 - val_loss: 0.0148 - val_accuracy:
0.0000e+00 - 25ms/epoch - 6ms/step
Epoch 5/100
4/4 - 0s - loss: 0.0109 - accuracy: 0.0303 - val_loss: 0.0148 - val_accuracy:
0.0000e+00 - 22ms/epoch - 5ms/step
Epoch 6/100
4/4 - 0s - loss: 0.0108 - accuracy: 0.0303 - val_loss: 0.0148 - val_accuracy:
0.0000e+00 - 21ms/epoch - 5ms/step
Epoch 7/100
4/4 - 0s - loss: 0.0108 - accuracy: 0.0303 - val_loss: 0.0148 - val_accuracy:
0.0000e+00 - 25ms/epoch - 6ms/step
Epoch 8/100
4/4 - 0s - loss: 0.0108 - accuracy: 0.0303 - val_loss: 0.0148 - val_accuracy:
0.0000e+00 - 23ms/epoch - 6ms/step
Epoch 9/100
4/4 - 0s - loss: 0.0108 - accuracy: 0.0303 - val_loss: 0.0148 - val_accuracy:
0.0000e+00 - 25ms/epoch - 6ms/step
Epoch 10/100
4/4 - 0s - loss: 0.0110 - accuracy: 0.0303 - val_loss: 0.0148 - val_accuracy:
0.0000e+00 - 22ms/epoch - 5ms/step
Epoch 11/100
```

```
4/4 - 0s - loss: 0.0110 - accuracy: 0.0303 - val_loss: 0.0148 - val_accuracy:
0.0000e+00 - 21ms/epoch - 5ms/step
Epoch 12/100
4/4 - 0s - loss: 0.0110 - accuracy: 0.0303 - val_loss: 0.0148 - val_accuracy:
0.0000e+00 - 20ms/epoch - 5ms/step
Epoch 13/100
4/4 - 0s - loss: 0.0107 - accuracy: 0.0303 - val_loss: 0.0148 - val_accuracy:
0.0000e+00 - 20ms/epoch - 5ms/step
Epoch 14/100
4/4 - 0s - loss: 0.0107 - accuracy: 0.0303 - val_loss: 0.0148 - val_accuracy:
0.0000e+00 - 26ms/epoch - 6ms/step
Epoch 15/100
4/4 - 0s - loss: 0.0109 - accuracy: 0.0303 - val_loss: 0.0148 - val_accuracy:
0.0000e+00 - 23ms/epoch - 6ms/step
Epoch 16/100
4/4 - 0s - loss: 0.0106 - accuracy: 0.0303 - val_loss: 0.0148 - val_accuracy:
0.0000e+00 - 21ms/epoch - 5ms/step
Epoch 17/100
4/4 - 0s - loss: 0.0107 - accuracy: 0.0303 - val_loss: 0.0147 - val_accuracy:
0.0000e+00 - 21ms/epoch - 5ms/step
Epoch 18/100
4/4 - 0s - loss: 0.0107 - accuracy: 0.0303 - val_loss: 0.0147 - val_accuracy:
0.0000e+00 - 22ms/epoch - 5ms/step
Epoch 19/100
4/4 - 0s - loss: 0.0107 - accuracy: 0.0303 - val_loss: 0.0147 - val_accuracy:
0.0000e+00 - 22ms/epoch - 5ms/step
Epoch 20/100
4/4 - 0s - loss: 0.0108 - accuracy: 0.0303 - val_loss: 0.0147 - val_accuracy:
0.0000e+00 - 24ms/epoch - 6ms/step
Epoch 21/100
4/4 - 0s - loss: 0.0110 - accuracy: 0.0303 - val_loss: 0.0147 - val_accuracy:
0.0000e+00 - 22ms/epoch - 5ms/step
Epoch 22/100
4/4 - 0s - loss: 0.0107 - accuracy: 0.0303 - val_loss: 0.0147 - val_accuracy:
0.0000e+00 - 22ms/epoch - 5ms/step
Epoch 23/100
4/4 - 0s - loss: 0.0107 - accuracy: 0.0303 - val_loss: 0.0147 - val_accuracy:
0.0000e+00 - 20ms/epoch - 5ms/step
Epoch 24/100
4/4 - 0s - loss: 0.0107 - accuracy: 0.0303 - val_loss: 0.0147 - val_accuracy:
0.0000e+00 - 20ms/epoch - 5ms/step
Epoch 25/100
4/4 - 0s - loss: 0.0106 - accuracy: 0.0303 - val_loss: 0.0147 - val_accuracy:
0.0000e+00 - 20ms/epoch - 5ms/step
Epoch 26/100
4/4 - 0s - loss: 0.0109 - accuracy: 0.0303 - val_loss: 0.0147 - val_accuracy:
0.0000e+00 - 23ms/epoch - 6ms/step
Epoch 27/100
```

```
4/4 - 0s - loss: 0.0107 - accuracy: 0.0303 - val_loss: 0.0147 - val_accuracy:
0.0000e+00 - 20ms/epoch - 5ms/step
Epoch 28/100
4/4 - 0s - loss: 0.0109 - accuracy: 0.0303 - val_loss: 0.0147 - val_accuracy:
0.0000e+00 - 20ms/epoch - 5ms/step
Epoch 29/100
4/4 - 0s - loss: 0.0109 - accuracy: 0.0303 - val_loss: 0.0147 - val_accuracy:
0.0000e+00 - 21ms/epoch - 5ms/step
Epoch 30/100
4/4 - 0s - loss: 0.0108 - accuracy: 0.0303 - val_loss: 0.0147 - val_accuracy:
0.0000e+00 - 21ms/epoch - 5ms/step
Epoch 31/100
4/4 - 0s - loss: 0.0108 - accuracy: 0.0303 - val_loss: 0.0147 - val_accuracy:
0.0000e+00 - 20ms/epoch - 5ms/step
Epoch 32/100
4/4 - 0s - loss: 0.0109 - accuracy: 0.0303 - val_loss: 0.0147 - val_accuracy:
0.0000e+00 - 23ms/epoch - 6ms/step
Epoch 33/100
4/4 - 0s - loss: 0.0106 - accuracy: 0.0303 - val_loss: 0.0147 - val_accuracy:
0.0000e+00 - 22ms/epoch - 5ms/step
Epoch 34/100
4/4 - 0s - loss: 0.0109 - accuracy: 0.0303 - val_loss: 0.0147 - val_accuracy:
0.0000e+00 - 20ms/epoch - 5ms/step
Epoch 35/100
4/4 - 0s - loss: 0.0109 - accuracy: 0.0303 - val_loss: 0.0147 - val_accuracy:
0.0000e+00 - 20ms/epoch - 5ms/step
Epoch 36/100
4/4 - 0s - loss: 0.0108 - accuracy: 0.0303 - val_loss: 0.0147 - val_accuracy:
0.0000e+00 - 20ms/epoch - 5ms/step
Epoch 37/100
4/4 - 0s - loss: 0.0107 - accuracy: 0.0303 - val_loss: 0.0147 - val_accuracy:
0.0000e+00 - 20ms/epoch - 5ms/step
Epoch 38/100
4/4 - 0s - loss: 0.0106 - accuracy: 0.0303 - val_loss: 0.0147 - val_accuracy:
0.0000e+00 - 21ms/epoch - 5ms/step
Epoch 39/100
4/4 - 0s - loss: 0.0106 - accuracy: 0.0303 - val_loss: 0.0147 - val_accuracy:
0.0000e+00 - 22ms/epoch - 5ms/step
Epoch 40/100
4/4 - 0s - loss: 0.0108 - accuracy: 0.0303 - val_loss: 0.0147 - val_accuracy:
0.0000e+00 - 21ms/epoch - 5ms/step
Epoch 41/100
4/4 - 0s - loss: 0.0107 - accuracy: 0.0303 - val_loss: 0.0147 - val_accuracy:
0.0000e+00 - 20ms/epoch - 5ms/step
Epoch 42/100
4/4 - 0s - loss: 0.0107 - accuracy: 0.0303 - val_loss: 0.0147 - val_accuracy:
0.0000e+00 - 22ms/epoch - 5ms/step
Epoch 43/100
```

```
4/4 - 0s - loss: 0.0106 - accuracy: 0.0303 - val_loss: 0.0147 - val_accuracy:
0.0000e+00 - 22ms/epoch - 5ms/step
Epoch 44/100
4/4 - 0s - loss: 0.0108 - accuracy: 0.0303 - val_loss: 0.0147 - val_accuracy:
0.0000e+00 - 24ms/epoch - 6ms/step
Epoch 45/100
4/4 - 0s - loss: 0.0106 - accuracy: 0.0303 - val_loss: 0.0147 - val_accuracy:
0.0000e+00 - 23ms/epoch - 6ms/step
Epoch 46/100
4/4 - 0s - loss: 0.0109 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 22ms/epoch - 5ms/step
Epoch 47/100
4/4 - 0s - loss: 0.0108 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 21ms/epoch - 5ms/step
Epoch 48/100
4/4 - 0s - loss: 0.0108 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 20ms/epoch - 5ms/step
Epoch 49/100
4/4 - 0s - loss: 0.0106 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 21ms/epoch - 5ms/step
Epoch 50/100
4/4 - 0s - loss: 0.0108 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 22ms/epoch - 6ms/step
Epoch 51/100
4/4 - 0s - loss: 0.0105 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 21ms/epoch - 5ms/step
Epoch 52/100
4/4 - 0s - loss: 0.0106 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 22ms/epoch - 5ms/step
Epoch 53/100
4/4 - 0s - loss: 0.0106 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 22ms/epoch - 5ms/step
Epoch 54/100
4/4 - 0s - loss: 0.0109 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 24ms/epoch - 6ms/step
Epoch 55/100
4/4 - 0s - loss: 0.0106 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 22ms/epoch - 6ms/step
Epoch 56/100
4/4 - 0s - loss: 0.0105 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 27ms/epoch - 7ms/step
Epoch 57/100
4/4 - 0s - loss: 0.0106 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 23ms/epoch - 6ms/step
Epoch 58/100
4/4 - 0s - loss: 0.0109 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 23ms/epoch - 6ms/step
Epoch 59/100
```

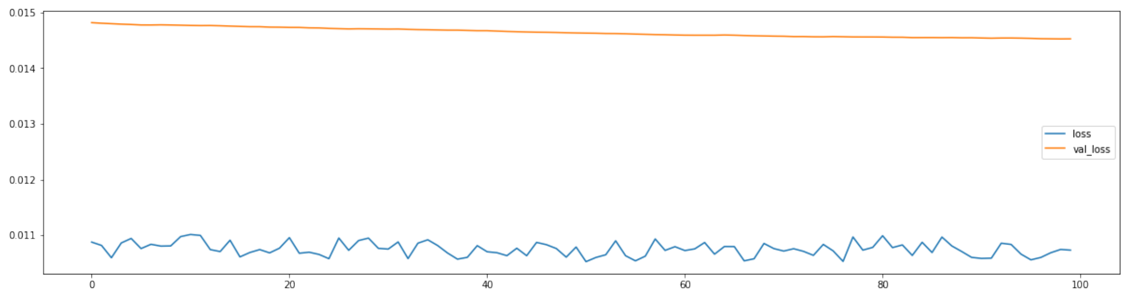
```
4/4 - 0s - loss: 0.0107 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 23ms/epoch - 6ms/step
Epoch 60/100
4/4 - 0s - loss: 0.0108 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 27ms/epoch - 7ms/step
Epoch 61/100
4/4 - 0s - loss: 0.0107 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 25ms/epoch - 6ms/step
Epoch 62/100
4/4 - 0s - loss: 0.0108 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 22ms/epoch - 6ms/step
Epoch 63/100
4/4 - 0s - loss: 0.0109 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 23ms/epoch - 6ms/step
Epoch 64/100
4/4 - 0s - loss: 0.0107 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 22ms/epoch - 5ms/step
Epoch 65/100
4/4 - 0s - loss: 0.0108 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 23ms/epoch - 6ms/step
Epoch 66/100
4/4 - 0s - loss: 0.0108 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 27ms/epoch - 7ms/step
Epoch 67/100
4/4 - 0s - loss: 0.0105 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 25ms/epoch - 6ms/step
Epoch 68/100
4/4 - 0s - loss: 0.0106 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 26ms/epoch - 7ms/step
Epoch 69/100
4/4 - 0s - loss: 0.0109 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 23ms/epoch - 6ms/step
Epoch 70/100
4/4 - 0s - loss: 0.0108 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 23ms/epoch - 6ms/step
Epoch 71/100
4/4 - 0s - loss: 0.0107 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 24ms/epoch - 6ms/step
Epoch 72/100
4/4 - 0s - loss: 0.0108 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 26ms/epoch - 6ms/step
Epoch 73/100
4/4 - 0s - loss: 0.0107 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 26ms/epoch - 7ms/step
Epoch 74/100
4/4 - 0s - loss: 0.0106 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 22ms/epoch - 6ms/step
Epoch 75/100
```

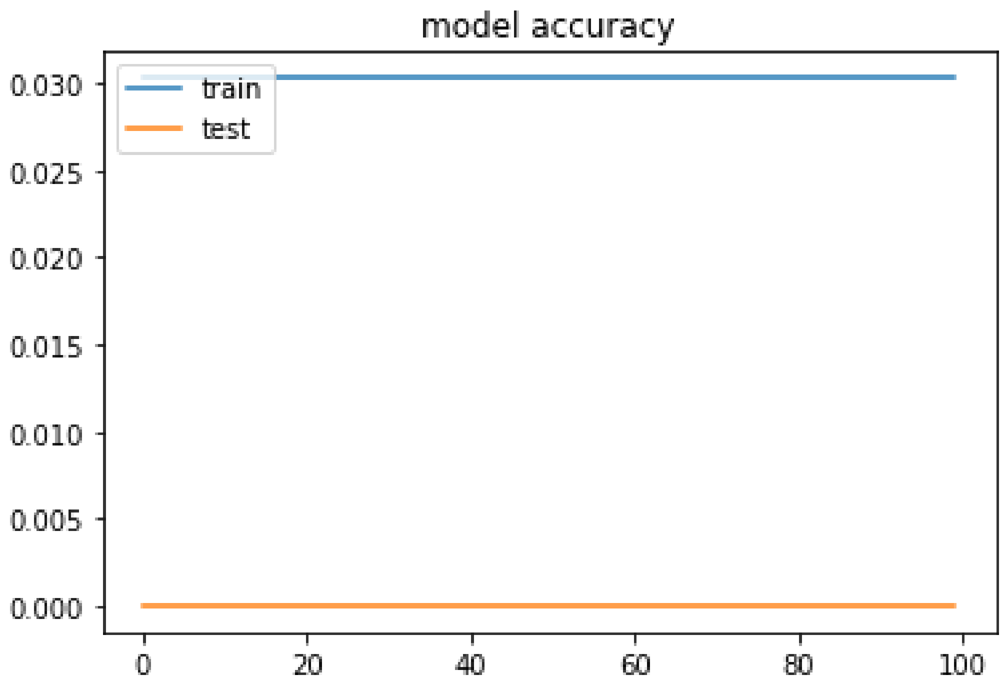
```
4/4 - 0s - loss: 0.0108 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 27ms/epoch - 7ms/step
Epoch 76/100
4/4 - 0s - loss: 0.0107 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 26ms/epoch - 7ms/step
Epoch 77/100
4/4 - 0s - loss: 0.0105 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 25ms/epoch - 6ms/step
Epoch 78/100
4/4 - 0s - loss: 0.0110 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 24ms/epoch - 6ms/step
Epoch 79/100
4/4 - 0s - loss: 0.0107 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 23ms/epoch - 6ms/step
Epoch 80/100
4/4 - 0s - loss: 0.0108 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 24ms/epoch - 6ms/step
Epoch 81/100
4/4 - 0s - loss: 0.0110 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 23ms/epoch - 6ms/step
Epoch 82/100
4/4 - 0s - loss: 0.0108 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 23ms/epoch - 6ms/step
Epoch 83/100
4/4 - 0s - loss: 0.0108 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 26ms/epoch - 7ms/step
Epoch 84/100
4/4 - 0s - loss: 0.0106 - accuracy: 0.0303 - val_loss: 0.0145 - val_accuracy:
0.0000e+00 - 26ms/epoch - 6ms/step
Epoch 85/100
4/4 - 0s - loss: 0.0109 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 23ms/epoch - 6ms/step
Epoch 86/100
4/4 - 0s - loss: 0.0107 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 24ms/epoch - 6ms/step
Epoch 87/100
4/4 - 0s - loss: 0.0110 - accuracy: 0.0303 - val_loss: 0.0145 - val_accuracy:
0.0000e+00 - 22ms/epoch - 6ms/step
Epoch 88/100
4/4 - 0s - loss: 0.0108 - accuracy: 0.0303 - val_loss: 0.0146 - val_accuracy:
0.0000e+00 - 23ms/epoch - 6ms/step
Epoch 89/100
4/4 - 0s - loss: 0.0107 - accuracy: 0.0303 - val_loss: 0.0145 - val_accuracy:
0.0000e+00 - 20ms/epoch - 5ms/step
Epoch 90/100
4/4 - 0s - loss: 0.0106 - accuracy: 0.0303 - val_loss: 0.0145 - val_accuracy:
0.0000e+00 - 20ms/epoch - 5ms/step
Epoch 91/100
```

```
4/4 - 0s - loss: 0.0106 - accuracy: 0.0303 - val_loss: 0.0145 - val_accuracy:
0.0000e+00 - 21ms/epoch - 5ms/step
Epoch 92/100
4/4 - 0s - loss: 0.0106 - accuracy: 0.0303 - val_loss: 0.0145 - val_accuracy:
0.0000e+00 - 21ms/epoch - 5ms/step
Epoch 93/100
4/4 - 0s - loss: 0.0109 - accuracy: 0.0303 - val_loss: 0.0145 - val_accuracy:
0.0000e+00 - 22ms/epoch - 5ms/step
Epoch 94/100
4/4 - 0s - loss: 0.0108 - accuracy: 0.0303 - val_loss: 0.0145 - val_accuracy:
0.0000e+00 - 27ms/epoch - 7ms/step
Epoch 95/100
4/4 - 0s - loss: 0.0107 - accuracy: 0.0303 - val_loss: 0.0145 - val_accuracy:
0.0000e+00 - 22ms/epoch - 5ms/step
Epoch 96/100
4/4 - 0s - loss: 0.0106 - accuracy: 0.0303 - val_loss: 0.0145 - val_accuracy:
0.0000e+00 - 21ms/epoch - 5ms/step
Epoch 97/100
4/4 - 0s - loss: 0.0106 - accuracy: 0.0303 - val_loss: 0.0145 - val_accuracy:
0.0000e+00 - 23ms/epoch - 6ms/step
Epoch 98/100
4/4 - 0s - loss: 0.0107 - accuracy: 0.0303 - val_loss: 0.0145 - val_accuracy:
0.0000e+00 - 25ms/epoch - 6ms/step
Epoch 99/100
4/4 - 0s - loss: 0.0107 - accuracy: 0.0303 - val_loss: 0.0145 - val_accuracy:
0.0000e+00 - 26ms/epoch - 7ms/step
Epoch 100/100
4/4 - 0s - loss: 0.0107 - accuracy: 0.0303 - val_loss: 0.0145 - val_accuracy:
0.0000e+00 - 23ms/epoch - 6ms/step
```

13. Plot the loss and accuracy in each epoch for the training set as well as the validation set. Comment on your results. Do you see signs of overfitting? If yes, explain. If no, explain.

[ ]:
```
# plot loss  and val_loss
plt.figure(figsize=(20, 5))
plt.plot(history.history['loss'], label='loss')
plt.plot(history.history['val_loss'], label='val_loss')
plt.legend()
plt.show()

# plot accuracy and val_accuracy
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label='val_accuracy')
plt.title('model accuracy')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```





There is no overfitting. The accuracy and val_accuracy are low and stable similar to loss and val_loss.

14. Do you see signs of vanishing or exploding gradient? Explain how the two would manifest in your graph.

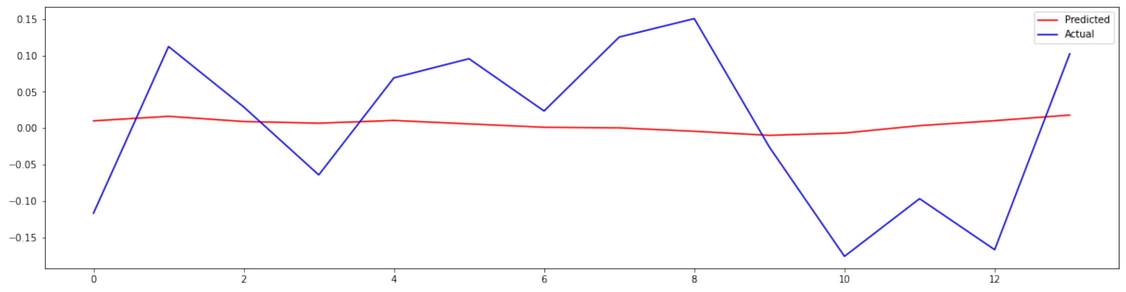
There is not exploding gradient because the loss doesn't variate much. There is not vanishing gradient because the loss doesn't decrease much when more layers are added to the network.

15. Compute the model predictions, undo the differencing and plot the results.

[ ]:
```
# Compute the model predictions
trainPredict = model.predict(Xtrain)
testPredict = model.predict(Xtest)
```

```
# reshape to 1D array
trainPredict = np.reshape(trainPredict, (trainPredict.shape[0],))
testPredict = np.reshape(testPredict, (testPredict.shape[0],))

# def rebuild_diffed(series: pd.DataFrame, first_element_original):
#     cumsum = series.cumsum()
#     return cumsum + first_element_original

print(testPredict)
print(df['Log1stDiff'].loc[test_idx].values[look_back:])

# plot testPredict alongside actual test values
plt.figure(figsize=(20, 5))
plt.plot(testPredict, color='red', label='Predicted')
plt.plot(df['Log1stDiff'].loc[test_idx].values[look_back:], color='blue',␣
 ↪label='Actual')
plt.legend()
plt.show()
```

```
[ 0.00999344  0.01621936  0.00909845  0.00681003  0.0105692   0.00582078
  0.0011528   0.00035984 -0.00434615 -0.00991009 -0.00673154  0.00343249
  0.01024912  0.01796216]
[-0.11716897  0.11224286  0.02919915 -0.06437866  0.06916336  0.09552712
  0.02358094  0.12528776  0.15067335 -0.02606011 -0.17639854 -0.0970834
 -0.1672513   0.10227885]
```



16. Now try other options and see if you can beat your initial predictions:

    a) Train the model for 500 epochs. Are your predictions better? Should you have stopped before the end of the 500 epochs judging by your loss and accuracy plots? If yes at what epoch would you stop and why?

    b) Try two, three, and four layers for your RNN with 50 and 100 neurons in each of the hidden layers (you should get altogether 6 models – 2 layers & 50 neurons, 2 layers & 100 neurons, 3 layers & 50 neurons, etc.). Train each of them for 100 epochs and save and plot the accuracies and the losses for each of the 6 combinations. Do you see signs of overfitting? Which model overfits most? Explain.

c) For the most severely overfitting model from the 6 options above use the Dropuot in combination with the L2 method, to regularize your overfitting model. Be creative, try different options to correct for the overfit.

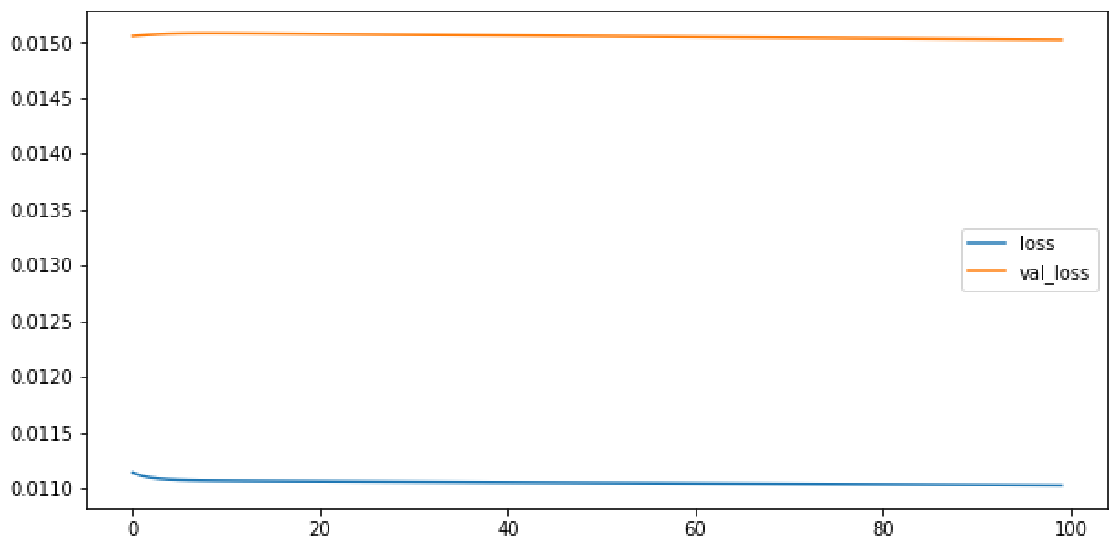
d) Could you create a model that beats the first one (the one with 1 layer and 25 units) and is not overfitting after regularization was applied?

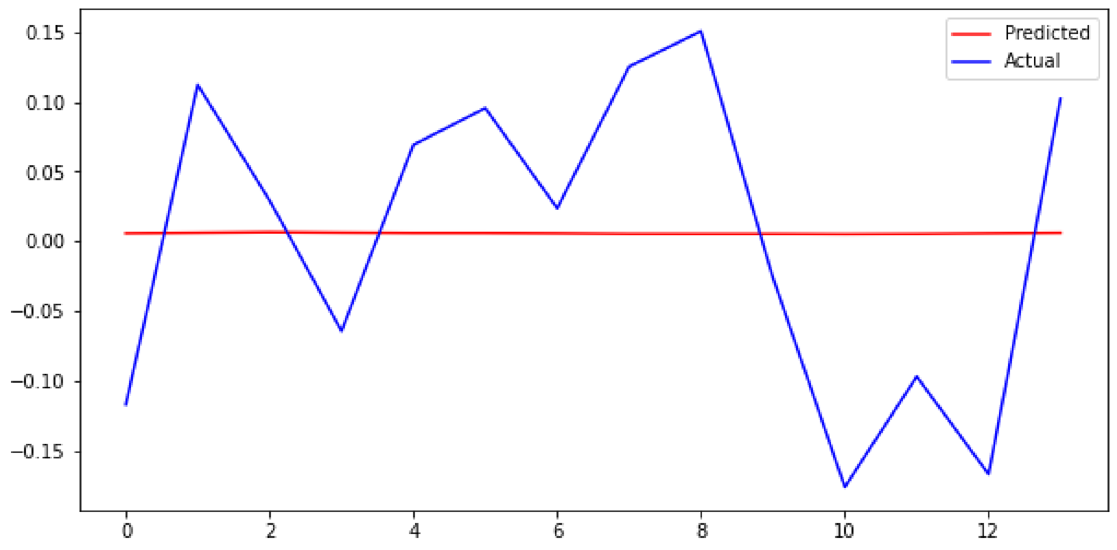
[ ]:
```
def predict(layers, epochs=100, optimizer='sgd', loss='mse',␣
↪metrics=['accuracy']) -> None:
    model = Sequential()
    for layer in layers:
        model.add(layer)
    model.add(Dense(1))
    model.compile(loss=loss, optimizer=optimizer, metrics=metrics)
    history = model.fit(Xtrain, Ytrain, epochs=epochs, batch_size=32,␣
↪validation_split=0.1, verbose=0, shuffle=False, validation_data=(Xtest,␣
↪Ytest))

    # plot loss  and val_loss
    plt.figure(figsize=(10, 5))
    plt.plot(history.history['loss'], label='loss')
    plt.plot(history.history['val_loss'], label='val_loss')
    plt.legend()
    plt.show()

    testPredict = model.predict(Xtest)
    testPredict = np.reshape(testPredict, (testPredict.shape[0],))

    plt.figure(figsize=(10, 5))
    plt.plot(testPredict, color='red', label='Predicted')
    plt.plot(df['Log1stDiff'].loc[test_idx].values[look_back:], color='blue',␣
↪label='Actual')
    plt.legend()
    plt.show()
```

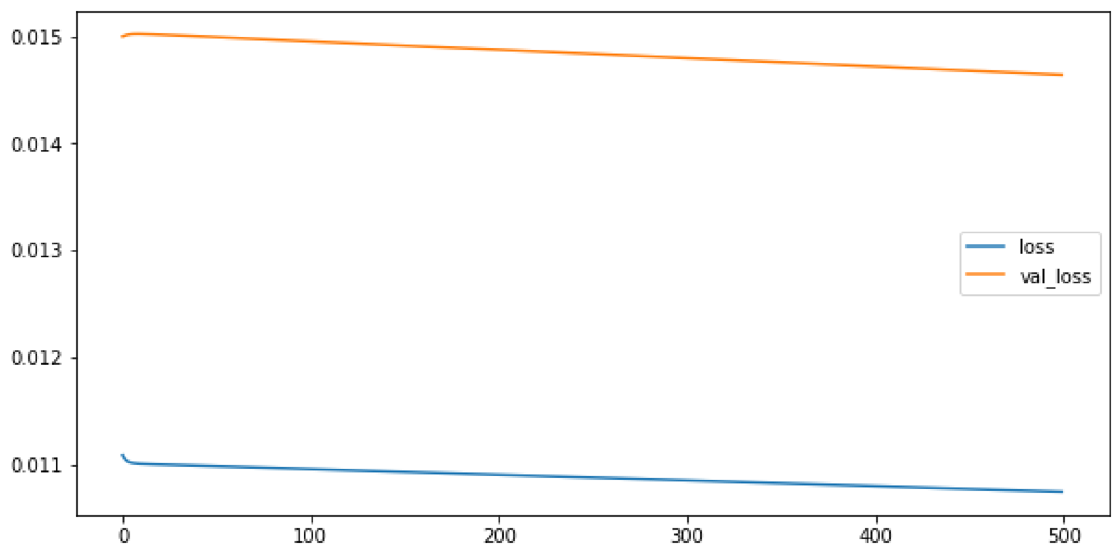
After the training using 500 epochs, we can see that the model is not overfitting and the results are not much better. The loss doesn't decrease much. We need to continue tuning the model.

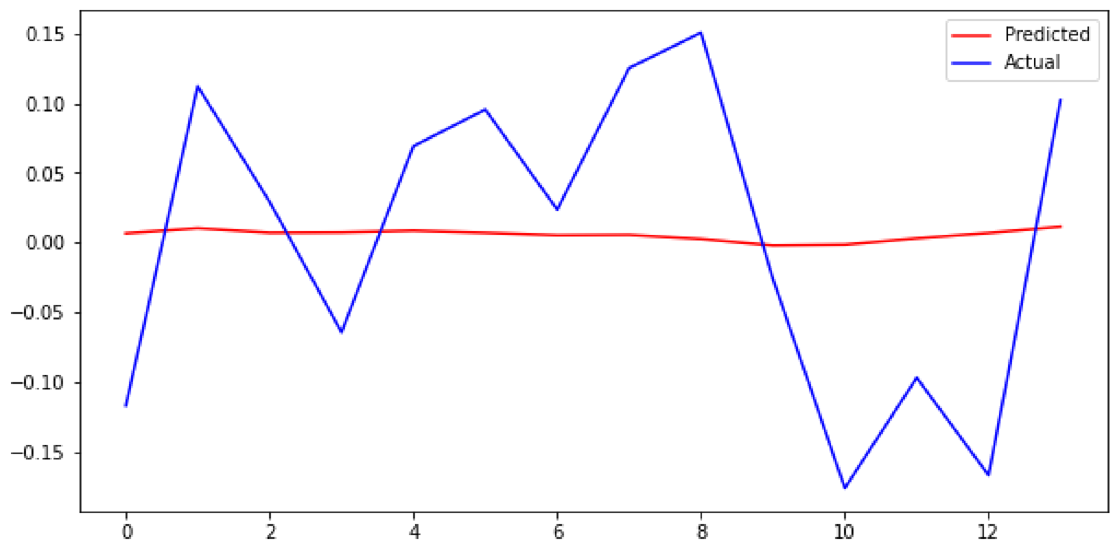
[ ]:
```
layers = [
    LSTM(50, input_shape=(look_back, n_features)),
    Dense(50, input_shape=(look_back, n_features)),
]
predict(layers, epochs=100)

layers = [
    LSTM(100, input_shape=(look_back, n_features)),
    Dense(100)
]
```

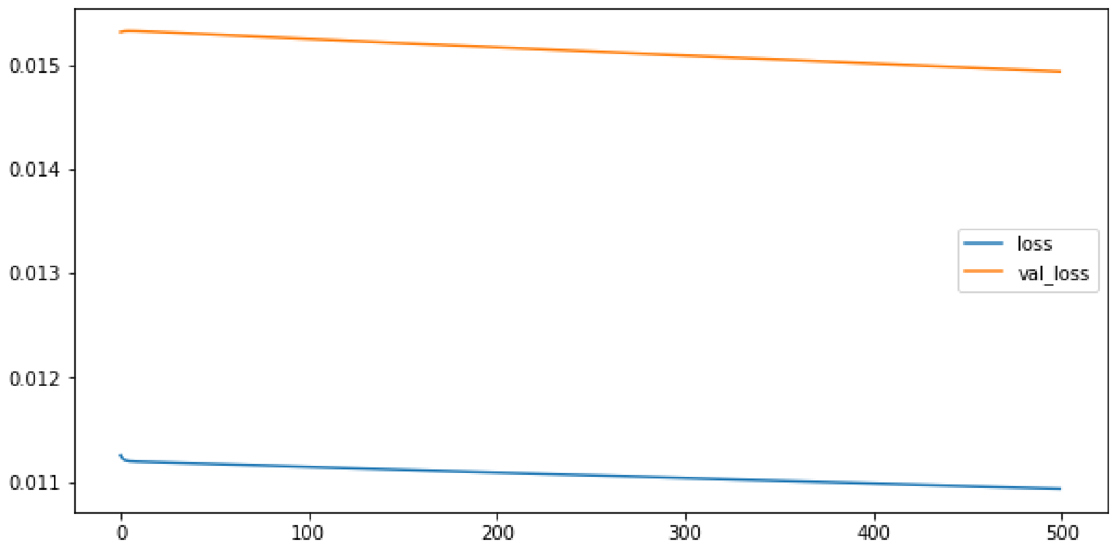
```
predict(layers, epochs=500)

# ============================

layers = [
    LSTM(50, input_shape=(look_back, n_features)),
    Dense(50),
    Dense(50)
]
predict(layers, epochs=500)

layers = [
    LSTM(100, input_shape=(look_back, n_features)),
    Dense(100),
    Dense(100)
]
predict(layers, epochs=500)


# ============================

layers = [
    LSTM(50, input_shape=(look_back, n_features)),
    Dense(50),
    Dense(50),
    Dense(50)
]
predict(layers, epochs=500)

layers = [
    LSTM(100, input_shape=(look_back, n_features)),
    Dense(100),
    Dense(100),
    Dense(100)
]
predict(layers, epochs=500)
```

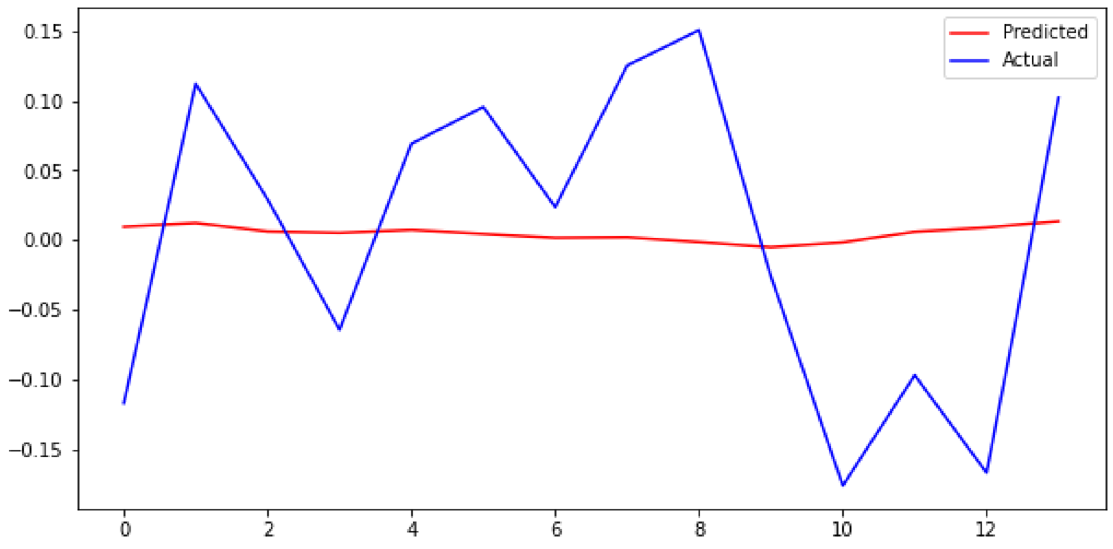

0.0150
0.0145
0.0140
0.0135
0.0130
0.0125
0.0120
0.0115
0.0110
0
20
40
60
80
100
loss
val_loss

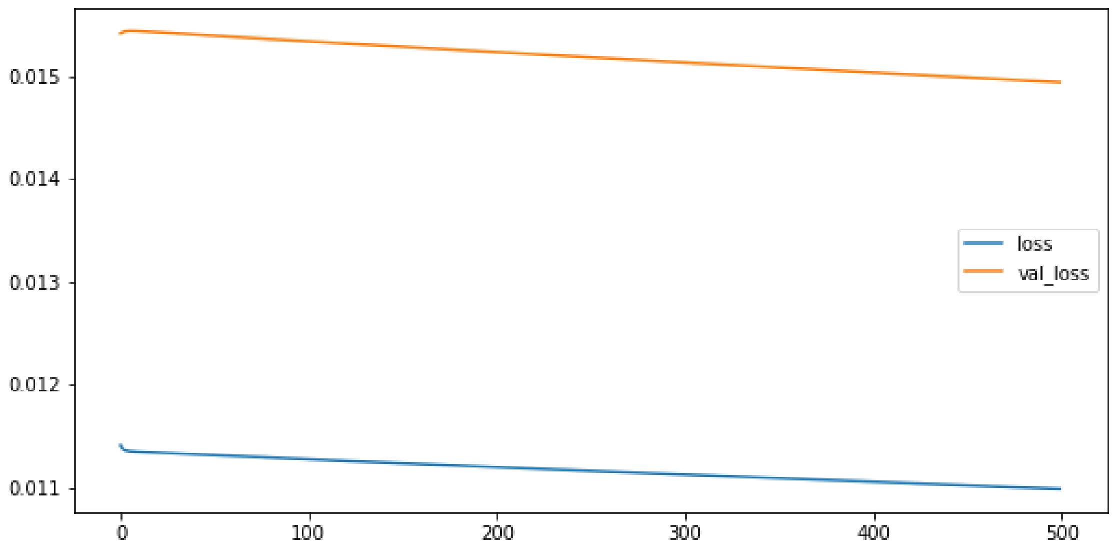

0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
0
2
4
6
8
10
12
Predicted
Actual

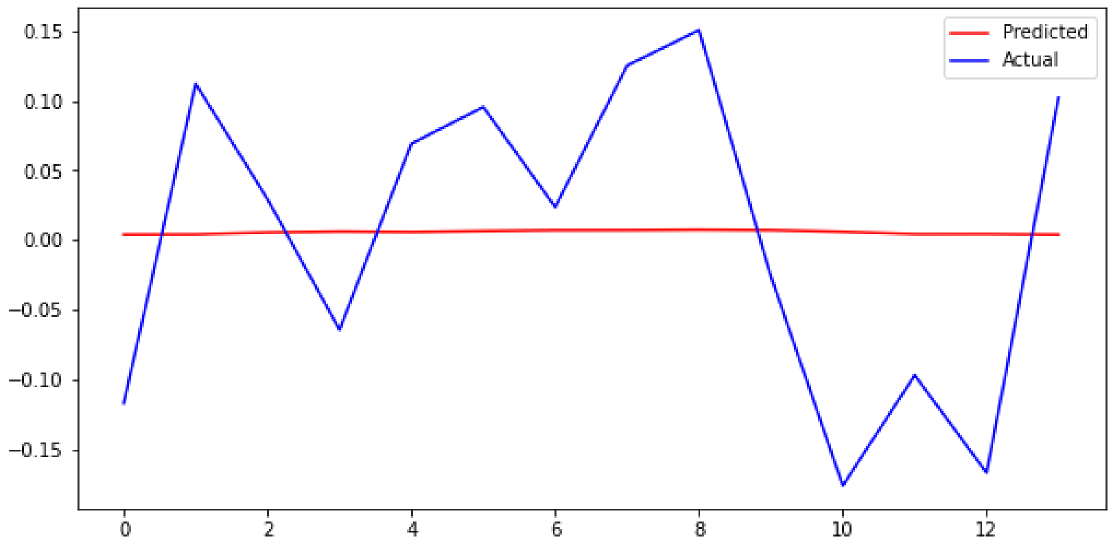

0.015
0.014
0.013
0.012
0.011
0
100
200
300
400
500
loss
val_loss

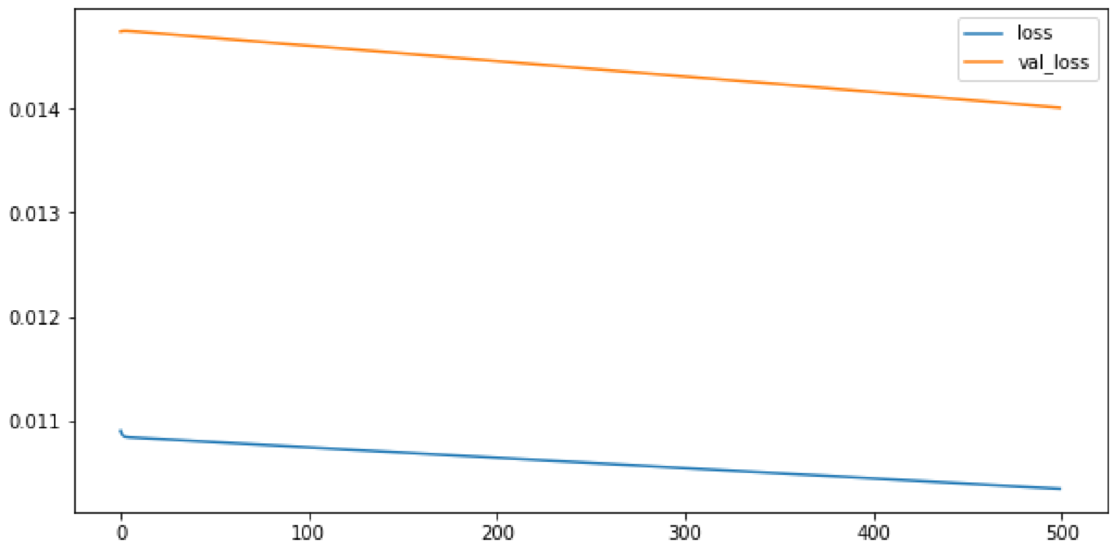

0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
0
2
4
6
8
10
12
Predicted
Actual

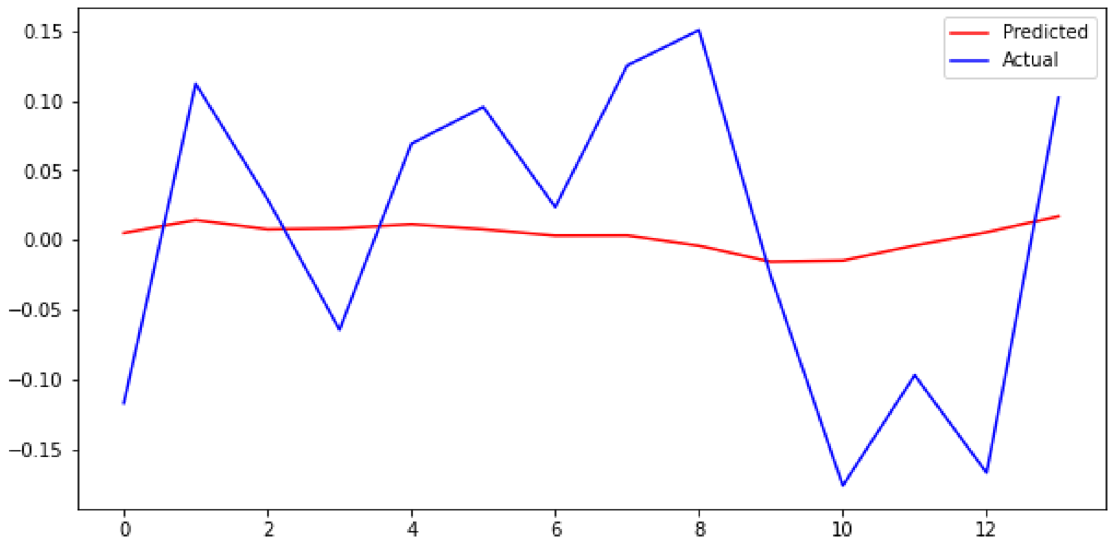

0.015
0.014
0.013
0.012
0.011
0
100
200
300
400
500
loss
val_loss

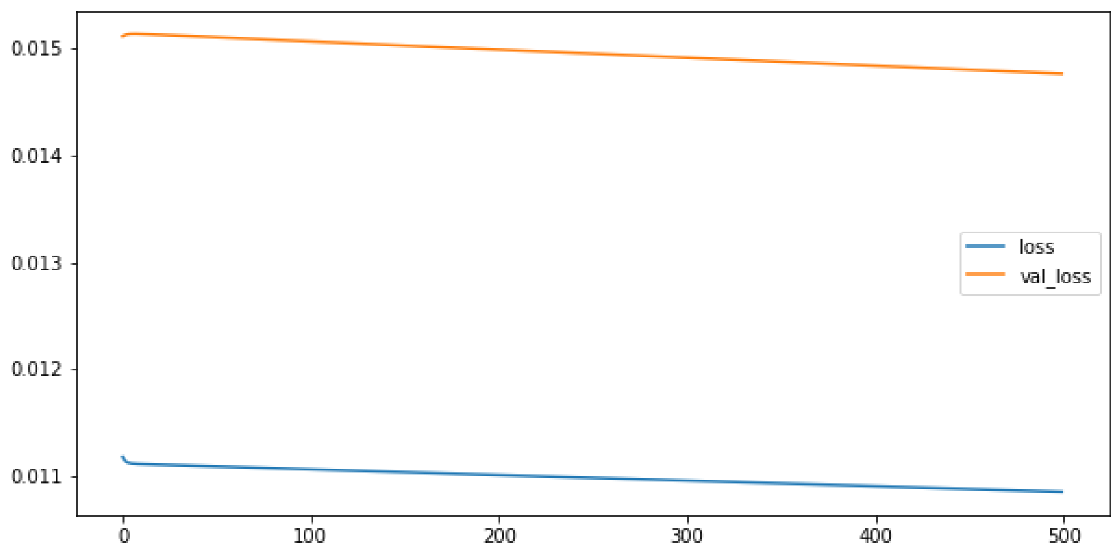

0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
0
2
4
6
8
10
12
Predicted
Actual

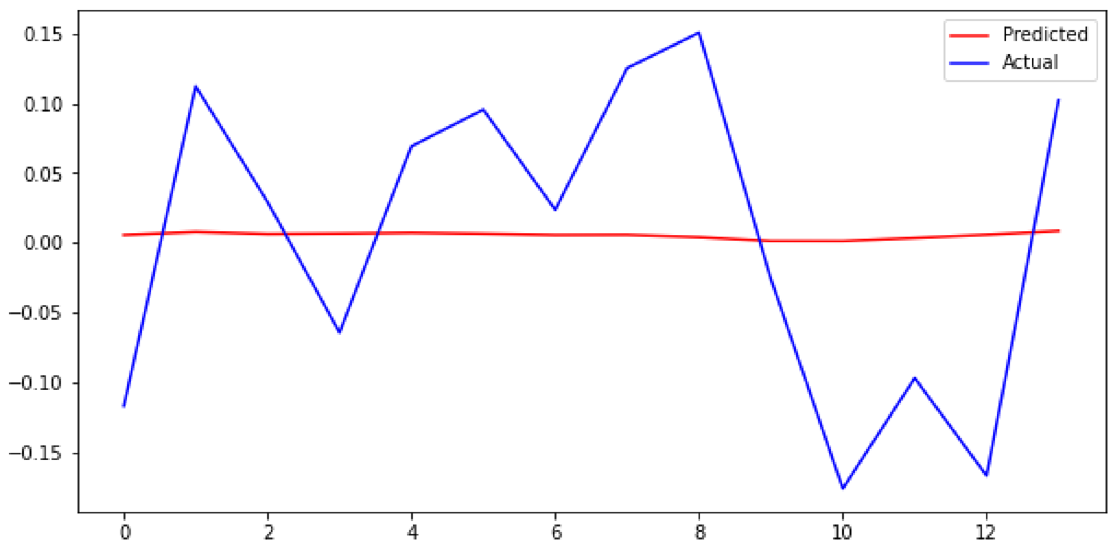

0.015
0.014
0.013
0.012
0.011
0
100
200
300
400
500
loss
val_loss


0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
0
2
4
6
8
10
12
Predicted
Actual


loss
val_loss
0.014
0.013
0.012
0.011
0
100
200
300
400
500

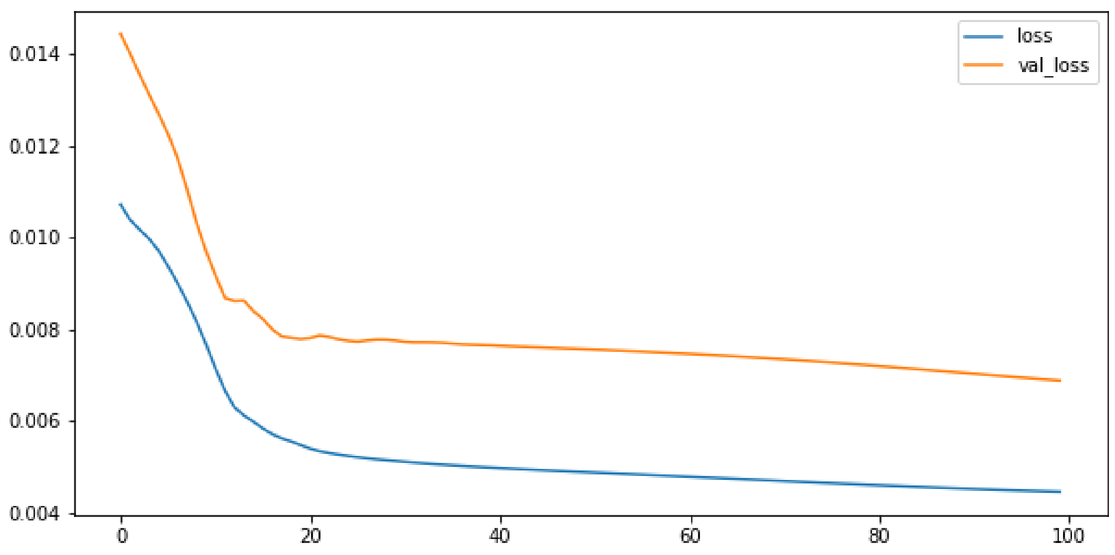

Predicted
Actual
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
0
2
4
6
8
10
12

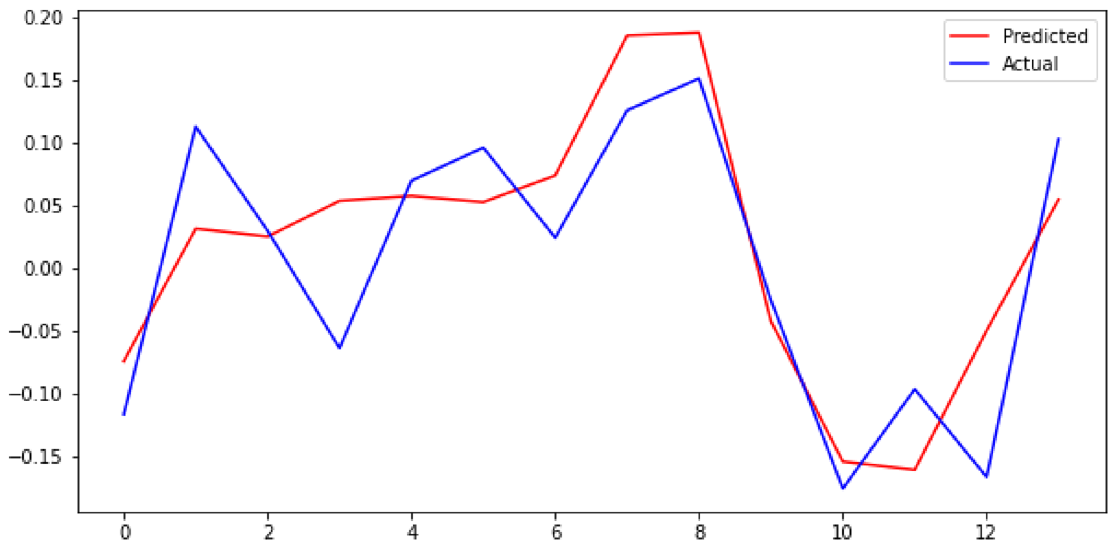




By looking at the plots, non of the models are overfitting.

[ ]:
```
# trying out the adam optimizer
layers = [
    LSTM(25),
]
predict(layers, epochs=100, loss='mse', optimizer='adam')
```





When trying out the adam optimisez, we can see that it returns a much better result.