# TIMESERIES FORECASTING – Traditional to Neural Networks and LSTM

**Mitali Bharali**
Graduate Student – UT Dallas – Master's in Business Analytics, minor in Data Science
2.5 yrs. experience in Data Science – Retail, Internet and Finance Industry

Timeseries forecasting by far seems to be one of the most common Data Science tasks – especially in retails or financials institutions. Personally, I was introduced to it when I began my internship in a Private Equity firm, and since then it is of my great interest to explore beyond the traditional methodologies and extend to Machine Learning – Deep Learning treatments. There are researches of the same around ML engineers and practitioners, but below is my attempt to experiment with timeseries in an ML environment.

## DATASET

I would use data from Capital IQ or Prequin where we would try to estimate NAVs, VARs and Cashflows of any given fund. I am trying to replicate the same with an open-source data on Google Stock Prices (timeseries forecasting used more for Stock prices in general) where I worked on predicting only the Open column. There are two different datasets for Train and Test and the basic summary statistics of the two are as below:

```
def return_desc(df):
    return print(df.describe()), print (df.dtypes),print (df.head(3)) ,print(df.isnull().sum()),print (df.shape)

print(return_desc(dataset_train))
```

```
              Open         High          Low
count  1258.000000  1258.000000  1258.000000
mean    533.709833   537.880223   529.007409
std     151.904442   153.008811   150.552807
min     279.120000   281.210000   277.220000
25%     404.115000   406.765000   401.765000
50%     537.470000   540.750000   532.990000
75%     654.922500   662.587500   644.800000
max     816.680000   816.680000   805.140000
Date       object
Open      float64
High      float64
Low       float64
Close      object
Volume     object
dtype: object
       Date    Open    High     Low   Close     Volume
0  1/3/2012  325.25  332.83  324.97  663.59  7,380,500
1  1/4/2012  331.27  333.87  329.08  666.45  5,749,400
2  1/5/2012  329.83  330.75  326.89  657.21  6,590,300
Date      0
Open      0
High      0
Low       0
Close     0
Volume    0
dtype: int64
(1258, 6)
```

**TRAIN SET**

```
print(return_desc(dataset_test))
```

```
             Open        High         Low       Close
count   20.000000   20.000000   20.000000   20.000000
mean   807.526000  811.926500  801.949500  807.904500
std     15.125428   14.381198   13.278607   13.210088
min    778.810000  789.630000  775.800000  786.140000
25%    802.965000  806.735000  797.427500  802.282500
50%    806.995000  808.640000  801.530000  806.110000
75%    809.560000  817.097500  804.477500  810.760000
max    837.810000  841.950000  827.010000  835.670000
Date       object
Open      float64
High      float64
Low       float64
Close     float64
Volume     object
dtype: object
       Date    Open    High     Low   Close     Volume
0  1/3/2017  778.81  789.63  775.80  786.14  1,657,300
1  1/4/2017  788.36  791.34  783.16  786.90  1,073,000
2  1/5/2017  786.08  794.48  785.02  794.02  1,335,200
Date      0
Open      0
High      0
Low       0
Close     0
Volume    0
dtype: int64
(20, 6)
```

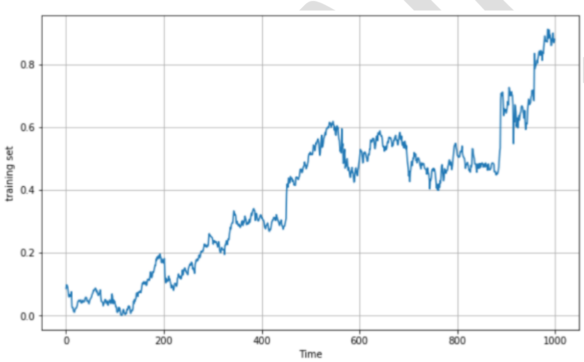
**TEST SET**

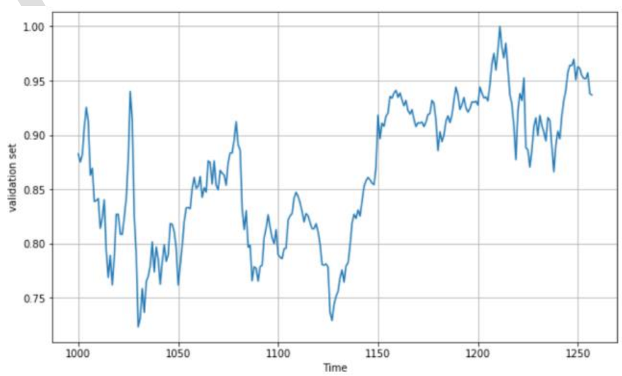
## TRADITIONAL FORECASTING

I started with traditional timeseries methods namely – ARIMA and Unobserved Components Model. A timeseries can be recognized by the following components – Trend, Seasonality, Noise and the below helper functions can help. I have used the time with np.arange for the series plotting in x axis.

```python
import matplotlib.pyplot as plt
#import tensorflow as tf
#from tensorflow import keras

def plot_series(time, series, format="-", start=0, end=None):
    plt.plot(time[start:end], series[start:end], format)
    plt.xlabel("Time")
    plt.ylabel("Value")
    plt.grid(True)

def trend(time, slope=0):
    return slope * time

def seasonal_pattern(season_time):
    """Just an arbitrary pattern, you can change it if you wish"""
    return np.where(season_time < 0.4,
                    np.cos(season_time * 2 * np.pi),
                    1 / np.exp(3 * season_time))

def seasonality(time, period, amplitude=1, phase=0):
    """Repeats the same pattern at each period"""
    season_time = ((time + phase) % period) / period
    return amplitude * seasonal_pattern(season_time)

def noise(time, noise_level=1, seed=None):
    rnd = np.random.RandomState(seed)
    return rnd.randn(len(time)) * noise_level

time = np.arange(4 * 365 + 1, dtype="float32")

baseline = 10

baseline = 10
amplitude = 40
slope = 0.05
noise_level = 5

time = time.reshape(1461,1)
time = time[:1258]
```

I further divided the data into train and dev, until 1000 for train and 1000 to 1258 (end of the dataset) for dev set. Plots of the same as below:

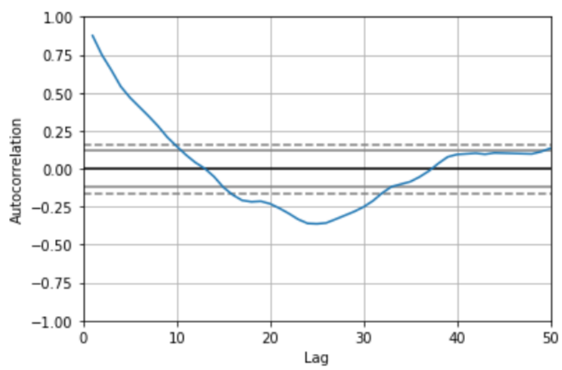


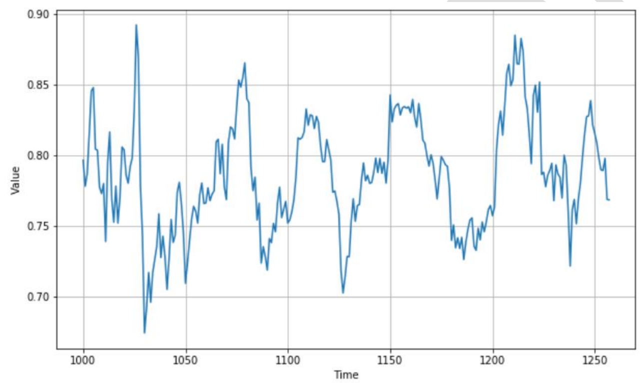

## ARIMA

Arima has three parts Auto Regression (p), Integration (d) and Moving Average (q).

An Autocorrelation plot helps me identify the right (p) value for the model. From the below graph, a (p) value between 7 – 10 looks best.




Clearly from the plots above, there is an upward trend, and we need to make it stationary with differencing. With trail and error, differencing method the actual dataset at time = 1000 gives us a stationary series.

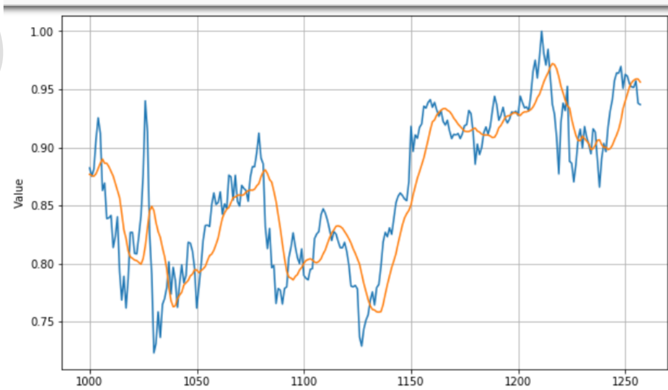



A general moving average with window size of 10 was implemented as below.

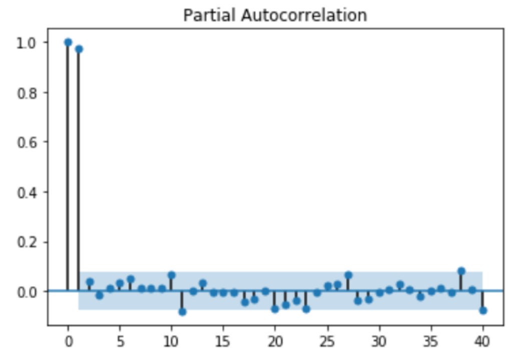
```
def moving_average_forecast(series, window_size):
    forecast = []
    for i in range(len(series) - window_size):
        forecast.append(series[i:i + window_size].mean())
    return np.array(forecast)
```

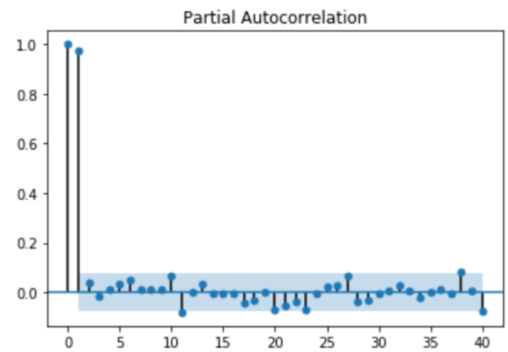
```
moving_avg = moving_average_forecast(training_set_scaled, 10)[split_time -
moving_avg = moving_avg.reshape(258,1)
```

```
plt.figure(figsize=(10, 6))
plot_series(time_valid, x_valid)
plot_series(time_valid, moving_avg)
```



For the differenced stationary series, I used the Partial Autocorrelation plot to figure out the (q) value, 2 as per the below:



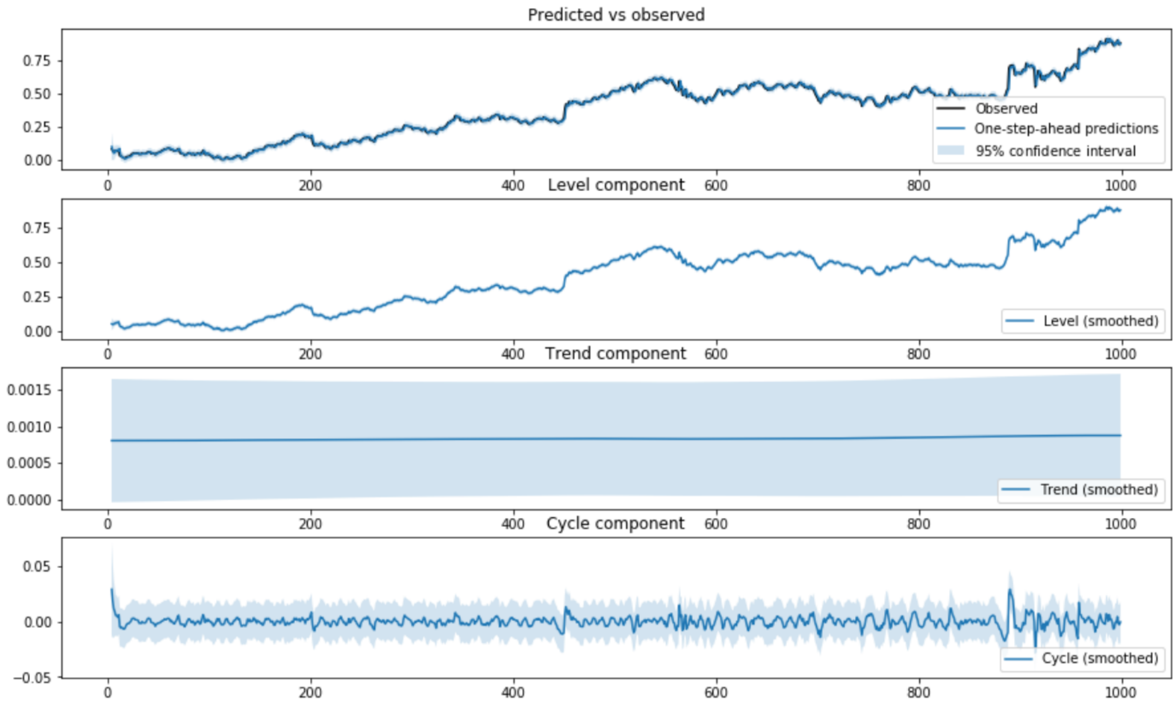

With p,q,d the model from statsmodels.tsa.arima_model is fit to give the below evaluations of AIC and BIC

```
print(model_fit.summary())
```

```
                             ARIMA Model Results
==============================================================================
Dep. Variable:                   D2.y   No. Observations:                  256
Model:                 ARIMA(9, 2, 2)   Log Likelihood                 640.946
Method:                       css-mle   S.D. of innovations              0.020
Date:                Wed, 26 Feb 2020   AIC                          -1255.892
Time:                        21:04:09   BIC                          -1209.804
Sample:                             2   HQIC                         -1237.355

==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const       3.641e-07   1.33e-05      0.027      0.978   -2.57e-05    2.65e-05
ar.L1.D2.y    -0.8255      0.381     -2.166      0.031      -1.573      -0.078
ar.L2.D2.y    -0.1121      0.081     -1.375      0.170      -0.272       0.048
ar.L3.D2.y    -0.0667      0.093     -0.718      0.474      -0.249       0.115
ar.L4.D2.y    -0.1187      0.084     -1.421      0.157      -0.282       0.045
ar.L5.D2.y    -0.1654      0.095     -1.748      0.082      -0.351       0.020
ar.L6.D2.y    -0.0501      0.084     -0.595      0.552      -0.215       0.115
ar.L7.D2.y     0.0026      0.083      0.031      0.975      -0.160       0.165
ar.L8.D2.y     0.0458      0.083      0.551      0.582      -0.117       0.209
ar.L9.D2.y    -0.0066      0.085     -0.078      0.938      -0.173       0.160
ma.L1.D2.y    -0.1607      0.378     -0.426      0.671      -0.901       0.579
ma.L2.D2.y    -0.8376      0.377     -2.223      0.027      -1.576      -0.099
```

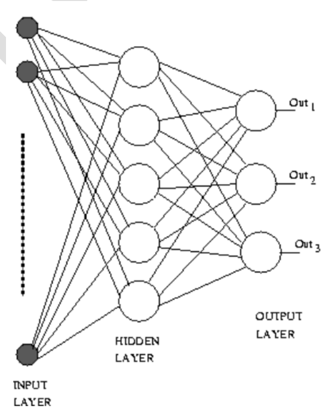
### Unobserved Component Models:

For the UCM, the below graphs depict the predicted against observed values entire training dataset.




### NEURAL NETWORKS

A general neural network architecture starts with the input layer of the feature vectors of ⍰ training examples, hidden layers and one output layers.




The input X matrix must be transformed into a nxm matrix, where n is the no of features and m is the no of training examples. In the given dataset, one column is augmented into a matrix of window size 60, thus making X matrix (60x1198) .

```
X_train
```

```
array([[0.08581368, 0.09701243, 0.09433366, ..., 0.07846566, 0.08034452,
        0.08497656],
       [0.09701243, 0.09433366, 0.09156187, ..., 0.08034452, 0.08497656,
        0.08627874],
       [0.09433366, 0.09156187, 0.07984225, ..., 0.08497656, 0.08627874,
        0.08471612],
       ...,
       [0.92106928, 0.92438053, 0.93048218, ..., 0.95475854, 0.95204256,
        0.95163331],
       [0.92438053, 0.93048218, 0.9299055 , ..., 0.95204256, 0.95163331,
        0.95725128],
       [0.93048218, 0.9299055 , 0.93113327, ..., 0.95163331, 0.95725128,
        0.93796041]])
```

```
print(X_train.shape, y_train.shape)
```

```
(1198, 60) (1198, 1)
```

The general methodology of neural network has the following steps:

i. Initialize Parameters: W, B
ii. Loop for Number of Iterations:
    a. Forward Propagation
    b. Cost Function
    c. Backward Propagation
iii. Update Parameters

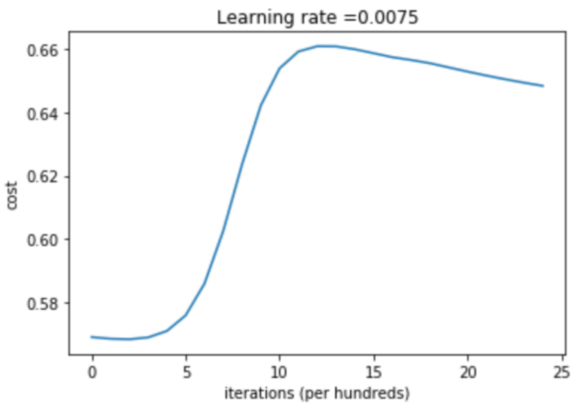
Activation Functions – Sigmoid, ReLu and tanh can be defined as well, but for numerical regression prediction, a linear activation is best suited.

```
def sigmoid(Z):

    """
    Implements the sigmoid activation in numpy

    """
    A = 1/(1+np.exp(-Z))
    cache = Z
    return A, cache


def relu(Z):

    """


    Implement the RELU function.


    """
    A = np.maximum(0,Z)
    assert(A.shape == Z.shape)
    cache = Z
    return A, cache


def relu_backward(dA, cache):

    """


    Implement the backward propagation for a single RELU unit.

    """
    Z = cache
    dZ = np.array(dA, copy=True) # just converting dz to a correct object.
    # When z <= 0, you should set dz to 0 as well.
    dZ[Z <= 0] = 0
    assert (dZ.shape == Z.shape)
    return dZ


def sigmoid_backward(dA, cache):

    """
    Implement the backward propagation for a single SIGMOID unit.

    """
    Z = cache
    s = 1/(1+np.exp(-Z))
    dZ = dA * s * (1-s)
    assert (dZ.shape == Z.shape)
    return dZ
```

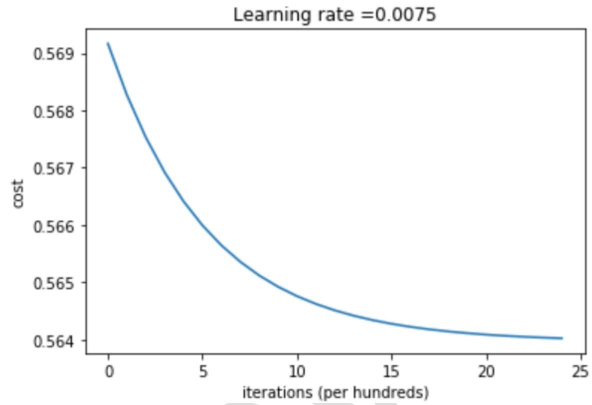
For a two layer model, after defining the loop for num_iterations:

```
In [48]: # GRADED FUNCTION: two_layer_model

def two_layer_model(X, Y, layers_dims, learning_rate = 0.0075, num_iterations = 3000, print_cost=False):

    np.random.seed(1)
    grads = {}
    costs = []                              # to keep track of the cost
    m = X.shape[1]                           # number of examples
    (n_x, n_h, n_y) = layers_dims

    # Initialize parameters dictionary, by calling one of the functions you'd previously implemented
    ### START CODE HERE ### (≈ 1 line of code)
    parameters = initialize_parameters(n_x, n_h, n_y)
    ### END CODE HERE ###

    # Get W1, b1, W2 and b2 from the dictionary parameters.
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]

    # Loop (gradient descent)
    for i in range(0, num_iterations):

        # Forward propagation: LINEAR -> RELU -> LINEAR -> SIGMOID. Inputs: "X, W1, b1, W2, b2". Output: "A1, cache1, A2, cache2
        ### START CODE HERE ### (≈ 2 lines of code)
        A1, cache1 = linear_activation_forward(X, W1, b1, activation='relu')
        A2, cache2 = linear_activation_forward(A1, W2, b2, activation='sigmoid')
        ### END CODE HERE ###

        # Compute cost
        ### START CODE HERE ### (≈ 1 line of code)
        cost = compute_cost(A2, Y)
        ### END CODE HERE ###

        # Initializing backward propagation
        dA2 = - (np.divide(Y, A2) - np.divide(1 - Y, 1 - A2))

        # Backward propagation. Inputs: "dA2, cache2, cache1". Outputs: "dA1, dW2, db2; also dA0 (not used), dW1, db1".
        ### START CODE HERE ### (≈ 2 lines of code)
        dA1, dW2, db2 = linear_activation_backward(dA2, cache2, activation='sigmoid')
        dA0, dW1, db1 = linear_activation_backward(dA1, cache1, activation='relu')
        ### END CODE HERE ###

        # Set grads['dWl'] to dW1, grads['db1'] to db1, grads['dW2'] to dW2, grads['db2'] to db2
        grads['dW1'] = dW1
        grads['db1'] = db1
        grads['dW2'] = dW2
        grads['db2'] = db2

        # Update parameters.
        ### START CODE HERE ### (approx. 1 line of code)
        parameters = update_parameters(parameters, grads, learning_rate=learning_rate)
        ### END CODE HERE ###

        # Retrieve W1, b1, W2, b2 from parameters
        W1 = parameters["W1"]
        b1 = parameters["b1"]
        W2 = parameters["W2"]
        b2 = parameters["b2"]

        # Print the cost every 100 training example
        if print_cost and i % 100 == 0:
            print("Cost after iteration {}: {}".format(i, np.squeeze(cost)))
        if print_cost and i % 100 == 0:
            costs.append(cost)

    # plot the cost
    plt.plot(np.squeeze(costs))
    plt.ylabel('cost')
    plt.xlabel('iterations (per hundreds)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters
```

The Learning rate curve for 2-layer and 4-layer model can be seen as below, with accuracy of 64, and 56 respectively for 2500 iterations.



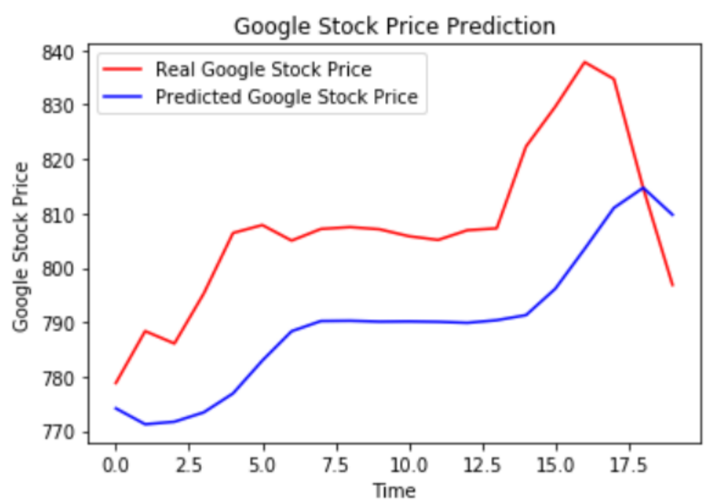

**2 Layer Model**




**4 Layer Model**

## Deep Learning - LSTM

The LSTM is a category of Recurrent Neural Network which has an additional memory unit that is passed on to other cells in the loop iterations. LSTMs are by far the most widely used neural nets for timeseries forecasting. I used the Keras Sequential and dense models for the layers and dropout of 0.2 for regularization. With epoch of 100, the accuracy measured in Mean squared Error is up to 0.0013. The predicted values of test against the actual test values can be visualized as below:



**Model**

```
# Initialising the RNN
regressor = Sequential()
# Adding the first LSTM layer and some Dropout regularisation
regressor.add(LSTM(units = 50, return_sequences = True, input_shape = (X_train.shape[1], 1)))
regressor.add(Dropout(0.2))

# Adding a second LSTM layer and some Dropout regularisation
regressor.add(LSTM(units = 50, return_sequences = True))
regressor.add(Dropout(0.2))

# Adding a third LSTM layer and some Dropout regularisation
regressor.add(LSTM(units = 50, return_sequences = True))
regressor.add(Dropout(0.2))

# Adding a fourth LSTM layer and some Dropout regularisation
regressor.add(LSTM(units = 50))
regressor.add(Dropout(0.2))

# Adding the output layer
regressor.add(Dense(units = 1))

# Compiling the RNN
regressor.compile(optimizer = 'adam', loss = 'mean_squared_error')
```