LSTM Example

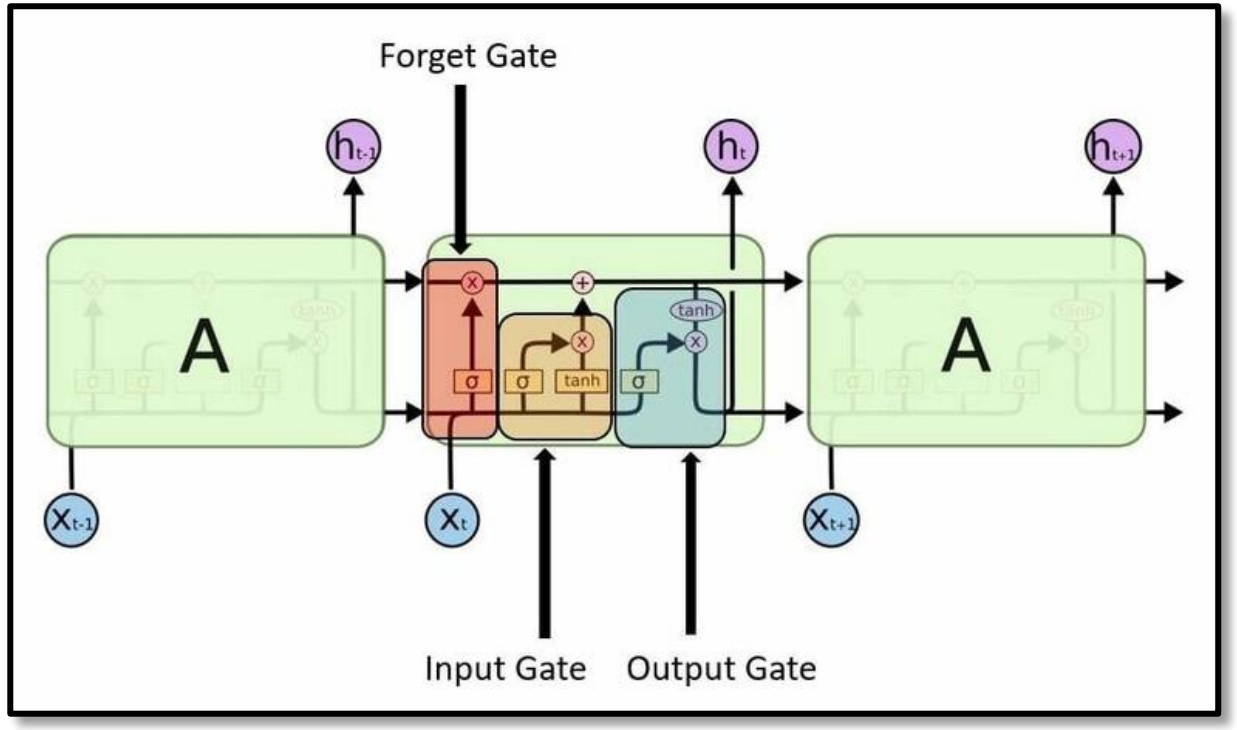
LSTM Example

LSTM

In previous presentations, we studied Long Short-Term Memory (LSTM) cells. These types of cells enable us to capture long-term dependencies that simple RNNs cannot effectively model.

We then examined the different gates in an LSTM cell, including the forget gate, input gate, and output gate. We explored how these gates work together to prepare both the current state and the output of the cell.

In this presentation, we will solve a problem using this architecture in Python.



LSTM Example

S&P500

The Standard and Poor's 500 (S&P500) index is one of the most widely followed indicators of the U.S. stock market performance. Accurately predicting the overall trend of this index is crucial for stock market traders and investment firms to make informed decisions. We can treat the S&P500 index as a time series and aim to forecast its next day's value using its historical data. To effectively capture potential long-term dependencies in this series, we will utilize Long Short-Term Memory (LSTM) networks. This model can provide valuable insights for stock market participants and contribute to more informed investment decisions

Import Modules

First, we need to import the modules that we are going to use to preprocess the dataset and build the model.
We need NumPy to reshape the data. We will import our data into the project using Pandas. We will use Matplotlib to visualize the data. We will use the Keras Sequential API along with Keras Dense and LSTM layers to build our model. We will use the Keras Sequential API along with Keras Dense and LSTM layers to build our model. We will use the mean squared error from sklearn to evaluate the model.

```
# Simple LSTM for a time series data
import numpy as np
import matplotlib.pyplot as plt
from pandas import read_csv
import math
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
import pylab
```

LSTM Example

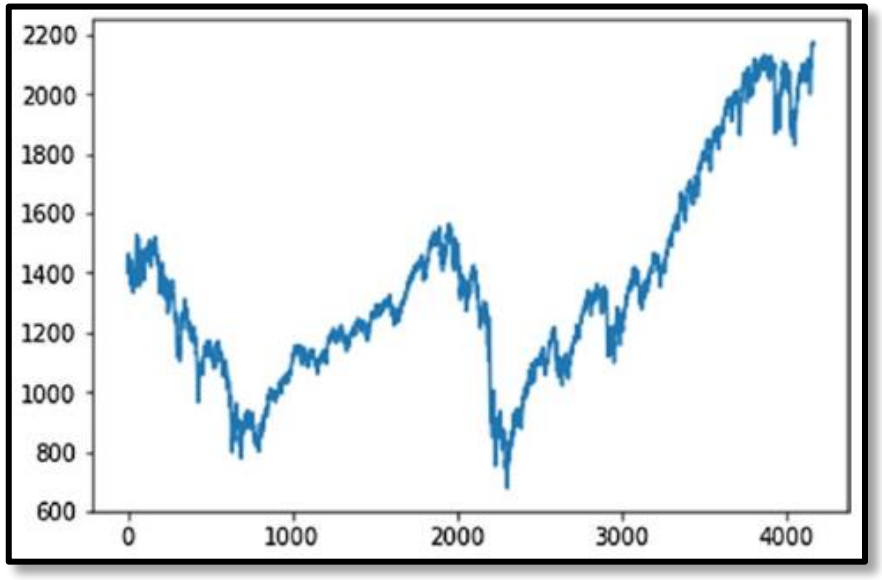
Time series

We will define a function that converts our dataset into a time series, which can be used to train and evaluate our LSTM model.
We will define two empty lists: one for inputs and one for labels. Then, we calculate the total number of possible data points by subtracting the lag value from the length of the dataset. for each possible time step, we create a list of the inputs and append it to the inputs list. Then, we append the corresponding label to the labels list. Finally, we create NumPy arrays from the two lists and return these NumPy arrays.

```
# convert an array of values into a timeseries data
def create_timeseries(series, ts_lag=1):
    dataX = []
    dataY = []
    n_rows = len(series)-ts_lag
    for i in range(n_rows-1):
        a = series[i:(i+ts_lag), 0]
        dataX.append(a)
        dataY.append(series[i + ts_lag, 0])
    X, Y = np.array(dataX), np.array(dataY)
    return X, Y
```

Load the dataset

First, we set the random seed to 230 for reproducibility. This ensures that we get the same result every time we run the code.
Next, we load the dataset from the CSV file using Pandas. To ensure that the dataset has been loaded without any errors, we can plot the dataset. Our dataset contains the value of the S&P500 index for the last 4000 days. By examining the resulting graph, we can confirm that the dataset was loaded correctly.




```
# fix random seed for reproducibility
np.random.seed(230)
# load dataset
dataframe = read_csv('sp500.csv', usecols=[0])
plt.plot(dataframe)
plt.show()
```

Preprocessing the data

Now that we have loaded the dataset, we will convert the values into float.
Then, we will use the StandardScaler to scale our dataset. This will facilitate the model training process and result in better accuracy after the training is completed.
Finally, we will split the dataset into training and testing sets. We will use the first 75 percent of the data as the training dataset and the remaining 25 percent as the testing dataset.

```
# Changing datatype to float32 type
series = dataframe.values.astype('float32')

# Normalize the dataset
scaler = StandardScaler()
series = scaler.fit_transform(series)
```

```
# split the datasets into train and test sets
train_size = int(len(series) * 0.75)
test_size = len(series) - train_size
train, test = series[0:train_size,:], series[train_
size:len(series),:]
```

Preprocessing the data

Now that we have preprocessed the dataset, we will use the function that we defined earlier to convert our testing and training datasets into time series data. Then, we will reshape both the testing and training inputs to a size that is expected by our model. After this step, our time series are ready, and we can proceed to define our model.

```
# reshape the train and test dataset into X=t and Y=t+1
ts_lag = 1
trainX, trainY = create_timeseries(train, ts_lag)
testX, testY = create_timeseries(test, ts_lag)

# reshape input data to be [samples, time steps, features]
trainX = np.reshape(trainX, (trainX.shape[0], 1, trainX.
shape[1]))
testX = np.reshape(testX, (testX.shape[0], 1, testX.shape[1]))
```

LSTM Example

Training the Model

Defining an LSTM model is not very different from defining a simple RNN model. We have previously covered how to define an RNN using Keras. To build an LSTM layer in Keras, we are going to use the Keras LSTM layer. We will use an LSTM layer with 10 cells. Then, we will add one dense layer with a single neuron, which will generate our output. As we are solving a regression problem, we will use the mean squared logarithmic error as our loss function and Adagrad as our optimizer. Finally, we will train the model for 500 epochs on the training dataset.

```
# Define the LSTM model
model = Sequential()
model.add(LSTM(10, input_shape=(1, ts_lag)))
model.add(Dense(1))
model.compile(loss='mean_squared_logarithmic_error',
optimizer='adagrad')

# fit the model
model.fit(trainX, trainY, epochs=500, batch_size=30)
```

Predict and Rescale

Now we will use the trained model to make predictions. We will predict the results for both the training and testing datasets. Then, we will rescale the predictions and the actual values to their initial scale using the inverse_transform function. This will help us have a more meaningful comparison and evaluation of our model's performance.
By rescaling the data back to its original form, we can interpret the results in the context of actual S&P500 index values. This step is crucial for assessing the practical significance of our predictions and for communicating the results to stakeholders who may not be familiar with scaled data.

```
# make predictions
trainPredict = model.predict(trainX)
testPredict = model.predict(testX)

# rescale predicted values
trainPredict = scaler.inverse_transform(trainPredict)
trainY = scaler.inverse_transform([trainY])
testPredict = scaler.inverse_transform(testPredict)
testY = scaler.inverse_transform([testY])
```

RMSE

Now that we have used the model to make predictions on the training and testing datasets and have rescaled the predictions and the actual values to their initial scale, we can calculate the Root Mean Squared Error (RMSE) of our model on both datasets.
We use the mean_squared_error function from sklearn.metrics to calculate the MSE. Then, we Apply the sqrt method to take the square root of the MSE. In the end we will print the results with 2 decimal places for easy interpretation.

```
# calculate root mean squared error
trainScore = math.sqrt(mean_squared_error(trainY[0],
trainPredict[:,0]))
print('Train Score: %.2f RMSE' % (trainScore))
testScore = math.sqrt(mean_squared_error(testY[0],
testPredict[:,0]))
print('Test Score: %.2f RMSE' % (testScore))
```

LSTM Example

Visual Comparison

In the end, we will have a visual comparison of the actual values and the predicted values. We will plot the actual values in blue. Then, we will plot the predicted values of the training dataset in orange and the predicted values of the testing dataset in green.
By examining the plot, we can visually confirm that the predicted values for both the testing and training datasets are very close to the actual values. This indicates that our model has learned effectively from the training data and is doing an acceptable job of predicting the next day's value of the S&P500.

```
# plot baseline and predictions
pylab.plot(trainPredictPlot)
pylab.plot(testPredictPlot)
pylab.show()
```