UAEM | Universidad Autónoma del Estado de México

UNIVERSIDAD AUTONOMA DEL ESTADO DE MEXICO

CENTRO UNIVERSITARIO UAEM ZUMPANGO

INGENIERIA EN COMPUTACION

UA: Redes Neuronales

Proyecto: Red neuronal que predice acciones en la bolsa de valores

Diciembre 2020

ALUMNO: ZAMORA RAMON OMAR

NC: 1624301

Como primer paso buscaremos un set de datos en donde se muestre una tabla de valores en los últimos 200 días, usaremos una api llamada alphavantage en donde por medio de un script creado en Python obtendremos toda la información de las acciones de cualquier empresa.

```
 8
 9  from alpha_vantage.timeseries import TimeSeries
10  import json
11
12  def save_dataset(symbol):
13      api_key ='C2KG22FO4I0U600O'
14
15      ts = TimeSeries(key=api_key, output_format='pandas')
16      data, meta_data = ts.get_daily(symbol, outputsize='full')
17      data.to_csv('./{}_daily.csv'.format(symbol))
18
19  simbolo = 'AAPL'
20  save_dataset(simbolo)
```

En la línea 9 se ve como se importa una librería de la api alphavantage, en la línea 10 se importa la librería json, seguido de la línea 12 declaramos una función para poder obtener nuestra data, en la línea 13 se obtiene la clave de api que te proporciona la aplicación, en la línea 19 y 20 se declara el símbolo de la acción de la cual obtendremos nuestra información. Al ejecutar el script nos devuelve un archivo en formato csv de nuestra información.

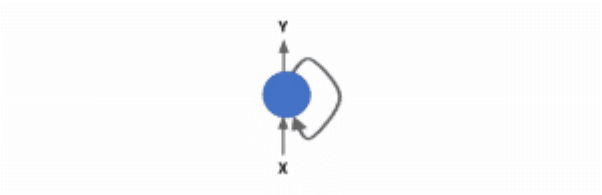
Para programar utilizaremos una red neuronal recurrente LTS

¿Qué es una red neuronal recurrente?

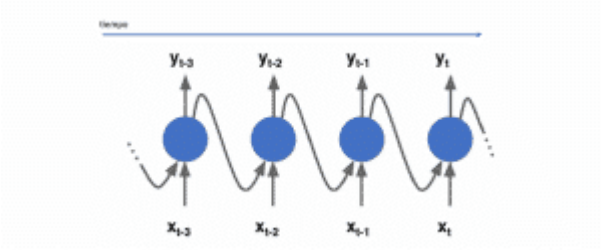
Las redes neuronales recurrentes, o *Recurrent Neural Networks*(RNN) en inglés, son una clase de redes para analizar datos de series temporales permitiendo tratar la dimensión de “tiempo”.

Las redes neuronales recurrentes (RNN) fueron ya concebidas en la década de 1980. Pero estas redes han sido muy difíciles de entrenar por sus requerimientos en computación y hasta la llegada de los avances de estos últimos años, no se han vuelto más accesibles y popularizado su uso por la industria.

Imaginemos la RNN más simple posible, compuesta por una sola neurona que recibe una entrada, produciendo una salida, y enviando esa salida a sí misma, como se muestra en la siguiente figura:



En cada instante de tiempo (también llamado *timestep* en este contexto), esta neurona recurrente recibe la entrada xde la capa anterior, así como su propia salida del instante de tiempo anterior para generar su salida y. Podemos representar visualmente esta pequeña red desplegada en el eje del tiempo como se muestra en la figura:



iguiendo esta misma idea, una capa de neuronas recurrentes se puede implementar de tal manera que, en cada instante de tiempo, cada neurona recibe dos entradas, la entrada correspondiente de la capa anterior y a su vez la salida del instante anterior de la misma capa.

Ahora cada neurona recurrente tienen dos conjuntos de parámetros, uno que lo aplica a la entrada de datos que recibe de la capa anterior y otro conjunto que lo aplica a la entrada de datos correspondiente al vector salida del instante anterior. Sin entrar demasiado en formulación, y siguiendo la notación explicada en la primera parte del libro, podríamos expresarlo de la siguiente manera:

$yt= f (Wxt+ Uyt−1+b)$

Donde x=(x1,…,xT) representa la secuencia de entrada proveniente de la capa anterior, W los pesos de la matriz y bel bias vistos ya en las anteriores capas. Las RNN extienden esta función con una conexión recurrente en el tiempo donde U es la matriz de pesos que opera sobre el estado de la red en el instante de tiempo anterior (yt−1) anterior. Ahora, en la fase de entrenamiento a través del Backpropagation también se actualizan los pesos de esta matriz.

Redes LSTM

Las LSTM son un tipo especial de redes recurrentes. La característica principal de las redes recurrentes es que la información puede persistir introduciendo bucles en el diagrama de la red, por lo que, básicamente, pueden «recordar» estados previos y utilizar esta información para decidir cuál será el siguiente. Esta característica las hace muy adecuadas para manejar series cronológicas. Mientras las redes recurrentes estándar pueden modelar dependencias a corto plazo (es decir, relaciones cercanas en la serie cronológica), las LSTM pueden aprender dependencias largas, por lo que se podría decir que tienen una «memoria» a más largo plazo

Pasos para crear nuestras NN

1.- Declaramos las librerías que vamos a ocupar

```
1  import numpy as np
2  np.random.seed(4)
3  import pandas as pd
4
```

2.- Importamos el set de datos a utilizar

```
5  # Lectura de los datos
6  #dataset = pd.read_csv('Apple.csv', index_col='Date', parse_dates=['Date'])
7  #dataset = pd.read_csv('Amazon.csv', index_col='Date', parse_dates=['Date'])
8  dataset = pd.read_csv('Facebook.csv', index_col='Date', parse_dates=['Date'])
9  dataset.head()
```

Para este proyecto aremos la prediccion con el valor mas alto de la accion

```
    ...: dataset.head()
Out[19]:
            1. Open  2. High   3. Low  4. Close  5. Volume
Date
2020-02-03   194.03   196.57  188.8500    196.44   24948955
2020-03-03   196.22   197.24  183.9700    185.89   27984104
2020-04-03   189.17   191.83  186.3900    191.76   23062473
2020-05-03   186.78   188.99  183.8901    185.17   19333425
2020-06-03   178.33   183.78  176.2600    181.09   24559550

In [20]:
```

3.- Para el entrenamiento usaremos 200 días que van desde 02/03/2020 hasta 15/10/2020 y para la predicción usaremos los datos de 16/10/2020 hasta 11/12/2020

```
10
11  # Sets de entrenamiento y validación
12  # La LSTM se entrenará con datos de 15/10/2020 hacia atrás. La validación se hará con datos de 16/10/
13  # En ambos casos sólo se usará el valor más alto de la acción para cada día
14  set_entrenamiento = dataset[:'2020'].iloc[:,1:2]
15  set_validacion = dataset['2020':].iloc[:,1:2]
16
```

4.- Para que se le facilite el procesamiento a nuestra neurona utilizaremos una técnica llamada normalización con la librería scikit learn

```
18  from sklearn.preprocessing import MinMaxScaler
19  sc = MinMaxScaler(feature_range=(0,1))
20  set_entrenamiento_escalado = sc.fit_transform(set_entrenamiento)
```

5.- Ahora utilizaremos una red LSTM debido a que estamos utilizando la variación del precio máximo de la acción con respecto al tiempo, para entrenarla utilizaremos bloques de 5 datos consecutivos almacenados en la variable x y el sexto dato será usado como salida o y

```
24  time_step = 5
25  X_train = []
26  Y_train = []
27  m = len(set_entrenamiento_escalado)
28
29  for i in range(time_step,m):
30      # X: bloques de "time_step" datos: 0-time_step, 1-time_step+1, 2-time_step+2, etc
31      X_train.append(set_entrenamiento_escalado[i-time_step:i,0])
32
33      # Y: el siguiente dato
34      Y_train.append(set_entrenamiento_escalado[i,0])
35  X_train, Y_train = np.array(X_train), np.array(Y_train)
```

6.- Antes de crear el modelo debemos de reajustar el tamaño del set de entrenamiento convirtiéndolo en un vector de 5x1

```
37  # Reshape X_train para que se ajuste al modelo en Keras
38  X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))
39
```

7.- Ahora si podemos crear la red, esta va a tener un tamaño de 50 neuronas, la función secuential permitirá crear el contenedor de la red lstm, después añadimos la red LSTM especificando el tamaño de las neuronas y el tamaño de la entrada, para la capa de salida utilizamos la función dense y especificamos que el dato de salida tendrá un tamaño igual a uno, para el entrenamiento utilizaremos el método rmsprop que funciona de manera similar al método del gadiente desendiente, la función de error será error cuadratico medio y usaremos lotes de 32 ejemplos con un total de 20 iteraciones.

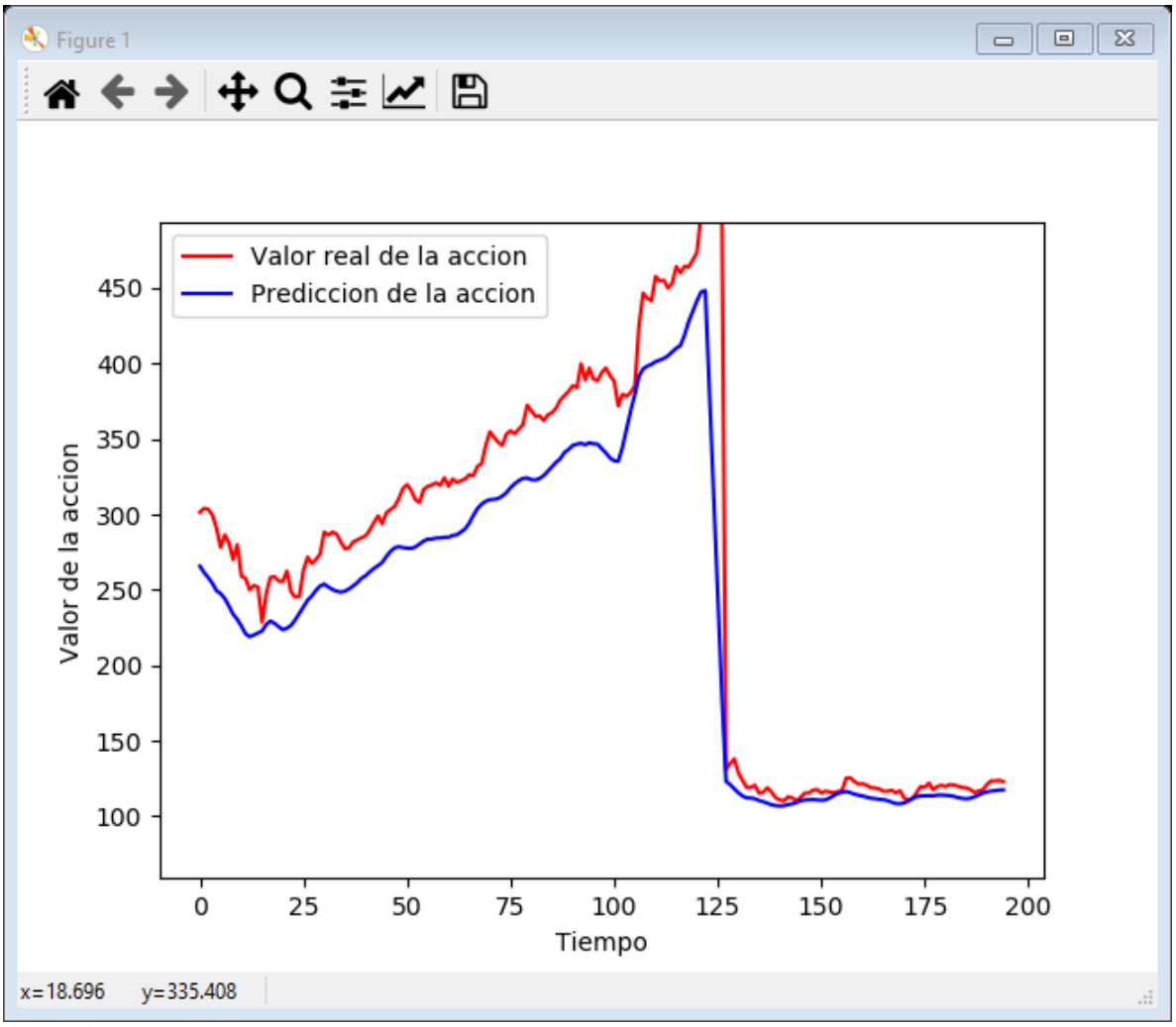
```python
# Red LSTM
from keras.models import Sequential
from keras.layers import Dense, LSTM
dim_entrada = (X_train.shape[1],1)
dim_salida = 1
na = 50
modelo = Sequential()
modelo.add(LSTM(units=na, input_shape=dim_entrada))
modelo.add(Dense(units=dim_salida))
modelo.compile(optimizer='rmsprop', loss='mse')
modelo.fit(X_train,Y_train,epochs=20,batch_size=32)
```

8.- Preparamos el set de validación normalizando los datos en el rango de 0 a 1, reorganizamos el set para crear bloques de 5 datos y finalmente realizamos la predicción usando la función predict y aplicamos la normalización inversa para que quede en la escala real de las acciones

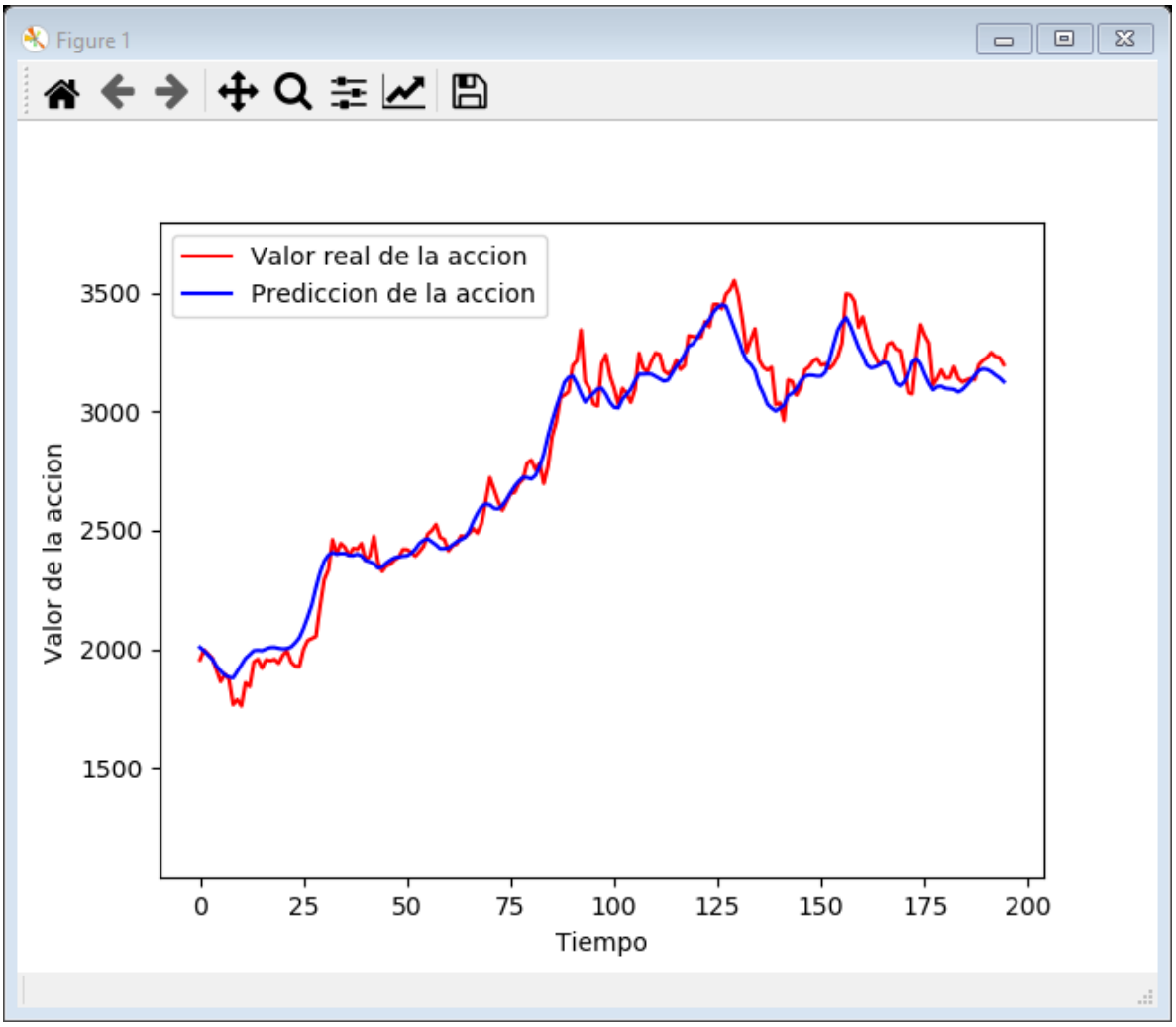
```python
# Validación (predicción del valor de las acciones)
x_test = set_validacion.values
x_test = sc.transform(x_test)

X_test = []
for i in range(time_step,len(x_test)):
    X_test.append(x_test[i-time_step:i,0])
X_test = np.array(X_test)
X_test = np.reshape(X_test, (X_test.shape[0],X_test.shape[1],1))

prediccion = modelo.predict(X_test)
prediccion = sc.inverse_transform(prediccion)
```

9.- Finalmente graficamos los datos reales y los datos predichos por nuestra red.

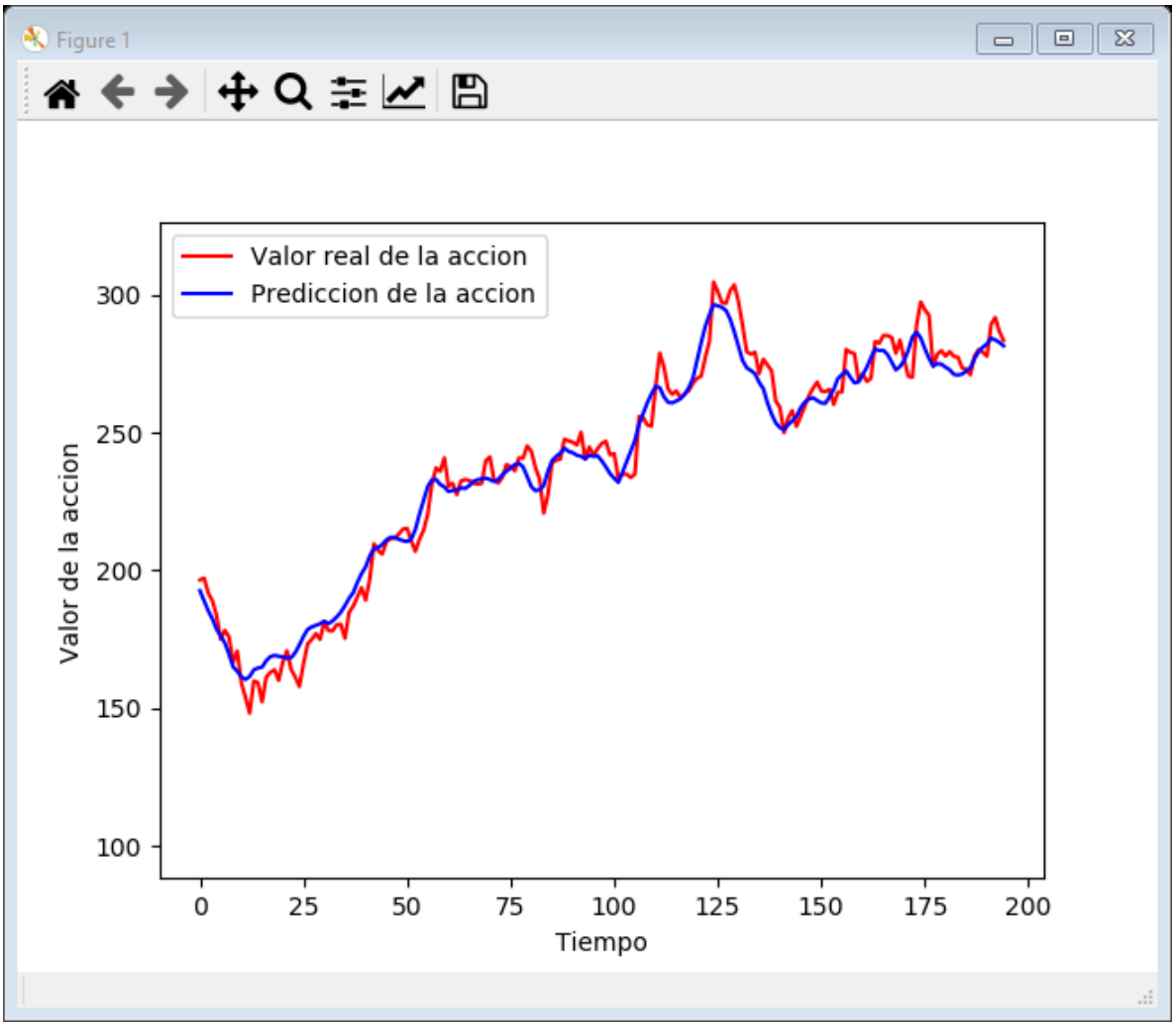
```python
def graficar_predicciones(real, prediccion):
    plt.plot(real[0:len(prediccion)],color='red', label='Valor real de la accion')
    plt.plot(prediccion, color='blue', label='Prediccion de la accion')
    plt.ylim(1.1 * np.min(prediccion)/2, 1.1 * np.max(prediccion))
    plt.xlabel('Tiempo')
    plt.ylabel('Valor de la accion')
    plt.legend()
    plt.show()

graficar_predicciones(set_validacion.values,prediccion)
```



En esta grafica estamos prediciendo las acciones de apple por lo cual nuestra neurona se equivoco aproximadamente en un 5% de los datos reales.



En la bolsa de acciones de amazon se demuestra que es casi exacto la predicción



Finalmente en nuestra tercer prueba también obtuvimos muy buenos resultados