Chapter 12 : Part 3

**Deep Learning**

**RNN: Building a RNN**

Python Implementation

**12.3.1 Problem Description**

* Here we predict the ***stock price*** of Google. If you have some notions in ***financial*** ***engineering***, you already know that it's *pretty* *challenging*, since indeed there is the ***Brownian Motion*** that states that the *future variations* of the *stock* *price* are *independent* from the *past*.
* So it's actually *impossible* to predict *exactly* the *future stock price* otherwise we would *all* become *billionaires* but it's actually possible to predict some trends.
* So we're gonna try to predict the upward and downward trends that exist in the Google stock price.
* Our model: The model that we will implement will be an LSTM.
* Our LSTM will try to ***capture*** the *downward* and *upward* *trend* of the ***Google stock price***.
* We're not gonna implement a *simple LSTM*, it's gonna be super *robust* with some *high-dimensionality*, several *layers*, it's gonna be a *stacked* *LSTM*, then we're gonna add some *dropout* *regularization* to avoid overfilling and we will use the *most powerful optimizer* that we have in the *Keras* Library.
* Our approach: We're gonna train our *LSTM* model on five years of the *Google stock price*, from the *beginning* of *2012* to the *end* of *2016*.
* Based on this training, and on the correlations identified by the LSTM of the *Google stock price*, we will try to predict the first month, January 2017.
* Remember, we're *not* going to *try* to *predict* exactly the *stock* *price*, we're gonna try to *predict* the *trend*, the *upward* or *downward* *trend* of the Google stock price.

|  |  |
| --- | --- |
| * Data-set: In our working directory, there are two ***.csv*** files, one for ***train*** and other for ***test***. * The train data, ***Google\_Stock\_Price\_Train.csv*** contains the ***Google stock price from beginning 2012 to the end of 2016***, and then the test data ***Google\_Stock\_Price\_Test.csv*** that contains the ***first month of 2017***, that is, the whole financial month of January 2017. * In the ***Google\_Stock\_Price\_Train.csv*** and we're gonna try to predict the **Open** i.e. the stock price at the beginning of the financial day. Let's go to Google Sheet, Excel or any spreadsheet to have closer look at the Google stock price. * MsExcel: Choose columns and Insert > Line : choose the style. So let's have a look at the Google stock price during this period. |  |

* You can see, the *beginning* of *2012* to the end of 2016 we have some *upward trends*, globally, with several *downward* *trends*. So that's the training set.
* Remember that these are financial days, so there's no Saturday or Sunday.
* The LSTM model predict the *test-data*. It will have no idea of what the *Google stock price* will be right after *2016*.
* Once our model is *trained*, we will predict the *Google* *stock* *price* for the whole month of *January*, and we will compare our predictions to the *actual* results that are in our *test-dataset*.
* We proceed the following steps:
* Data-preprocessing: We do it from scratch. Different from ANN/CNN.
* Building RNN-Architecture: Define the layer, no. of neurons, output-layer.
* Prediction: We use our RNN to predict the test-dataset. We will make the predictions on January 2017.
* Visualization: Finally we visualize and compare our Train & Test results.

**12.3.2 Data-preprocessing**

It is different than ANN & CNN. We're gonna implement it from scratch.

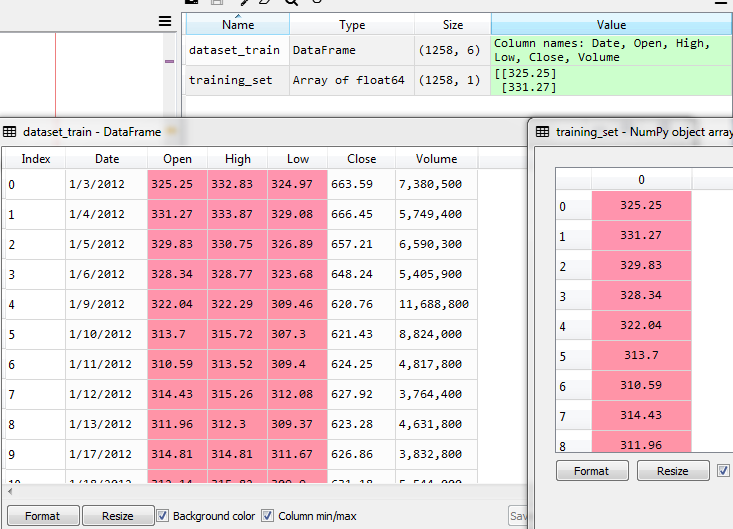
* Libraries: Following are the essential libraries we're gonna use to implement the RNN.

**import** pandas **as** pd

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

* Import the training sets: Notice, we are importing the ***training-set*** and not the whole ***data-set***. Because we are gonna train our *RNN* on only the *training* *sets*. The RNN will have *no idea* of what's going on in a *test set*.



* It's like the test-set doesn't exist for the RNN (not even for validation). We're not importing the training-set right now, we do that after train the RNN. Once the training is done, we will introduce the test-set to the RNN.

dataset\_train = **pd.read\_csv**('Google\_Stock\_Price\_Train.csv')     # *Notice train-set is now our main dataset*

training\_set = dataset\_train.iloc[:, 1:2].values

* First line import the data as a DataFrame using Pandas.
* The second line creates the NumPy array using the right columns. We did two things, selecting the ***right*** ***column*** and creating a ***NumPy*** ***array***.
* It's the real ***training*** ***set*** containing the *input* data of the *NN*.
* ***iloc[ :, 1:2]*** Start from column *indexed* 1 and end at ***(2-1)=1*** *indexed* column and *exclude* column *indexed* ***2***. First '**:**' take all the rows,
* ***data.iloc[3]***

Gives the 4th row.

* ***data.iloc[1:3]***

will give row indexed ***1***, ***2*** and exclude 3 (i.e. 2nd, 3rd rows excluding 4th row.). ***iloc[k:m]*** starts from ***k*** (including ***k***), ends at ***m-1*** excluding ***m***.

* ***data.iloc[ i:j, n:m]***

Gives rows starting from i and ends at ***j-1*** excluding ***j***. And columns starting from ***n*** (including ***n***), ends at ***m-1*** excluding ***m***.

* "**.values**" transform the data to NumPy-array. So we don't have a vector, we have a NumPy array of one column.

**12.3.3 Feature scaling**

We know there are *standardization* and *normalization*. What are we going to use this time for *RNN*? In previous chapter we used *standardization*.

|  |  |
| --- | --- |
| Standardization | Normalization |
|  |  |

* Normalization is more relevant for RNN: Whenever you build an ***RNN*** and especially if there is a ***sigmoid function*** as the ***activation*** function in the ***output layer*** of you RNN, well I recommend to apply Normalization.
* We're going to use the ***MinMaxScaler*** class from *preprocessing* module of *sklearn*.

**from** sklearn.preprocessing **import** MinMaxScaler

* Next we create scale object ***sc***, we have to input some arguments, ***feature\_range*** equals ***(0, 1)***, that's the default feature range.

sc = **MinMaxScaler**(feature\_range= (0, 1))

* The new *scaled* *stock* *prices* will be between *zero* and *one*.
* The last thing we need to do, is of course to apply this ***sc*** object to apply normalization to ***training\_set***.

train\_set\_scled = **sc.fit\_transform**(training\_set)

* We create a new variable ***train\_set\_scaled***, because it is recommended to keep your *original datasets*.
* Then we apply ***fit\_transform*** as we did in previous chapters, which will not only fit your object **sc** to the **training\_set**, also *transform* it, that is scale it.
* Here fit means it just going to get the *min* and the *max* of the *data*.
* The transform method, it's going to compute for each of the stock prices of the training set, the scaled stock prices according to normalization formula.

# *feature scaling*

**from** sklearn.preprocessing **import** MinMaxScaler

sc = **MinMaxScaler**(feature\_range= (0, 1))

train\_set\_scled = **sc.fit\_transform**(training\_set)

* After *executing* the code, we obtained our training set *scaled/normalized* between ***0*** and ***1***. The last values are close to ***1*** because remember, the stock price was going up between 2012 and 2016 i.e. getting closer to max value.

|  |  |
| --- | --- |
|  |  |

* Now we have now the *right values* for our future *RNN* that we're going to *build*.

**12.3.4 Time steps – Data-structure**

In this step, we'll create a specific Data Structure, that's the most important step actually of data pre-processing for RNN.

* We're going to create a Data Structure specifying what the RNN will need to remember when *predicting* the *next stock price*.
* And this is called the ***number of time steps***. This is important to have the right number of ***time steps*** because a wrong number of time steps could lead to over fitting or nonsense predictions.
* time-steps: We have to create a *special data structure* with, *60 time-steps* and *one* *output*.
* 60 time-steps means that at each time ***T***, the RNN is going to look at the ***60*** stock prices before ***time T***, that is ***the stock prices*** between ***60*** days before time ***T*** and time ***T***.
* Based on the ***trends*** of these ***60*** previous ***timesteps***, it will try to predict the ***next*** ***output***.
* So ***60 timesteps*** of the ***past*** ***information*** from which our RNN is gonna try to learn and understand some correlations, or some trends, and based on its understanding, it's going to try to predict the next output. That is, the stock price at time ***T + 1***.
* Why 60 time steps: It is fixed by some experiments, trying different number of time steps.
* *One* time *step* first, which is completely stupid, because it led to *overfitting*. The model won't learn anything.
* Then *20 timesteps*, which was *not enough* to be able to *capture* some *trends*, then 30, 40.
* Eventually the **best number** of **timesteps** we ended up with was ***60***.
* ***60 timesteps*** correspond to the ***60*** previous *financial* *days*, and since ***20*** ***financial*** ***days*** in one month, well ***60 timesteps*** correspond to three months.
* That means that each day we're gonna look at the *three previous month* to try to *predict* the stock price the *next day*.
* So we're gonna have ***60 timesteps*** and ***one*** ***output***, which will be the stock price at time ***T + 1***.
* **X\_train** and **y\_train**: These are created differently in RNN. In previous chapters we just choose the columns from data-set. And *splitted* those using *train-test-split*.
* Here in RNN we have to make these manually. Let ***n*** is our data-size.

1. We first create a list of list, ***2D-list***. Each list is a row of ***60 days data***. This will be ***X\_train*** matrix. (i.e. *matrix*).
2. Then we also create another vector a ***1D-list***, starting from ***61st data***, i.e. (i.e. vector). This will be ***y\_train*** *vector*.
3. Then we Convert these to NumPy array to implement in our RNN.

* ***X\_train*** will be the input of the neural network, and ***y\_train***, will contain the output.
* For each observation, that is for each *financial* *day*, ***X\_train*** will contain the ***60*** previews stock prices, before that financial day.
* ***y\_train*** will contain the stock price the next *financial* *day*.
* We use a ***for loop*** to append data points to ***X\_train*** and ***y\_train***.

**print**(f"Size of train set : {train\_set\_scled.shape}")

**print**(f"No. of data-points : {train\_set\_scled.shape[0]}")

# *creating a data structure with 60 time-steps and 1 output.*

data\_size = train\_set\_scled.shape[0]

X\_train = []

y\_train = []

**for** i **in** **range**(60, data\_size):

**X\_train.append**(train\_set\_scled[(i-60):i, 0])    # *append a list of 60 data-points in one row and (data\_size-60) rows*

**y\_train.append**(train\_set\_scled[i, 0]) # *vector of (data\_size-60) rows*

# *Convert X\_train and y\_train to NumPy array*

X\_train, y\_train = **np.array**(X\_train), **np.array**(y\_train)



1. We can see the 60 previous stock prices in ***X\_train***, and the next stock price in ***y\_train***.
2. Notice in ***y\_train*** 1st entry is 61st data point, then 62nd data-point and so on.
3. In ***y\_train*** each **n**-th element is the last element of ***X\_train***'s **(n+1)-**th row.
4. So there is total ***1258-60 = 1198*** *time-steps*, and as *time step* increase as *1-stride* the row's *element's* move *1-stride* to the *left*.

* The first line of observation here corresponds to time . At the 60th financial day of our training dataset.
* Because since here we're getting the 60 stock prices before , and in 2nd row, the 60 stock prices before , hence we have 59 stock prices in common.
* Because we have this sliding window of size ***60***, sliding with a stride of one at each observation, from one observation to the next.
* That's exactly the idea of the RNN. It is memorizing what's happening in the ***60*** previous timesteps to predict the next value at time ***T+1***.

That's the approach, the data structure we need to create as the input of a RNN.

**12.3.5 Adding Extra Dimension to our Data Structure**

This is the last step of the Data Preprocessing, we will add a *new* *dimension* to this *structure*, which makes it more *powerful*.

* You don't have to do it with only one column, ***Open*** stock price, but with several other indicators.
* These *indicators* can be some *other* *columns* of our dataset, like for example the ***Close*** column, or the ***Volume***.
* Or even some other ***different stock prices/other-companies*** related to Google. For example: Apple is related to Samsung.
* This step is about *reshaping* the *data* that is adding some even *more* *dimensionality* to the *data* *structure* that we just made.
* This dimension is actually the unit, that is, the number of predictors used to predict Google stock price at **T+1**. In this financial engineering problem, where we try to predict the *trend* of the *Google Stock Price*, these predictors are indicators.
* Right now we have *one* *indicator*, which is the ***Open*** Google Stock Price.
* Now in this *new* *dimension* that we're gonna add to our *data* *structure*, we will be able to add some *more* *indicators*.
* That could help predict even better the upward and downward trends of the Google Stock Price (we're not gonna do it in this implementation. We will just use the ***open*** *Google Stock Price*).
* Implementing Reshape:
* It's actually really simple, it just takes one line of code. We're gonna use the ***reshape()*** function. Anytime you want to add a dimension in a NumPy array you always need to use the ***reshape()*** function.
* We just need to do this for ***X\_train***, because ***X\_train*** actually contains the inputs of the ***NN***.

|  |
| --- |
| * Also we need to create this new dimensionality of this new data structure, because this shape is expected by the future RNN that we're gonna build in the next. * So that's not only for use some more indicators, but also to be compatible with the input format/shape of the RNN. |

* ***reshape()*** is taken from ***NumPy*** library. In this ***reshape()*** function, we need to input two arguments.
* The first argument is our current-data-structure that we want to reshape.
* In the second argument we need to specify this new structure. We're gonna include three elements (to add these three dimensions), because right now, our data structure has two dimensions, a NumPy array of 1198 rows and 60 columns.
* After adding a new dimension it will be a Stack of NumPy array.

# *Reshaping*

X\_train = **np.reshape**(X\_train, (X\_train.shape[0], X\_train.shape[1], 1))

* X\_train.shape[0] = no. of rows of X\_train = 1198
* X\_train.shape[1] = no. of columns of X\_train = 60
* No. of indicators = 1. We set it 1, because we're not gonna use other indicators like "Close" or "Volume", we stick to our only indicator "Open".
* Our *2D matrix* is now the *first-layer* of a *3D* *tensor*.

|  |
| --- |
| * Function ***numpy.reshape*** gives a new shape to an array without changing its data. It is a ***numpy*** package function. First of all, it needs to know what to reshape, which is the first argument of this function (in your case, you want to reshape ***X\_train***). * Then it needs to know what is the *shape* of your *new matrix*. This argument needs to be a ***tuple***. For 2D reshape you can pass ***(W,H)***, for three dimensional you can pass ***(W,H,D)***, for four dimensional you can pass ***(W,H,D,T)*** and so on. * However, you can also call ***reshape*** a Numpy matrix by ***X\_train.reshape((W,H,D))***. In this case, since ***reshape*** function is a method of ***X\_train*** object, then you do not have to pass it and only pass the new shape. * It is also worth mentioning that the total number of element in a matrix with the new shape, should match your original matrix. For example, your 2D **X\_train** has **X\_train.shape[0]**  **X\_train.shape[1]** elements. This value should be equal to . |

|  |
| --- |
| * Tensor is like 3D version of an array. The input should be a 3D array, containing the following three dimensions.   ***(batch\_size, time\_steps, input\_dim)***   * First dimension is ***batch\_size***, which will correspond to the total number of observations we have (no. of rows of ***X\_train***). * The second dimension is the ***time\_steps***, (no. of columns of ***X\_train***). * The third dimension, ***input\_dim*** is the no. of indicators that we're adding (no. of the predictors i.e. ***Open***, so we used ***1***). We are doing it because our RNN takes a 3D tensor, but: * This can be some *new financial indicators*, that could help predict the Google stock price trends. * For example, that can be the "***Closed***" Google stock price or even some other stock prices from other companies that are related to Google. * Other example can be ***Apple*** and ***Samsung***, you know that in *an iPhone's* most of the *material* is coming from *Samsung*. And therefore *Apple* is highly *dependent* on *Samsung*, but at the same time, *Samsung* is highly dependent on *Apple*, because simply, *Apple* is their *best customer*. And therefore the stock prices of Apple and Samsung might be highly correlated. * Since we have one indicator, So we just inputted ***1***, but don't forget to change it if you have several indicators. * Instead of using numbers we use ***X\_train.shape*** so that later we can use any size of data-structure. |

* So now we have our final structure, that is expected by the neural network, our future RNN.

1. First dimension corresponding to the number of *stock* *prices*.
2. The second dimension corresponding to the number of *time* *steps*.
3. And the third dimension corresponding to the number of *indicators*.



***You can see its changing the axis here.***

|  |  |  |
| --- | --- | --- |
| Dimension 1 (Axis : 1) | Dimension 2 (Axis : 2) | Dimension 3 (Axis : 3) |
|  |  | * At each time ***T***, starting from ***60***. The ***60th*** financial day. |

**12.3.6 Building the RNN:** Structure of our RNN

Now we're going to build the RNN, the whole architecture of our LSTM.

* It's not gonna be an LSTM, it's gonna be a stacked LSTM with several LSTM layers and we're gonna make it perfect.
* We're gonna make it robust by adding some *dropout regularization* to avoid *overfitting*.
* But keep in mind: "All models are wrong, but some are useful."
* Libraries and Packages:

# *importing keras libraries and packages*

**from** keras.models **import** Sequential

**from** keras.layers **import** Dense

**from** keras.layers **import** LSTM

**from** keras.layers **import** Dropout

* **Sequential** class will allow us to create a neural network object representing a sequence of layers (other representation is graph).
* **Dense** class to add the output layer.
* **LSTM** class, to add the LSTM layers.
* **Dropout** class to add some dropout regularization.
* Initialize our RNN: We gonna initialize our RNN as a sequence of layers.

# *Initialize the RNN*

regressor\_rnn = **Sequential**()

* We're gonna use the ***Sequential*** class from ***keras*** to introduce our ***regressor*** as a *sequence of layers*. [Since we're dealing with regression problem, we used 'regressor', for classification problem we used 'classifier']
* This time we're predicting a continuous output, the Google stock price, And therefore we are doing some regression.
* Remember, regression is about predicting the continuous value and classification is about predicting a category/class.

|  |
| --- |
| * We also gonna build Computational Graph in *following* *chapters*. To build some computational graphs, we use **Pytorch**, because for this, which is much more powerful tool for dynamic graphs. |

* Add The Different Layers: And now we gonna add the different layers, to make it a powerful stacked LSTM. We add the first ***LSTM*** *layer*, of our ***RNN*** which was introduced as a *sequence* of layers and also some ***Dropout*** *regularization* to avoid ***Overfitting***.

#*adding first LSTM layer & some Dropout-Regularization*

**regressor\_rnn.add**(**LSTM**(units= 50, return\_sequences= **True**, input\_shape = (X\_train.shape[1], 1)))

**regressor\_rnn.add**(**Dropout**(rate = 0.2))

* Inside the ***add()*** method, we need to *input* the *layer*, the *type of layer* we want to add.
* Since we want to add is an LSTM layer, so we use ***LSTM*** class to add first layer (and ***Dense*** to add output layer).
* Parameters:
* **units:** It is the number of LSTM cells or units in our LSTM layer. We set a very high number of LSTM cells, or memory units, (for simplicity's let's just call them ***neurons***).

|  |
| --- |
| * Why more dimensions: Now *already* our model will have a very *high dimensionality*, because we are going to ***stack*** many ***layers***. But we can increase this *dimensionality* even more by including a *large* *number* of *neurons* in *each* of the *LSTM layers*. * Since capturing the trends of a stock price is pretty complex, we need to have this high dimensionality and therefore we also need to have a large number of neurons in each of the multiple LSTM layers. * And therefore, number of neurons we'll choose for this first LSTM layer is gonna be 50. The next layers, will also have 50 neurons, that will get us a model with high dimensionality and will lead us to better results. |

* **return\_sequences:** We set it ***True***, because we are building a *stacked LSTM* which have *several LSTM layers*. When you add another LSTM layer one after another, you have to set *return\_sequences*= **True**.
* Once you are done with your LSTM layers, you are not gonna add another one after that, you will set it *return\_sequences*= **False**, (however we don’t have to do it because it's a default value of ***return\_sequences***).
* **input\_shape:** It is exactly the shape of the input containing ***X\_train*** that we created in the last step of the data preprocessing part. It's an input shape in 3D, corresponding to the ***observations***, the ***time\_steps***, and the ***indicators***.
* But in this third argument of the LSTM class, we won't have to include all the three dimensions, only the ***time\_steps*** and the ***indicators***,
* i.e. from (X\_train.shape[0], X\_train.shape[1], 1) we need to input **(X\_train.shape[1], 1)**. The first one X\_train.shape[0], corresponding to the observations, will be automatically taken into account.
* Adding Dropout-Regularization: We recommend to use 20% i.e. 0.2 because it's quite relevant. 20% of the neurons i.e. 10 out of 50 neurons of the LSTM layer, will be ignored during the training, during the forward propagation and back propagation.

**regressor\_rnn.add**(**Dropout**(rate = 0.2))

We're gonna add a total of 4 LSTM layers, so that will make a big, stacked LSTM.

**12.3.7 Building the RNN:** extra LSTM layers

Now we add some extra LSTM layers, with ***dropout regularization***.

* Adding 2nd LSTM layer: We only need to do one change, which is the ***input\_shape***.

**regressor\_rnn.add**(**LSTM**(units= 50, return\_sequences= **True**))

**regressor\_rnn.add**(**Dropout**(rate = 0.2))

* We had to specify the ***input\_shape*** in *first* input *layer* but we don’t need it for the *second* *layer*, because it's recognized automatically. Thanks to this ***units*** argument, which tells exactly that we have ***50 neurons*** in the ***previous layer***. So no need to specify any ***input\_shape*** here, when you're adding your next LSTM layers, after the first one.
* *50 neurons* in *2nd hidden layer* adds some *high dimensionality* to our model, to be able to handle the *complexity* of the *problem*.
* *Augmenting* the *dimensionality* of our model, will *augment* at the same time the *complexity*, and therefore will *respond* *better* to the *complexity* of the *problem* and that will eventually lead us to *better results*.
* We're keeping a ***20%*** ***dropout*** for the regularization, since that's a relevant choice.
* Adding 3rd LSTM layer: We only need to copy the 2nd LSTM layer. All are just same.

**regressor\_rnn.add**(**LSTM**(units= 50, return\_sequences= **True**))

**regressor\_rnn.add**(**Dropout**(rate = 0.2))

* Adding 4th LSTM layer: We only need to copy the 2nd LSTM layer. The parameter ***return\_sequences*** is set to ***False*** because it is the *last LSTM layer* and next *layer* is the *output layer*.

regressor\_rnn.add(LSTM(units= 50, return\_sequences= **False**))

* Because we're ***not going to return anymore sequences***, and therefore, since the default value of the ***return\_sequence***'s parameter is **False**, we just removing this parameter.
* Be careful, this is not the final layer of our RNN, this is the fourth LSTM layer, but after that, comes the output layer, with the output dimension, ***units*** will be ***1*** of course, because we're predicting just ***one value***, the ***value*** of the ***stock*** price at ***time T+1***.

**regressor\_rnn.add**(**LSTM**(units= 50))

**regressor\_rnn.add**(**Dropout**(rate = 0.2))

|  |
| --- |
| * However we can specify the number of neurons in the LSTM layers, and we're keeping ***50 neurons*** to have the same goal of having a high dimensionality. * Also we're *keeping* the *20% dropout regularization*. We can also change that. |

**12.3.8 Building the RNN:** output layer, compile & fit

* output-layer: For an output-layer we're not adding an LSTM layer, but a classic *fully connected layer* (similar to CNN). The output layer is a fully connected to the previous (4th-last) LSTM layer.
* To make a full connection we need to use the ***Dense*** class exactly as we did for ANN and CNN.

# *the output Layer*

**regressor\_rnn.add**(**Dense**(units = 1))

* We specified ***units = 1***, because we're predicting a *real value* corresponding to the *stock price*, i.e. the *output* has only *one dimension*. So there will be *one neuron* in *output-layer*. This will ***output*** our stock price, at time ***T +1***.

Now we're done with the architecture of our super robust LSTM RNN.

* Compiling the RNN: Our optimizer will be ***'adam'*** and the loss-function will be the ***mean\_squared\_error*** because we're doing some regression.
* We use the ***compile()*** method, it is another method of the ***Sequential*** class.

# *------ Compiling the Model ----------*

**regressor\_rnn.compile**(optimizer='adam', loss='mean\_squared\_error')

* optimizer: For RNN ***rmsprop*** is recommended in the ***keras*** *documentation*.
* ***rmsprop*** is some kind of an advanced ***SGD*** optimizer that usually a good choice for ***RNN***.
* However, with our implementation we're not gonna use an ***rmsprop***, here we use the ***adam*** optimizer.
* The ***adam*** optimizer is always a safe choice for ***SGD***. It's always a *good choice* because it is very powerful and it always performs some relevant updates of the weights.
* loss: Since, we're not doing classification anymore. We do not use cross-entropy. So the loss is not gonna be '**binary\_crossentropy**' or '**categorical\_crossentropy**'.
* Now we're dealing with a regression *problem* because we have to predict a *continuous* *value* and the loss for this kind of problem is the ***mean\_squared\_error***.
* So that the error can be measured by the *mean* of the *squared* *differences* between the *predictions* and the *targets*. Targets means the ***real values***.
* Mean squared error, also sometimes called MSE.

Now our ***regressor*** is compiled with a powerful optimizer ***'adam'*** and the right loss function ' *mean\_squared\_error* '.

* Fit/train this RNN to our training set: Notice we did not use ***training\_set*** or ***train\_set\_scled***. Our *training set* is composed of ***X\_train***, that's the right *data structure* that is *expected* by the *neural networks*.
* So we need to take ***X\_train*** and not ***training\_set*** or ***train\_set\_scled***.
* We also need to *specify* the *output* (i.e ***y\_train***), when fitting the *regressor* to our *training sets* because the *output* contains the *ground truth* that is the stock price at time ***T +1***.
* We're training the RNN on the ***true stock price*** at time ***T +1***, after the ***60 previous stock prices*** during the ***60 previous financial days***. So that's why we also need to include the *ground truth* (dependent - variable) and therefore ***y\_train***.
* It is the final step of part two, *building* the *RNN*. Here we fit our LSTM-RNN to training set, on ***X\_train*** and ***y\_train***.
* The *training* will happen in *this part* and in the end we'll have a *robust RNN*, but mostly, we will have a smart one. Ours *RNN* will be *intelligent*, *trained*, neural network, to be able to predict in some way, the upward trends, and downward trends of the Google stock price.
* We need to input four arguments, which are going to be: Feature/independent ***X\_train***, output/dependent ***y\_train***, no. of ***epochs*** and ***batch\_size*** for the sample.
* X\_train: The *independent* *variables* of the *training* set. Which will be the *input* of the *neural network* and will be *forward propagated* to the *output*, which will be the *prediction* and that will compare to the *ground* *truth* that is contained in ***y\_train***.
* y\_train: Are the real values for predictions, it is the dependent variables.
* number of epochs: That is on how many iterations do you want your RNN to be trained. Or how many times do you want the whole training data to be forward propagated inside the NN and then back propagated to update the weight.
* What is the best number: You have to test it several times. For example:
* I tried with first ***25 epochs*** and noticed that there wasn't a convergence of the loss.
* Then I tried, 50 still not some convergence.
* Finally I tried 100, and there I observed some convergence of the loss.
* So I think 100 is the right number of epochs. For this kind of problem, besides we don't have that much data. We only have the Google stock prize for 5 years.
* batch\_size: We set this so that our *RNN* trained on *batches* of *observations/stock prices* going into the NN. So instead of updating the weight for each observation (stock price) during *forward-propagation* generating *error* and *back-propagation*, we do that, every 32 stock prices.

|  |
| --- |
| * If you want you can train this *RNN* on even *more* than *5 years*. The data is available online, it's the real *Google stock price* that you can take from many financial sources. One of them can be for example, *Yahoo finance*. So feel free to play with it and experiment even more to create some maybe even more robust RNN. * But anyway with *5 years* of training data well *100 epochs* turned out to be a good choice, with some *convergence*. |

**All code at once (only fit)**

# *Recurrent Neural Netwark: RNN*

**import** pandas **as** pd

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

# *------------------- Data Preprocessing ---------------------*

dataset\_train = **pd.read\_csv**('Google\_Stock\_Price\_Train.csv')     # *Notice train-set is now our main dataset*

training\_set = dataset\_train.iloc[:, 1:2].values

# *feature scaling*

**from** sklearn.preprocessing **import** MinMaxScaler

sc = **MinMaxScaler**(feature\_range= (0, 1))

train\_set\_scled = **sc.fit\_transform**(training\_set)

**print**(f"Size of train set : {train\_set\_scled.shape}")

**print**(f"No. of data-points : {train\_set\_scled.shape[0]}")

# *creating a data structure with 60 time-steps and 1 output.*

data\_size = train\_set\_scled.shape[0]

X\_train = []

y\_train = []

**for** i **in** **range**(60, data\_size):

**X\_train.append**(train\_set\_scled[(i-60):i, 0])    # *append a list of 60 data-points in list of (data\_size-60) rows*

**y\_train.append**(train\_set\_scled[i, 0]) # *vector of (data\_size-60) rows*

# *Convert X\_train and y\_train to NumPy array*

X\_train, y\_train = **np.array**(X\_train), **np.array**(y\_train)

# *Reshaping*

X\_train = **np.reshape**(X\_train, (X\_train.shape[0], X\_train.shape[1], 1))

# *------------------- Building RNN ---------------------*

# *importing keras libraries and packages*

**from** keras.models **import** Sequential

**from** keras.layers **import** Dense

**from** keras.layers **import** LSTM

**from** keras.layers **import** Dropout

# *Initialize the RNN*

regressor\_rnn = **Sequential**()

# *adding first LSTM layer & some Dropout-Regularization*

**regressor\_rnn.add**(**LSTM**(units= 50, return\_sequences= **True**, input\_shape = (X\_train.shape[1], 1)))

**regressor\_rnn.add**(**Dropout**(rate = 0.2))

# *second LSTM layer with Dropout-Regularization*

**regressor\_rnn.add**(**LSTM**(units= 50, return\_sequences= **True**))

**regressor\_rnn.add**(**Dropout**(rate = 0.2))

# *third LSTM layer with Dropout-Regularization*

**regressor\_rnn.add**(**LSTM**(units= 50, return\_sequences= **True**))

**regressor\_rnn.add**(**Dropout**(rate = 0.2))

# *forth LSTM layer (last LSTM layer) with Dropout-Regularization*

**regressor\_rnn.add**(**LSTM**(units= 50))

**regressor\_rnn.add**(**Dropout**(rate = 0.2))

# *the output Layer*

**regressor\_rnn.add**(**Dense**(units = 1))

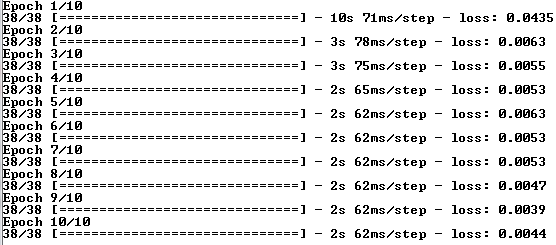
# *------ Compiling the Model ----------*

**regressor\_rnn.compile**(optimizer='adam', loss='mean\_squared\_error')

# *------- Fit the model to Training dataset (model learns) ----------*

# *The training will happen in this part.*

**regressor\_rnn.fit**(X\_train, y\_train, epochs= 10, batch\_size=32)



|  |
| --- |
| * Notice the *convergence* of ***loss*** values during the *training*. So we started with a ***loss*** near ***4%, 0.0435***. The final loses in the end in the last 20 epochs, it's staying around ***0.0015*** i.e. ***0.15%***. * Too small loss indicates overfitting: So if you want to try more epochs, I think the loss remain around 0.0015. * That’s because we added a Dropout Regularization. And if you obtain a ***loss*** too small in the end well you might get some ***over fitting***. * In that case you might get some *small loss* on the *training data* and really *large loss* on the *test data*. So be careful not to obtain *overfitting* and therefore *not to try to* *decrease* the *loss* as much as possible. |

* And that's why here it seems that we get really *good* *results*. Next we're going to make the *Predictions* and *Visualizing* the Results to observe the *predictions*.

**12.3.9 Prediction from the RNN**

In this part we'll make the predictions and visualize the results. There is three steps:

1. First, we're gonna get the Real Google stock price of 2017-January from Google\_Stock\_Price\_Test.csv.
2. In the second step, we're gonna get the Predicted Google stock price of 2017-January from our trained model.
3. And then in the final step, we will visualize the results.

* Getting the real Google stock price of January 2017: We simply get it from the test dataset that we have, in the CSV file, ***Google\_Stock\_Price\_Test.csv***.
* So first we create a data-frame using Pandas
* Then we will select the right column, the ***Open*** Google stock price.
* And make it a NumPy array.

dataset\_test = **pd.read\_csv**('Google\_Stock\_Price\_Test.csv')

real\_stock\_price = dataset\_test.iloc[:, 1:2].values

* Data-preprocess for Test-Set before prediction: This part is little bit tricky. There are some important tricks that we will make sure to understand and apply, and some mistakes to absolutely avoid.
* Here we're gonna use our *regressor*, and we're gonna *predict* the Google *stock prices* of *January, 2017*.
* But the thing is, we *trained* our *model* to be able to *predict* the *stock price* at time ***T +1***, based on the ***60*** previous *stock prices*, and, therefore, to *predict* each *stock price* of *each* financial day of *January, 2017*, we will need the *60 previous stock prices* of the 60 previous financial days, before the actual day.
* In order to get at *each* day of *January, 2017*, the *60* previous *stock prices* of the *60 previous days*, well, *we* will *need* *both* the ***training*** ***set*** and the ***test*** ***set***, because we will have *some of the 60 days* that will be *from* the *training set*, because they will be from *December 2016*, and we will also have some *stock prices* of the *test set*, because some of them will come *from January 2017*.
* Therefore, we need to do some concatenation of the ***training set*** and the ***test set***,
* How do we concatenate: If you think of making this concatenation by concatenating the training-set " **training\_set** " and the test-set " **real\_stock\_price** ", it will lead us to a problem and we have to ***re-scale*** the result(we have to apply the ***fit\_transform*** method again). We should never do this, we have to keep the actual test values as they are.
* The trick is, here we actually concatenate the original DataFrames, ***dataset\_train*** and ***dataset\_test***.
* From this concatenation, we will get the input of each prediction. Then we will ***scale*** it by our ***sc*** object. That's the way we are only *scaling* the *input*, and not changing the *actual test values*.
* We have to scale the input, because our ***RNN*** model was trained on the scaled values of the train set. That's why we use scaled input to get our predictions.
* The scaled input should be based on the same scaling i.e. normalization with our ***sc*** object.
* Here ***dataset\_total*** will contain the whole concatenated dataset. We're gonna use ***concat()*** function from this ***Pandas*** library.
* We need to input two arguments:

1. The first one are the *two DataFrames* we want to concatenate,
2. The *second* argument is the *axis* along which we want to make that concatenation. i.e. do we want to concatenate the *lines/rows-wise*? Or do we want to concatenate the *column-wise*?

* ***dataset\_train['Open']*** contains the 'Open' ***Google stock prices*** from 2012-2016, and and ***dataset\_test['Open']***, contains 'Open' ***Google stock prices*** of January, 2017.

dataset\_total = **pd.concat**((dataset\_train['Open'], dataset\_test['Open']), axis = 0)

* The trick is to specify a column in the DataFrame we can use their name inside **[]**, as we did ***['Open']***.
* So *(dataset\_train['Open'], dataset\_test['Open'])* is the first argument of the ***concat*** function.
* Now the ***axis*** we want to ***concatenate*** is the ***lines***, i.e. we concatenate the ***rows***.
* Therefore, we need to make a concatenation along the vertical axis. To do this we need to use ***axis = 0***. Because ***vertical*** ***axis*** is labeled by ***0*** .
* For a ***horizontal*** concatenation (columns), use ***axis = 1***,
* For ***vertical*** concatenation use ***axis = 0***.

# *getting the Predicted Google stock price of 2017-January from our trained model*

dataset\_total = **pd.concat**((dataset\_train['Open'], dataset\_test['Open']), axis = 0)

inputs\_for\_predict = dataset\_total[**len**(dataset\_total)-**len**(dataset\_test)-60 : ].values

# *values makes it array, otherwise it is just a series*

# *it is 60 + 20 = 80 stock-prices, combining october, november, December 2016 and january 2017*

**print**(**len**(inputs\_for\_predict))

* To get the inputs, we need to specify the index. Since we need ***60*** from ***test-set*** and we get ***20*** from ***train-set***, our input size will be ***80***.

inputs\_for\_predict = dataset\_total[**len**(dataset\_total)-**len**(dataset\_test)-60 : ].values



* Here **len**(dataset\_test)+60 = 80. Thus we starting from the index **len**(dataset\_total)- 80, to get last 80 data-points.
* Because to predict each financial day of *January 2017*, we need to get the 60 previous stock prices from *October, November, December* of *2016* ( the 60 previous financial days).
* That’s why we fix the lower-bound as: **len**(dataset\_total) - **len**(dataset\_test)-60. And there is no upper-bound because we wand all values to the last of the January 2017.
* ***.values*** is used to convert the selected data-points into ***NumPy*** ***Arrays***.
* Since we haven't used the ***iloc*** method from Pandas to get these inputs, and therefore, it is not still shaped the right way. It is not properly *shaped* like a *NumPy Array*. So we need to use a *simple reshape* of the ***inputs\_for\_predict*** object (not the NumPy-reshape to convert 3D tensor) just to format the data:

# *simple reshape for the input: to make it a vector, similar job as 'iloc'*

inputs\_for\_prd = **inputs\_for\_predict.reshape**(-1, 1)

* Next we use NumPy's ***reshape*** to create 3D structure of our inputs (observations, time\_steps, indicators). We'll just copy what we've done before and make the proper changes.
* Scaling the prediction-inputs: Before we reshape our input to 3D format to put in RNN we must scale those inputs. We do not use ***fit\_transform()*** because we did it already with training set ***train\_set\_scled***. And we just need to use ***transform()*** to bring input ***inputs\_for\_predict*** same scale of ***train\_set\_scled***.

inputs\_for\_prd = **sc.transform**(inputs\_for\_prd) # *scaling, only "transform" , no "fit"*

* Since the RNN was trained on the scaled values. we need to scale the inputs.
* So we're not gonna use the ***fit\_transform()*** because our sc object was already fitted to the training set ***train\_set\_scled***. I'm directly gonna use the ***transform()*** method because the scaling we need to apply to our input must be the *same scaling* that was applied to the *training set*.
* Therefore we must not ***fit*** our *scaling object* ***sc*** again. We must directly apply the ***transform*** method to get the *previous scaling* on which our *regressor* was *trained*.
* The Data Pre-Processing for Prediction is given below:

# *getting the Real Google stock price of 2017-January from Google\_Stock\_Price\_Test.csv*

dataset\_test = **pd.read\_csv**('Google\_Stock\_Price\_Test.csv')

real\_stock\_price = dataset\_test.iloc[:, 1:2].values

# *getting the Predicted Google stock price of 2017-January from our trained model*

dataset\_total = **pd.concat**((dataset\_train['Open'], dataset\_test['Open']), axis = 0)

inputs\_for\_predict = dataset\_total[**len**(dataset\_total)-**len**(dataset\_test)-60 : ].values

# *values makes it array, otherwise it is just a series*

# *it is 60 + 20 = 80 stock-prices, combining october, november, December 2016 and january 2017*

**print**(**len**(inputs\_for\_predict))

# *simple reshape for the input: to make it a vector, similar job as 'iloc'*

inputs\_for\_prd = **inputs\_for\_predict.reshape**(-1, 1)

inputs\_for\_prd = **sc.transform**(inputs\_for\_prd) # *scaling, only "transform" , no "fit"*

* Data-Preprocess for test-set, creating 3D data-structure: Here we prepare the special 3D structure of ***inputs\_for\_prd*** that expected by the NN for the training, but also for the predictions. It is similar as we did before:

# *creating the data structure for test-set*

test\_data\_size = inputs\_for\_prd.shape[0]

X\_test = []

# *There is no "y\_test = []" we'll predict it*

**for** i **in** **range**(60, test\_data\_size):

**X\_test.append**(inputs\_for\_prd[(i-60):i, 0])

# *Convert X\_test to NumPy array*

X\_test = **np.array**(X\_test)

* We don’t need "**y\_test = []**", we'll predict it. Instead of ***train\_set\_scled*** we use our input for test ***inputs\_for\_prd***.
* Then we convert it to ***NumPy*** array.
* Reshaping: It is exactly same as we did for ***X\_train***, but here we do it for ***X\_test***.

X\_test = **np.reshape**(X\_test, (X\_test.shape[0], X\_test.shape[1], 1))

* Getting the Predicted Google stock price of January 2017: The predicted values are returned in Scaled format, so we need to inverse-scale it to get the real values

# *---- Making Prediction !!! -----*

y\_pred = **regressor\_rnn.predict**(X\_test)

predicted\_stock\_price = **sc.inverse\_transform**(y\_pred)

**12.3.10 Visualizing the result**

We are just going to use the ***plot*** function to separately plot the *real Google stock price* and then the *predicted Google stock price*. We will give a *title* and different *colors* and a *label* for the *legend*, and a title to our chart and some labels for the x- axis and the y- axis.

* Keep in mind that we're *plotting* the *first month* of *January 2017*, and the *predictions* of *January 2017*.

**plt.plot**(real\_stock\_price, color = 'red', label = 'Real Google Stock price of January 2017')

**plt.plot**(predicted\_stock\_price, color = 'blue', label = 'Predicted Stock price of January 2017')

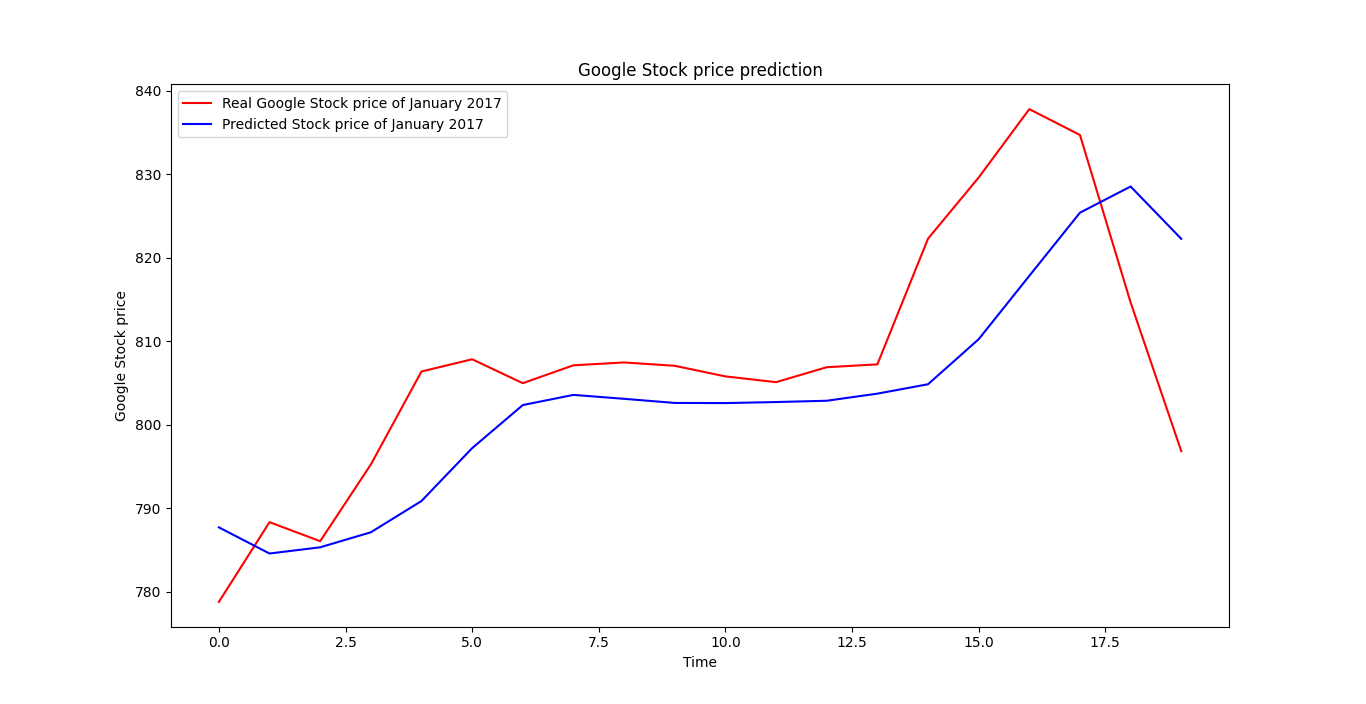
**plt.title**('Google Stock price prediction')

**plt.xlabel**('Time')

**plt.ylabel**('Google Stock price')

**plt.legend**()

**plt.show**()



* We have the real *Google* *stock* *price* in *red* and our *predicted Google stock price* in *blue*. And we get this comparison of the real and the predicted Google stock prices for the whole month of January 2017.
* In some parts of the *predictions*, our predictions *lagging* behind the *actual values*.
* We see a big spike, like a stock time singularity at last week of January, and our predictions did not follow that, but that is completely normal. (There is another spike around the first week also.)
* Our model just lags behind because it cannot *react* so *fast*, these kind of *nonlinear changes*.
* That's totally fine because, indeed, according to the *Brownian Motion* Mathematical Concept in *financial engineering*, the *future variations* of the *stock price* are *independent* from the *past*.
* And therefore, this *future variation* that we see here around the *spike*, well, is a *variation* that is indeed totally *independent* from the *previous stock prices*.
* The good news is that our *RNN* *model* reacts okay to *smooth* *changes*.
* For the parts of the predictions *containing* *smooth* *changes*, our model reacts *pretty* *well* and manages to follow the *upward* and *downward* *trends*.
* It manages to follow the *upward trend*, then the *stable trend*, and again, the *upward trend*. Then, there is a *downward trend* in the *last* financial days of *January*, and it started to capture it.

**What is Next?**

In the next chapters of unsupervised deep learning, we are going to start to play with one of the most powerful tools for deep learning and artificial intelligence, the **Pytorch**,

**Pytorch** is much more powerful than *Keras*, thanks to the *dynamic* *graphs*.

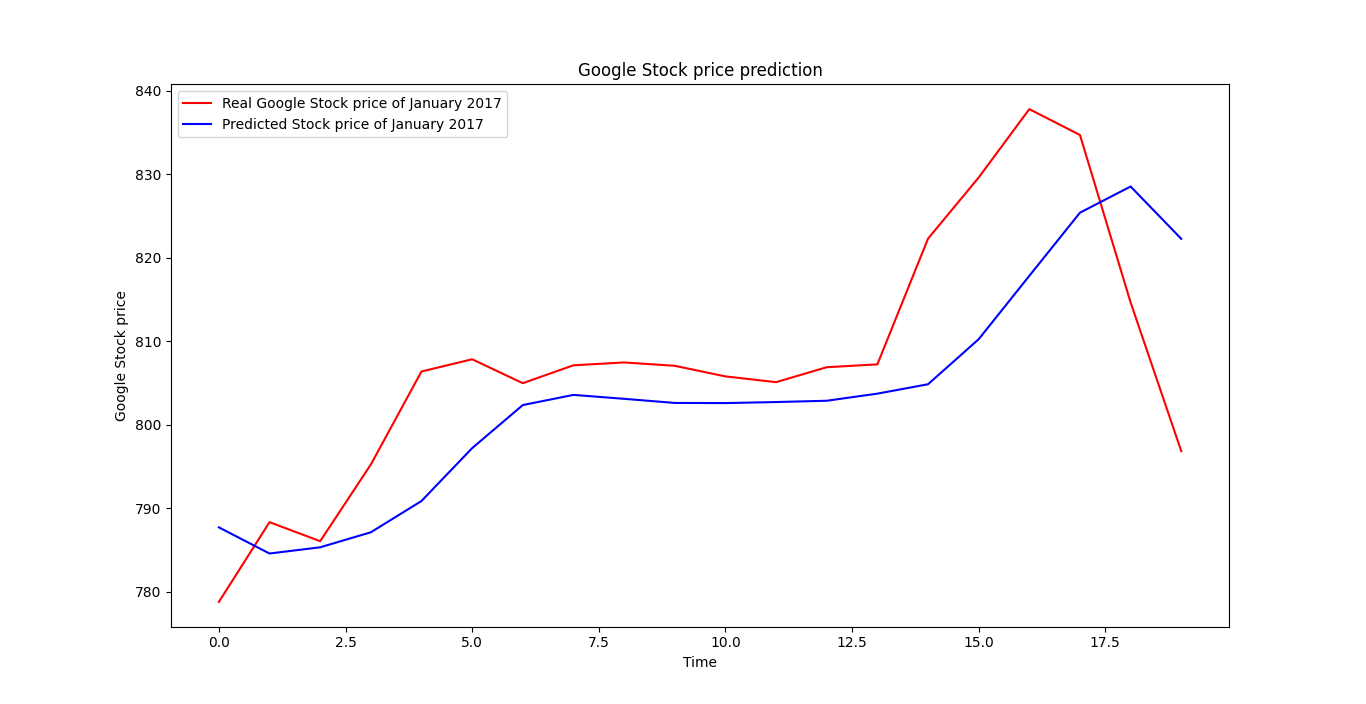
With these *dynamic graphs*, we will be able to build some very *powerful*, *unsupervised, deep learning models* especially the Boltzmann machines and the Auto-encoders which will implement, build two different recommended systems. One that will predict the **ratings** given by the users to movies, and one other that will predict if a user will **like** yes or no, a movie.

***Using 10 epochs only.***

|  |  |
| --- | --- |
|  |  |

**Using 100 Epochs**

|  |  |  |
| --- | --- | --- |
| To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.  Epoch 1/100  38/38 [==============================] - 7s 60ms/step - loss: 0.0360  Epoch 2/100  38/38 [==============================] - 2s 61ms/step - loss: 0.0062  Epoch 3/100  38/38 [==============================] - 2s 61ms/step - loss: 0.0052  Epoch 4/100  38/38 [==============================] - 2s 60ms/step - loss: 0.0054  . . . . . . . . .  . . . . . . . . .  Epoch 97/100  38/38 [==============================] - 2s 61ms/step - loss: 0.0015  Epoch 98/100  38/38 [==============================] - 2s 61ms/step - loss: 0.0013  Epoch 99/100  38/38 [==============================] - 2s 61ms/step - loss: 0.0012  Epoch 100/100  38/38 [==============================] - 2s 62ms/step - loss: 0.0014  1/1 [==============================] - 1s 1s/step | ------ Real Stock prices---------  [[778.81]  [788.36]  [786.08]  [795.26]  [806.4 ]  [807.86]  [805. ]  [807.14]  [807.48]  [807.08]  [805.81]  [805.12]  [806.91]  [807.25]  [822.3 ]  [829.62]  [837.81]  [834.71]  [814.66]  [796.86]]  ------ --------- | ------ Predicted Stock prices---------  [[787.7355 ]  [784.6061 ]  [785.3503 ]  [787.1467 ]  [790.8928 ]  [797.22 ]  [802.37866]  [803.59607]  [803.12555]  [802.6338 ]  [802.61597]  [802.74005]  [802.89484]  [803.7443 ]  [804.8699 ]  [810.2623 ]  [817.8498 ]  [825.4131 ]  [828.5456 ]  [822.2784 ]]  ------ --------- |



***All RNN code at once (practiced version)***

# *Recurrent Neural Netwark: RNN*

**import** pandas **as** pd

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

# *------------------- Data Preprocessing ---------------------*

dataset\_train = **pd.read\_csv**('Google\_Stock\_Price\_Train.csv')     # *Notice train-set is now our main dataset*

training\_set = dataset\_train.iloc[:, 1:2].values

# *feature scaling*

**from** sklearn.preprocessing **import** MinMaxScaler

sc = **MinMaxScaler**(feature\_range= (0, 1))

train\_set\_scled = **sc.fit\_transform**(training\_set)

**print**(f"Size of train set : {train\_set\_scled.shape}")

**print**(f"No. of data-points : {train\_set\_scled.shape[0]}")

# *creating a data structure with 60 time-steps and 1 output.*

data\_size = train\_set\_scled.shape[0]

X\_train = []

y\_train = []

**for** i **in** **range**(60, data\_size):

**X\_train.append**(train\_set\_scled[(i-60):i, 0])    # *append a list of 60 data-points in list of (data\_size-60) lists*

**y\_train.append**(train\_set\_scled[i, 0]) # *vector of (data\_size-60) rows*

# *Convert X\_train and y\_train to NumPy array*

X\_train, y\_train = **np.array**(X\_train), **np.array**(y\_train)

# *Reshaping*

X\_train = **np.reshape**(X\_train, (X\_train.shape[0], X\_train.shape[1], 1))

# *------------------- Building RNN ---------------------*

# *importing keras libraries and packages*

**from** keras.models **import** Sequential

**from** keras.layers **import** Dense

**from** keras.layers **import** LSTM

**from** keras.layers **import** Dropout

# *Initialize the RNN*

regressor\_rnn = **Sequential**()

# *adding first LSTM layer & some Dropout-Regularization*

**regressor\_rnn.add**(**LSTM**(units= 50, return\_sequences= **True**, input\_shape = (X\_train.shape[1], 1)))

**regressor\_rnn.add**(**Dropout**(rate = 0.2))

# *second LSTM layer with Dropout-Regularization*

**regressor\_rnn.add**(**LSTM**(units= 50, return\_sequences= **True**))

**regressor\_rnn.add**(**Dropout**(rate = 0.2))

# *third LSTM layer with Dropout-Regularization*

**regressor\_rnn.add**(**LSTM**(units= 50, return\_sequences= **True**))

**regressor\_rnn.add**(**Dropout**(rate = 0.2))

# *forth LSTM layer (last LSTM layer) with Dropout-Regularization*

**regressor\_rnn.add**(**LSTM**(units= 50))

**regressor\_rnn.add**(**Dropout**(rate = 0.2))

# *the output Layer*

**regressor\_rnn.add**(**Dense**(units = 1))

# *------ Compiling the Model ----------*

**regressor\_rnn.compile**(optimizer='adam', loss='mean\_squared\_error')

# *------- Fit the model to Training dataset (model learns) ----------*

# *The training will happen in this part.*

**regressor\_rnn.fit**(X\_train, y\_train, epochs= 100, batch\_size=32)

# *------------------- Making Prediction ---------------------*

# *getting the Real Google stock price of 2017-January from Google\_Stock\_Price\_Test.csv*

dataset\_test = **pd.read\_csv**('Google\_Stock\_Price\_Test.csv')

real\_stock\_price = dataset\_test.iloc[:, 1:2].values

# *getting the Predicted Google stock price of 2017-January from our trained model*

dataset\_total = **pd.concat**((dataset\_train['Open'], dataset\_test['Open']), axis = 0)

inputs\_for\_predict = dataset\_total[**len**(dataset\_total)-**len**(dataset\_test)-60 : ].values

# *values makes it array, otherwise it is just a series*

# *it is 60 + 20 = 80 stock-prices, combining october, november, December 2016 and january 2017*

**print**(**len**(inputs\_for\_predict))

# *simple reshape for the input: to make it a vector, similar job as 'iloc'*

inputs\_for\_prd = **inputs\_for\_predict.reshape**(-1, 1)

inputs\_for\_prd = **sc.transform**(inputs\_for\_prd) # *scaling, only "transform" , no "fit"*

# *creating the data structure for test-set*

test\_data\_size = inputs\_for\_prd.shape[0]

X\_test = []

# *There is no "y\_test = []" we'll predict it*

**for** i **in** **range**(60, test\_data\_size):

**X\_test.append**(inputs\_for\_prd[(i-60):i, 0])

# *Convert X\_test to NumPy array*

X\_test = **np.array**(X\_test)

# *Reshaping*

X\_test = **np.reshape**(X\_test, (X\_test.shape[0], X\_test.shape[1], 1))

# *---- Making Prediction !!! -----*

y\_pred = **regressor\_rnn.predict**(X\_test)

predicted\_stock\_price = **sc.inverse\_transform**(y\_pred)

# *------------------- Visualizing the Result ---------------------*

**print**(f"\n------ Real Stock prices--------- \n ")

**print**(f"{real\_stock\_price}")

**print**(f"\n------ --------- \n ")

**print**(f"\n------ Predicted Stock prices--------- \n ")

**print**(f"{predicted\_stock\_price}")

**print**(f"\n------ --------- \n ")

**plt.plot**(real\_stock\_price, color = 'red', label = 'Real Google Stock price of January 2017')

**plt.plot**(predicted\_stock\_price, color = 'blue', label = 'Predicted Stock price of January 2017')

**plt.title**('Google Stock price prediction')

**plt.xlabel**('Time')

**plt.ylabel**('Google Stock price')

**plt.legend**()

**plt.show**()

# *python prctc\_rnn.py*