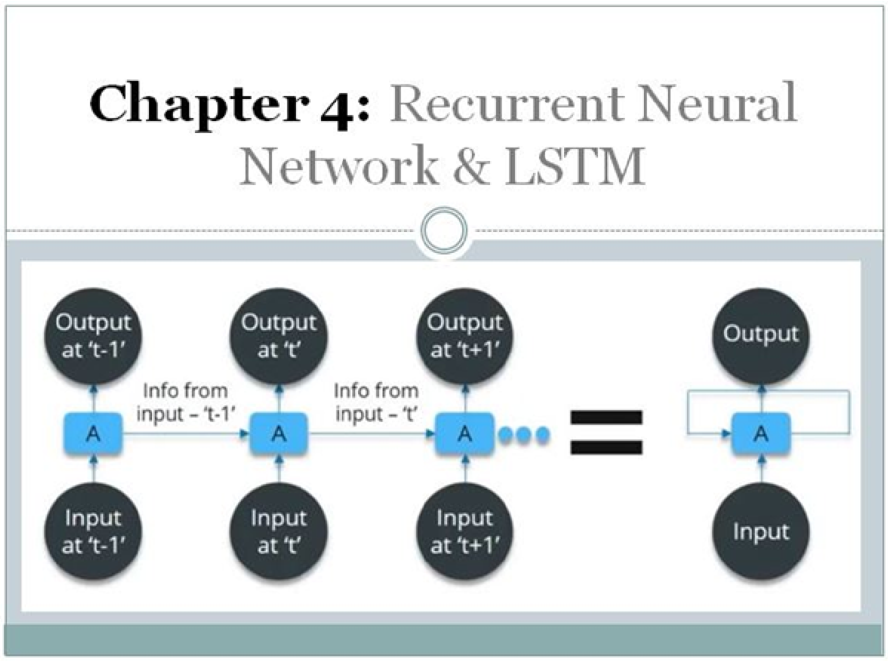
Recurrent Neural Network & LSTM with Practical Implementation

Amir Ali Follow
May 23, 2019 · 21 min read



In this Fourth Chapter of Deep Learning, we will discuss the Recurrent Neural Network and LSTM. It is a Supervised Deep Learning technique and we will discuss both theoretical and Practical Implementation from Scratch.

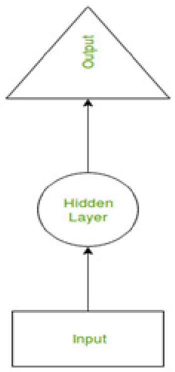
This chapter spans 7 parts:

1. What is the Recurrent Neural Network?
2. Understand a Recurrent Neural Network in Detail.
3. Forward Propagation in Recurrent Neural Network.
4. Backpropagation is an RNN (BPTT).
5. Vanishing and Exploding Gradient Problem.
6. Long Short Term Memory (LSTM).

7. Practical Implementation of Recurrent Neural Network & LSTM.

1. What is the Recurrent Neural Network?

Let's say the task is to predict the next word in a sentence. Let's try accomplishing it using an MLP. So what happens in an MLP. In the simplest form, we have an input layer, a hidden layer, and an output layer. The input layer receives the input, the hidden layer activations are applied and then we finally receive the output.



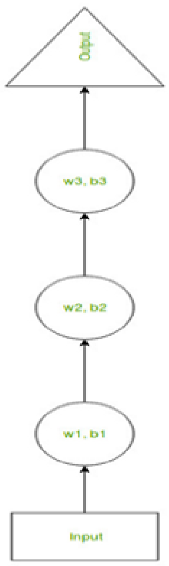
Let's have a deeper network, where multiple hidden layers are present. So here, the input layer receives the input, the first hidden layer activations are applied and then these activations are sent to the next hidden layer, and successive activations through the layers to produce the output. Each hidden layer is characterized by its owights and biases.

Since each hidden layer has its owights and activations, they behave independently. Now the objective is to identify the relationship between successive inputs. Can we supply the inputs to hidden layers? Yes, we can!

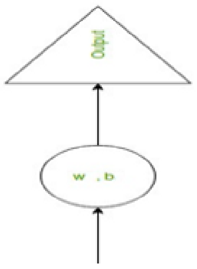
Here, the weights and bias of these hidden layers are different. And hence each of these layers behaves independently and cannot be combined too combine these hidden layers toe shall have the same weights and bias for these hidden layers.

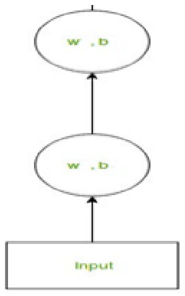
We can now combine these layers tohat the weights and bias of all the hidden layers are the same. All these hidden layers can be rolled in together in a single recurrent layer.

So it's like supplying the input to the hidden layer. At all the time steps weights of the recurrent neuron would be the same since it's a single neuron now. So a recurrent neuron stores the state of a previous input and combines with the current input thereby preserving some relationship of the current input with the previous input.

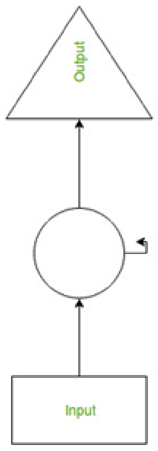


Here, the weights and bias of these hidden layers are different. And hence each of these layers behaves independently and cannot be combined together. To combine these hidden layers together, we shall have the same weights and bias for these hidden layers.





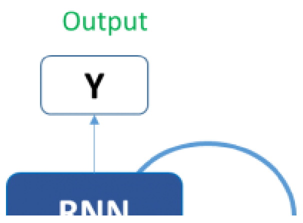
We can now combines these layers together, that the weights and bias of all the hidden layers is the same. All these hidden layers can be rolled in together in a single recurrent layer.

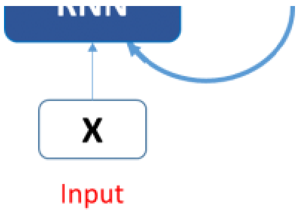


So it's like supplying the input to the hidden layer. At all the time steps weights of the recurrent neuron would be the same since it's a single neuron now. So a recurrent neuron stores the state of a previous input and combines with the current input thereby preserving some relationship of the current input with the previous input.

2. Understand a Recurrent Neural Network in Details.

Let's take a simple task at first. Let's take a character level RNN where we have the word "Hello". So we provide the first 4 letters i.e. h,e,l,l and ask the network to predict the last letter i.e.' o'. So here the vocabulary of the task is just 4 letters {h,e,l,o}. In real case scenarios involving natural language processing, the vocabularies include the words in the entire Wikipedia database or all the words in a language. Here for simplicity, we have taken a very small set of vocabulary.





Let’s see how the above structure be used to predict the fifth letter in the word “hello”. In the above structure, the blue RNN block applies something called a recurrence formula to the input vector and also its previous state. In this case, the letter “h” has nothing preceding it, let’s take the letter “e”. So at the time the letter “e” is supplied to the network, a recurrence formula is applied to the letter “e” and the previous state which is the letter “h”. These are known as various time steps of the input. So if at time t, the input is “e”, at time t-1, the input was “h”. The recurrence formula is applied to e and h both. and we get a new state.

The formula for calculating current state:

$$h_t = f(h_{t-1}, x_t)$$

Where:

h_t -> Current state

h_{t-1} -> Previous state

X_t -> Input state

We now have a state of the previous input instead of the input itself, because the input neuron would have applied the transformations on our previous input. So each successive input is called a time step.

In this case, we have four inputs to be given to the network, during a recurrence formula, the same function and the same weights are applied to the network at each time step.

Let’s take a look at how we can calculate these states in Excel and get the output.

The formula for applying Activation function (tanh):

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

Where:

W_{hh} -> Weight at the recurrent neuron

W_{xh} -> Weight at input neuron

Taking the simplest form of a recurrent neural network, let's say that the activation function is tanh, the weight at the recurrent neuron is Whh and the weight at the input neuron is Wxh.

The Recurrent neuron, in this case, is just taking the considering the immediately previous stater sequences, the equation can involve multiple such states. Once the final state is calculated we can go on to produce the output

Now, once the current state is calculated we can calculate the output state as

The formula for calculating output:

$$y_t = W_{hy}h_t$$

Where:

y_t -> Output

W_{hy} -> Weight at the output layer

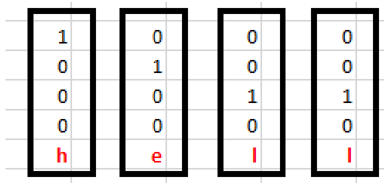
Let me summarize the steps in a recurrent neuron

1. A single time step of the input is supplied to the network i.e. xt is supplied to the network
2. We then calculate its current state using a combination of the current input and the previous state i.e. we calculate ht
3. The current ht becomes ht-1 for the next time step

4. We can go as many time steps as the problem demands and combine the information from all the previous states
5. Once all the time steps are completed the final current state is used to calculate the output yt
6. The output is then compared to the actual output and the error is generated
7. The error is then backpropagated to the network to update the weights(we shall go into the details of backpropagation in further sections) and the network is trained

3. Forward Propagation in a Recurrent Neuron.

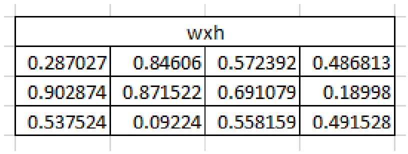
Let's take a look at the inputs first –



1	0	0	0
0	1	0	0
0	0	1	1
0	0	0	0
h	e	l	l

The inputs are one-hot encoded. Our entire vocabulary is {h,e,l,o} and hence we can easily one-hot encode the inputs.

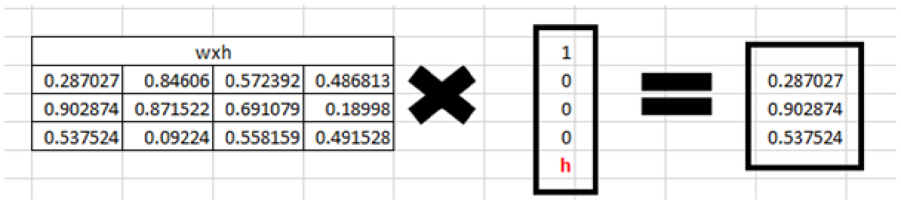
Now the input neuron would transform the input to the hidden state using the weight wxh. We have randomly initialized the weights as a 3*4 matrix –



wxh			
0.287027	0.84606	0.572392	0.486813
0.902874	0.871522	0.691079	0.18998
0.537524	0.09224	0.558159	0.491528

Step 1:

Now for the letter "h", for the hidden state, we would need Wxh*Xt. By matrix multiplication, we get it as –



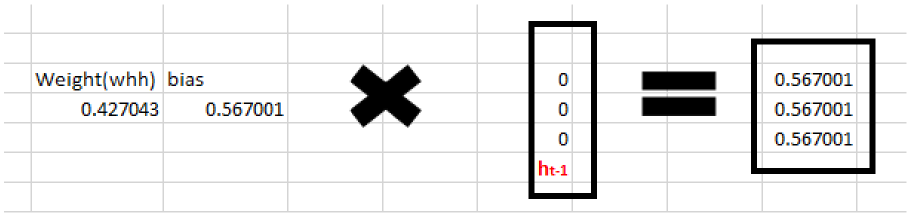
wxh							
0.287027	0.84606	0.572392	0.486813	×	1	=	0.287027
0.902874	0.871522	0.691079	0.18998		0		0.902874
0.537524	0.09224	0.558159	0.491528		0		0.537524
					0		
					h		

Step 2:

Now moving to the recurrent neuron, we have **Whh** as the weight which is a 1*1 matrix as 0.42703 and the bias which is also a 1*1 matrix as 0.56700

For the letter "h", the previous state is [0,0,0] since there is no letter prior to it.

So beforelate -> (whh*ht-1+bias)

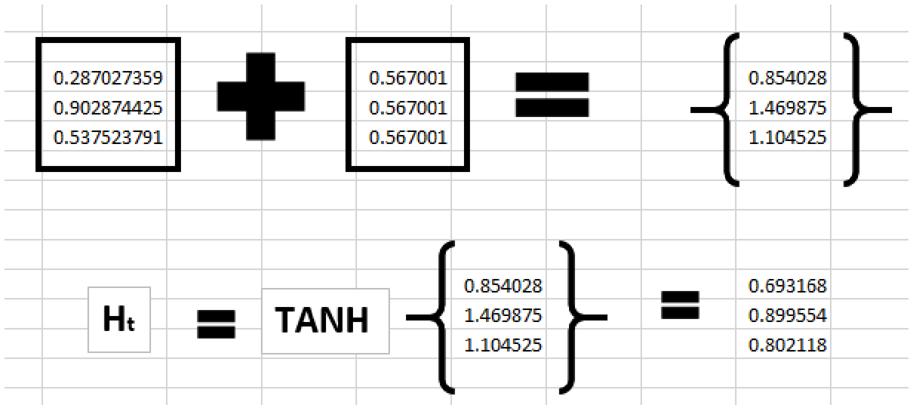


Step 3:

Now we can get the current state as –

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

Since for h, there is no previous hidden state we apply the tanh function to this output and get the current state –

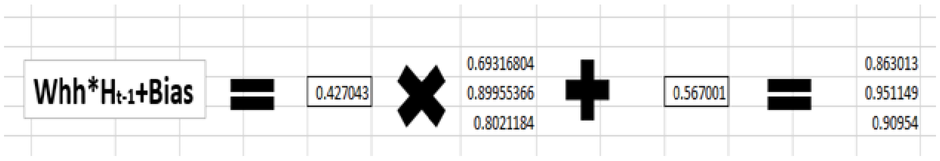


Step 4:

Now we go on to the next state. “e” is now supplied to the network. The processed output of ht, now becomes ht-1, while the one-hot encoded e, is xt. Let’s now calculate the current state ht.

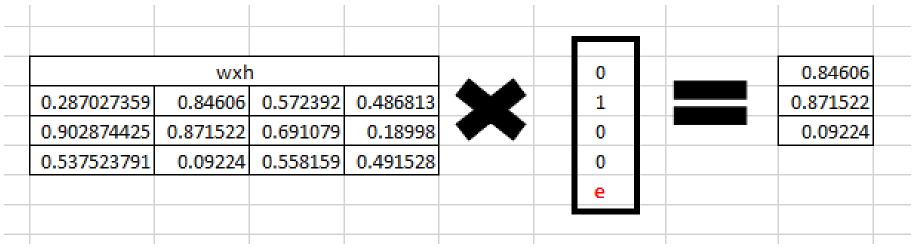
$$h_t = \tanh\left(W_{hh}h_{t-1} + W_{xh}x_t\right)$$

Whh*ht-1 +bias will be –



$$Whh*H_{t-1}+Bias = 0.427043 \times \begin{bmatrix} 0.69316804 \\ 0.89955366 \\ 0.8021184 \end{bmatrix} + 0.567001 = \begin{bmatrix} 0.863013 \\ 0.951149 \\ 0.90954 \end{bmatrix}$$

Wxh*xt will be –

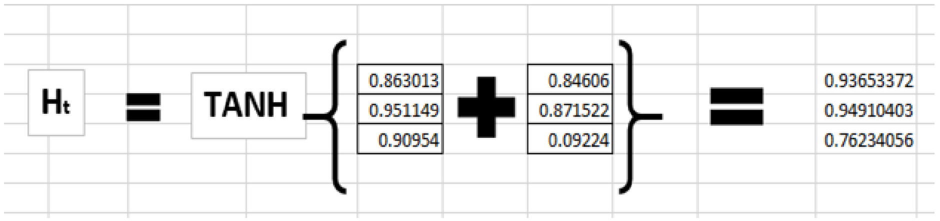


wxh			
0.287027359	0.84606	0.572392	0.486813
0.902874425	0.871522	0.691079	0.18998
0.537523791	0.09224	0.558159	0.491528

$$\times \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0.84606 \\ 0.871522 \\ 0.09224 \end{bmatrix}$$

Step 5:

Now calculating ht for the letter “e”,



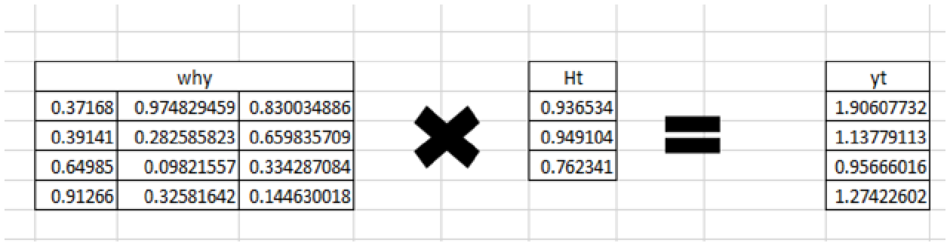
$$H_t = \mathrm{TANH}\left\{ \begin{bmatrix} 0.863013 \\ 0.951149 \\ 0.90954 \end{bmatrix} + \begin{bmatrix} 0.84606 \\ 0.871522 \\ 0.09224 \end{bmatrix} \right\} = \begin{bmatrix} 0.93653372 \\ 0.94910403 \\ 0.76234056 \end{bmatrix}$$

Now, this would become ht-1 for the next state and the recurrent neuron would use this along with the new character to predict the next one.

Step 6:

At each state, the recurrent neural network would produce the output as well. Let’s calculate yt for the letter e.

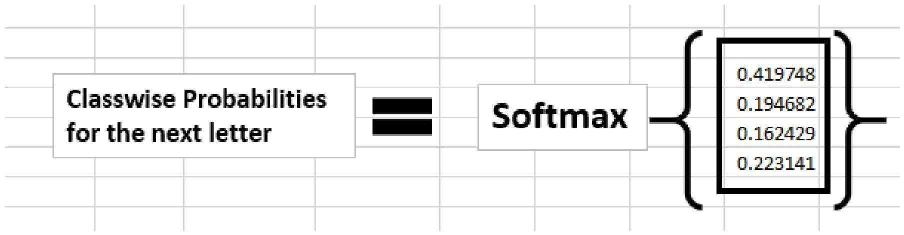
$$y_t = W_{hy}h_t$$



why				Ht		yt
0.37168	0.974829459	0.830034886	✖	0.936534	=	1.90607732
0.39141	0.282585823	0.659835709		0.949104		1.13779113
0.64985	0.09821557	0.334287084		0.762341		0.95666016
0.91266	0.32581642	0.144630018				1.27422602

Step 7:

The probability for a particular letter from the vocabulary can be calculated by applying the softmax function. so we shall have softmax(yt)



If we convert these probabilities to understand the prediction, we see that the model says that the letter after “e” should be h, since the highest probability is for the letter “h”. Does this mean we have done something wrong? No, so here we have hardly trained the network. We have just shown it two letters. So it pretty much hasn’t learned anything yet.

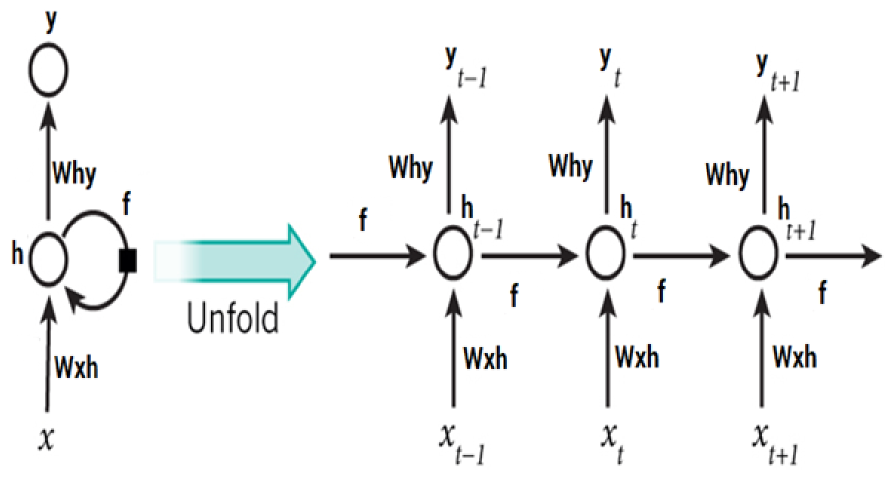
Now the next BIG question that faces us is how does Backpropagation work in case of a Recurrent Neural Network. How are the weights updated while there is a feedback loop?

4. Backpropagation is an RNN (BPTT).

To imagine how weights would be updated in case of a recurrent neural network, might be a bit of a challenge. So to understand and visualize the backpropagation, let’s unroll the network at all the time steps. In an RNN we may or may not have outputs at each time step.

In the case of forwarding propagation, the inputs enter and move forward at each time step. In case of backward propagation, in this case, we are figuratively going back in time to change

the weights, hence we call it the Backpropagation through time(BPTT).



In case of an RNN, if yt is the predicted value ȳt is the actual value, the error is calculated as a cross-entropy loss –

Et(ȳt,yt) = —ȳt log(yt)

E(ȳ,y) = —∑ ȳt log(yt)

We typically treat the full sequence (word) as one training example, so the total error is just the sum of the errors at each time step (character). The weights as we can see are the same at each time step. Let's summarize the steps for backpropagation

1. The cross-entropy error is first computed using the current output and the actual output
2. Remember that the network is unrolled for all the time steps
3. For the unrolled network, the gradient is calculated for each time step with respect to the weight concerninghat the weight is the same for all the time steps the gradients can be combined together for all time stepshts are then updated for both recurrent neuron and the dense layers

The unrolled network looks much like a regular neural network. And the backpropagation algorithm is similar to a regular neural network, just that we combine the gradients of the error for all time steps. Now, what do you think might happen, if there are 100s of time steps. This would basically take really long tworkverge since after unrolling the network becomes really huge.

In case you dosh to deep dive into the math of backpropagation, all you need to understand is that backpropagation through time works similarly as it does in a regular neural network once you unroll the recurrent neuron in your network. However, I shall be coming up with a detailed article on Recurrent Neural networks with scratch with would have the detailed mathematics of the backpropagation algorithm in a recurrent neural network.

5. Vanishing and Exploding Gradient Problem.

Recurrent Neural Networks use a **backpropagation algorithm** for training, **but** it is **applied** for every **timestamp.** It is commonly known as **Back-propagation Through Time (BTT) which we already discussed above.**

There are **some issues** with Back-propagation such as:

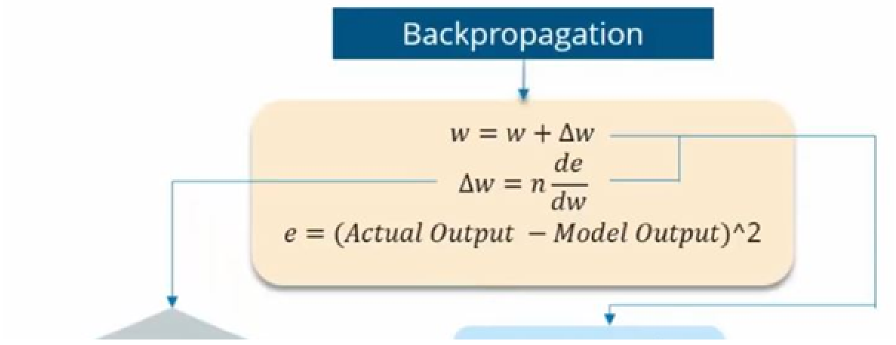
- **Vanishing Gradient**
- **Exploding Gradient**

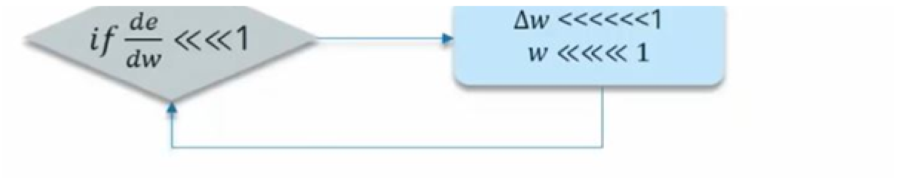
Let us consider each of these to understand what is going on

5.1: Vanishing Gradient

When making use of back-propagation the goal is to calculate the error which is actually found out by findie difference between the actual output and the model output and raising that to a power of 2.

Consider the following diagram:


Backpropagation
$w = w + \Delta w$
$\Delta w = n\frac{de}{dw}$
$e = (Actual\ Output\ - Model\ Output)$^2



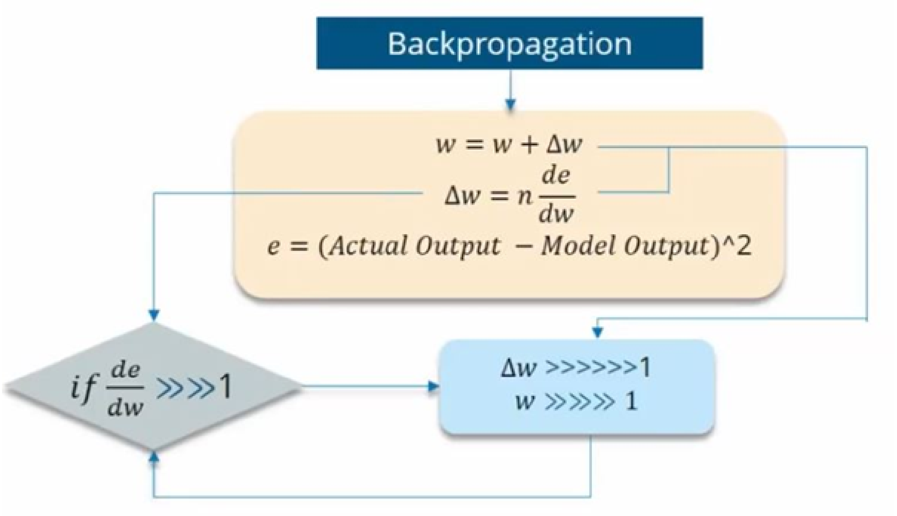
With the error calculated, the changes in the error with respect to the change iconcerninge calculated. But with each learning rate, this has to be multiplied with the same.

So, the product of the learning rate with the change leads to the value which is the actual change in the weight. This change in weight is added to the old set of weights for every training iteration as shown in the figure. The issue here is when the change in weight is multiplied, the value is very less. Consider you are predicting a sentence say, “I am going to Rahim Yar Khan” and you want to predict “I am going to Rahim Yar Khan, the language spoken there is _____”

A lot of iterations will cause the new weights to be extremely negligible and this leads to the weights not being updated.

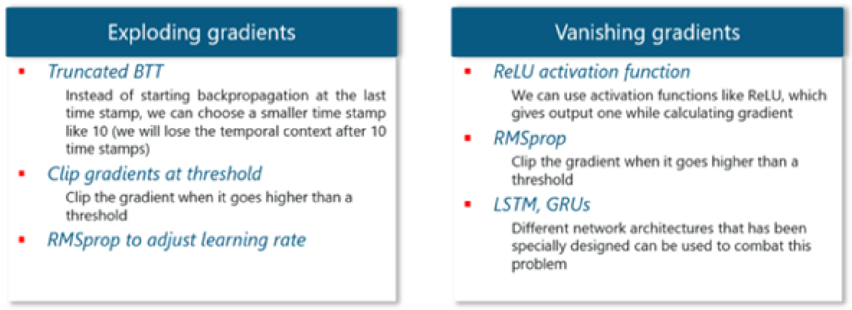
5.2. Exploding Gradient

The working of the exploding gradient is similar but the weights here change drastically instead of negligible change. Notice the small change in the diagram below:



Backpropagation
$w = w + \Delta w$
$\Delta w = n\frac{de}{dw}$
$e = (Actual\ Output - Model\ Output)^2$
$if \frac{de}{dw} \ggg 1$
$\Delta w >>>>>>1$
$w \ggg\ggg 1$

We need to overcome both of these and it is a bit of a challenge at first. Consider the following chart



Exploding gradients

- *Truncated BTT*
  Instead of starting backpropagation at the last time stamp, we can choose a smaller time stamp like 10 (we will lose the temporal context after 10 time stamps)
- *Clip gradients at threshold*
  Clip the gradient when it goes higher than a threshold
- *RMSprop to adjust learning rate*

Vanishing gradients

- *ReLU activation function*
  We can use activation functions like ReLU, which gives output one while calculating gradient
- *RMSprop*
  Clip the gradient when it goes higher than a threshold
- *LSTM, GRUs*
  Different network architectures that has been specially designed can be used to combat this problem

6. Long Short Term Memory (LSTM).

Long Short-Term Memory networks are usually just called “LSTMs”.

They are a special kind of Recurrent Neural Networks that are capable of learning long-term dependencies.

What are long-term dependencies?

Many times only recent data is needed in a model to perform operations. But there might be a requirement from data which was obtained in the past.

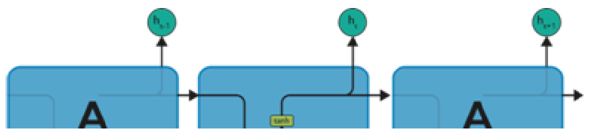
Let’s look at the following example:

Consider a language model trying to predict the next word based on the previous ones. If we are trying to predict the last word in the sentence say “The clouds are in the sky”.

The context here was pretty simple and the last word ends up being sky all the time. In such cases, the gap between the past information and the current requirement can be bridged really easily by using Recurreal Networks.

So, problems like Vanishing and Exploding Gradients do not exist and this makes LSTM networks handle long-term dependencies easily.

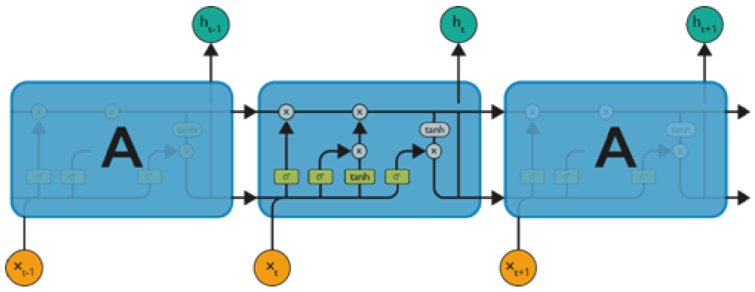
LSTM has a chain-like neural network layer. In a standard recurrent neural network, the repeating module consists of one single function as shown in the below figure:



As shown above, there is a tanh function present in the layer. This function is a squashing function. So, what squashingion?

It is a function that is basically used in the range of nd to manipulate the values based on the inputs.

Now, let us consider the structure of an LSTM network:

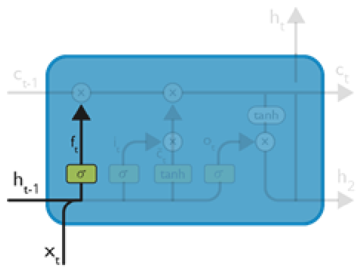


As denoted from the image, each of the functions in the layers has its own structures when it comes to networks. The cell state is the horizontal line in the figure and it acts like a conveyer belt carrying certain data linearly across the data channel.

Let us consider a step-by-step approach to understand LSTM networks better.

Step 1:

The first step in the LSTM is to identify that information which is not required and will be thrown away from the cell state. This decision is made by a sigmoid layer called a forget gate layer.



The highlighted layer in the above is the sigmoid layer which is previously mentioned.

The calculation is done by considering the new input and the previous timestamp which eventually leads to the output of a number between 0 and 1 for each number in that cell state. As a typical binary, 1 represents to keep the cell state while 0 represents to trash it.

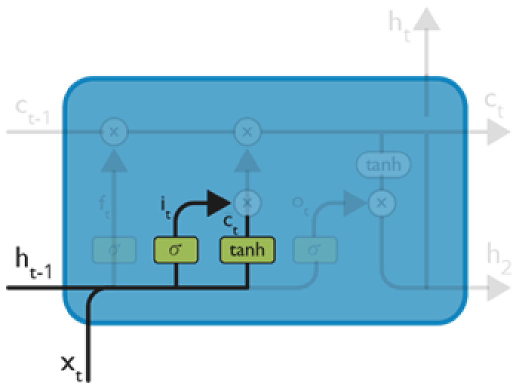
$$f_t = \sigma(w_f[h_{t-1}, x_t] + b_f)$$

$w_f = Weight$
$h_{t-1} = Output\ from\ previous\ timestamp$
$x_t = New\ input$
$b_f = Bias$

Consider gender classification, it is really important to consider the latest and correct gender when the network is being used.

Step 2:

The next step is to decide, what new information we’re going to store in the cell state. This whole process comprises of following steps:



$$i_t = \sigma(w_i[h_{t-1}, x_t] + b_i)$$

$$c\tilde{}_t = tanh(w_c[h_{t-1}, x_t] + b_c)$$

A sigmoid layer called the “input gate layer” decides which values will be updated.

The tanh layer creates a vector of new candidate values that could be added to the state.

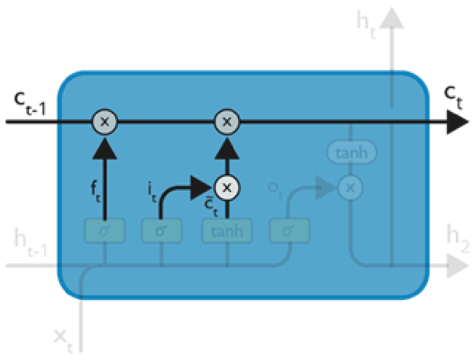
The input from the previous timestamp and the new input are passed through a sigmoid function which gives the value i(t). This value is then multiplied by c(t) and then added to the cell state.

In the next step, these two are combined to update the state.

Step 3:

Now, we will update the old cell state Ct−1, into the new cell state Ct.

First, we multiply the old state (Ct−1) by f(t), forgetting the things we decided to leave behind earlier.



$$c_t = f_t * c_{t-1} + i_t{}^* \, c\tilde{}_t$$

We add i_t* c˜_t. This is the new candidate values, scaled by how much we decided to update each state value.

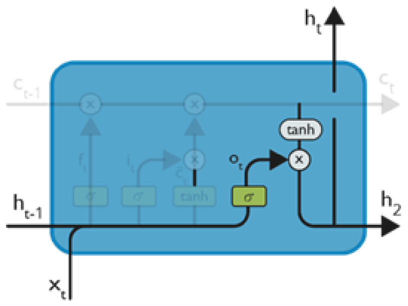
In the second step, we decided to do make use of the data which is only required at that stage.

In the third step, we actually implement it.

In the languagemple which was previously discussed, there is where the old gender would be dropped and the new gender would be considered.

Step 4:

We will run a sigmoid layer which decides what parts of the cell state we're going to output. Then, we put the cell state through tanh (push the values to be between −1 and 1). Later, we multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.



$$o_t = \sigma(w_o[h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * \tanh(c_t)$$

The calculation in this step is pretty much straightforward which eventually leads to the output. However, the output consists of only the outputs there were decided to be carried forwarded in the previous steps and not all the outputs at once.

Summing up all the 4 steps:

1. In the first step, we found out what was needed to be dropped.

2. The second step consisted of what new inputs are added to the network.

3. The third step was to combine the previously obtained inputs to generate the new cell states.

4. Lastly, we arrived at the output as per requirement.

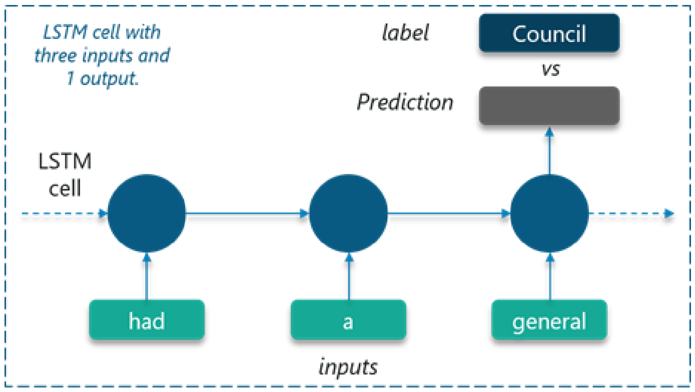
Let's consider an interesting use-case.

Use Case: Long Short-Term Memory Networks

The use case we will be **considering** is to **predict** the **next word** in a sample short story.

We can start by **feeding** an **LSTM** Network with **correct sequences** from the text of 3 **symbols** as **inputs** and 1 labeled symbol.

Eventually, the neural network will **learn** to **predict** the next symbol **correctly!**



Dataset:

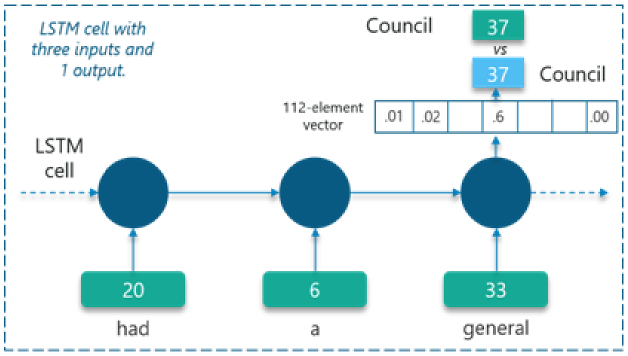
The LSTM is trained using a **sample short story** which consists of **112 unique symbols. Comma** and **period** are also **considered** as **unique symbols** in this **case.**

"Long ago, the mice had a general council to consider what measures they could take to outwit their common enemy, the cat. some said this, and some said that but at last a young mouse got up and said he had a proposal to make, which he thought would meet the case . You will all agree, said he that our chief danger consists in the sly and treacherous manner in whicslyly and treacherouslyf we could receive some signal of her approach, we could easily escape from her. I venture, therefore, to propose that a small bell be procured, and attached by a ribbon around the neck of the cat. By this means we should always know when she was about, and could easily retire while she was in the neighborhood. This proposal met with general applause until an old mouse got up and said that it is all very well, but who is to bell the cat? The mice looked at one another and nobody spoke. Then the old mouse said it is easy to propose impossible remedies."

Training:

We already know that **LSTMs** can only **understand real numbers.** So, the first **requirement** is to **convert** the unique symbols into **unique integer** values based on the **frequency** of **occurrence.**

Doing this will create a **customized dictionary** that we can **make use** later on to **map** the values.



In the above figure, **certain symbols** are mapped to be **integers** as shown.

The network will create a **112-element vector** consisting of the **probability** of **occurrence** of each of these unique integer values.

Note: If you want this article check out my **academia.edu** profile.

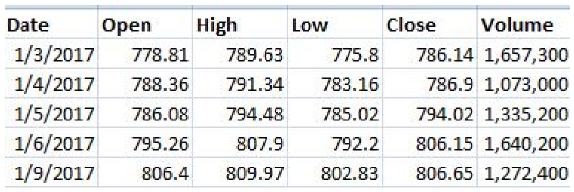
7. Practical Implementation of Recurrent Neural Network and LSTM.

Stock Price Prediction

In this part, we will create one of the most powerful Deep Learning models. We will even go as far as saying that we will create the Deep Learning model closest to “Artificial Intelligence”. Why is that? Because this model will have long-term memory, just like us, humans.

The branch of Deep Learning which facilitates this is Recurrent Neural Networks. Classic RNNs have a short memory and were neither popular nor powerful for this exact reason. But a recent major improvement in Recurrent Neural Networks gave rise to the popularity of LSTMs (Long Short Term Memory RNNs) which has completely changed the playing field. In this part, we will learn how to implement this ultra-powerful model, and we will take the challenge to use it to predict the real Google stock price. A similar challenge has already been faced by researchers at Stanford University and we will aim to do at least as good as them.

Dataset sample:



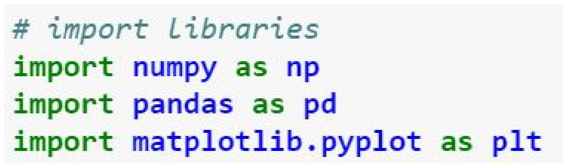
Date	Open	High	Low	Close	Volume
1/3/2017	778.81	789.63	775.8	786.14	1,657,300
1/4/2017	788.36	791.34	783.16	786.9	1,073,000
1/5/2017	786.08	794.48	785.02	794.02	1,335,200
1/6/2017	795.26	807.9	792.2	806.15	1,640,200
1/9/2017	806.4	809.97	802.83	806.65	1,272,400

Let’s solve the problem

Part 1: Data Preprocessing

1.1 Import the Libraries:

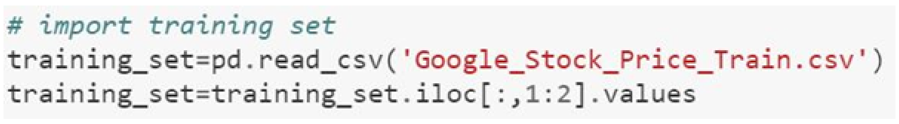
In this step, we import three Libraries in Data Preprocessing part. Basically, Library is a tool that you can ua specific job. First of all, we import the **numpy** library used for multidimensional array then import the **pandas** library used to import the dataset and in last we import **matplotlib** library used for plotting the graph.



```
# import libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

1.2 Import the dataset

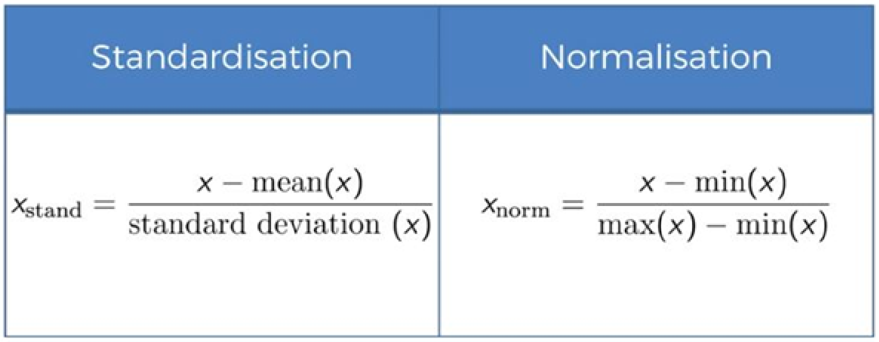
In this step, we import the dataset to do that we use the **pandas** library. After import, our dataset and we take only one attribute of Google stock price prediction which is open Google stock price which we want to predict.



```
# import training set
training_set=pd.read_csv('Google_Stock_Price_Train.csv')
training_set=training_set.iloc[:,1:2].values
```

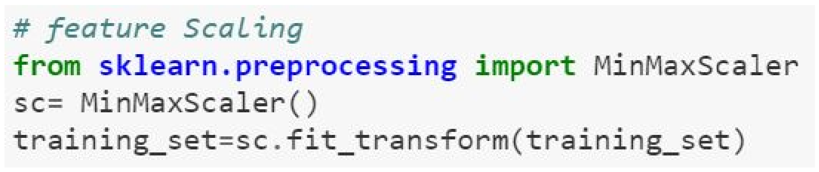
1.3 Feature Scaling

Feature Scaling is the most important part of data preprocessing. If we see our dataset then some attribute contains information in Numeric value some value very high and some are very low if we see the age and estimated salary. This will cause some issues in our machinery model to solve that problem we set all values on the same scale there are two methods to solve that problem first one is Normalize and Second is Standard Scaler.



Standardisation	Normalisation
$x_{\text{stand}} = \frac{x - \text{mean}(x)}{\text{standard deviation }(x)}$	$x_{\text{norm}} = \frac{x - \min(x)}{\max(x) - \min(x)}$

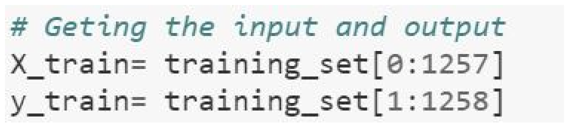
Here we use Normalize Scalar because in Google Stock Price Prediction we build the LSTM model which has several sigmoid functions (which is 0 or 1) that are why we choose Normalize Scalar here.



```
# feature Scaling
from sklearn.preprocessing import MinMaxScaler
sc= MinMaxScaler()
training_set=sc.fit_transform(training_set)
```

1.4 Getting the Input and Output

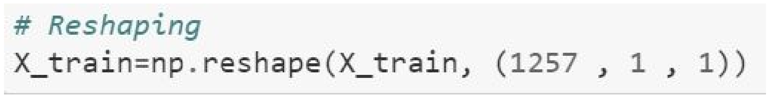
In this step, we create our input and output from training data. Here X_train is our input which is **t** and Y_train is our output which is **t+1**.



```
# Geting the input and output
X_train= training_set[0:1257]
y_train= training_set[1:1258]
```

1.5 Reshaping

In this step, we reshape our input. Why I Reshape here Because our input shape have 2 dimension one dimension corresponds to observation and second correspond to feature which has only one feature here and we convert our input into three-dimension last 1 corresponds to a timestamp because our input is t and output is t+1. So t+1-t= 1 that's why 1 here. So here 1257 is observation, 1 is time step, 1 is feature scaling.



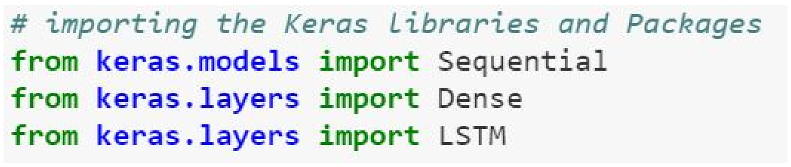
```
# Reshaping
X_train=np.reshape(X_train, (1257 , 1 , 1))
```

Part 2: Building our Model

In this part, we model our Recurrent Neural Network model.

2.1 Import the Libraries

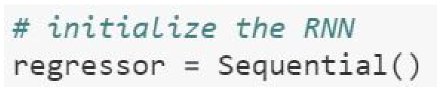
In this step, we import the Library which will build our RNN model. We import **Keras** Library which will build a deep neural network based on Tensorflow because we use Tensorflow backhand. Here we import the three modules from Keras. The first one is **Sequential** used for initializing our RNN model and second is **Dense** used for adding different layers of RNN and third is **LSTM** which we use in the RNN model.



```
# importing the Keras libraries and Packages
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
```

2.2 Initialize our RNN model

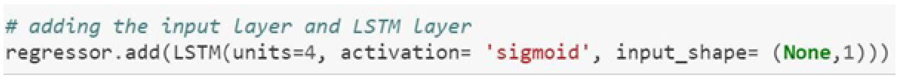
In this step, we initialize our Recurrent Neural Network model to do that we use sequential modules.



```
# initialize the RNN
regressor = Sequential()
```

2.3 Adding the input layer and LSTM layer

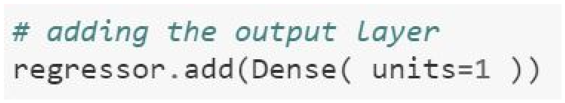
In this step, we add the LSTM layer and the input. Here we pass several parameters pass here first is **unit=4** which corresponds to number ofry unit I choose 4 here second is **activation = sigmoid** which is activation function I choose both tanh & sigmoid but I choose sigmoid activation here and the third parameter is **input shape** and in input shape I pass 2 arguments here 1 for timestep and 1 for feature.



```
# adding the input layer and LSTM layer
regressor.add(LSTM(units=4, activation= 'sigmoid', input_shape= (None,1)))
```

2.4 Adding the output layer

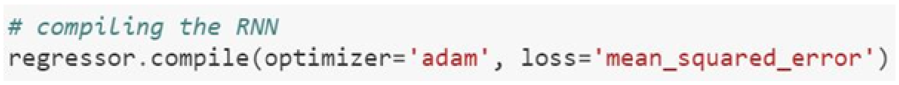
In this step, we add the output layer to do we use Dense class here. Here in output, we pass only one argument here which is **unit** corresponds to the number of neurons in the output layer that corresponds to the dimension of outcome since the outcome is stock price t+1 so the outcome has 1 dimension there we need put here units=1 stock price t+1.



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

2.5 Compiling the RNN

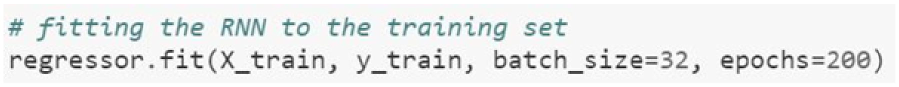
In this step, we compile the RNN to do that we use the compile method and add several parameters the first parameter is **optimizer = Adam** here use the optimal number of weights. So for choosing the optimal number of weights, there are various algorithms of Stochastic Gradient Descent but very efficient one which is Adam so that's why we use Adam optimizer here. The second parameter is loss this corresponds to loss function here we use the **mean_squared_error** because the dataset is regression value so that why we loss function means square error here. It's basically the sum of the squared distance between youd stock price and the real stock price.

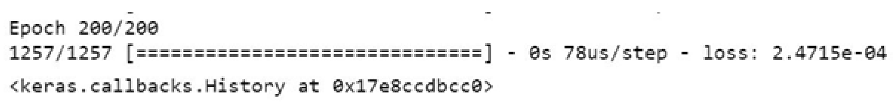


```
# compiling the RNN
regressor.compile(optimizer='adam', loss='mean_squared_error')
```

2.6 Fitting the RNN to the Training set

In this step we fit the training data our model X_train, y_train is our training data. Here a **batch size** is basically a number of observations after which you wanseveralts here we take batch size 32. And the final parameter is **epoch** is basically when whole the training set passed through the RNN here we choose the 200 number of the epoch.


```
# fitting the RNN to the training set
regressor.fit(X_train, y_train, batch_size=32, epochs=200)
```

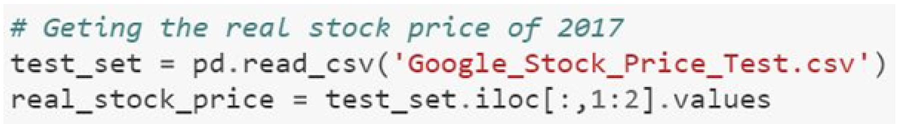

```
Epoch 200/200
1257/1257 [==============================] - 0s 78us/step - loss: 2.4715e-04
<keras.callbacks.History at 0x17e8ccdbcc0>
```

Part 3: Making the Prediction and Visualize the Result.

In this step, we make a prediction of our test set of Google Stock Price and then visualize the result.

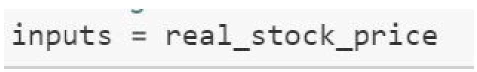
3.1 Import the test set dataset of Google Stock Price

In this step, we import our test set dataset of Google stock price.

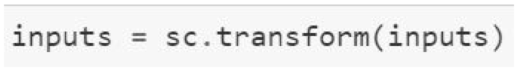

```
# Geting the real stock price of 2017
test_set = pd.read_csv('Google_Stock_Price_Test.csv')
real_stock_price = test_set.iloc[:,1:2].values
```

3.2 Getting the Predicted Stock Price of 2017

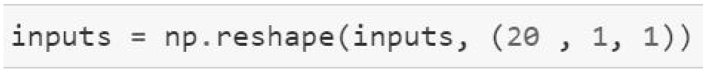
First, we need input as we have already done in part 1


```
inputs = real_stock_price
```

Then Scale our value as we have already done in part 1


```
inputs = sc.transform(inputs)
```

Then Reshape as we have already done in part 1


```
inputs = np.reshape(inputs, (20 , 1, 1))
```

Now predicted our test set result uses the regressor model which we already train in part 2.

```
predicted_stock_price = regressor.predict(inputs)
```

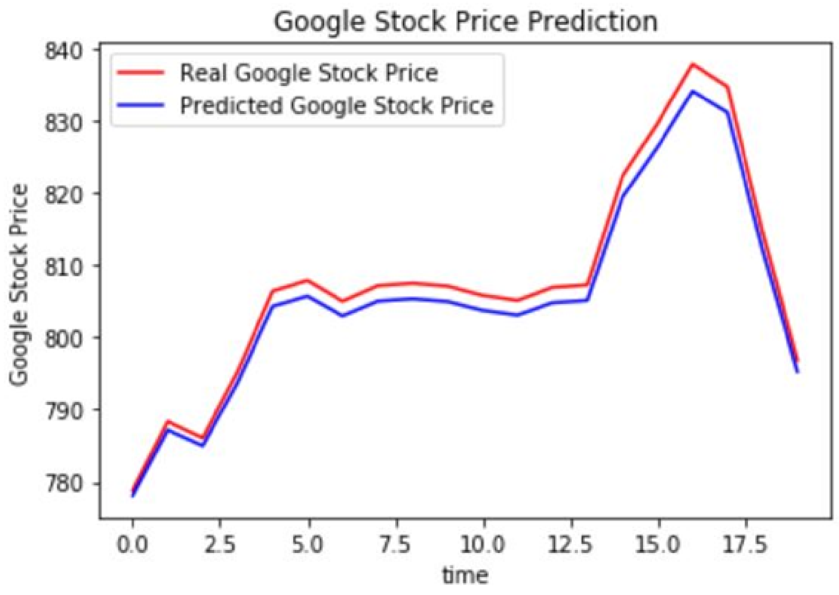
The predicted result is scale value now we convert these values into original values using the inverse method.

```
predicted_stock_price = sc.inverse_transform(predicted_stock_price)
```

3.3 Visualize the Result

In this step, we compare our predicted results with real stock prices. As you can see below.

```
# Visulising the Result
plt.plot( real_stock_price , color = 'red' , label = 'Real Google Stock Price')
plt.plot( predicted_stock_price , color = 'blue' , label = 'Predicted Google Stock Price')
plt.title('Google Stock Price Prediction')
plt.xlabel( 'time' )
plt.ylabel( 'Google Stock Price' )
plt.legend()
plt.show()
```



If you want dataset and code you also check my **Github** Profile.

End Notes

If you liked this article, be sure to click ❤ below to recommend it and if you have any questions, **leave a comment** and I will do my best to answer.

For being more aware of the world of machine learning, **follow me**. It's the best way to find out when I write more articles like this.

You can also follow me on **Github** for code & dataset follow on **Aacademia.edu** for this article, **Twitter** and **Email** me directly or find me on **LinkedIn.** I'd love to hear from you.

That's all folks, Have a nice day :)

Recurrent Neural Network Long Short Term Memory Vanishing Gradient Exploding Gradient

Backpropagation