**Website Traffic Analysis**

**Project Submission Part 3: Development Part 1**

**Title:DAC\_Phase 3**

Member name: C.Shanmugapriya/au812121106029

Time-series Database is the fastest-growing category of databases in the past two years, and both traditional and emerging technology industries have been generating more and more time-series data.

Now let’s try to understand the business use case that we tackling here.

The web traffic is basically the number of sessions in a given time frame, and it varies a lot with respect to what time of the day it is, what day of the week it is, and so on, and how much web traffic of platform can withstand depends on the size of the servers that are supporting the platform.

If the traffic is more than what the servers can handle, the website might show this 404 error, which is something we don’t want to happen. It will make the visitors go away.

One solution to this problem is to increase the number of servers. However, the downside of the solution is the cause can go up, which is again undesirable. So, what is the solution?

You can dynamically a lot of servers based on the historical visitor’s volume data or based on the historical web traffic data. And that brings us to the data science problem, which is basically forecasting the web traffic or a number of sessions based on the historical data.

We will deep dive into the web traffic data set and look at how we can use LSTM to solve this time series forecasting problem.

Now we will cover the problem statement of web traffic forecasting, and how it will help in scaling the resources, backing the outline publishing platform.

We will work with the web traffic dataset. It is a six-month series data set the link is given here [**Link**](https://www.kaggle.com/kajal1/web-traffic-forecast-dataset).

Load Dataset for Web Traffic Forecasting

Here we are reading the dataset by using pandas. It has over 4800 observations.

import pandas as pd

import numpy as np

data=pd.read\_csv('webtraffic.csv')

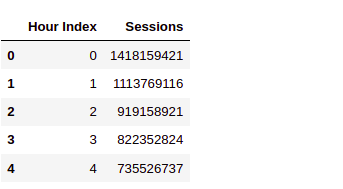
Check the shape of the data

data.shape

https://editor.analyticsvidhya.com/uploads/80765Screenshot%20from%202021-09-08%2009-35-41.png

To print the first records of the dataset.

data.head()



The first column is the hours as in this is the first hours, this is the second hour and so on.

And the second column session is the volume of traffic at an hourly level.

For example:- this is the number of sessions in the second hours and so on.

Data Exploration for Web Traffic Forecasting

Now let’s explore the data, we will use the below code to plot the entire time series there you go.

import matplotlib.pyplot as plt

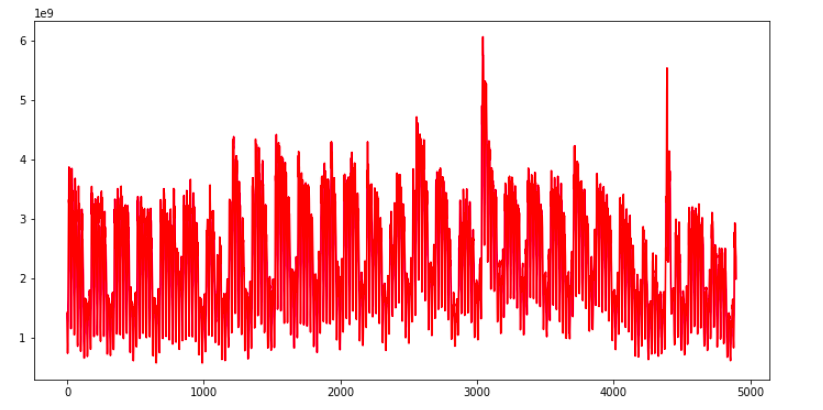
sessions = data['Sessions'].values

ar = np.arange(len(sessions))

plt.figure(figsize=(22,10))

plt.plot(ar, sessions,'r')

plt.show()



Each point of this curve is an early session count and you can see there are some repeating patterns throughout the time series.

The traffic volume comes down, after almost equal intervals of time. Apart from that, there are a couple of spikes as well in the traffic, In this plot.

Let’s explore this data, at a more granular level, we can use the below code and replace the entire time series, with a subset of it.

#first week web traffic

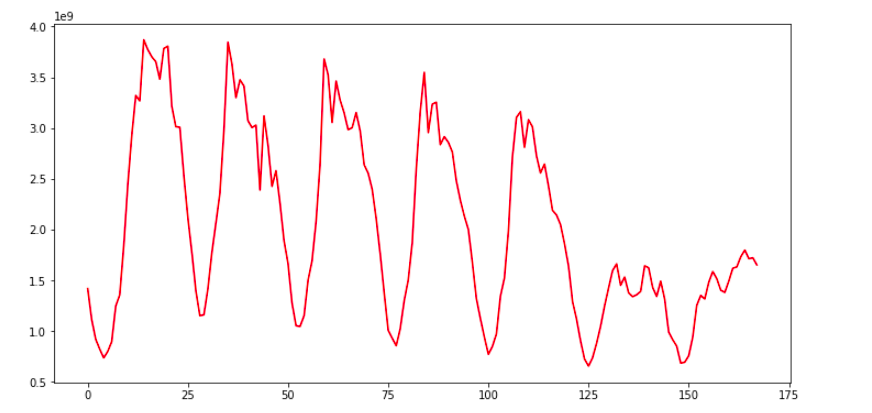
sample = sessions[:168]

ar = np.arange(len(sample))

plt.figure(figsize=(22,10))

plt.plot(ar, sample,'r')

plt.show()

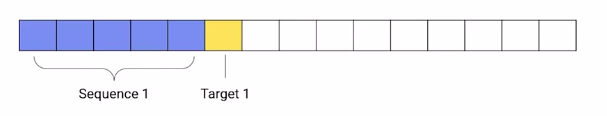


Here we are plotting the first week’s data only, now the repeating pattern can be seen more clearly, and these dips in the plot in web traffic are may be occurring once every 24 hours. So clearly there are two instances of time in a day, when we have a huge traffic volume, like during a few times and when we have a modest level of traffic on the website. As in here, I will help you to explore this data as much as possible, before getting started with model building.

Data Preparation for Web Traffic Forecasting

Moving on now let’s prepare the data for model training, here we will create input sequences, from the block traffic data. Let’s say this is a time series. Each cell would have some number or value. Let’s create sequences of length five, so the first five observations, will form the first sequence, and the sixth observation, this one will be treated as the target.

The second sequence will start from the second element, till the sixth element and the target will be the seventh element.





Now the subsequent sequences will be extracted, by moving this window, one step at a time.

def prepare\_data(seq,num):

x=[]

y=[]

for i in range(0,(len(seq)-num),1):

input\_ = seq[i:i+num]

output = seq[i+num]

x.append(input\_)

y.append(output)

return np.array(x), np.array(y)

In this function, prepare\_data. We are using the same technique to create sequences from time-series data. We have specified the sequence length of one week or 168 hours.

num=168

x,y= prepare\_data(sessions,num)

print(len(x))

Now here we are calling this function to create sequences. The sequence length we have specified is 168 hours and that is equivalent to one week. So we are creating sequences of one week, as our input sequences. Now the number of sequences are well over 4700.

Split the Dataset

Next, we have to split the data into a training set and validation set and we will do this in the ratio, 90 is to 10. Now that sense it is a time serious problem, we are not splitting the data randomly, we are splitting it in a sequential manner. As you can see the code below.

ind = int(0.9 \* len(x))

x\_tr = x[:ind]

y\_tr = y[:ind]

x\_val=x[ind:]

y\_val=y[ind:]

nowhere in the code we would scale the data, Both the input sequences and the target values, will be scaled because of scaling the data, speeds because of scaling the data, speeds of the model training process.

from sklearn.preprocessing import StandardScaler

#normalize the inputs

x\_scaler= StandardScaler()

x\_tr = x\_scaler.fit\_transform(x\_tr)

x\_val= x\_scaler.transform(x\_val)

#reshaping the output for normalization

y\_tr=y\_tr.reshape(len(y\_tr),1)

y\_val=y\_val.reshape(len(y\_val),1)

#normalize the output

y\_scaler=StandardScaler()

y\_tr = y\_scaler.fit\_transform(y\_tr)[:,0]

y\_val = y\_scaler.transform(y\_val)[:,0]

After that, we are reshaping the data from two dimensional to 3 dimensional.

#reshaping input data

x\_tr= x\_tr.reshape(x\_tr.shape[0],x\_tr.shape[1],1)

x\_val= x\_val.reshape(x\_val.shape[0],x\_val.shape[1],1)

print(x\_tr.shape)

https://editor.analyticsvidhya.com/uploads/24413Screenshot%20from%202021-09-08%2009-49-03.png

The first dimension of our data is the number of sequences, and the second dimension is the number of elements in the sequences. But LSTM layer accepts only three-dimensional data.



These three dimensions are the number of sequences number of time steps and the length of the features. So third dimension is the length of the vectors of the sequence elements.

Let’s say I have five elements in my sequence and each of these elements has a vector length of 10. So this third dimension will become 10. If you can recall in the case of the auto-tagging projects the length of the sequence elements was nothing but the length of the word embeddings.

However in this dataset, the sequence elements are real number values and therefore the feature length is just one, hence we would reshape both the training set and the validation set as shown in the above code. Now, the data is ready for model training.

In the next section, we will build our deep learning model to predict traffic using LSTM.

Model Building for Web Traffic Forecasting

In the previous section, we covered how time-series data is converted into sequences to train data.

And now we will use these sequences to train a deep learning model to predict future web traffic. The first thing that we do is define the model architecture.

from keras.models import \*

from keras.layers import \*

from keras.callbacks import \*

from tensorflow import keras

# define model

model = Sequential()

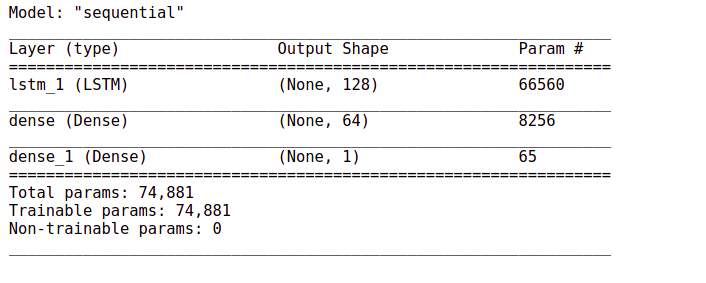
model.add(LSTM(128,input\_shape=(168,1)))

model.add(Dense(64,activation='relu'))

model.add(Dense(1,activation='linear'))

Have a look at the activation at the final layer above the code It is linear. This is because we have to predict a continuous value and not some class tag or category as it is a regression problem and not a classification problem. Other than that we are using a single layer of LSTM here and the input shape is 168 that is one week.

model.summary()



The number of train parameters is just around 74000.

# Define the optimizer and loss

model.compile(loss='mse',optimizer='adam')

#Define the callback to save the best model during the training

mc = ModelCheckpoint('best\_model.hdf5', monitor='val\_loss',

verbose=1, save\_best\_only=True, mode='min')

# Train the model for 30 epochs with batch size of 32:

history=model.fit(x\_tr, y\_tr ,epochs=30, batch\_size=32,

validation\_data=(x\_val,y\_val), callbacks=[mc])

The output will be:-

Epoch 1/30  
133/133 [==============================] – 15s 101ms/step – loss: 0.1409 – val\_loss: 0.0339

Epoch 00001: val\_loss improved from inf to 0.03390, saving model to best\_model.hdf5  
Epoch 2/30  
133/133 [==============================] – 17s 126ms/step – loss: 0.0394 – val\_loss: 0.0286

Epoch 00002: val\_loss improved from 0.03390 to 0.02865, saving model to best\_model.hdf5  
Epoch 3/30  
133/133 [==============================] – 19s 142ms/step – loss: 0.0369 – val\_loss: 0.0265

Epoch 00003: val\_loss improved from 0.02865 to 0.02647, saving model to best\_model.hdf5  
Epoch 4/30  
133/133 [==============================] – 19s 142ms/step – loss: 0.0332 – val\_loss: 0.0269

Epoch 00004: val\_loss did not improve from 0.02647  
Epoch 5/30  
133/133 [==============================] – 18s 137ms/step – loss: 0.0329 – val\_loss: 0.0240

Epoch 00005: val\_loss improved from 0.02647 to 0.02402, saving model to best\_model.hdf5  
Epoch 6/30  
133/133 [==============================] – 19s 143ms/step – loss: 0.0303 – val\_loss: 0.0250

**Blogathon is Live AGAIN !**

Get Assured **Rs 2000** for Every Published Article!

Get **2x Rewards** for Gen AI Articles!

[Register Now](https://datahack.analyticsvidhya.com/blogathon/?utm_source=blog&utm_medium=banner_b/w_articles&utm_campaign=Blogathon37)

Epoch 00006: val\_loss did not improve from 0.02402  
Epoch 7/30  
133/133 [==============================] – 21s 155ms/step – loss: 0.0283 – val\_loss: 0.0248

Epoch 00007: val\_loss did not improve from 0.02402  
Epoch 8/30  
133/133 [==============================] – 21s 156ms/step – loss: 0.0290 – val\_loss: 0.0211

Epoch 00008: val\_loss improved from 0.02402 to 0.02107, saving model to best\_model.hdf5  
Epoch 9/30  
133/133 [==============================] – 19s 146ms/step – loss: 0.0269 – val\_loss: 0.0231

Epoch 00009: val\_loss did not improve from 0.02107  
Epoch 10/30  
133/133 [==============================] – 19s 140ms/step – loss: 0.0269 – val\_loss: 0.0241

Epoch 00010: val\_loss did not improve from 0.02107  
Epoch 11/30  
133/133 [==============================] – 18s 138ms/step – loss: 0.0252 – val\_loss: 0.0201

Epoch 00011: val\_loss improved from 0.02107 to 0.02014, saving model to best\_model.hdf5  
Epoch 12/30  
133/133 [==============================] – 19s 144ms/step – loss: 0.0237 – val\_loss: 0.0199

Epoch 00012: val\_loss improved from 0.02014 to 0.01994, saving model to best\_model.hdf5  
Epoch 13/30  
133/133 [==============================] – 27s 201ms/step – loss: 0.0215 – val\_loss: 0.0160

Epoch 00013: val\_loss improved from 0.01994 to 0.01602, saving model to best\_model.hdf5  
Epoch 14/30  
133/133 [==============================] – 23s 170ms/step – loss: 0.0197 – val\_loss: 0.0202

Epoch 00014: val\_loss did not improve from 0.01602  
Epoch 15/30  
133/133 [==============================] – 17s 127ms/step – loss: 0.0188 – val\_loss: 0.0168

Epoch 00015: val\_loss did not improve from 0.01602  
Epoch 16/30  
133/133 [==============================] – 17s 127ms/step – loss: 0.0179 – val\_loss: 0.0173

Epoch 00016: val\_loss did not improve from 0.01602  
Epoch 17/30  
133/133 [==============================] – 17s 125ms/step – loss: 0.0179 – val\_loss: 0.0188

Epoch 00017: val\_loss did not improve from 0.01602  
Epoch 18/30  
133/133 [==============================] – 17s 127ms/step – loss: 0.0171 – val\_loss: 0.0140

Epoch 00018: val\_loss improved from 0.01602 to 0.01395, saving model to best\_model.hdf5  
Epoch 19/30  
133/133 [==============================] – 18s 133ms/step – loss: 0.0165 – val\_loss: 0.0155

Epoch 00019: val\_loss did not improve from 0.01395  
Epoch 20/30  
133/133 [==============================] – 17s 127ms/step – loss: 0.0165 – val\_loss: 0.0216

Epoch 00020: val\_loss did not improve from 0.01395  
Epoch 21/30  
133/133 [==============================] – 17s 126ms/step – loss: 0.0160 – val\_loss: 0.0166

Epoch 00021: val\_loss did not improve from 0.01395  
Epoch 22/30  
133/133 [==============================] – 17s 125ms/step – loss: 0.0165 – val\_loss: 0.0158

Epoch 00022: val\_loss did not improve from 0.01395  
Epoch 23/30  
133/133 [==============================] – 17s 126ms/step – loss: 0.0161 – val\_loss: 0.0166

Epoch 00023: val\_loss did not improve from 0.01395  
Epoch 24/30  
133/133 [==============================] – 17s 128ms/step – loss: 0.0158 – val\_loss: 0.0150

Epoch 00024: val\_loss did not improve from 0.01395  
Epoch 25/30  
133/133 [==============================] – 17s 127ms/step – loss: 0.0164 – val\_loss: 0.0164

Epoch 00025: val\_loss did not improve from 0.01395  
Epoch 26/30  
133/133 [==============================] – 17s 131ms/step – loss: 0.0158 – val\_loss: 0.0179

Epoch 00026: val\_loss did not improve from 0.01395  
Epoch 27/30  
133/133 [==============================] – 17s 127ms/step – loss: 0.0155 – val\_loss: 0.0159

Epoch 00027: val\_loss did not improve from 0.01395  
Epoch 28/30  
133/133 [==============================] – 17s 127ms/step – loss: 0.0153 – val\_loss: 0.0141

Epoch 00028: val\_loss did not improve from 0.01395  
Epoch 29/30  
133/133 [==============================] – 17s 128ms/step – loss: 0.0149 – val\_loss: 0.0260

Epoch 00029: val\_loss did not improve from 0.01395  
Epoch 30/30  
133/133 [==============================] – 17s 128ms/step – loss: 0.0154 – val\_loss: 0.0145

Epoch 00030: val\_loss did not improve from 0.01395

Load the weights of the best model prior to predictions. Now here we will use the mean squared error and we are using model checkpoint again to save the best model weight.

model.load\_weights('best\_model.hdf5')

finally, the model training starts and moving on the evaluation starts here and moving on to the evaluation part, the mean squared error for the validation data is just 0.013. Evaluate the performance of the model on the validation data.

mse = model.evaluate(x\_val,y\_val)

print("Mean Square Error:",mse)

MSE 

Now, whenever we are working on a project it is always a good practice to have a baseline model, just to have an idea of how good your model is with respect to the baseline predictions.

Baseline Model with Forecasting

So here we are using a simple moving average as the baseline model. So what we will do, we will take a sequence and its length is the same 168 elements. And then we take average this sequence and we compare this average with the target value.

# build a simple moving average model

def compute\_moving\_average(data):

pred=[]

for i in data:

avg=np.sum(i)/len(i)

pred.append(avg)

return np.array(pred)

# reshape the data

x\_reshaped = x\_val.reshape(-1,168)

# get predictions

y\_pred = compute\_moving\_average(x\_reshaped)

So this function computes the average of the input sequences and over here we are extracting the predictions.

# evaluate the performance of model on the validation data

mse = np.sum ( (y\_val - y\_pred) \*\*2 ) / (len(y\_val))

print("Mean square of error:- ",mse)

Now we calculate the mean squared error for this model. On the same validation data, we get a score of 0.554 which is way higher than this previous error.

https://editor.analyticsvidhya.com/uploads/47327Screenshot%20from%202021-09-08%2013-47-57.png

So our LSTM based model has done exceptionally well as compared to the baseline model,

Web Traffic Forecasting

Now moving on to forecasting. These are the steps that we will follow:-

1. first, initialize an array with weeks data,
2. Predict the next hour traffic volume
3. Append the predicted value at the end of the array ‘data
4. Skip the first element of the array ‘data’
5. Repeating steps, from the second step till the fourth step for the specified number of iterations.

This is how we can forecasting for any number of hours in future. This function forecast performs the steps just discuss and it returns the predicted sequence of numbers.

def forecast(x\_val, no\_of\_pred, ind):

predictions=[]

#intialize the array with a weeks data

temp=x\_val[ind]

for i in range(no\_of\_pred):

#predict for the next hour

pred=model.predict(temp.reshape(1,-1,1))[0][0]

#append the prediction as the last element of array

temp = np.insert(temp,len(temp),pred)

predictions.append(pred)

#ignore the first element of array

temp = temp[1:]

return predictions

It’s time to forecast the traffic for the next 24 hours based on the previous week data.

no\_of\_pred =24

ind=72

y\_pred= forecast(x\_val,no\_of\_pred,ind)

y\_true = y\_val[ind:ind+(no\_of\_pred)]

# Lets convert back the normalized values to the original dimensional space

y\_true= y\_scaler.inverse\_transform(y\_true)

y\_pred= y\_scaler.inverse\_transform(y\_pred)

Now let’s look at the plot of real vs forecast values.

def plot(y\_true,y\_pred):

ar = np.arange(len(y\_true))

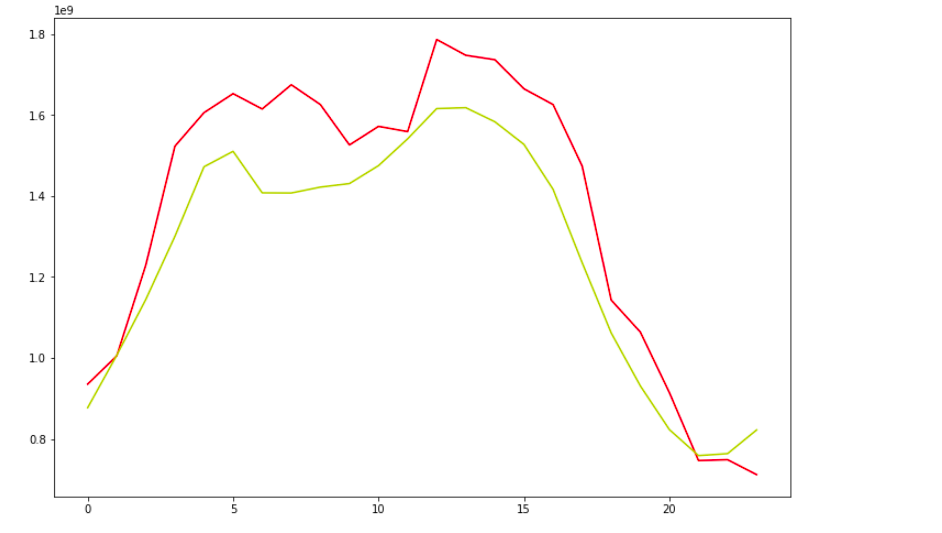
plt.figure(figsize=(22,10))

plt.plot(ar, y\_true,'r')

plt.plot(ar, y\_pred,'y')

plt.show()

plot(y\_true,y\_pred)



It looks great. Our model has been successful in capturing the trend.

This red curve is the actual value and this yellow curve are the predicted values both are pretty much close to each other.

Similarly, we can use a CNN based model in place of LSTM to perform the same task. Let’s see how it is done.

CNN Model with Forecasting

Now here we are using Conv1D layers in the model architecture. And these layers are followed by a flattening layer. This layer converts the input to a One Dimensional array, which is then passed on to this set of dense layers.

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import \*

from tensorflow.keras.callbacks import \*

model= Sequential()

model.add(Conv1D(64, 3, padding='same', activation='relu',input\_shape=(num,1)))

model.add(Conv1D(32, 5, padding='same', activation='relu',input\_shape=(num,1)))

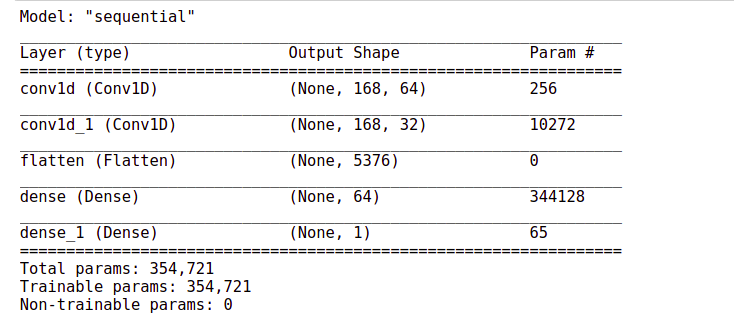
model.add(Flatten())

model.add(Dense(64,activation='relu'))

model.add(Dense(1,activation='linear'))

model.summary()

The output will be:-



# Define the optimizer and loss:

model.compile(loss='mse',optimizer='adam')

# Define the callback to save the best model during the training

mc = ModelCheckpoint('best\_model.hdf5', monitor='val\_loss', verbose=1,

save\_best\_only=True, mode='min')

# Train the model for 30 epochs with batch size of 32:

history=model.fit(x\_tr, y\_tr ,epochs=30, batch\_size=32, validation\_data=(x\_val,y\_val),

callbacks=[mc])

Here again, you can see that the training process is processing fast. It is hardly taking one second to finish and pip off.

Epoch 1/30

133/133 [==============================] - 4s 23ms/step - loss: 0.0899 - val\_loss: 0.0482

Epoch 00001: val\_loss improved from inf to 0.04819, saving model to best\_model.hdf5

Epoch 2/30

133/133 [==============================] - 3s 22ms/step - loss: 0.0249 - val\_loss: 0.0209

Epoch 00002: val\_loss improved from 0.04819 to 0.02086, saving model to best\_model.hdf5

Epoch 3/30

133/133 [==============================] - 3s 22ms/step - loss: 0.0186 - val\_loss: 0.0190

Epoch 00003: val\_loss improved from 0.02086 to 0.01899, saving model to best\_model.hdf5

Epoch 4/30

133/133 [==============================] - 3s 23ms/step - loss: 0.0157 - val\_loss: 0.0161

Epoch 00004: val\_loss improved from 0.01899 to 0.01610, saving model to best\_model.hdf5

Epoch 5/30

133/133 [==============================] - 3s 20ms/step - loss: 0.0132 - val\_loss: 0.0149

Epoch 00005: val\_loss improved from 0.01610 to 0.01490, saving model to best\_model.hdf5

Epoch 6/30

133/133 [==============================] - 3s 21ms/step - loss: 0.0134 - val\_loss: 0.0158

Epoch 00006: val\_loss did not improve from 0.01490

Epoch 7/30

133/133 [==============================] - 3s 22ms/step - loss: 0.0122 - val\_loss: 0.0138

Epoch 00007: val\_loss improved from 0.01490 to 0.01385, saving model to best\_model.hdf5

Epoch 8/30

133/133 [==============================] - 3s 23ms/step - loss: 0.0112 - val\_loss: 0.0146

Epoch 00008: val\_loss did not improve from 0.01385

Epoch 9/30

133/133 [==============================] - 3s 21ms/step - loss: 0.0106 - val\_loss: 0.0177

Epoch 00009: val\_loss did not improve from 0.01385

Epoch 10/30

133/133 [==============================] - 3s 21ms/step - loss: 0.0092 - val\_loss: 0.0131

Epoch 00010: val\_loss improved from 0.01385 to 0.01314, saving model to best\_model.hdf5

Epoch 11/30

133/133 [==============================] - 3s 23ms/step - loss: 0.0089 - val\_loss: 0.0167

Epoch 00011: val\_loss did not improve from 0.01314

Epoch 12/30

133/133 [==============================] - 3s 22ms/step - loss: 0.0083 - val\_loss: 0.0148

Epoch 00012: val\_loss did not improve from 0.01314

Epoch 13/30

133/133 [==============================] - 3s 21ms/step - loss: 0.0078 - val\_loss: 0.0154

Epoch 00013: val\_loss did not improve from 0.01314

Epoch 14/30

133/133 [==============================] - 3s 21ms/step - loss: 0.0073 - val\_loss: 0.0142

Epoch 00014: val\_loss did not improve from 0.01314

Epoch 15/30

133/133 [==============================] - 3s 21ms/step - loss: 0.0064 - val\_loss: 0.0144

Epoch 00015: val\_loss did not improve from 0.01314

Epoch 16/30

133/133 [==============================] - 3s 22ms/step - loss: 0.0057 - val\_loss: 0.0153

Epoch 00016: val\_loss did not improve from 0.01314

Epoch 17/30

133/133 [==============================] - 3s 22ms/step - loss: 0.0050 - val\_loss: 0.0158

Epoch 00017: val\_loss did not improve from 0.01314

Epoch 18/30

133/133 [==============================] - 3s 22ms/step - loss: 0.0051 - val\_loss: 0.0155

Epoch 00018: val\_loss did not improve from 0.01314

Epoch 19/30

133/133 [==============================] - 3s 20ms/step - loss: 0.0044 - val\_loss: 0.0153

Epoch 00019: val\_loss did not improve from 0.01314

Epoch 20/30

133/133 [==============================] - 3s 21ms/step - loss: 0.0040 - val\_loss: 0.0144

Epoch 00020: val\_loss did not improve from 0.01314

Epoch 21/30

133/133 [==============================] - 3s 22ms/step - loss: 0.0033 - val\_loss: 0.0147

Epoch 00021: val\_loss did not improve from 0.01314

Epoch 22/30

133/133 [==============================] - 3s 20ms/step - loss: 0.0031 - val\_loss: 0.0153

Epoch 00022: val\_loss did not improve from 0.01314

Epoch 23/30

133/133 [==============================] - 3s 22ms/step - loss: 0.0030 - val\_loss: 0.0161

Epoch 00023: val\_loss did not improve from 0.01314

Epoch 24/30

133/133 [==============================] - 3s 22ms/step - loss: 0.0026 - val\_loss: 0.0150

Epoch 00024: val\_loss did not improve from 0.01314

Epoch 25/30

133/133 [==============================] - 3s 23ms/step - loss: 0.0025 - val\_loss: 0.0161

Epoch 00025: val\_loss did not improve from 0.01314

Epoch 26/30

133/133 [==============================] - 3s 21ms/step - loss: 0.0026 - val\_loss: 0.0151

Epoch 00026: val\_loss did not improve from 0.01314

Epoch 27/30

133/133 [==============================] - 3s 20ms/step - loss: 0.0026 - val\_loss: 0.0151

Epoch 00027: val\_loss did not improve from 0.01314

Epoch 28/30

133/133 [==============================] - 3s 21ms/step - loss: 0.0023 - val\_loss: 0.0160

Epoch 00028: val\_loss did not improve from 0.01314

Epoch 29/30

133/133 [==============================] - 3s 22ms/step - loss: 0.0022 - val\_loss: 0.0145

Epoch 00029: val\_loss did not improve from 0.01314

Epoch 30/30

133/133 [==============================] - 3s 23ms/step - loss: 0.0020 - val\_loss: 0.0161

Epoch 00030: val\_loss did not improve from 0.01314

Load the weights of the best model prior to predictions.

model.load\_weights('best\_model.hdf5')

Let’s check out the performance of this model on the validation set. Evaluate the performance of a model on the validation data.

mse = model.evaluate(x\_val,y\_val)

print("Mean Square Error:",mse)

mse | Web Traffic Forecasting

The mean squared error has improved the width from 0.015 to 0.013.

Comparison with the baseline model

Now let’s compare this performance with the baseline model.

#build a simple model

def compute\_moving\_average(data):

pred=[]

for i in data:

avg=np.sum(i)/len(i)

pred.append(avg)

return np.array(pred)

x\_reshaped = x\_val.reshape(-1,168)

y\_pred = compute\_moving\_average(x\_reshaped)

mse = np.sum ( (y\_val - y\_pred) \*\*2 ) / (len(y\_val))

print("Mean Square Error:",mse)

MSE

The baseline score was 0.55. So our CNN based model is also much better than the baseline model.

Forecasting

Now let’s see how well it forecast the web traffic for a period of 24 hours.

def forecast(x\_val, no\_of\_pred, ind):

predictions=[]

#intialize the array with previous weeks data

temp=x\_val[ind]

for i in range(no\_of\_pred):

#predict for the next hour

pred=model.predict(temp.reshape(1,-1,1))[0][0]

#append the prediction as the last element of array

temp = np.insert(temp,len(temp),pred)

predictions.append(pred)

#ignore the first element of array

temp = temp[1:]

return predictions

It’s time to forecast the traffic for the next 24 hours based on the previous week data.

no\_of\_pred =24

ind=72

y\_pred= forecast(x\_val,no\_of\_pred,ind)

y\_true = y\_val[ind:ind+(no\_of\_pred)]

Let’s convert back the normalized values to the original dimensional space.

y\_true= y\_scaler.inverse\_transform(y\_true)

y\_pred= y\_scaler.inverse\_transform(y\_pred)

def plot(y\_true,y\_pred):

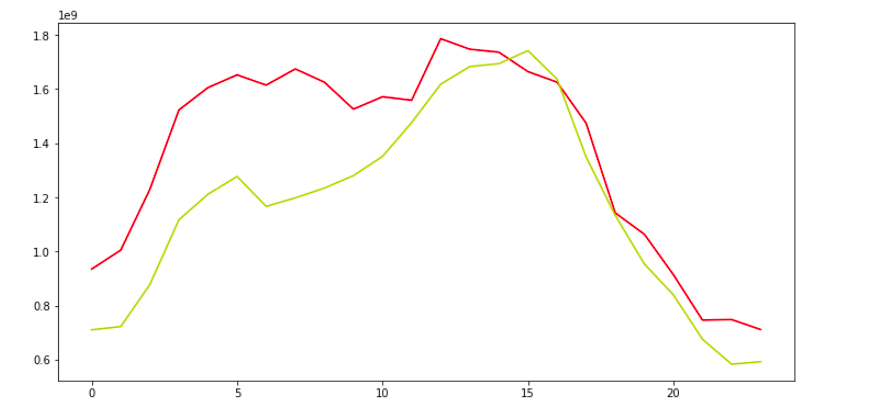
ar = np.arange(len(y\_true))

plt.figure(figsize=(22,10))

plt.plot(ar, y\_true,'r')

plt.plot(ar, y\_pred,'y')

plt.show()



The forecasted values are almost close to the actual values. Well, the performance seems pretty much similar to that of the LSTM in the based model.

If You can recall in the Auto-tagging system project we saw that the CNN-based model outperform the LSTM based model by a huge margin.

But here in this case both the models have performed mode or less the same.

Conclusion

However, the CNN-based model still has the advantage of speed. Now, this is not the end of the row. We can further improve this model by taking measures like, we can make the time series data stationary and then use it for making sequences and then later use it in the model input sequences. To learn more about stationarity charity and other time series-related concepts, you can check out this [link](https://www.analyticsvidhya.com/blog/2016/02/time-series-forecasting-codes-python/)**.**

Apart from that, we can also try a different number of hidden units r different numbers of hidden layers. To see how our model performs under different settings. In addition, we can also try to change the learning rate. Even that might help in improving the model.