Time Series Forecasting

Ananya Sharma
Department of Computer Science, University of Southern California,
Los Angeles, California 90089, USA

Abstract:

In this report I first describe the kinds of data we have, how to ensure it's all in compliance and can gel along with each other. Then I start by testing a simple Recurrent Neural Network (RNN) model that will have only one parameter 'temperature' along with its date and time, to predict the temperature for the next 24-120 hours (1-5 days). Then by measuring its performance metrics namely - Precision, recall, FScore, Accuracy. Then, I compare this model against 3 other models, one which again uses the same RNN but with all other parameters namely humidity, pressure, wind direction,etc. The other 2 models again employ these parameters but have different architectures. Thus, finally deciding which model is better by fine tuning the parameters and eventually testing a new dataset to predict the temperature. Further, data normalization is also done and hence it's found which neural network works better and if inclusion of other parameters actually helps the model.

Introduction :

For this task, we are given the following information :
Datetime, temperature, humidity, pressure, weather, wind direction and wind speed.

By training my model, these datasets just need to be run on it to get predictions.

I started by importing the libraries required and the data (1 csv files storing the data)

```
In [1]: #import modules and load the csv files given

        import pandas as pd
        import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
        import seaborn as sns
        import string
        import nltk
        import tensorflow as tf
        import re

        d = pd.read_csv('/Users/ananyasharma/Downloads/usc-dsci552-section-32415d-spring-2021-ps6/ps6_trainvalid.csv')
```

FIG 1 : How to load data

Then, to know what kind of data we have

In [4]: d

Out[4]:

	datetime	temperature	humidity	pressure	weather	wind_direction	wind_speed
1	2012-10-01 13:00:00	291.870000	88.0	1013.0	mist	0.0	0.0
2	2012-10-01 14:00:00	291.868186	88.0	1013.0	sky is clear	0.0	0.0
3	2012-10-01 15:00:00	291.862844	88.0	1013.0	sky is clear	0.0	0.0
4	2012-10-01 16:00:00	291.857503	88.0	1013.0	sky is clear	0.0	0.0
5	2012-10-01 17:00:00	291.852162	88.0	1013.0	sky is clear	0.0	0.0
...	...	...	...	...	...	...	...
click to scroll output; double click to hide		295.440000	17.0	1017.0	sky is clear	345.0	1.0
45009	2017-11-19 21:00:00	296.020000	16.0	1016.0	sky is clear	345.0	1.0
45010	2017-11-19 22:00:00	296.510000	17.0	1015.0	sky is clear	345.0	1.0
45011	2017-11-19 23:00:00	297.090000	17.0	1014.0	sky is clear	324.0	0.0
45012	2017-11-20 00:00:00	296.690000	25.0	1014.0	sky is clear	0.0	2.0

45012 rows × 7 columns

FIG 2 : View the data

Before proceeding further, let's explore the data given.
I created graphs of the data as per the columns to see the trend of data in the same.

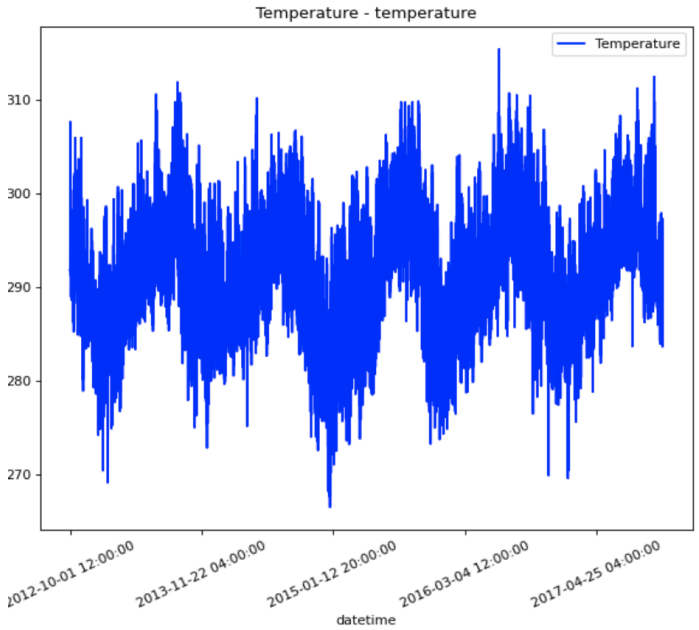


FIG 3 : Temperature versus datetime

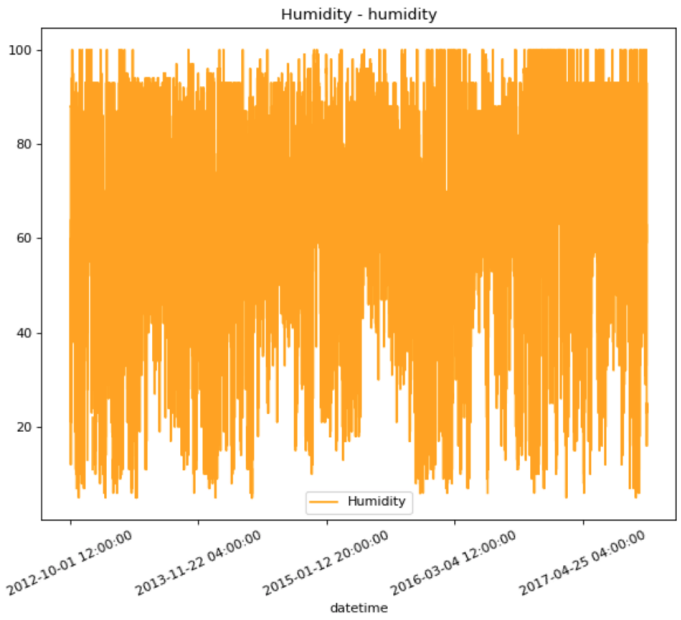


FIG 4 : Temperature versus humidity

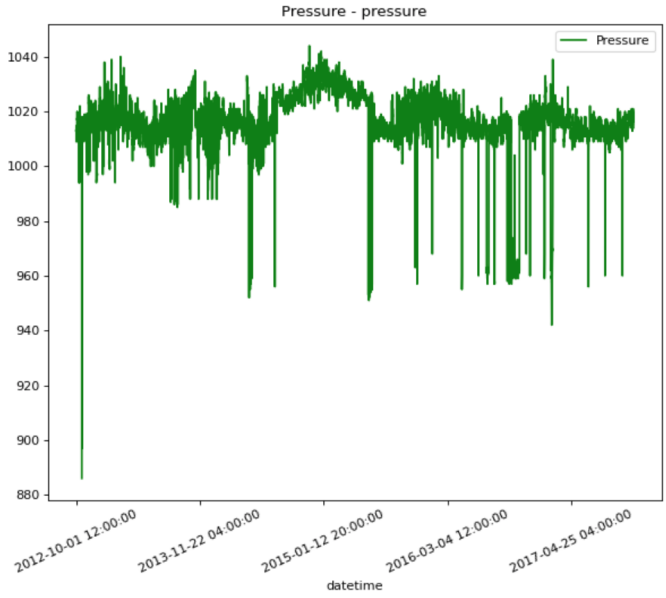


FIG 5 : Temperature versus pressure

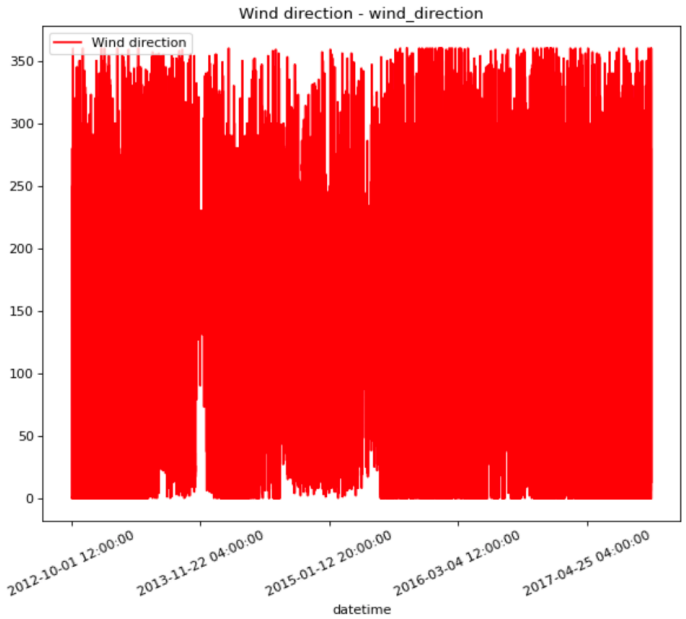


FIG 6 : Temperature versus wind direction

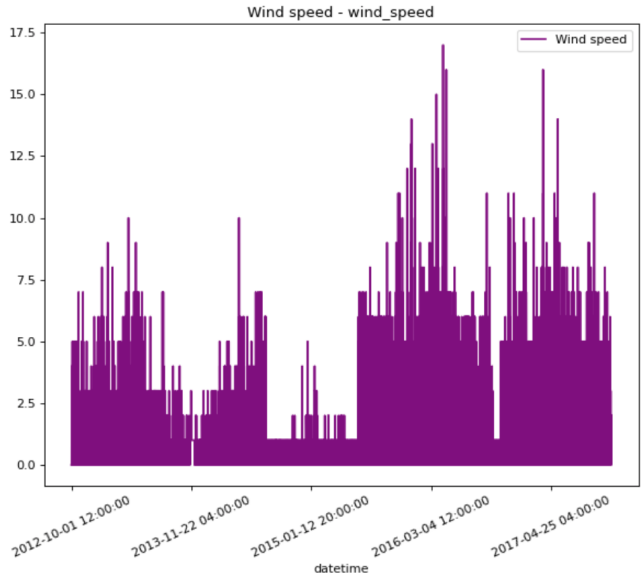


FIG 7 : Temperature versus wind speed

However, the plots above are a bit noisy, because they contain all the daily data. If we look carefully into the data points, we could see that there is only a minor change between the current date and the next date for all columns.
Smoothing is a technique applied to time series to remove the fine-grained variation between time steps. Smoothing is used to remove noise and better expose the signal of the underlying causal processes. Moving averages are a simple and common type of smoothing used in time series analysis and time series forecasting.
The rolling() function will automatically group observations into a window. We can specify the window size, and by default, a trailing window is created.
Below are 2 examples of transforming the temperature dataset into a moving average with a window size of 30 and 120 days, chosen arbitrarily.

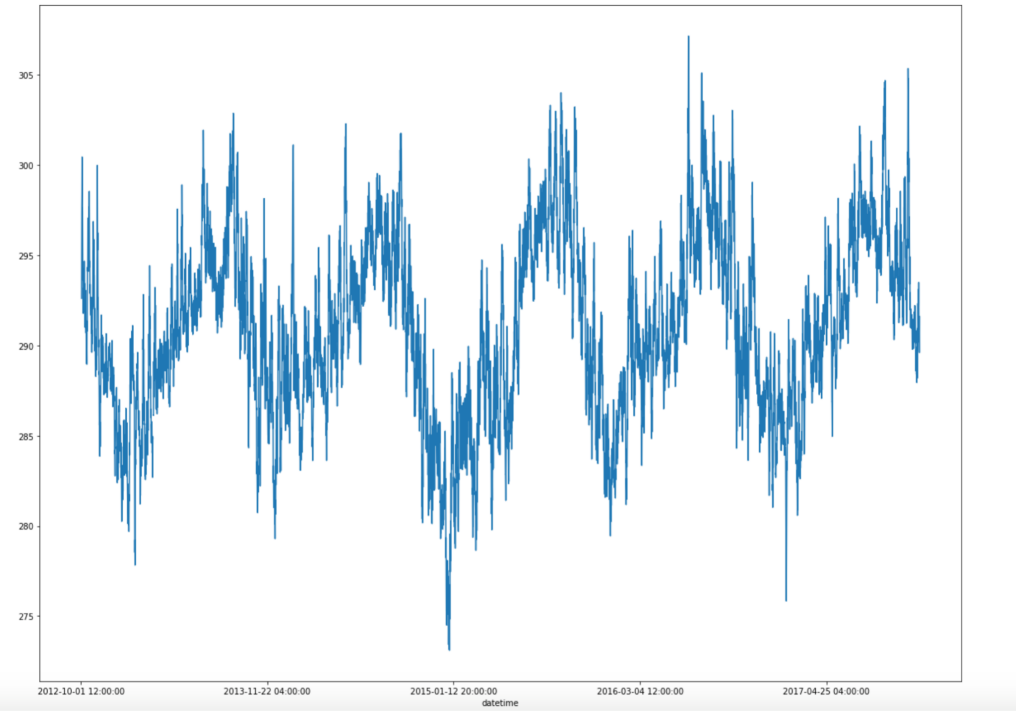


FIG 8 : Temperature versus datetime with 30 days data rolled together

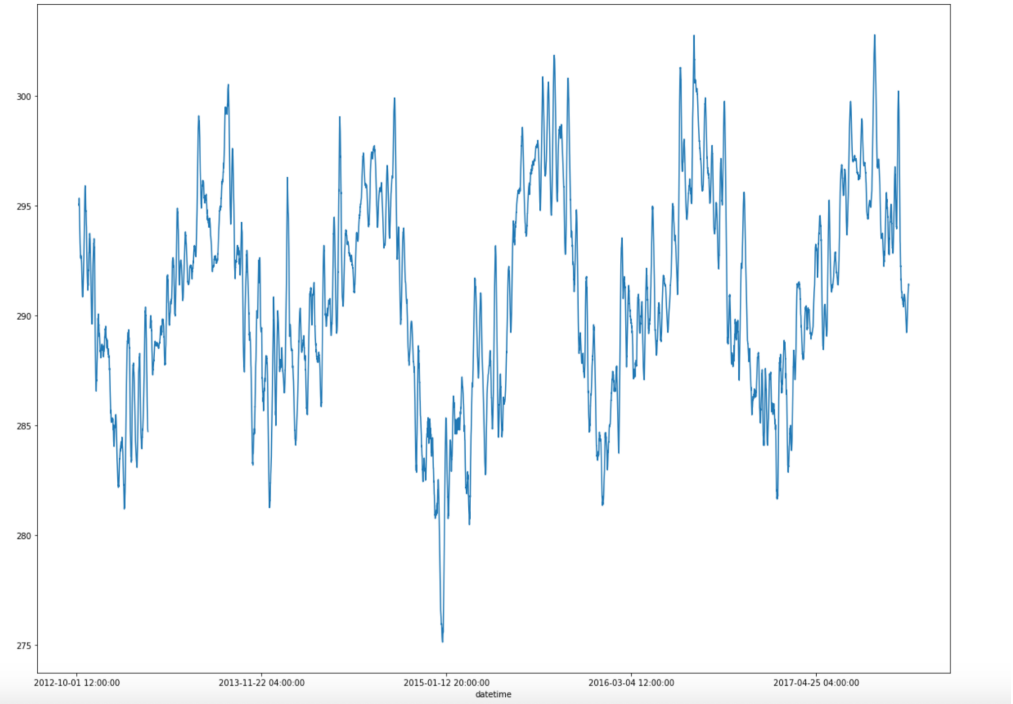


FIG 9 : Temperature versus datetime with 120 days data rolled together

We see that although there isn't a significant difference between Figures 8 and 9, there is however a less crowded graph in Fig 9. Further, as compared to Figure 3, there is a significant difference in both the figures. Some distinguishable patterns appear when we plot the data. The time-series has a seasonality pattern, such as temperatures are always low at the beginning of the year and high in the middle of the year.

Next, the heat map shows the correlation between different features.

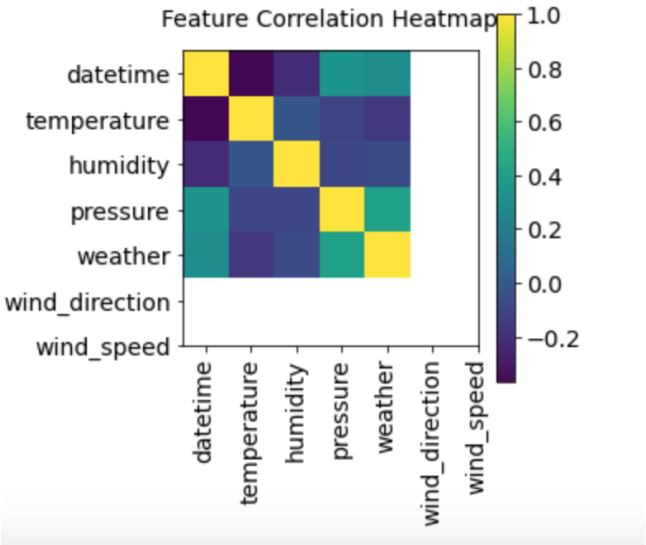


FIG 10 : Feature Correlation Heatmap

Now, let's have a model with only the temperature parameter (sans the pressure, humidity,etc given) and see how well it predicts.

The data now looks like :

```
In [23]: df
```

Out[23]:

datetime	temperature
2012-10-01 13:00:00	291.870000
2012-10-01 14:00:00	291.868186
2012-10-01 15:00:00	291.862844
2012-10-01 16:00:00	291.857503
2012-10-01 17:00:00	291.852162
...	...
2017-11-19 20:00:00	295.440000
2017-11-19 21:00:00	296.020000
2017-11-19 22:00:00	296.510000
2017-11-19 23:00:00	297.090000
2017-11-20 00:00:00	296.690000

45010 rows × 1 columns

```
In [30]: train_df,test_df=df[1:36008],df[36008:]
```

FIG 11 : Only temperature data

Further, the data has been divided into 80% training data and 20% testing data.

Now, the training data looks like :

```
In [31]: train_df
```

Out[31]:

datetime	temperature
2012-10-01 14:00:00	291.868186
2012-10-01 15:00:00	291.862844
2012-10-01 16:00:00	291.857503
2012-10-01 17:00:00	291.852162
2012-10-01 18:00:00	291.846821
...	...
2016-11-09 18:00:00	302.440000
2016-11-09 19:00:00	304.740000
2016-11-09 20:00:00	306.570000
2016-11-09 21:00:00	306.880000
2016-11-09 22:00:00	306.480000

36007 rows × 1 columns

FIG 12 : Training dataset

The data is now normalized so that there isn't too much of a variance in between the data. It is given the range 0 to 1.

```
In [35]: #Feature Scaling
         from sklearn.preprocessing import MinMaxScaler
         sc = MinMaxScaler(feature_range=(0,1))
         train_df_scaled = sc.fit_transform(train_df)

In [36]: train_df_scaled

Out[36]: array([[0.51799915],
                [0.51789007],
                [0.51778099],
                ...,
                [0.81824247],
                [0.82457335],
                [0.81640447]])
```

FIG 13: Normalized training data

The code for the same is :

In [12]:
```
from sklearn.preprocessing import MinMaxScaler


dataset = d.temperature.values #numpy.ndarray
dataset = dataset.astype('float32')
dataset = np.reshape(dataset, (-1, 1))
scaler = MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(dataset)
train_size = int(len(dataset) * 0.80)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:], dataset[train_size:len(dataset),:]

def create_dataset(dataset, look_back=1):
    X, Y = [], []
    for i in range(len(dataset)-look_back-1):
        a = dataset[i:(i+look_back), 0]
        X.append(a)
        Y.append(dataset[i + look_back, 0])
    return np.array(X), np.array(Y)

# reshape into X=t and Y=t+1
look_back = 30
X_train, Y_train = create_dataset(train, look_back)
X_test, Y_test = create_dataset(test, look_back)

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

FIG 14 : Preprocessing the data before training the model

The following data pre-processing and feature engineering is done before constructing the model.

- Create the dataset, ensure all data is float.
- Normalize the features.
- Split into training and test sets.
- Convert an array of values into a dataset matrix.
- Reshape into X=t and Y=t+1.
- Reshape input to be 3D (num_samples, num_timesteps, num_features).

The model is as follows :

```
In [15]: from keras.models import Sequential
         from keras.layers import LSTM,Dense ,Dropout,Bidirectional
         from keras.callbacks import ModelCheckpoint, EarlyStopping

         model = Sequential()
         model.add(LSTM(100, input_shape=(X_train.shape[1], X_train.shape[2])))
         model.add(Dropout(0.2))
         model.add(Dense(1))
         model.compile(loss='mean_squared_error', optimizer='adam')

         history = model.fit(X_train, Y_train, epochs=20, batch_size=70, validation_data=(X_test, Y_test),
                             callbacks=[EarlyStopping(monitor='val_loss', patience=10)], verbose=1, shuffle=False)

         model.summary()
```

```
Epoch 1/20
511/511 [==============================] - 4s 4ms/step - loss: 0.0201 - val_loss: 0.0014
Epoch 2/20
511/511 [==============================] - 2s 4ms/step - loss: 0.0023 - val_loss: 0.0011
Epoch 3/20
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 100)               52400
_________________________________________________________________
dropout_1 (Dropout)          (None, 100)               0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 101
=================================================================
Total params: 52,501
Trainable params: 52,501
Non-trainable params: 0
_________________________________________________________________
```

FIG 15 : The RNN model with 3 layers

- The LSTM with 100 neurons in the first hidden layer and 1 neuron in the output layer for predicting temperature. The input shape will be 1 time step with 30 features.
- Dropout 20%.
- Use the Mean Squared Error loss function and the efficient Adam version of stochastic gradient descent.
- The model will be fit for 20 training epochs with a batch size of 70.

Here, the type of model used is sequential. A sequential model helps stack layers one on top of the other.

Layers *early* in the network architecture (i.e., closer to the actual input image) learn *fewer* convolutional filters while layers *deeper* in the network (i.e., closer to the output predictions) will learn *more* filters.

A common LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. The cell remembers values over arbitrary time intervals and the three gates regulate the flow of information into and out of the cell. The two technical problems overcome by LSTMs are vanishing gradients and exploding gradients, both related to how the network is trained.

The Long Short Term Memory architecture was motivated by an analysis of error flow in existing RNNs which found that long time lags were inaccessible to existing architectures, because back propagated error either blows up or decays exponentially.

An LSTM layer consists of a set of recurrently connected blocks, known as memory blocks. These blocks can be thought of as a differentiable version of the memory chips in a digital computer. Each one contains one or more recurrently connected memory cells and three multiplicative units – the input, output and forget gates – that provide continuous analogues of write, read and reset operations for the cells. The network can only interact with the cells via the gates. Learning rate and network size are the most crucial tunable LSTM hyperparameters. The hyperparameters can be tuned independently. In particular, the learning rate can be calibrated first using a fairly small network, thus saving a lot of time.

The forget gate decides what information should be thrown away or kept. Information from the previous hidden state and information from the current input is passed through the sigmoid function. Values come out between 0 and 1. The closer to 0 means to forget, and the closer to 1 means to keep.

Dropout can be applied between each time step.It randomly sets input units 0 with a frequency of rate at each step during training time, which helps prevent overfitting. In Keras, all recurrent layers and all cells have a dropout hyperparameter and a recurrent_dropout hyperparameter: the former defines the dropout rate to apply to the inputs (at each time step), and the latter defines the dropout rate for the hidden states (also at each time step). So there is no need to create a custom cell to apply dropout at each time step in an RNN, which makes this step less tedious.

Dense implements the operation:
output = activation(dot product (input, kernel) + bias)
where activation is the element-wise activation function passed as the activation argument, kernel is a weights matrix created by the layer, and bias is a bias vector created by the layer.

Let’s check the Mean Absolute Error and Root Mean Squared Error for both the training and the test set:

```
Train Mean Absolute Error: 1.0812188071860815
Train Root Mean Squared Error: 1.445130079502583
Test Mean Absolute Error: 0.86648091993852
Test Root Mean Squared Error: 1.1126274020013491
```

FIG 16 : MAE and RMSE for both Training and Test Data

Further, let’s also test the model accuracy on training and testing data both. This shows that for the test data the values differ by 0.866, which is an acceptable error rate.
The loss for both the sets is :

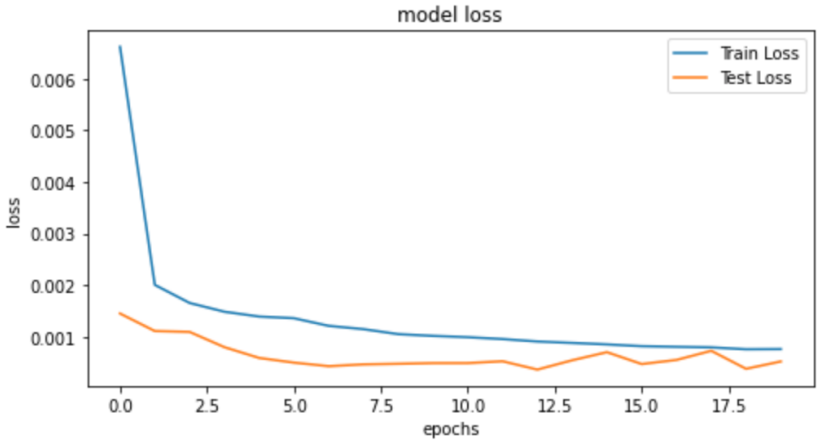


FIG 17 : Training and Test Loss for Model 1

As can be seen, the loss for both the sets is almost the same and decreases with increase in the number of epochs.

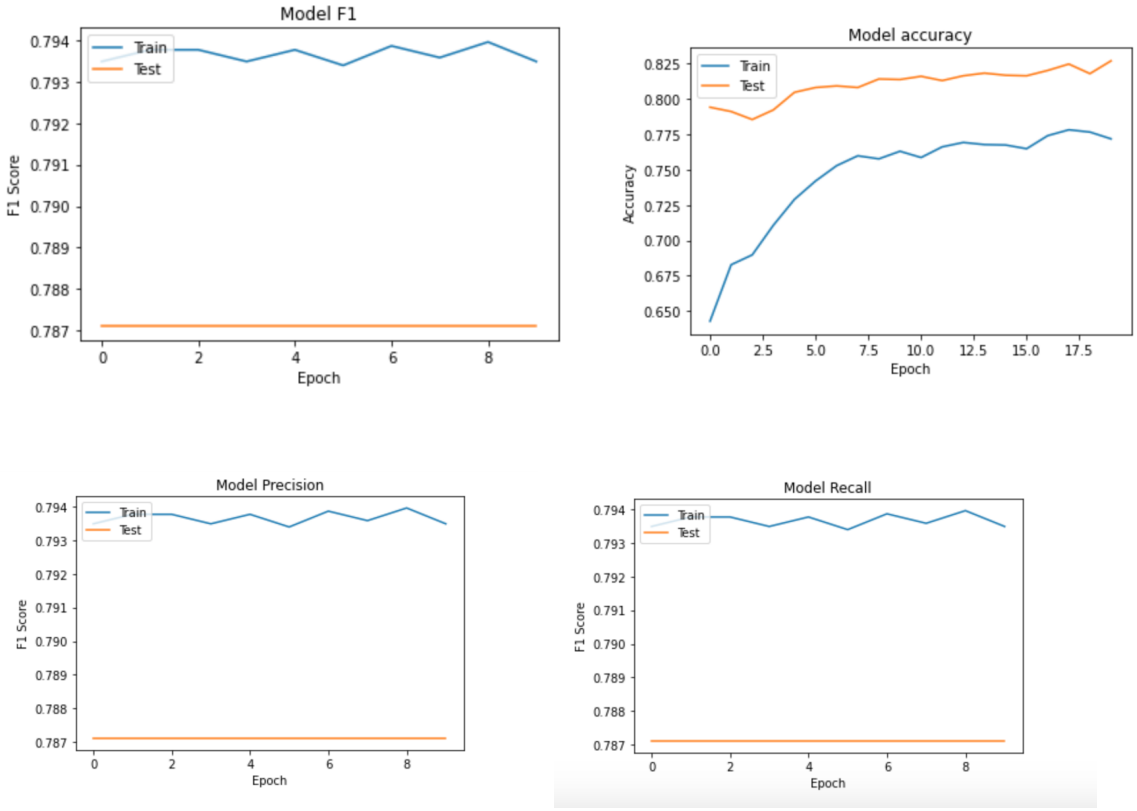


FIG 18 : Model 1 parametrics

As can be seen, the metrics look good enough.

Accuracy is one metric for evaluating classification models. It is the fraction of predictions our model got right. Accuracy = Number of correct predictions/ Total number of predictions.

The Loss Function is one of the important components of Neural Networks. Loss is a prediction error of the Neural Net. And the method to calculate the loss is called LossFunction. Loss is used to calculate the gradients. And gradients are used to update the weights of the Neural Net.

Precision is the number of true positives (i.e. the number of items correctly labelled as belonging to the positive class) divided by the total number of elements labelled as belonging to the positive class (i.e. the sum of true positives and false positives). It is the degree of refinement with which an operation is performed.

When the recall is high, it means the model can classify all the positive samples correctly as Positive. Thus, the model can be trusted in its ability to detect positive samples.

The F1 score can be interpreted as a weighted average of the precision and recall, where an F1 score reaches its best value at 1 and worst at 0.

We'll see the prediction for whichever model is the best.

The second model has the same architecture as model 1 but takes all parameters available except 'Weather'.
This is how the data looks:

```
In [4]: d = d.dropna(axis = 0, how ='any')
        d
```

Out[4]:

datetime	temperature	humidity	pressure	wind_direction	wind_speed
2012-10-01 13:00:00	291.870000	88.0	1013.0	0.0	0.0
2012-10-01 14:00:00	291.868186	88.0	1013.0	0.0	0.0
2012-10-01 15:00:00	291.862844	88.0	1013.0	0.0	0.0
2012-10-01 16:00:00	291.857503	88.0	1013.0	0.0	0.0
2012-10-01 17:00:00	291.852162	88.0	1013.0	0.0	0.0
...	...	...	...	...	...
2017-11-19 20:00:00	295.440000	17.0	1017.0	345.0	1.0
2017-11-19 21:00:00	296.020000	16.0	1016.0	345.0	1.0
2017-11-19 22:00:00	296.510000	17.0	1015.0	345.0	1.0
2017-11-19 23:00:00	297.090000	17.0	1014.0	324.0	0.0
2017-11-20 00:00:00	296.690000	25.0	1014.0	0.0	2.0

44671 rows × 5 columns

FIG 19 : Data for Model 2

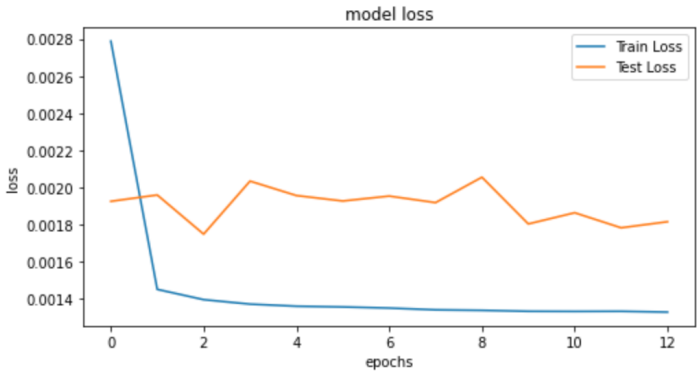


FIG 20 : Loss for Model 2

Here we see that the Test loss is greater than the Training Loss which shows that the model is overfitting with such an architecture and hence I do not consider it further for any predictions.

For the 3rd model, data is split in the following manner:

In [26]:
```
def split_series(series, n_past, n_future):
  #
  # n_past ==> no of past observations
  #
  # n_future ==> no of future observations
  #
  X, y = list(), list()
  for window_start in range(len(series)):
    past_end = window_start + n_past
    future_end = past_end + n_future
    if future_end > len(series):
      break
    # slicing the past and future parts of the window
    past, future = series[window_start:past_end, :], series[past_end:future_e
    X.append(past)
    y.append(future)
  return np.array(X), np.array(y)
```

In [27]:
```
n_past = 10
n_future = 5
n_features = 5
```

In [28]:
```
X_train, y_train = split_series(train.values,n_past, n_future)
X_train = X_train.reshape((X_train.shape[0], X_train.shape[1],n_features))
y_train = y_train.reshape((y_train.shape[0], y_train.shape[1], n_features))
X_test, y_test = split_series(test.values,n_past, n_future)
X_test = X_test.reshape((X_test.shape[0], X_test.shape[1],n_features))
y_test = y_test.reshape((y_test.shape[0], y_test.shape[1], n_features))
```

FIG 21 : Data splitting for Model 3

It has the following architecture :

In [33]:
```
# E1D1
# n_features ==> no of features at each timestep in the data.
#
encoder_inputs = tf.keras.layers.Input(shape=(n_past, n_features))
encoder_l1 = tf.keras.layers.LSTM(100, return_state=True)
encoder_outputs1 = encoder_l1(encoder_inputs)

encoder_states1 = encoder_outputs1[1:]

#
decoder_inputs = tf.keras.layers.RepeatVector(n_future)(encoder_outputs1[0])

#
decoder_l1 = tf.keras.layers.LSTM(100, return_sequences=True)(decoder_inputs,initial_state = encoder_states1)
decoder_outputs1 = tf.keras.layers.TimeDistributed(tf.keras.layers.Dense(n_features))(decoder_l1)

#
model_e1d1 = tf.keras.models.Model(encoder_inputs,decoder_outputs1)

#
model_e1d1.summary()
```

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 10, 5)]           0
_________________________________________________________________
lstm (LSTM)                  [(None, 100), (None,      42400
_________________________________________________________________
repeat_vector (RepeatVector) (None, 5, 100)            0
_________________________________________________________________
lstm_1 (LSTM)                (None, 5, 100)            80400
_________________________________________________________________
time_distributed (TimeDistribut (None, 5, 5)           505
=================================================================
Total params: 123,305
Trainable params: 123,305
Non-trainable params: 0
```

FIG 22 : Model 3 with 4 layers

Similarly, architecture for model 4:

In [34]:
```
# E2D2
# n_features ==> no of features at each timestep in the data.
#
encoder_inputs = tf.keras.layers.Input(shape=(n_past, n_features))
encoder_l1 = tf.keras.layers.LSTM(100,return_sequences = True, return_state=True)
encoder_outputs1 = encoder_l1(encoder_inputs)
encoder_states1 = encoder_outputs1[1:]
encoder_l2 = tf.keras.layers.LSTM(100, return_state=True)
encoder_outputs2 = encoder_l2(encoder_outputs1[0])
encoder_states2 = encoder_outputs2[1:]
#
decoder_inputs = tf.keras.layers.RepeatVector(n_future)(encoder_outputs2[0])
#
decoder_l1 = tf.keras.layers.LSTM(100, return_sequences=True)(decoder_inputs,initial_state = encoder_states1)
decoder_l2 = tf.keras.layers.LSTM(100, return_sequences=True)(decoder_l1,initial_state = encoder_states2)
decoder_outputs2 = tf.keras.layers.TimeDistributed(tf.keras.layers.Dense(n_features))(decoder_l2)
#
model_e2d2 = tf.keras.models.Model(encoder_inputs,decoder_outputs2)
#
model_e2d2.summary()
```

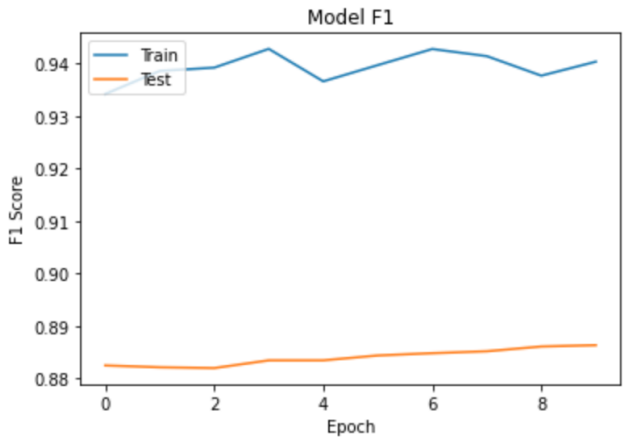
```
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_2 (InputLayer)         [(None, 10, 5)]           0         
_________________________________________________________________
lstm_2 (LSTM)                [(None, 10, 100), (N 42400     
_________________________________________________________________
lstm_3 (LSTM)                [(None, 100), (None, 80400     
_________________________________________________________________
repeat_vector_1 (RepeatVector)  (None, 5, 100)       0         
_________________________________________________________________
lstm_4 (LSTM)                (None, 5, 100)       80400     
_________________________________________________________________
lstm_5 (LSTM)                (None, 5, 100)       80400     
_________________________________________________________________
time_distributed_1 (TimeDistrib (None, 5, 5)         505       
=================================================================
Total params: 284,105
Trainable params: 284,105
Non-trainable params: 0
_________________________________________________________________
```

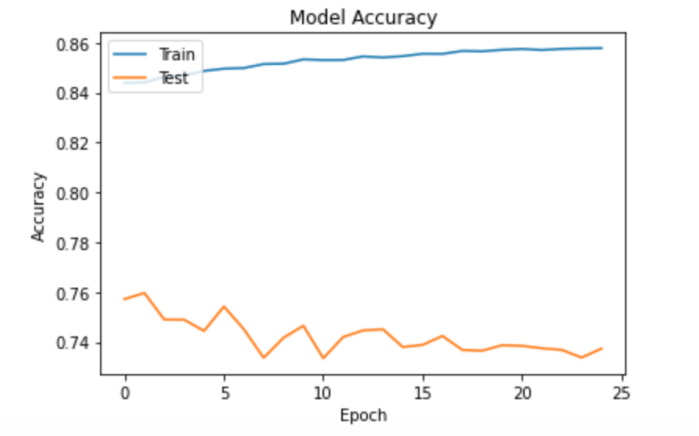
FIG 23 : Model 4 with 7 layers

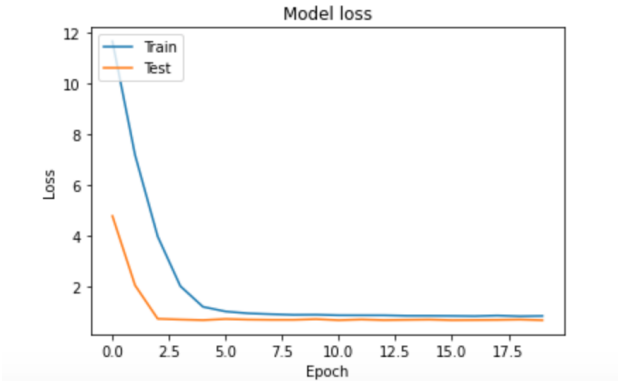
In [60]:
```
from keras import backend as K

def recall_m(y_true, y_pred):
    true_positives = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    possible_positives = K.sum(K.round(K.clip(y_true, 0, 1)))
    recall = true_positives / (possible_positives + K.epsilon())
    return recall

def precision_m(y_true, y_pred):
    true_positives = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    predicted_positives = K.sum(K.round(K.clip(y_pred, 0, 1)))
    precision = true_positives / (predicted_positives + K.epsilon())
    return precision

def f1_m(y_true, y_pred):
    precision = precision_m(y_true, y_pred)
    recall = recall_m(y_true, y_pred)
    return 2*((precision*recall)/(precision+recall+K.epsilon()))


reduce_lr = tf.keras.callbacks.LearningRateScheduler(lambda x: 1e-3 * 0.90 ** x)
model_e1d1.compile(optimizer=tf.keras.optimizers.Adam(), loss=tf.keras.losses.Huber(),metrics=['acc',f1_m,precision_
history_e1d1=model_e1d1.fit(X_train,y_train,epochs=25,validation_data=(X_test,y_test),batch_size=32,verbose=0,callba
model_e2d2.compile(optimizer=tf.keras.optimizers.Adam(), loss=tf.keras.losses.Huber(),metrics=['acc',f1_m,precision_
history_e2d2=model_e2d2.fit(X_train,y_train,epochs=25,validation_data=(X_test,y_test),batch_size=32,verbose=0,callba

loss_1, accuracy_1, f1_score_1, precision_1, recall_1 = model_e1d1.evaluate(X_test,y_test, verbose=0)
loss_2, accuracy_2, f1_score_2, precision_2, recall_2 = model_e2d2.evaluate(X_test,y_test, verbose=0)
```

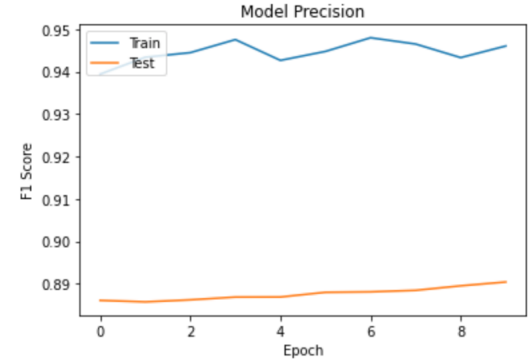
FIG 24 : Compiling both models 3 and 4 and calculating their metrics

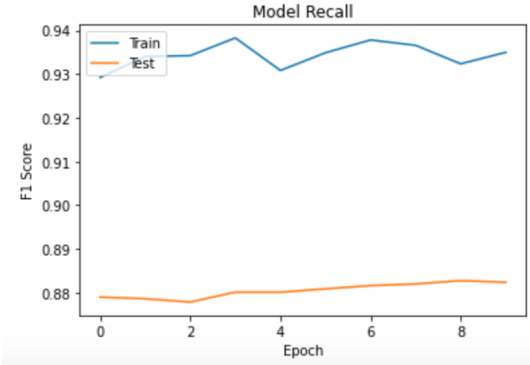
For model 3:




Model Accuracy
Train
Test
Accuracy
Epoch
0.86
0.84
0.82
0.80
0.78
0.76
0.74
0
5
10
15
20
25


Model loss
Train
Test
Loss
Epoch
12
10
8
6
4
2
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


Model Precision
Train
Test
F1 Score
Epoch
0.95
0.94
0.93
0.92
0.91
0.90
0.89
0
2
4
6
8


Model Recall
Train
Test
F1 Score
Epoch
0.94
0.93
0.92
0.91
0.90
0.89
0.88
0
2
4
6
8

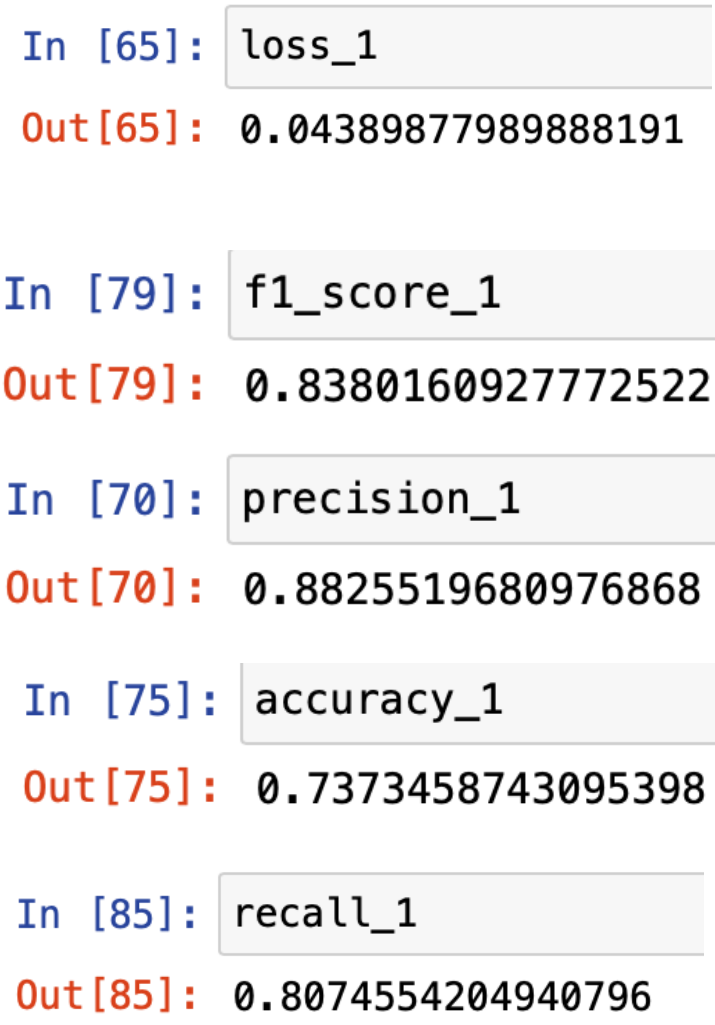
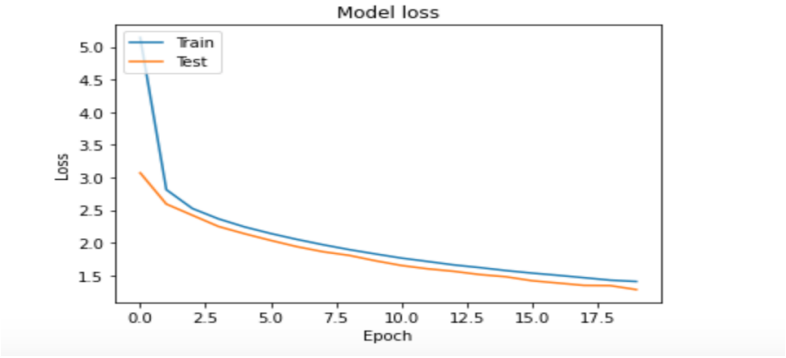
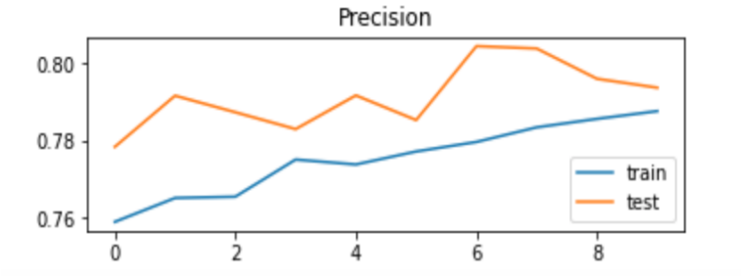
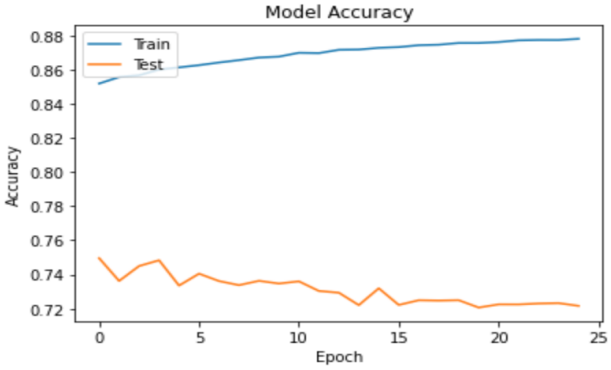


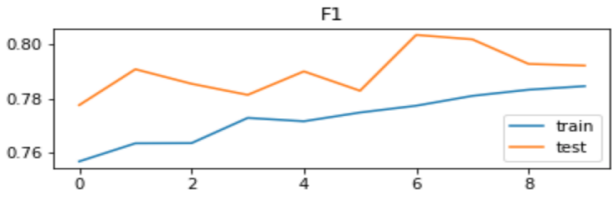
FIG 25 : Model 3 metrics and graphs

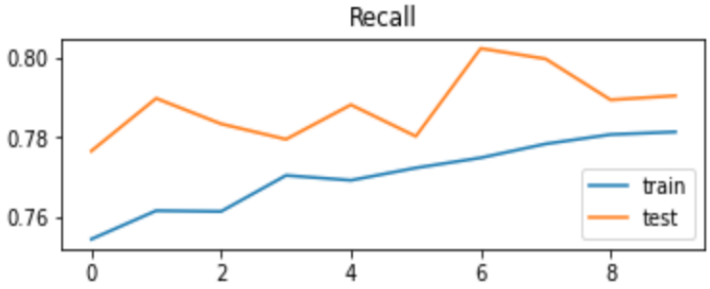
For model 4:


Model loss
Train
Test
Loss
Epoch
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


Precision
0.80
0.78
0.76
train
test
0
2
4
6
8


Model Accuracy
Train
Test
Accuracy
Epoch
0.88
0.86
0.84
0.82
0.80
0.78
0.76
0.74
0.72
0
5
10
15
20
25






```
In [66]: loss_2
Out[66]: 0.04928551986813545

In [71]: precision_2
Out[71]: 0.8242448568344116

In [76]: accuracy_2
Out[76]: 0.7216095328330994

In [80]: f1_score_2
Out[80]: 0.8086850047111511

In [86]: recall_2
Out[86]: 0.8217270374298096
```

FIG 26 : Model 4 Metrics and graphs

It can be seen that Model 3 is the one with the best metrics and hence is the best model out of the 4 models shown.

So, the prediction by model 3 is as follows :

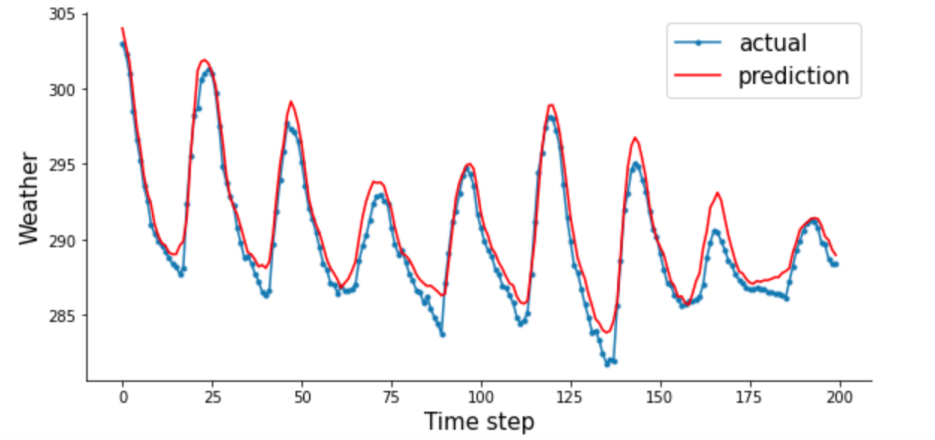


FIG 27 : Prediction by best fit model 3

Similarly, to make sure model 3 is good, i also checked for the following predicted weather versus the actual weather :

```
In [48]: predicted_temperature
Out[48]: array([[296.38995],
                [296.1101 ],
                [295.57486],
                [294.89908],
                [293.76154]], dtype=float32)
```

	temperature
datetime	
2017-11-19 20:00:00	295.44
2017-11-19 21:00:00	296.02
2017-11-19 22:00:00	296.51
2017-11-19 23:00:00	297.09
2017-11-20 00:00:00	296.69

FIG 28 : Predicted vs Actual weather for model 3

Conclusion:

So, 4 models have been compared on the basis of their performance metrics and data.

The first model had only the temperature data and the second model has the same architecture as the first, but takes all other columns except Weather. The last two models take all columns except Weather and differ in their architecture. With the addition of more features and a much more deeper RNN model, it definitely helps the analysis. What remains to be seen is how the column 'weather' would've affected the model(s).

I also showed which model is the best to use (the third one with 4 layers) on the basis of performance metrics, which were the best for this model, with the least loss and highest accuracy. Accordingly, the prediction by model 3 is also shown.

[1]https://www.analyticsvidhya.com/blog/2020/10/multivariate-multi-step-time-series-forecasting-using-stacked-lstm-sequence-to-sequence-autoencoder-in-tensorflow-2-0-keras/
[2]https://towardsdatascience.com/time-series-analysis-visualization-forecasting-with-lstm-77a905180eba
[3]https://machinelearningmastery.com/how-to-develop-lstm-models-for-time-series-forecasting/
[4]https://www.tensorflow.org/guide/keras/rnn
[5]https://towardsdatascience.com/time-series-forecasting-with-rnns-ff22683bbbb0
[6]https://medium.com/analytics-vidhya/weather-forecasting-with-recurrent-neural-networks-1eaa057d70c3
[7]https://keras.io/examples/timeseries/timeseries_weather_forecasting/
[8]https://medium.com/@llmkhoa511/time-series-analysis-and-weather-forecast-in-python-e80b664c7f71
[9]https://www.tensorflow.org/tutorials/structured_data/time_series

DATA AVAILABLE AT
https://www.kaggle.com/c/usc-dsci552-section-32415d-spring-2021-ps6/overview

CODE AVAILABLE AT
https://github.com/usc-dsci552-32415D-spring2021/problem-set-06-AnanyaSharma25/blob/main/PS6.py