**Time-series Forecasting using LSTM (PyTorch implementation)**

Exploring implementation of long short-term memory network using PyTorch and weather dataset

[[](https://medium.com/@ozdogar?source=post_page-----86169d74942e--------------------------------)](https://medium.com/@ozdogar?source=post_page-----86169d74942e--------------------------------)

[Can Ozdogar](https://medium.com/@ozdogar?source=post_page-----86169d74942e--------------------------------)

·

Follow

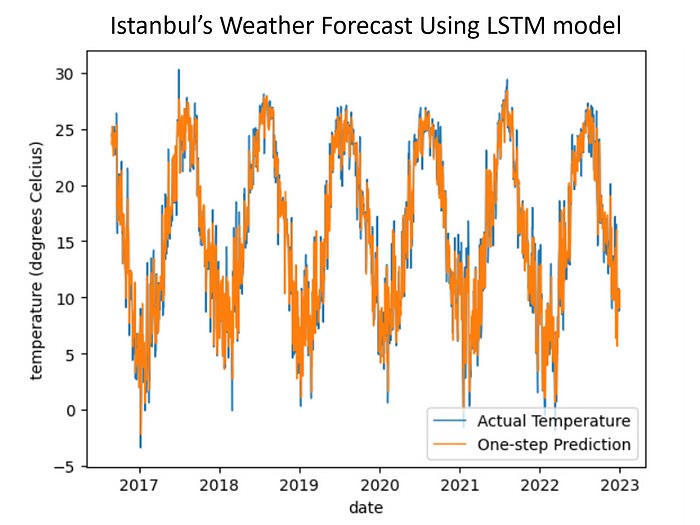
8 min read

·

Sep 10

66

2



In my previous time series post, I explored SARIMA for monthly weather forecasting. In this new post, I will be using LSTM for daily weather forecasting and show that LSTM is performing better for weather data as (i) can easily utilize multidimensional data, (ii) can make daily prediction where SARIMA fails to be feasible when seasonality is 365. This post will be focusing on the implementation part. Previous post: <https://medium.com/@ozdogar/time-series-forecasting-using-sarima-python-8db28f1d8cfc>

LSTM is a recurrent neural network that tries to solve vanishing/exploding gradient problem of RNN. For this purpose it uses:  
\* Forget gate: deciding the % of long-term memory that will be remembered  
\* Input gate: deciding the change in the long-term memory  
\* Output gate: outputs the short-term memory (and also the result from the unit)  
StatQuest channel on Youtube clearly explains the concept in detail:  
<https://www.youtube.com/watch?v=YCzL96nL7j0>

In compared to less flexible models in time-series forecasting, LSTM has poor interpretability. Additionally, you need a large dataset to perform LSTM without worrying about overfitting. Yet, it is widely used in time-series forecasting and yielding good results.

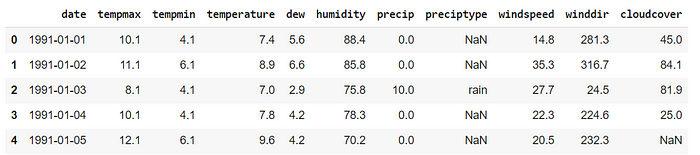
You may reach the **full Colab Notebook** I created for this study at: <https://github.com/cnzdgr/Weather-Forecast/blob/main/Weather_Prediction_using_LSTM.ipynb>

The dataset was purchased from “[www.visualcrossing.com](http://www.visualcrossing.com/)”, therefore cannot be shared. You may replicate the study on the Colab Notebook with other time series.

**Part 1: Data Preparation**

LSTM model can utilize multidimensional data. Therefore, we can use all parameters that we see relevant for our study.

weather = pd.read\_csv("drive/MyDrive/Colab Notebooks/datasets/weather/istanbul\_historical\_weather.csv", on\_bad\_lines='skip')  
weather = weather[['datetime', 'tempmax', 'tempmin', 'temp', 'dew', 'humidity', 'precip', 'preciptype', 'windspeed', 'winddir', 'cloudcover']]  
weather = weather.rename(columns={'temp': 'temperature', 'datetime': 'date'})  
weather['date'] = pd.to\_datetime(weather['date'])  
weather.set\_index('date')  
weather.head(5)



Data description:

* date: YYYY-MM-DD
* temperature: average temperature in Istanbul at the given date, in degrees Celsius
* tempmax: maximum temperature in Istanbul at the given date, in degrees Celsius
* tempmin: minimum temperature in Istanbul at the given date, in degrees Celsius
* dew: dew point; required temperature to have dew in the air
* humidity: average humiditiy in Istanbul at the given date, as percentage
* precip: precipitation, in mm
* preciptype: type of precipitation, such as rain, snow
* windspeed: average windspeed in kilometers per hour
* winddir: direction of the wind, between 0∘ and 360∘
* cloudcover: percentage of sky covered in cloud, daily average

There are a few NaN cells and we cannot feed LSTM any NaN values. Therefore, we need to delete these rows. Replacing NaN values with column’s mean/median value is also possible, but not necessary considering it’s less than %0.1 of total data points in our case.

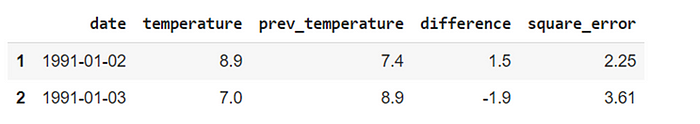
Additionally, we need to separate the month value of each date as the network will be using it. The reason is: the network needs to estimate its approximate location on a given month and a series of months is good enough. If we choose T (number of timesteps to predict) as 5, feeding the model with [Jan, Jan, Jan, Feb, Feb] can help the model to understand it’s the beginning of February. If T is close to 30, the model can make better predictions by predicting according to period of the month.

weather.dropna(inplace=True)  
weather['month'] = weather['date'].dt.month

**Part 2: Checking the Naive prediction’s error**

We need a benchmark to compare LSTM network’s performance that we are going to build in Part 3. For this task, I think the best alternative is the naive prediction’s performance. A naive temperature prediction would be “tomorrow’s average temperature will be the same as today”. We will calculate the error term for this prediction and use it as the benchmark of our LSTM model.

weather\_naive = weather[['date', 'temperature']].copy(deep=True)  
weather\_naive['prev\_temperature'] = weather\_naive['temperature'].shift(1)  
weather\_naive.drop([0], inplace=True)  
weather\_naive['difference'] = weather\_naive['temperature'] - weather\_naive['prev\_temperature']  
weather\_naive['square\_error'] = weather\_naive['difference'] \*\* 2  
weather\_naive.head(2)  
  
square\_error = weather\_naive['square\_error'].mean()  
print(f'Square Error of the Naive Approach is {square\_error:.3f}')

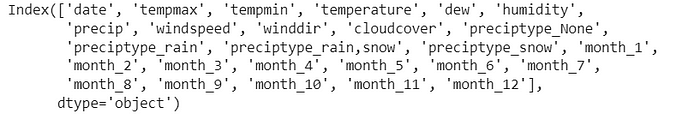




**Part 3: LSTM model for Weather Prediction**

First, we need to one-hot-encode both precipitation type and month columns.

#One-hot-encoding precipitation type and month  
weather\_LSTM = weather.copy(deep=True)  
weather\_LSTM = pd.get\_dummies(weather, columns = ['preciptype', 'month'])  
weather\_LSTM.columns



df columns

All columns will be used for the input vector as need all available data from (t-1-T) to (t-1) to predict temperature at t.  
Our targets will be temperature values at each timestep(t).  
For our analysis, we will be using T=20. T does not need to be higher than 30 in our case and any value above 10 is generating good result (empirically).  
As a note, our input tensor will have dimensions of TxDxN. At each timestep, we will be needing TxD data.  
In our case, dimensions are 20 × 25 × 11666

input\_data = weather\_LSTM.drop(['date'], axis=1)  
targets = weather\_LSTM['temperature'].values  
T = 20 #Number of timesteps to look while predicting  
D = input\_data.shape[1] #Dimensionality of the input  
N = len(input\_data) - T

We need to prepare our dataset by separating it as train/test. In this example, we will be using 80% of the data for training and the last 20% of data (nearly 6 years) for testing.  
Also, we need to preprocess our data and StandardScaler of scikit-learn is suitable for this task. It standardizes features by removing the mean and scaling to unit variance.  
As the last step, we need to convert our data from numpy to Tensor by using torch.from\_numpy() method.

#Train size: 80% of the total data size  
train\_size = int(len(input\_data) \* 0.80)  
  
# Normalization of the inputs  
scaler = StandardScaler()  
scaler.fit(input\_data[:train\_size + T - 1])  
input\_data = scaler.transform(input\_data)  
  
# Preparing X\_train and y\_train  
X\_train = np.zeros((train\_size, T, D))  
y\_train = np.zeros((train\_size, 1))  
  
for t in range(train\_size):  
 X\_train[t, :, :] = input\_data[t:t+T]  
 y\_train[t] = (targets[t+T])  
  
# Preparing X\_test and y\_test  
X\_test = np.zeros((N - train\_size, T, D))  
y\_test = np.zeros((N - train\_size, 1))  
  
for i in range(N - train\_size):  
 t = i + train\_size  
 X\_test[i, :, :] = input\_data[t:t+T]  
 y\_test[i] = (targets[t+T])  
  
# Make inputs and targets  
X\_train = torch.from\_numpy(X\_train.astype(np.float32))  
y\_train = torch.from\_numpy(y\_train.astype(np.float32))  
X\_test = torch.from\_numpy(X\_test.astype(np.float32))  
y\_test = torch.from\_numpy(y\_test.astype(np.float32))

We will be using a simple LSTM structure.

class LSTM(nn.Module):  
 def \_\_init\_\_(self, input\_dim, hidden\_dim, layer\_dim, output\_dim):  
 super(LSTM, self).\_\_init\_\_()  
 self.M = hidden\_dim  
 self.L = layer\_dim  
  
 self.rnn = nn.LSTM(  
 input\_size=input\_dim,  
 hidden\_size=hidden\_dim,  
 num\_layers=layer\_dim,  
 batch\_first=True)  
 #batch\_first to have (batch\_dim, seq\_dim, feature\_dim)  
 self.fc = nn.Linear(hidden\_dim, output\_dim)  
  
 def forward(self, X):  
 # initial hidden state and cell state  
 h0 = torch.zeros(self.L, X.size(0), self.M).to(device)  
 c0 = torch.zeros(self.L, X.size(0), self.M).to(device)  
  
 out, (hn, cn) = self.rnn(X, (h0.detach(), c0.detach()))  
  
 # h(T) at the final time step  
 out = self.fc(out[:, -1, :])  
 return out

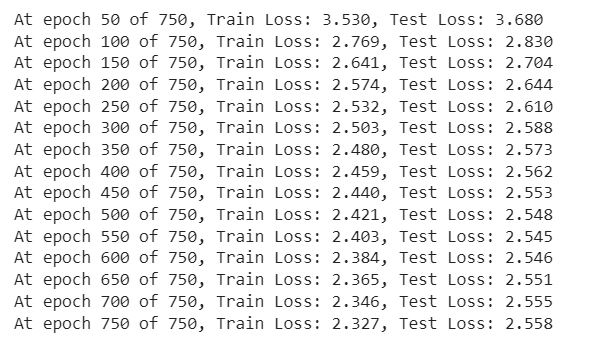
model = LSTM(D, 512, 2, 1)  
model.to(device)

For our model, I have chosen hidden dimension as 512. Considering our dataset, 512 can be considered as upper bound. Yet, you need to try different sizes and pick the most suitable one.

Now, we will be writing a standard LSTM training code. For optimizer, SGD worked better than Adam , but you may try alternatives (Adam, AdamW, etc.) for your case as finding the best optimizer often requires trial/error.

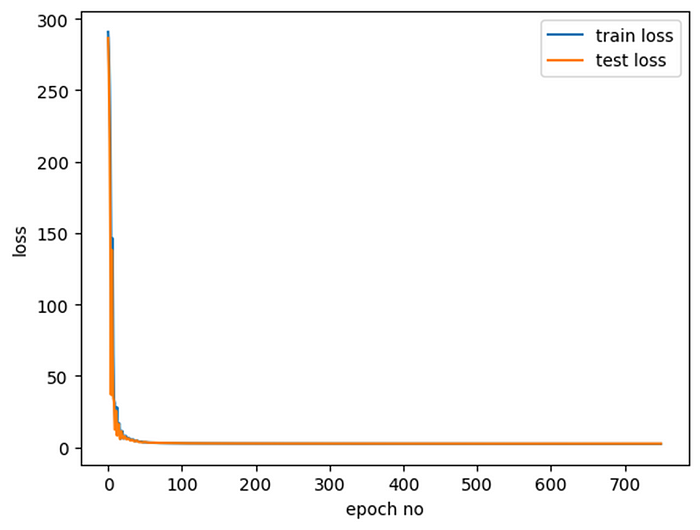
# Training  
def train(model,  
 learning\_rate,  
 X\_train,  
 y\_train,  
 X\_test,  
 y\_test,  
 epochs=200):  
  
 # Loss and optimizer  
 criterion = nn.MSELoss()  
 optimizer = torch.optim.SGD(model.parameters(), lr=learning\_rate, momentum=0.9, weight\_decay=1e-4)  
  
 train\_losses = np.zeros(epochs)  
 test\_losses = np.zeros(epochs)  
  
 for epoch in range(epochs):  
 optimizer.zero\_grad()  
  
 # Forward pass  
 outputs = model(X\_train)  
 loss = criterion(outputs, y\_train)  
  
 # Backpropagation  
 loss.backward()  
 optimizer.step()  
  
 #Train loss  
 train\_losses[epoch] = loss.item()  
  
 # Test loss  
 test\_outputs = model(X\_test)  
 test\_loss = criterion(test\_outputs, y\_test)  
 test\_losses[epoch] = test\_loss.item()  
  
 if (epoch + 1) % 50 == 0:  
 print(f'At epoch {epoch+1} of {epochs}, Train Loss: {loss.item():.3f}, Test Loss: {test\_loss.item():.3f}')  
  
 return train\_losses, test\_losses

# move data to GPU  
X\_train, y\_train = X\_train.to(device), y\_train.to(device)  
X\_test, y\_test = X\_test.to(device), y\_test.to(device)  
  
train\_losses, test\_losses = train(model,  
 0.01,  
 X\_train,  
 y\_train,  
 X\_test,  
 y\_test,  
 epochs=750)



Train/test loss

# Plot the train loss and test loss per iteration  
plt.plot(train\_losses, label='train loss')  
plt.plot(test\_losses, label='test loss')  
plt.xlabel('epoch no')  
plt.ylabel('loss')  
plt.legend()  
plt.show()



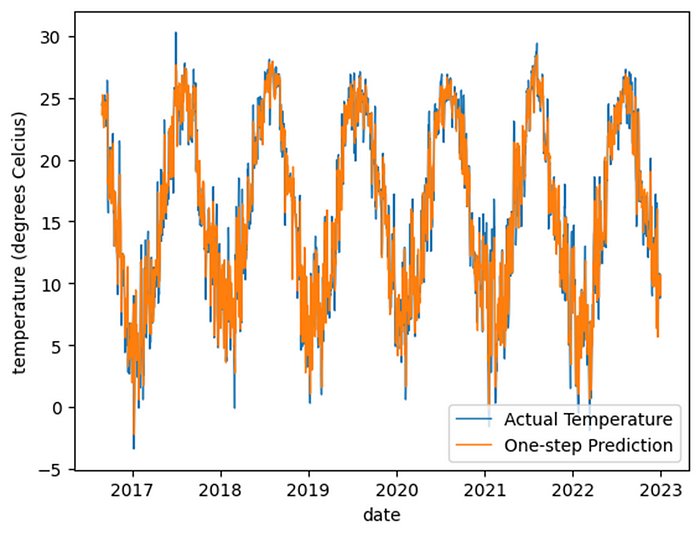
Train and test loss at each epoch

We can see that around 500 epochs are enough to properly train our model. After that, test loss is not decreasing and our model starts to overfit.

**Part 4: Checking LSTM’s Performance**

We created a working LSTM model and trained it. We saw that both training and test losses decreased substantially, so we are pretty sure that our model learned something. However, we need to verify that our model is decently performing: beating the benchmark performance with a meaningful margin.

#Checking one-step prediction performance of the model  
test\_target = y\_test.cpu().detach().numpy()  
test\_predictions = []  
  
for i in range(len(test\_target)):  
 input\_ = X\_test[i].reshape(1, T, D)  
 p = model(input\_)[0,0].item()  
  
 # update the predictions list  
 test\_predictions.append(p)  
  
plot\_len = len(test\_predictions)  
plot\_df = weather[['date', 'temperature']].copy(deep=True)  
plot\_df = plot\_df.iloc[-plot\_len:]  
plot\_df['prediction'] = test\_predictions  
plot\_df.set\_index('date', inplace=True)  
  
plt.plot(plot\_df['temperature'], label='Actual Temperature', linewidth=1)  
plt.plot(plot\_df['prediction'], label='One-step Prediction', linewidth=1)  
plt.xlabel('date')  
plt.ylabel('temperature (degrees Celsius)')  
plt.legend(loc='lower right')



Istanbul’s daily temperature

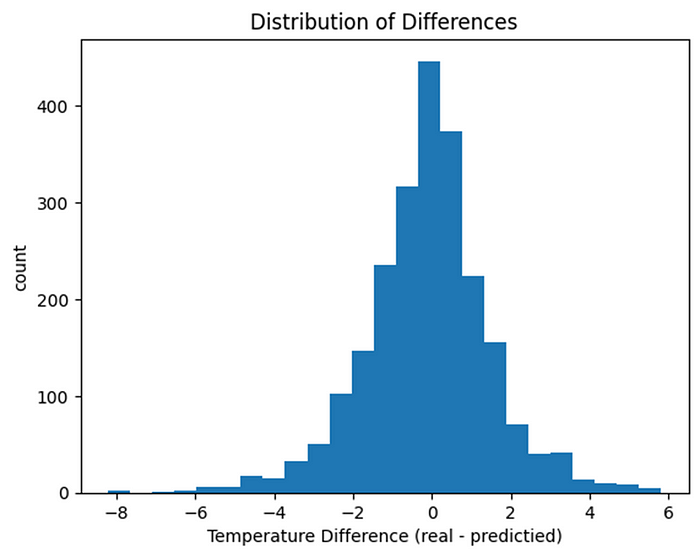
After visually checking, predictions seem fine. Yet, we need to check MSE for comparison as well.

LTSM\_error = pd.DataFrame(test\_target, columns = ['targets'])  
LTSM\_error['predictions'] =test\_predictions  
LTSM\_error['error'] = LTSM\_error['targets'] - LTSM\_error['predictions']  
LTSM\_error['error\_square'] = LTSM\_error['error'] \*\* 2  
err = LTSM\_error['error\_square'].mean()  
print(f'Mean square error is: {err:.3f}')



It is always good to check the distribution of errors. If the distribution is not resembling a normal distribution, there can be a problem with the model selection or model hyper parameters.

plt.hist(LTSM\_error['error'], bins=25)  
plt.xlabel('Temperature Difference (real - predictied)')  
plt.ylabel('count')  
plt.title('Distribution of Differences')



Distribution of SMEs

**Part 5: Discussion**

The LSTM model is better in predicting one-step-ahead average weather, in compare to the naive model. The naive model’s mean square error was 3.54, LSTM reduced it to 2.56.

This model is not fine-tuned. There are several hyper parameters that can be changed, such as hidden layer size, epochs, T selection, optimizer selection, and so on. I believe further reduction in error is possible. However, considering we only use data from Istanbul’s previous weather (20-day) and weather prediction is a complex task, I believe it is not possible to reduce square error below 2.0.

The code that is used in this example can be used for different time series forecasting tasks as well given the data size is sufficient.