Using LSTM Neural Networks to predict time series data

Mourad Askar

Table of Contents

Introduction

The project aims to predict the volume of traffic flow between Minneapolis and St. Paul at a specific point in Minnesota. My task is to build a multi-step RNN with LSTM model that makes a single prediction point of the traffic volume 2 hours into the future, given the previous 6-hour window. This can be demonstrated in (Figure 1), where the inputs represent the data point for a given 6 hours, and the label is the expected output 2 hours later.

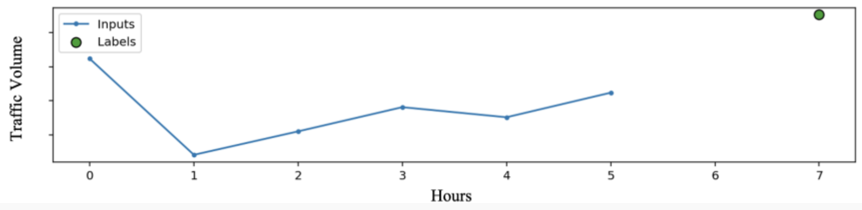


Figure 1

Exploratory Data Analysis

Data Description

This project uses the Metro Interstate Traffic Volume Data Set. Hourly Interstate 94 Westbound traffic volume for MN DoT ATR station 301, roughly midway between Minneapolis and St Paul, MN. Hourly weather features and holidays included for impacts on traffic volume.

The source dataset has a total of **48,204 records**, with **9 features**:

1. **holiday**: string (None or name of the holiday)
2. **temp**: in degrees kelvin
3. **rain_1h**: in mm for the last hour
4. **snow_1h**: in mm for the last hour
5. **clouds**: percent
6. **weather_main**: short descriptive text
7. **weather_description**: longer descriptive text
8. **date_time**: in Y/m/d H:M:S format
9. **traffic_volume**: # of cars in the last hour

The expected target feature is the hourly predictions of **traffic_volume**.

The project was a part of a class-managed Kaggle competition as part of the Neural Networks and Deep Learning course. The competition leaderboard is based on the Mean Absolute Error (MAE) of the predictions of the last 5000 records in the source dataset.

Data Challenges

1. Duplicate Records

The data was expected to be captured at 1-hour intervals, but duplicate hourly entries were found with further inspection. There were 7629 duplicate hourly entries. This raised a concern regarding the data consistency to make the predictions accurate for the specified Kaggle competition 5000 records.

Problem Details: The actual case was that for some records at a specific day-hour, I found duplicate records that have all the fields repeated except for the weather_main and weather_description, where it stated different weather conditions for the same hour. So, the traffic volume was repeated many times for the same day-hour.

Initially, I was aware of that, but I decided to treat the dataset as an hour per record, not as actual time-series set with a time unit indexed dataset, but this led to modest validation results (mid 300s MAE). Then, I decided to preprocess the training and validation dataset properly as time-series indexed at 1-hour intervals, and the validation results were much better (low 200s MAE). But when I submitted for the Kaggle predictions, I got a high MAE.

Problem Details: The problem is that the last 5000 records for the Kaggle competition submission are not 5000 hours, and if we do detailed data preprocessing and cleaning, we'll collapse the duplicate day-hour records into a single day-hour record, leading to a shifted record ID locations for the Kaggle submission.

To solve this problem, I decided to make my final predictions using an appropriately preprocessed testing dataset that is time-series indexed at 1-hour intervals, which collapsed the number of records down to 4083 records. Then, before the actual submission, I used the original day-hour locations from the source testing dataset (5000 records) to augment a repeated prediction for the repeated day-hours.

I had to do the time-series indexing after all the initial cleansing and data splitting into training, validation, and testing. To keep track of the records belonging to the right dataset.

2. Missing Records

The data was also found to suffer from time gaps presented as missing records for specific periods. The largest span was a gap between 2014-08-08 and 2015-06-11 (Figure 2), with nearly ten months of missing data. This mainly affects the validation dataset records range. I had no solution for the long gaps other than seeing how the neural networks will deal with such gaps.

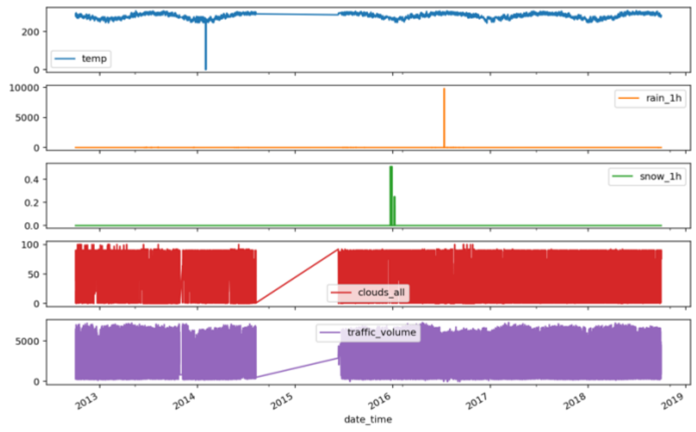


Figure 2

3. Outliers

A few records in two days had the temperature field set at zero, which didn't seem right (Figure 3). I fixed this by setting the missing values with the temperature average of each day (Figure 4).

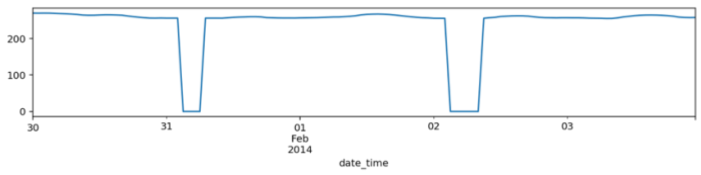


Figure 3

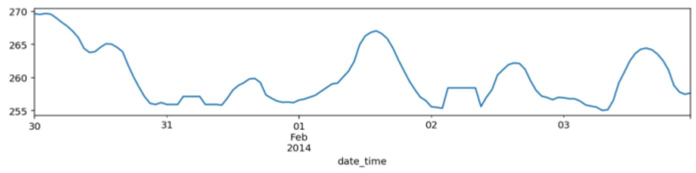


Figure 4

Rain field also had a single extreme value, which I also set to the mean of that day.
Snow had extreme values, but I could not determine if the values were to be outliers or not; it could be an exceptional winter, and as I'm unaware of the weather conditions in that area, I decided to keep it as is.

Features Engineering and Data Transformations

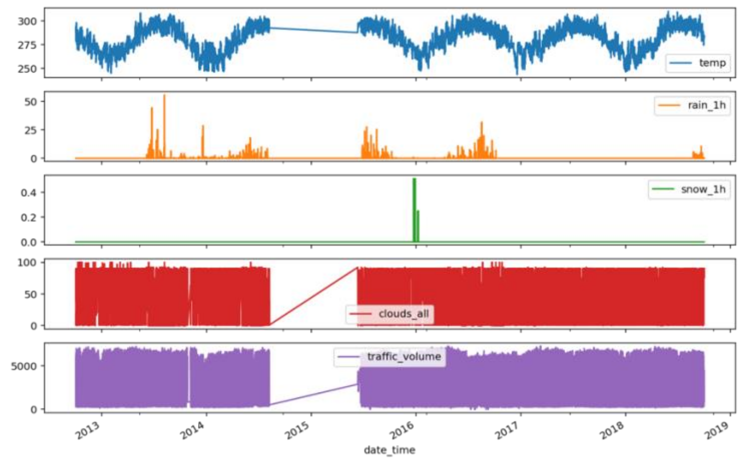


Figure 5

A quick overview of the data after the initial cleansing (Figure 5) shows a better situation than the initial view (Figure 2).

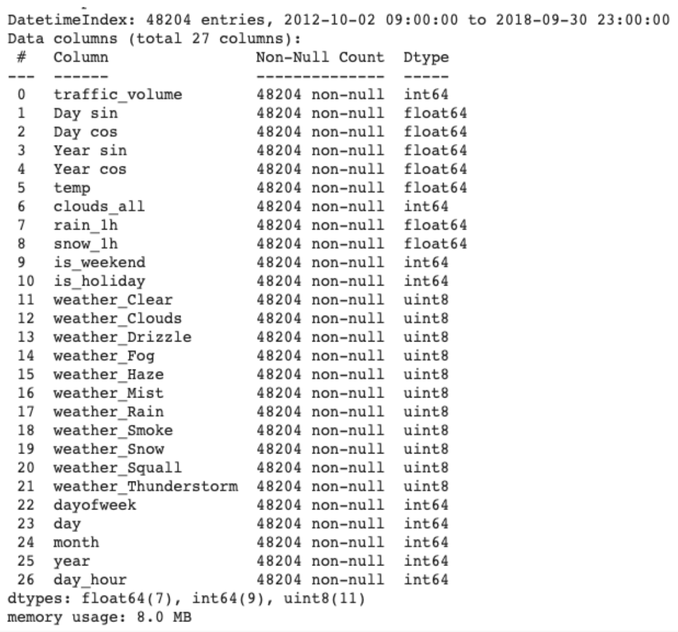
1. New Features

I decided to transform the **weather_main** into one-hot encoded variables and to drop the **weather_description** as I see that it adds a kind of redundant information with weather_main. Also, I think the valuable information to capture is whether the day is a holiday or a weekend. We don't need to keep track of which holiday. So, a new feature **is_holiday** is created, and the old feature **holiday** was dropped. Similarly, we don't need to keep track of which weekend it is. So, I created a new feature, **is_weekend**.

The date_time field was converted into a signal using sin and cos to convert the time to clear "Time of day" and "Time of year" signals. This gives the model access to the most important frequency features.

Lastly, I broke down the date_time components into its other elemental fields, dayofweek, day, month, year, and day_hour.

We end with 27 features instead of the initial 9 features.



```
DatetimeIndex: 48204 entries, 2012-10-02 09:00:00 to 2018-09-30 23:00:00
Data columns (total 27 columns):
 #   Column               Non-Null Count  Dtype  
---  ------               --------------  -----  
 0   traffic_volume       48204 non-null  int64  
 1   Day sin              48204 non-null  float64
 2   Day cos              48204 non-null  float64
 3   Year sin             48204 non-null  float64
 4   Year cos             48204 non-null  float64
 5   temp                 48204 non-null  float64
 6   clouds_all           48204 non-null  int64  
 7   rain_1h              48204 non-null  float64
 8   snow_1h              48204 non-null  float64
 9   is_weekend           48204 non-null  int64  
 10  is_holiday           48204 non-null  int64  
 11  weather_Clear        48204 non-null  uint8  
 12  weather_Clouds       48204 non-null  uint8  
 13  weather_Drizzle      48204 non-null  uint8  
 14  weather_Fog          48204 non-null  uint8  
 15  weather_Haze         48204 non-null  uint8  
 16  weather_Mist         48204 non-null  uint8  
 17  weather_Rain         48204 non-null  uint8  
 18  weather_Smoke        48204 non-null  uint8  
 19  weather_Snow         48204 non-null  uint8  
 20  weather_Squall       48204 non-null  uint8  
 21  weather_Thunderstorm 48204 non-null  uint8  
 22  dayofweek            48204 non-null  int64  
 23  day                  48204 non-null  int64  
 24  month                48204 non-null  int64  
 25  year                 48204 non-null  int64  
 26  day_hour             48204 non-null  int64  
dtypes: float64(7), int64(9), uint8(11)
memory usage: 8.0 MB
```

Figure 6

2. Timeseries index resampling

As mentioned in the data challenges section, there were 7629 duplicate hourly entries. And I decided to fix this by resampling the data to be on a 1-hour basis so that each record resembles only one hour. Duplicate hour records were averaged within the same hour. The transformations were done after the dataset split into training, validation, and testing.

Dataset split before timeseries resampling:

train_df:	33204
val_df:	10000
test_df:	5000

Dataset split after 1-hour timeseries resampling:

train_df:	40110
val_df:	8359
test_df:	4083
kaggle_df:	5000

I kept the kaggle_df in its original form, including the duplicates to be the reference has the original data_time records and ID locations, to be used later for matching the predications to the correct location in the final submission CSV for the Kaggle competition.

3. Data Normalization

I got the best results using Min-Max Normalization vs. Standard Scaling.

Experimentation Journey

I started my experiments having the same objective that I had in previous assignments. My experiments aimed to evaluate the effect of tweaking each hyperparameter individually on the performance of the RNN and LSTM based models. I decided not to use automated hyperparameter search methods to understand better how each hyperparameter affects the model.

Given that the data preprocessing consumed many hours, I had to apply quick and dirty methods to help me understand how the LSTM networks worked. Alternative to the hybrid automated systematic hyperparameters evaluation that I used in my previous assignments. I applied systematic manual testing of the hyperparameters using extremely small and large values for the number of LSTM units. Then, I added other layer types such as Convolutional, Dense, GRU, and Bi-directional layers. I also experimented with different batch sizes. Then, I started mixing and matching my observations based on how the layers and the hyperparameters affected the models.

I tried to find well-known LSTM model architectures as I previously found for the Convolutional Neural Networks. Still, I couldn't find anything beyond a couple of LSTM layers as references.

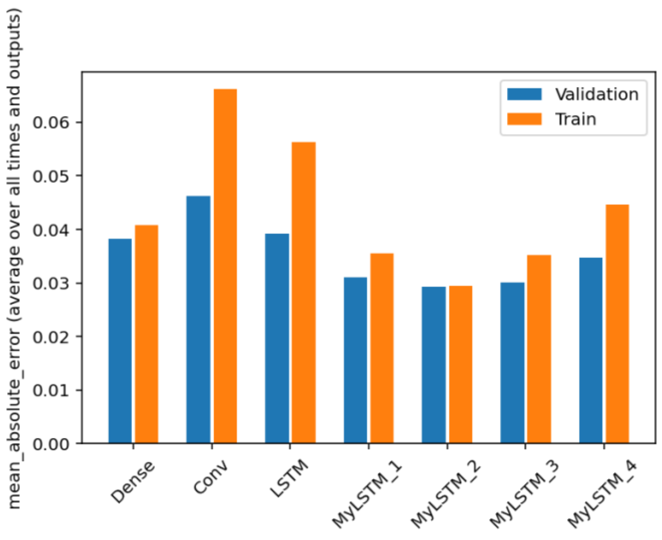
The hyperparameters that were systematically tested using various values are:

- Number of LSTM units and layers, and Dense units at the final layers (Model 1)
- Bi-Directional LSTM (Model 2) – **My Kaggle submission**
- GRU (Model 3)
- Convolutional (Model 4)

Models Building

I kept a variation of three reference models provided in the TensorFlow tutorial (Dense, Conv, and LSTM). I built three other models (MyLSTM_1, MyLSTM_2, and MyLSTM_3) that performed better than the reference models and a fourth model (MyLSTM_4) that had debatable results because of its high variance.

All models' comparison

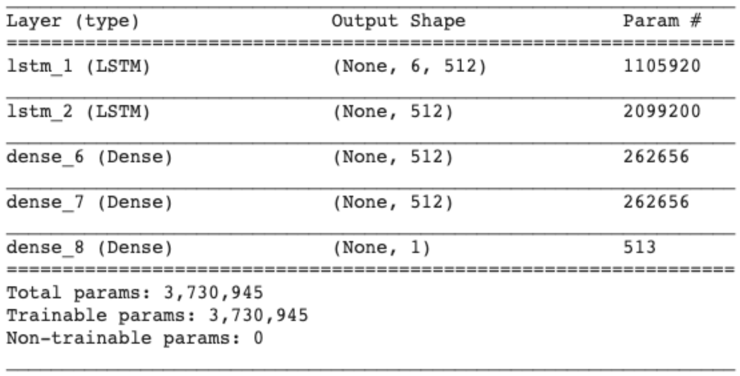


```
Dense   : 296.9976
Conv    : 481.6037
LSTM    : 410.0791
MyLSTM_1: 258.4220
MyLSTM_2: 214.6449
MyLSTM_3: 255.8801
MyLSTM_4: 325.1835
```

Figure 7

Base model (MyLSTM_1)

My base model is Model 1 (MyLSTM_1)



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
lstm_1 (LSTM)                (None, 6, 512)            1105920   
_________________________________________________________________
lstm_2 (LSTM)                (None, 512)               2099200   
_________________________________________________________________
dense_6 (Dense)              (None, 512)               262656    
_________________________________________________________________
dense_7 (Dense)              (None, 512)               262656    
_________________________________________________________________
dense_8 (Dense)              (None, 1)                 513       
=================================================================
Total params: 3,730,945
Trainable params: 3,730,945
Non-trainable params: 0
_________________________________________________________________
```

Training and Validation Results:

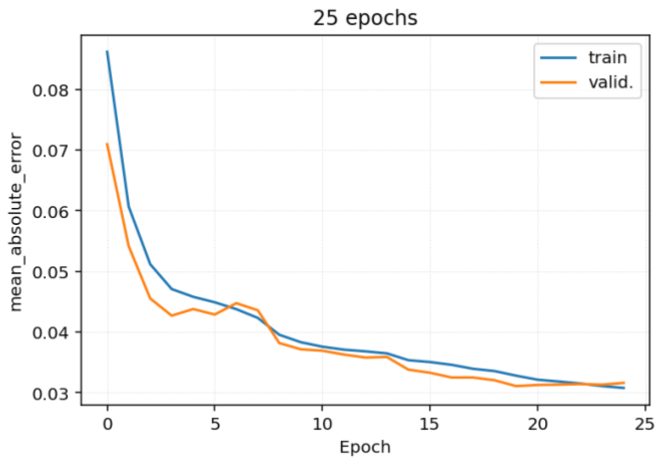


Figure 8

Validation Dataset Predictions vs. True Result (Last 500 hours):

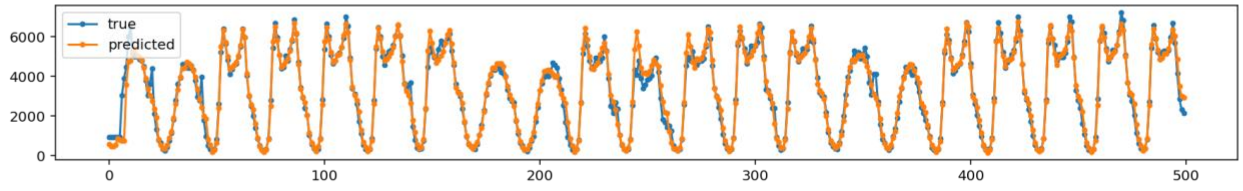
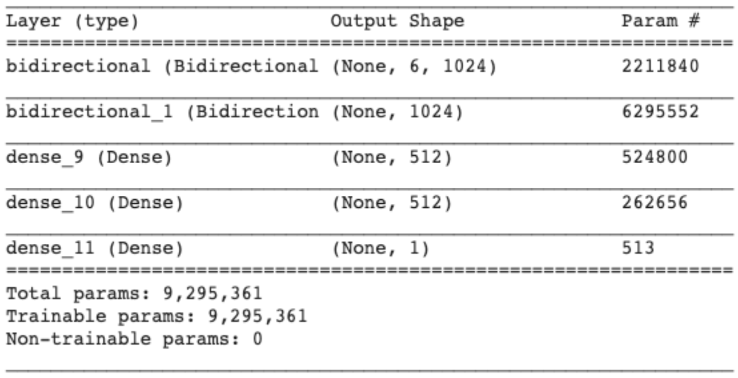


Figure 9

Best model (MyLSTM_2) (Kaggle submission)

My best model is Model 2 (MyLSTM_2)



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
bidirectional (Bidirectional (None, 6, 1024)           2211840
_________________________________________________________________
bidirectional_1 (Bidirection (None, 1024)              6295552
_________________________________________________________________
dense_9 (Dense)              (None, 512)               524800
_________________________________________________________________
dense_10 (Dense)             (None, 512)               262656
_________________________________________________________________
dense_11 (Dense)             (None, 1)                 513
=================================================================
Total params: 9,295,361
Trainable params: 9,295,361
Non-trainable params: 0
_________________________________________________________________
```

Training and Validation Results:

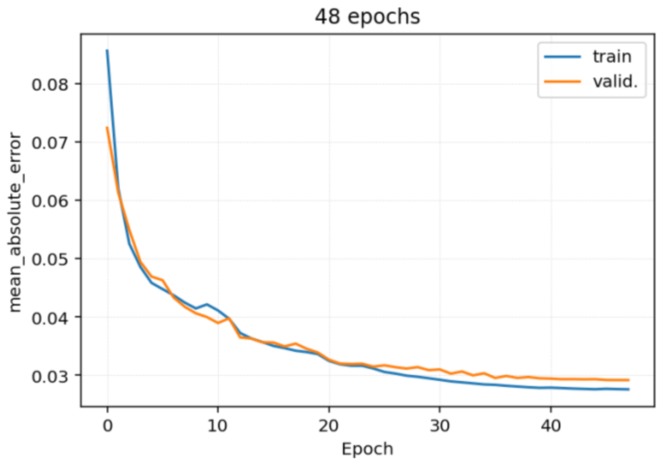


Figure 10

Validation Dataset Predictions vs. True Result (Last 500 hours):

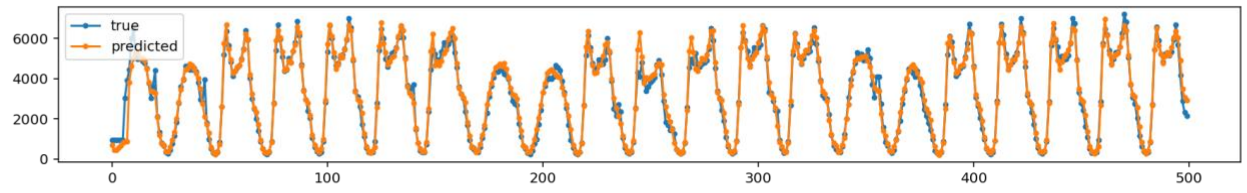


Figure 11

My best model utilized bidirectional LSTMs with two custom forward and backward layers, and two dense layers with 512 units each, and a dense output layer with a single unit. The model performed the best and maintained low variance for more epochs during training (Figure 10). The predictions in (Figure 11) show that the model captured all the significant patterns with minor misses of some anomalies. The model is probably an overkill, and I might have got similar results with less units, but I chose not to optimize the model as much as maximize the gains.

Conclusion

- The LSTM Bi-Directional model performed the best with the least variance.
- I could not make LSTM networks gain better results by going deeper but having more LSTM units made a difference.
- LSTM and GRU should help with the Vanishing Gradient Descent problem in deep networks, but I did not gain any benefits in building deeper networks. Maybe I was not patient enough this time.
- I noticed that most of the models with fewer LSTM units were hardly overfitting and showed better validation scores than the training score. Which I was not able to interpret.

My reflections

- Time-series is an interesting type of problem. Though, I had no prior experience in dealing with it. So, I had to understand a few basic concepts to deal with the time-series data in the best manner. Such as utilizing pandas DateTimeIndex and resampling.
- A new thing that I learned and applied was the callback functions of TensorFlow/Keras. I used three of them, and worth mentioning that one of them was a custom implementation. The first one would be for "early stopping" the training if there were no progress for a certain number of epochs. The second one was to reduce the "learning rate" when there was no enhancement for a certain number of epochs. The third one is a custom implementation; other online sources influenced that to help me visualize the training/validation progress after each epoch. This one was fun and exciting to use. It's similar to using TensorBoard but was simple and easily customizable.