Forecasting using spatio-temporal data with combined Graph Convolution + LSTM model

Run the latest release of this notebook: launch binder Open in Colab

The dynamics of many real-world phenomena are spatio-temporal in nature. Traffic forecasting is a quintessential example of spatio-temporal problems for which we present here a deep learning framework that models speed prediction using spatio-temporal data. The task is challenging due to two main inter-linked factors: (1) the complex spatial dependency on road networks, and (2) non-linear temporal dynamics with changing road conditions.

To address these challenges, here we explore a neural network architecture that learns from both the spatial road network data and time-series of historical speed changes to forecast speeds on road segments at a future time. In the following we demo how to forecast speeds on road segments through a `graph convolution` and `LSTM` hybrid model. The spatial dependency of the road networks are learnt through multiple graph convolution layers stacked over multiple LSTM, sequence to sequence model, layers that leverage the historical speeds on top of the network structure to predicts speeds in the future for each entity.

The architecture of the GCN-LSTM model is inspired by the paper: T-GCN: A Temporal Graph Convolutional Network for Traffic Prediction.

The authors have made available the implementation of their model in their GitHub repository. There has been a few differences in the architecture proposed in the paper and the implementation of the graph convolution component, these issues have been documented here and here. The `GCN_LSTM` model in `StellarGraph` emulates the model as explained in the paper while giving additional flexibility of adding any number of `graph convolution` and `LSTM` layers.

Concretely, the architecture of `GCN_LSTM` is as follows:

1. User defined number of graph convolutional layers (Reference: Kipf & Welling (ICLR 2017)).
2. User defined number of LSTM layers. The TGCN uses GRU instead of LSTM. In practice there are not any remarkable differences between the two types of layers. We use LSTM as they are more frequently used.
3. A Dropout and a Dense layer as they experimentally showed improvement in performance and managing over-fitting.

References:

- T-GCN: A Temporal Graph Convolutional Network for Traffic Prediction
- https://github.com/lehaifeng/T-GCN
- Semi-Supervised Classification with Graph Convolutional Networks

Note: this method is applicable for uni-variate timeseries forecasting.

```
# install StellarGraph if running on Google Colab
import sys
if 'google.colab' in sys.modules:
  %pip install -q stellargraph[demos]==1.2.1
```

```
# verify that we're using the correct version of StellarGraph for this notebook
import stellargraph as sg

try:
    sg.utils.validate_notebook_version("1.2.1")
except AttributeError:
    raise ValueError(
        f"This notebook requires StellarGraph version 1.2.1, but a different version {sg.__ve
    ) from None
```

```
import os
import sys
import urllib.request

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.lines as mlines

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import Sequential, Model
from tensorflow.keras.layers import LSTM, Dense, Dropout, Input
```

Data

We apply the GCN-LSTM model to the **Los-loop** data. This traffic dataset contains traffic information collected from loop detectors in the highway of Los Angeles County (Jagadish et al., 2014). There are several processed versions of this dataset used by the research community working in Traffic forecasting space.

This demo is based on the preprocessed version of the dataset used by the TGCN paper. It can be directly accessed from there github repo.

This dataset contains traffic speeds from Mar.1 to Mar.7, 2012 of 207 sensors, recorded every 5 minutes.

In order to use the model, we need:

- A N by N adjacency matrix, which describes the distance relationship between the N sensors,
- A N by T feature matrix, which describes the (f_1, .., f_T) speed records over T timesteps for the N sensors.

A couple of other references for the same data albeit different time length are as follows:

- DIFFUSION CONVOLUTIONAL RECURRENT NEURAL NETWORK: DATA-DRIVEN TRAFFIC FORECASTING: This dataset consists of 207 sensors and collect 4 months of data ranging from Mar 1st 2012 to Jun 30th 2012 for the experiment. It has some missing values.
- ST-MetaNet: Urban Traffic Prediction from Spatio-Temporal Data Using Deep Meta Learning. This work uses the DCRNN preprocessed data.

Loading and preprocessing the data

```
import stellargraph as sg
```

This demo is based on the preprocessed version of the dataset used by the TGCN paper.

```
dataset = sg.datasets.METR_LA()
```

(See the "Loading from Pandas" demo for details on how data can be loaded.)

```
speed_data, sensor_dist_adj = dataset.load()
num_nodes, time_len = speed_data.shape
print("No. of sensors:", num_nodes, "\nNo of timesteps:", time_len)
```

```
    No. of sensors: 207
    No of timesteps: 2016
```

Let's look at a sample of speed data.

```
speed_data.head()
```

	0	1	2	3	4	5	6	7	8	
773869	64.375	62.666667	64.00	61.777778	59.555556	57.333333	66.500	63.625	68.750	6
767541	67.625	68.555556	63.75	65.500000	67.250000	69.000000	63.875	67.250	65.250	6

As you can see above, there are 2016 observations (timesteps) of speed records over 207 sensors. Speeds are recorded every 5 minutes. This means that, for a single hour, you will have 12 observations. Similarly, a single day will contain 288 (12x24) observations. Overall, the data consists of speeds recorded every 5 minutes over 207 for 7 days (12X24X7).

Forecasting with spatio-temporal data as a supervised learning problem

Time series forecasting problem can be cast as a supervised learning problem. We can do this by using previous timesteps as input features and use the next timestep as the output to predict. Then, the spatio-temporal forecasting question can be modeled as predicting the feature value in the future, given the historical values of the feature for that entity as well as the feature values of the entities "connected" to the entity. For example, the speed prediction problem, the historical speeds of the sensors are the timeseries and the distance between the sensors is the indicator for connectivity or closeness of sensors.

Train/test split

Just like for modeling any standard supervised learning problem, we first split the data into mutually exclusive train and test sets. However, unlike, a standard supervised learning problem, in timeseries analysis, the data is in some chronological time respecting order and the train/test happens along the timeline. Lets say, we use the first `T_t` observations for training and the remaining `T - T_t` of the total `T` observations for testing.

In the following we use first 80% observations for training and the rest for testing.

```
def train_test_split(data, train_portion):
    time_len = data.shape[1]
    train_size = int(time_len * train_portion)
    train_data = np.array(data.iloc[:, :train_size])
    test_data = np.array(data.iloc[:, train_size:])
    return train_data, test_data


train_rate = 0.8


train_data, test_data = train_test_split(speed_data, train_rate)
print("Train data: ", train_data.shape)
print("Test data: ", test_data.shape)
```

```
Train data:  (207, 1612)
Test data:  (207, 404)
```

Scaling

It is generally a good practice to rescale the data from the original range so that all values are within the range of 0 and 1. Normalization can be useful and even necessary when your time series data has input values with differing scales. In the following we normalize the speed timeseries by the maximum and minimum values of speeds in the train data.

Note: `MinMaxScaler` in `scikit learn` library is typically used for transforming data. However, in timeseries data since the features are distinct timesteps, so using the historical range of values in a particular timestep as the range of values in later timesteps, may not be correct. Hence, we use the maximum and the minimum of the entire range of values in the timeseries to scale and transform the train and test sets respectively.

```
def scale_data(train_data, test_data):
    max_speed = train_data.max()
    min_speed = train_data.min()
    train_scaled = (train_data - min_speed) / (max_speed - min_speed)
    test_scaled = (test_data - min_speed) / (max_speed - min_speed)
    return train_scaled, test_scaled
```

```
train_scaled, test_scaled = scale_data(train_data, test_data)
```

Sequence data preparation for LSTM

We first need to prepare the data to be fed into an LSTM. The LSTM model learns a function that maps a sequence of past observations as input to an output observation. As such, the sequence of observations must be transformed into multiple examples from which the LSTM can learn.

To make it concrete in terms of the speed prediction problem, we choose to use 50 minutes of historical speed observations to predict the speed in future, lets say, 1 hour ahead. Hence, we would first reshape the timeseries data into windows of 10 historical observations for each segment as the input and the speed 60 minutes later is the label we are interested in predicting. We use the sliding window approach to prepare the data. This is how it works:

- Starting from the beginning of the timeseries, we take the first 10 speed records as the 10 input features and the speed 12 timesteps head (60 minutes) as the speed we want to predict.
- Shift the timeseries by one timestep and take the 10 observations from the current point as the input features and the speed one hour ahead as the output to predict.

- Keep shifting by 1 timestep and picking the 10 timestep window from the current time as input feature and the speed one hour ahead of the 10th timestep as the output to predict, for the entire data.
- The above steps are done for each sensor.

The function below returns the above transformed timeseries data for the model to train on. The parameter `seq_len` is the size of the past window of information. The `pre_len` is how far in the future does the model need to learn to predict.

For this demo:

- Each training observation are 10 historical speeds (`seq_len`).
- Each training prediction is the speed 60 minutes later (`pre_len`).

```
seq_len = 10
pre_len = 12


def sequence_data_preparation(seq_len, pre_len, train_data, test_data):
    trainX, trainY, testX, testY = [], [], [], []

    for i in range(train_data.shape[1] - int(seq_len + pre_len - 1)):
        a = train_data[:, i : i + seq_len + pre_len]
        trainX.append(a[:, :seq_len])
        trainY.append(a[:, -1])

    for i in range(test_data.shape[1] - int(seq_len + pre_len - 1)):
        b = test_data[:, i : i + seq_len + pre_len]
        testX.append(b[:, :seq_len])
        testY.append(b[:, -1])

    trainX = np.array(trainX)
    trainY = np.array(trainY)
    testX = np.array(testX)
    testY = np.array(testY)

    return trainX, trainY, testX, testY


trainX, trainY, testX, testY = sequence_data_preparation(
    seq_len, pre_len, train_scaled, test_scaled
)
print(trainX.shape)
print(trainY.shape)
print(testX.shape)
print(testY.shape)
```

```
(1591, 207, 10)
(1591, 207)
```

```
(383, 207, 10)
(383, 207)
```

StellarGraph Graph Convolution and LSTM model

```
from stellargraph.layer import GCN_LSTM
```

```
gcn_lstm = GCN_LSTM(
    seq_len=seq_len,
    adj=sensor_dist_adj,
    gc_layer_sizes=[16, 10],
    gc_activations=["relu", "relu"],
    lstm_layer_sizes=[200, 200],
    lstm_activations=["tanh", "tanh"],
)
```

```
x_input, x_output = gcn_lstm.in_out_tensors()
```

```
model = Model(inputs=x_input, outputs=x_output)
```

```
model.compile(optimizer="adam", loss="mae", metrics=["mse"])
```

```
history = model.fit(
    trainX,
    trainY,
    epochs=100,
    batch_size=60,
    shuffle=True,
    verbose=0,
    validation_data=[testX, testY],
)
```

```
model.summary()
```

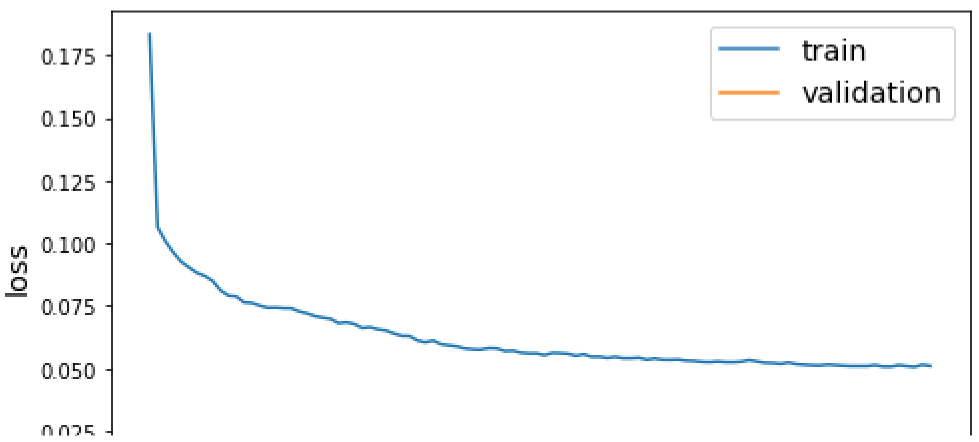
```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 207, 10)]         0
_________________________________________________________________
tf_op_layer_ExpandDims (Tens [(None, 207, 10, 1)]      0
_________________________________________________________________
reshape (Reshape)            (None, 207, 10)           0
_________________________________________________________________
fixed_adjacency_graph_convol (None, 207, 16)           43216
_________________________________________________________________
```

```
fixed_adjacency_graph_convol (None, 207, 10)           43216
_________________________________________________________________
reshape_1 (Reshape)          (None, 207, None, 1)      0
_________________________________________________________________
permute (Permute)            (None, None, 207, 1)      0
_________________________________________________________________
reshape_2 (Reshape)          (None, None, 207)         0
_________________________________________________________________
lstm (LSTM)                  (None, None, 200)         326400
_________________________________________________________________
lstm_1 (LSTM)                (None, 200)               320800
_________________________________________________________________
dropout (Dropout)            (None, 200)               0
_________________________________________________________________
dense (Dense)                (None, 207)               41607
=================================================================
Total params: 775,239
Trainable params: 689,541
Non-trainable params: 85,698
_________________________________________________________________
```

```
print(
    "Train loss: ",
    history.history["loss"][-1],
    "\nTest loss:",
    history.history["val_loss"][-1],
)
```

```
Train loss:  0.05109991133213043
Test loss: 0.0
```

```
sg.utils.plot_history(history)
```



```
ythat = model.predict(trainX)
yhat = model.predict(testX)
```

Rescale values

Rescale the predicted values to the original value range of the timeseries.

```
## Rescale values
max_speed = train_data.max()
min_speed = train_data.min()

## actual train and test values
train_rescref = np.array(trainY * max_speed)
test_rescref = np.array(testY * max_speed)
```

```
## Rescale model predicted values
train_rescpred = np.array((ythat) * max_speed)
test_rescpred = np.array((yhat) * max_speed)
```

Measuring the performance of the model

To understand how well the model is performing, we compare it against a naive benchmark.

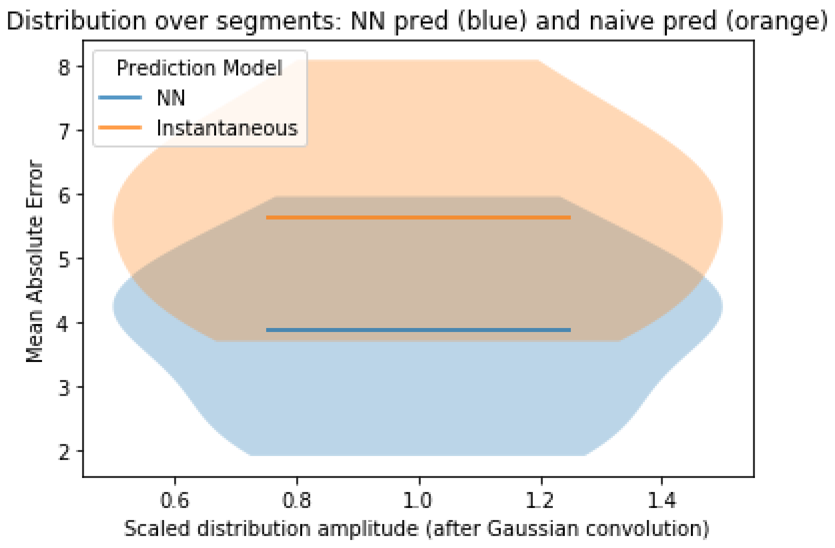
1. Naive prediction: using the most recently **observed** value as the predicted value. Note, that albeit being **naive** this is a very strong baseline to beat. Especially, when speeds are recorded at a 5 minutes granularity, one does not expect many drastic changes within such a short period of time. Hence, for short-term predictions naive is a reasonable good guess.

Naive prediction benchmark (using latest observed value)

```
## Naive prediction benchmark (using previous observed value)

testnpred = np.array(testX)[
    :, :, -1
]  # picking the last speed of the 10 sequence for each segment in each sample
testnpredc = (testnpred) * max_speed


## Performance measures

seg_mael = []
seg_masel = []
seg_nmael = []

for j in range(testX.shape[-1]):

    seg_mael.append(
        np.mean(np.abs(test_rescref.T[j] - test_rescpred.T[j]))
    )  # Mean Absolute Error for NN
    seg_nmael.append(
        np.mean(np.abs(test_rescref.T[j] - testnpredc.T[j]))
    )  # Mean Absolute Error for naive prediction
    if seg_nmael[-1] != 0:
        seg_masel.append(
            seg_mael[-1] / seg_nmael[-1]
        )  # Ratio of the two: Mean Absolute Scaled Error
    else:
        seg_masel.append(np.NaN)

print("Total (ave) MAE for NN: " + str(np.mean(np.array(seg_mael))))
print("Total (ave) MAE for naive prediction: " + str(np.mean(np.array(seg_nmael))))
print(
    "Total (ave) MASE for per-segment NN/naive MAE: "
    + str(np.nanmean(np.array(seg_masel)))
)
print(
    "...note that MASE<1 (for a given segment) means that the NN prediction is better than th
)
```

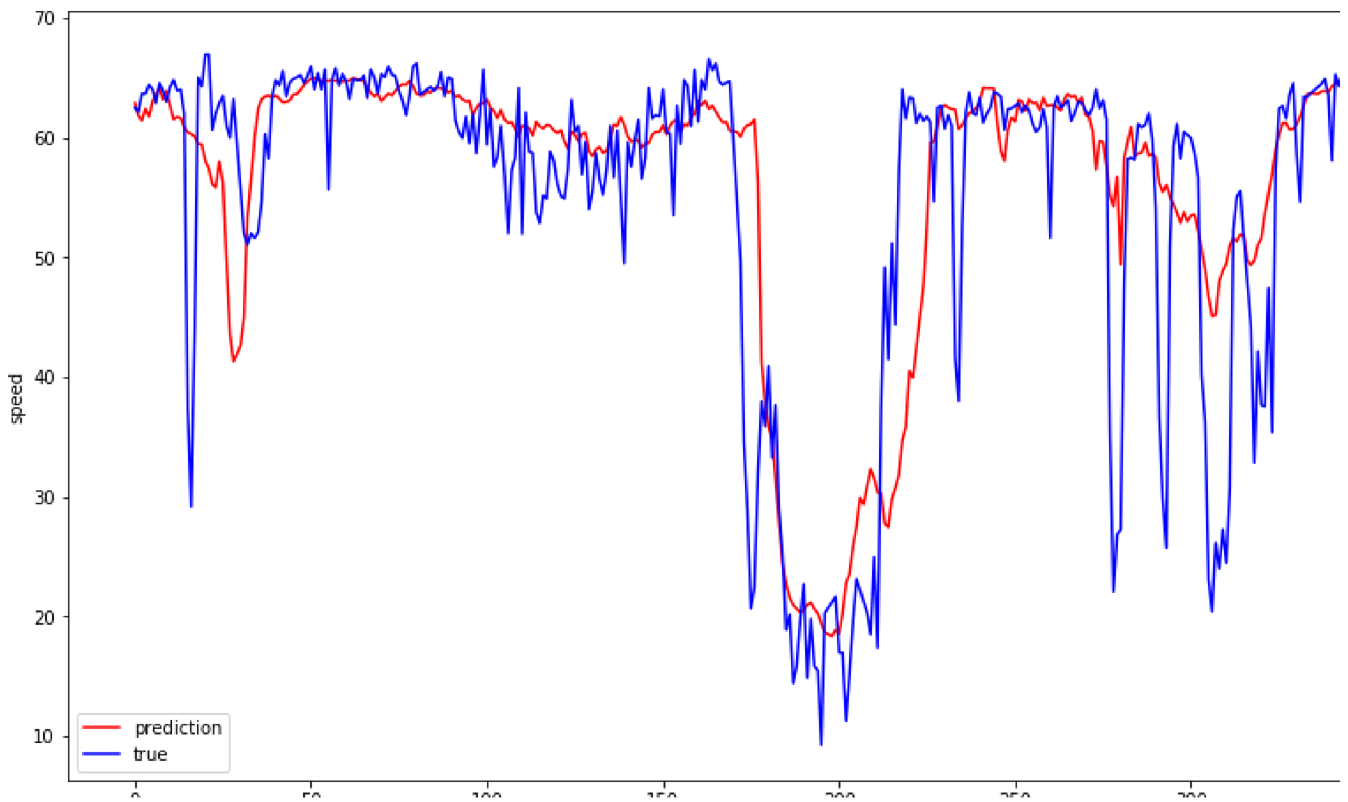
```
Total (ave) MAE for NN: 3.8822542511393068
Total (ave) MAE for naive prediction: 5.619645381284217
Total (ave) MASE for per-segment NN/naive MAE: 0.6846969434482696
...note that MASE<1 (for a given segment) means that the NN prediction is better than th
```

```
# plot violin plot of MAE for naive and NN predictions
fig, ax = plt.subplots()
# xl = minsl

ax.violinplot(
    list(seg_mael), showmeans=True, showmedians=False, showextrema=False, widths=1.0
)

ax.violinplot(
    list(seg_nmael), showmeans=True, showmedians=False, showextrema=False, widths=1.0
)

line1 = mlines.Line2D([], [], label="NN")
line2 = mlines.Line2D([], [], color="C1", label="Instantaneous")

ax.set_xlabel("Scaled distribution amplitude (after Gaussian convolution)")
ax.set_ylabel("Mean Absolute Error")
ax.set_title("Distribution over segments: NN pred (blue) and naive pred (orange)")
plt.legend(handles=(line1, line2), title="Prediction Model", loc=2)
plt.show()
```



Plot of actual and predicted speeds on a sample sensor

```
##all test result visualization
fig1 = plt.figure(figsize=(15, 8))
#    ax1 = fig1.add_subplot(1,1,1)
a_pred = test_rescpred[:, 100]
a_true = test_rescref[:, 100]
plt.plot(a_pred, "r-", label="prediction")
plt.plot(a_true, "b-", label="true")
plt.xlabel("time")
plt.ylabel("speed")
plt.legend(loc="best", fontsize=10)
plt.show()
```



Run the latest release of this notebook: launch binder Open in Colab