Master Science de la matière
École Normale Supérieure de Lyon
Université Claude Bernard Lyon I

Stage 2019–2020
Fabio Mensi
M2 Physique - Systèmes complexes

Deep learning methods and graph-based approaches for traffic speed prediction in the Lyon area

Summary : *In the present work, the problem of traffic speed prediction is explored in all its phases, from data processing to forecast analysis. A detailed pipeline is described covering the necessary steps to obtain a meaningful subset of roads with which to tackle the prediction problem. A series of baselines has been implemented, from traditional techniques to machine learning approaches, to offer comparisons with current and future state-of-the-art methods. Finally, graph neural networks are introduced in the context of traffic speed prediction, with a focus on the recently proposed Diffusion Convolution Recurrent Neural Network (DCRNN), a reference in this field. This deep learning architecture has been carefully analysed, adopted and partially reimplemented for further developing purposes.*

Key words : *graph neural networks, machine learning on graphs, traffic prediction, time series forecasting, complex systems*

Internship supervised by :

Rémy Cazabet
remy.cazabet@gmail.com
Univ de Lyon, CNRS, Université Lyon 1, LIRIS, UMR5205 Villeurbanne, France
20, Avenue Albert Einstein, 69621 Villeurbanne, France
https://liris.cnrs.fr/

Angelo Furno
angelo.furno@univ-eiffel.fr
Univ. Lyon, Univ. Gustave Eiffel, LICIT UMR_T9401, Lyon, France
25, Avenue François Mitterand, 69675 Bron, France
https://www.licit.ifsttar.fr/

Acknowledgments

I would like to express the deepest appreciation to the ENS de Lyon institution, its professors and my fellow students, who all have contributed in the making of an inclusive and challenging environment, that I am grateful to have been a part of. In particular, I would like to thank Professor Pierre Borgnat, the responsible for the M2 Complex Systems track, and Professor Cendrine Moskalenko, whose guidance during an experimental laboratory project has been very important.
My profound gratitude goes to my supervisors, Professor Rémy Cazabet and Professor Angelo Furno. I cannot be grateful enough for their support, both academical and on a human level, that I have found crucial, especially during the lockdown period. My appreciation also extends to the people I met at the LIRIS and LICIT laboratories, who made me feel welcome and part of the team despite the brevity of my in-presence work, and to LabEx IMU, for funding this project.
I want to thank the wonderful friends that I have made during my stay in Lyon, with whom I shared laughs, thoughts and meals, the unforgettable *gnari* from my hometown and all the serendipitous encounters I made during my journeys.
Finally, I cannot put into words the gratitude I have towards my family, who has been supportive and comprehensive about my sometimes-questionable choices, and I will never cease to be proud to be a part of it.

Contents

1 Introduction

Traffic forecasting covers a central role in the context of Intelligent Transportation Systems (ITSs). Accurate predictions may contribute to increase the efficiency of traffic operation tasks and driver's route-planning, as well as to reduce carbon emissions and traffic congestion related problems. In this regard analytical modeling has been widely used to address driving routing and signals timing optimization but, with the increasing availability of traffic data and computational power, data-driven approaches are currently being explored [1].
Time-series traffic data, consisting of flow, occupancy and speed of road segments, derives from processes where spatial and temporal correlations, as well as the underlying road network topology, are of major importance.
Traditional methods, like ARIMA [2] and VAR [3], often rely on restrictive assumptions on the data and may fail to capture non-stationarity, multimodality and the aforementioned spatio-temporal dependencies. Recently developed deep learning techniques, able to preserve and exploit the road network structure, are suitable candidates to improve traffic forecasting capabilities and with this study we investigate their application to speed prediction, analysing floating car data acquired in the Lyon area. The aim of this project is to analyse such methods critically and discussing whether their application is meaningful with the respect to the problem they intend to solve. On this line of thought, data processing and filtering pipelines are applied in order to focus the study on roads whose speed forecasting is a challenging task, subjected to heavy congestions and where traditional methods are expected to fail more often.
A series of baselines have been implemented, from traditional techniques to machine learning approaches, to offer comparison and introduction to more sophisticated methods, relying on the idea of Recurrent Neural Networks (RNN). We introduce Graph Neural Networks (GNN), a class of neural network architectures designed to address tasks where data lies on graphs. In particular, we provide a detailed study of the Diffusion Convolution Recurrent Neural Network (DCRNN), a recently proposed method aimed at combining both the concepts of RNNs and GNNs, along with a reimplementation of its basic architecture and suggestions for further developments.
Finally, we provide the forecasting performance of all the introduced methods, with respect to commonly used error metrics.

2 Data exploration

2.1 Data description

Data is provided by BeMobile and it covers traffic speed of Lyon, urban and peripheral areas, of a year, from October 2017 to September 2018. The total extent of area can be seen in figure 1, inscribed in a rectangle whose coordinates are shown in table 1.
The original data consisted in floating car data (GPS trajectory measurements) which is generated, on an average working day, by roughly 20000 vehicles, non-homogeneously distributed, both spatially and temporally.
The area covered is divided into 317693 road segments, characterized by a unique identifier and different road attributes, such as length, free flow speed (ffs), position coordinates, road structure information, functional road class (frc) and netclass. The last two features describe the type of road and its importance in the overall road network.

Latitude Sud : 45.5154189	**Latitude Nord** : 45.9954442
Longitude West : 4.5817533	**Longitude Est** : 5.1722470

Table 1: Lyon area coordinates

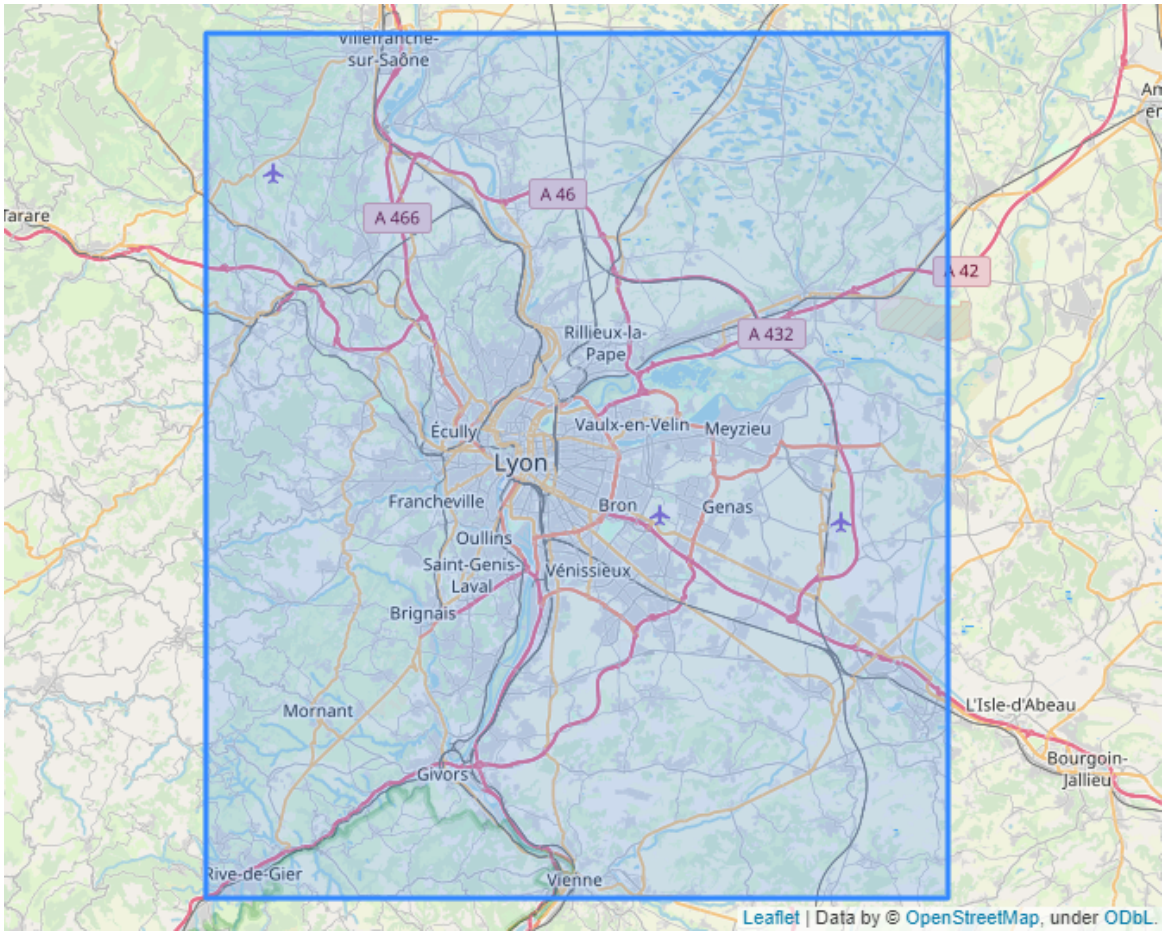

Figure 1: Lyon area map

2.1.1 Speed dataset

BeMobile extracted from raw floating car data an estimation of the speed measured on each road and aggregated in 3-minute time slots. In figure 2 an example of a speed time series, along with its road features.

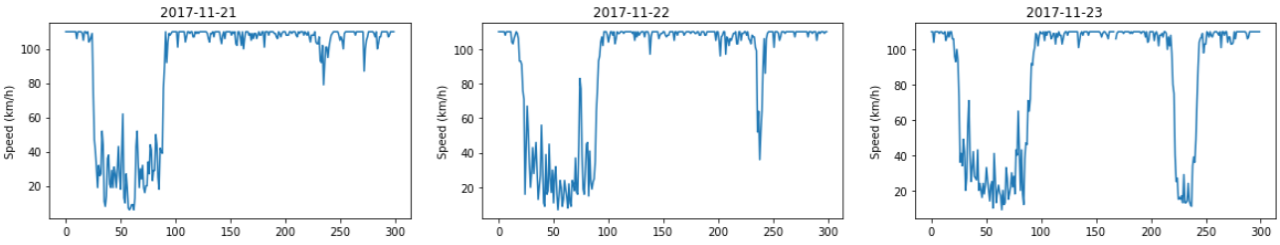

Figure 2: Speed data for three consecutive days, on the x axis the time instances, separated by 3 minutes. 0 := 5:00 and 300 := 19:57 for road 12500009456155 – length: 579 m, ffs: 110 km/h, area: Chaponnay.

It is worth mentioning that the measured speed never surpasses the free flow speed value. This is valid for every road and every time instance. Most probably during the aggregation process the speed value is capped in order not to be sensible to occasional speeding drivers rather than an impeccably lawful behavior by Lyonnais drivers. While this is a deliberate and legitimate choice it may not always represent the true speed on the road segments.

2.2 Graph Road Network

Road network is naturally represented as a graph G, where each point of interest (most often an intersection, but it may simply be a point of the road in case of highways or long roads) is a node, and each road segment is a directed edge, connecting two nodes if and only if there exist a direct connection between the two.
In the present study, to address the problem of traffic speed forecasting, it is more useful to switch to a line graph G_L representation. Starting from the original graph G the line graph G_L is constructed following these rules:

- Each node of G_L represents an edge of G
- Two vertices of G_L are adjacent if and only if their corresponding edges in G are incident

This procedure is done so that the road features can be represented as node attributes and the speed values as node signals, a common representation in the framework of GNN for traffic speed prediction. The graph G for the Lyon network counts 156466 nodes and 317693 edges. The derived line graph G_L counts 317693 nodes and 746888 edges.
In the following discussion we will use the words "node" and "edge" to refer to the entities of the line graph G_L, as they are the most suitable description of the road network.

2.3 Road selection

Selecting the nodes to analyze is a crucial step and it must be driven having in mind the aim of the project.
In order to apply and check the performances of GNN like algorithms on the Lyon speed dataset we should select road segments which present a well-defined connectivity (to be represented as graph) and that are in a reasonable quantity (for computational reasons), few hundreds at most.
Moreover, the nodes should present interesting behavior, whose prediction is not a trivial task. For example, if the speed can be forecasted with a sufficiently good accuracy based only on an historical average then a GNN approach becomes unnecessary. These goals may sometimes conflict with each other, so compromises are to be made.
Nevertheless, simple selection criteria have been preferred in order to make the road selection pipeline adaptable to other case scenarios.

2.3.1 Early selection (missing values)

As it happens with most datasets, the speed dataset suffered extensively from missing and noisy data. Because it was generated starting from the trajectories followed by 20000 cars each day, on average, some of the road segments, the nodes of G_L, did not have a single measurement for entire days. This made them inadequate for any speed forecasting purpose. Moreover, during nighttime the number of measurements dropped significantly for every road, since traffic becomes sparser or vanishes. In urban areas traffic lights or other factors have an impact on the speed measurements, that in some cases may not be representative of the actual traffic conditions. Time series of these road segments appear noisy and uninformative therefore a change of framework may be necessary in analyzing them. As a result of a coarse data exploration process, a restriction on the data was applied, considering only daytime, from 5.00 AM to 19.57 PM (300 time slots per day). This is in line with the aim of the project, since congestion phenomena appear in this time interval. Based on a week of data, assumed to be representative of the entire dataset, we firstly restrict our study to those roads presenting at least 90% of actual measurements (non-missing values) in the period mentioned before. The number of nodes satisfying this criterion is 11404 out the initial 317693 (3.6%).

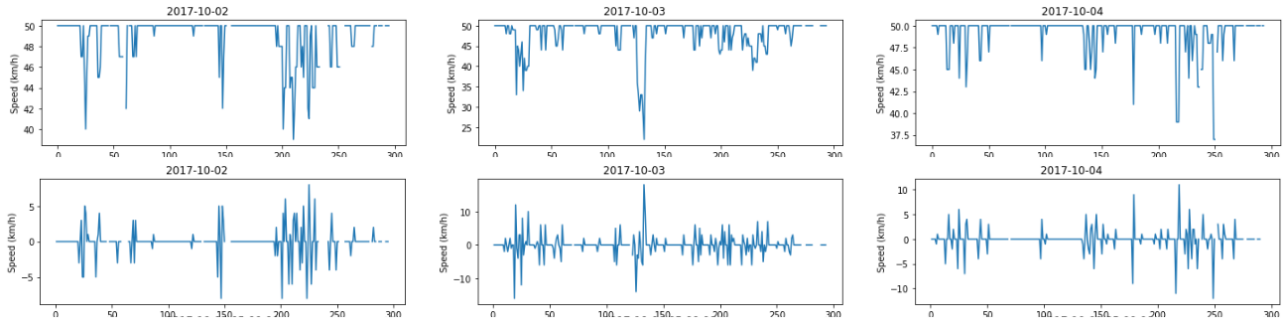

Figure 3: Speed data for three consecutive days (up) and its first difference (down). On the x axis the time instances, separated by 3 minutes. 0 := 5.00 and 300 := 19.57 for road 12500009271034 – length: 126 m, ffs: 50 km/h, area: Villeurbanne.

2.3.2 Selection based on non-stationarity

A road segment having interesting characteristics must be non-stationary. The time series of its speed values will not have a constant mean and variance over time[1] if there are congestions or seasonal patterns, as it happens with the speed drop associated to peak hours. However, the goal of this selection step is mainly to exclude nodes that present highly noisy time series (white noise is a stationary process by definition). Nodes with very regular time series, e.g. constant speed, will also be considered stationary but may be kept in further steps to ensure connectivity. A selection based on a stationarity criterion can be implemented via the KPSS (Kwiatkowski–Phillips–Schmidt–Shin) hypothesis testing [4]. The KPSS test is based on the following hypothesis:

- H_0 (null hypothesis): the time series is originated from stationary processes (possibly presenting a trend).
- H_1 (alternative hypothesis): the time series is originated from non-stationary processes.

Nodes whose time series are most likely non-stationary will reject the null hypothesis. The test was conducted on the time series associated to the nodes that have passed the early selection, considering data from working days ranging from October 2017 to February 2018. Whenever data was found to be missing linear interpolation between neighboring time slots was adopted. 2652 nodes out of 11404 (23.2%) had the null hypothesis rejected, with a p-value less than 0.05. KPSS is known to have a high rate of type I error so a best practice is to combine it with an ADF (augmented Dickey–Fuller) test. However, in this context, the aim was not to have a strict selection so the adoption of only KPSS was considered sufficient.

2.3.3 Selection based on first difference

The first difference of a time series is the series of changes from one period to the next. In our context it means the change of the measured speed from a 3-minute time slot to the next. In figure 3 an example of a time series with its corresponding first difference.
Traffic conditions usually are stable on a 3-minute time scale, there might be sudden drops due to peak hours or accidents or increases but on average the first difference is expected to have low values. By considering data from working days ranging from October 2017 to February 2018 an average of the

[1]These are some of the necessary conditions for a time series to be considered stationary. To compute the mean and the variance of a time series, in the sense of the stationarity definition, we would need to have multiple time series instances generated from the the same stochastic process. The mean and the variance would then be computed over those multiple time series, for each of the time slots composing the time series. If the mean and the variance does not depend on the particular time slot then the time series may be stationary, otherwise it is not. In practice we do not have multiple time series for the same process (one could consider multiple days as multiple time series but it would not be as rigorous) so stationarity testing does not follow strictly the definition.

first difference was calculated for the 11404 road segments that passed the early selection. Normalized data was used, the speed values were divided by ffs (free flow speed) and whenever data is found to be missing, a value of 1, equivalent to a change of speed of ffs, is imputed.

2.3.4 Hybrid selection

A selection based on the previously described criteria was conducted with the objective to select nodes holding meaningful and exploitable data. Among the 11404 road segments that passed the early selection, those who satisfied all the following conditions were kept:

- Have the H_0 of KPSS test rejected or the average first difference is less than 10% of the ffs.
- Missing values do not exceed 15% of the data (working days from Oct 2017 to Feb 2018).
- The average first difference is less than 20% of the ffs.

These conditions can be synthetized by saying that the nodes containing valuable information are the most stable (first difference less than 10% of ffs) or the most non-stationary (H_0 of KPSS rejected) with a softer condition on the first difference (less than 20% of the ffs). The condition on the missing values aimed at ensuring data richness on a larger time interval (from Oct 17 to Feb 18). 4393 out of 11404 (38.5%) satisfied these conditions and are kept for later analysis. It is worth noting that most roads of urban areas were excluded after this selection.

2.4 Spotting congested areas

The selected roads should have useful features for a forecasting analysis. It is now important to find what nodes present the most interesting behavior, i.e. that are more prone to congestion phenomena. A simple way to do this was to count how many times the measured speed fell under a third of the usual ffs (heavy congestion). The usual interval, Oct 17 to Feb 18, was adopted and in figure 4 the selected nodes having highlighted the areas containing the nodes with most congestions, while in figure 5 the map having marked the 100 nodes with the heaviest congestions.

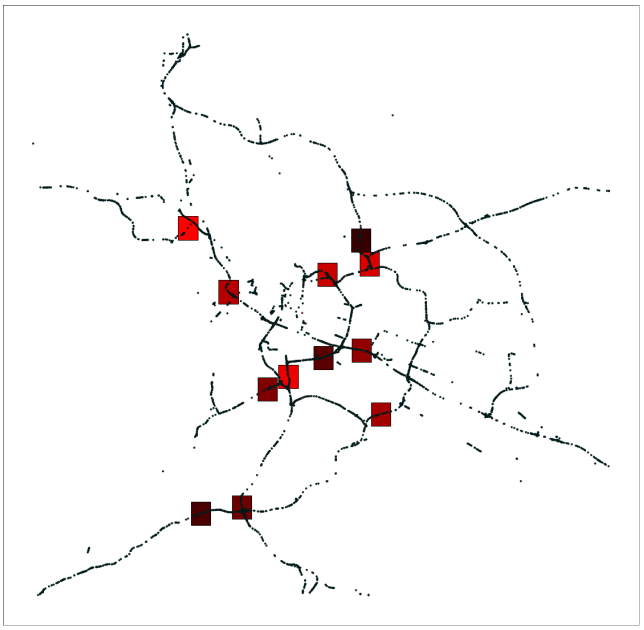

Figure 4: Selected nodes are represented as black dots. The red squares denote the areas where most congested nodes have been detected, the brighter the more congested.

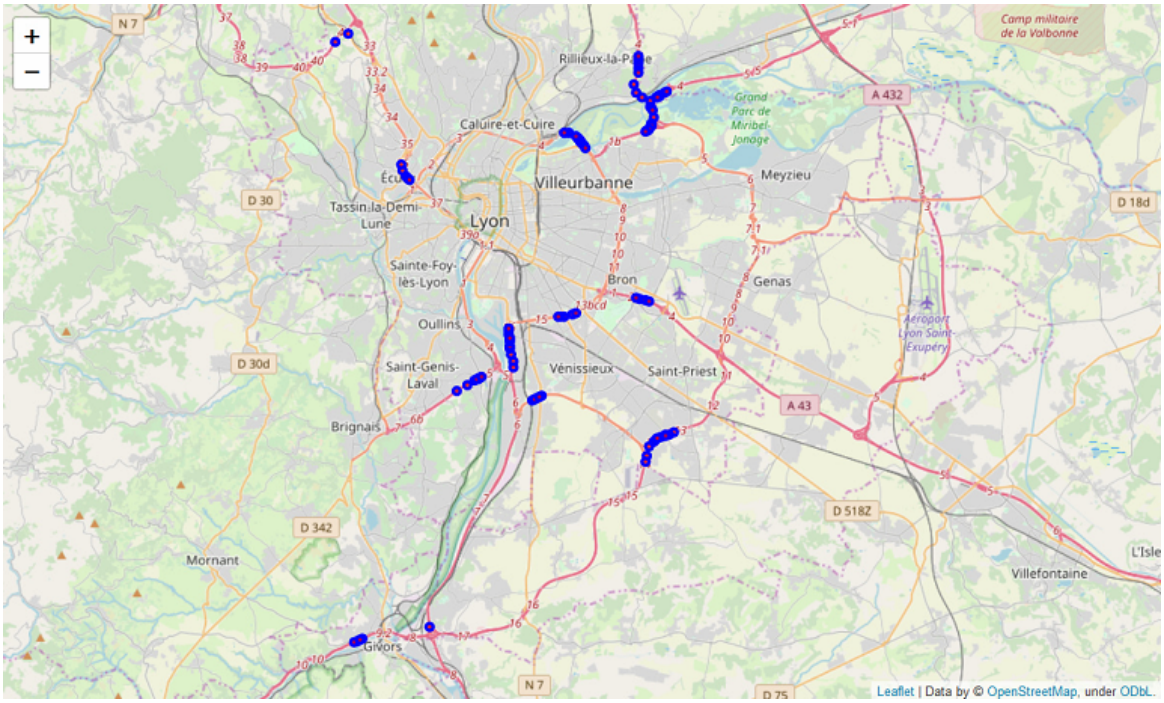

Figure 5: The blue markers indicate the 100 nodes with the most congestions, they are all contained inside the red squares of figure 4

As it can be seen, most of the congested nodes are placed in close proximity of intersections and exchanges of highways or other major roads. The focus now is to build a graph with a well-defined connectivity that is able to include most of the areas depicted in figure 4.

2.5 Subgraph building

To build a connected (at least weakly) subgraph that includes nodes in the areas shown in figure 4 starting from G_L it is necessary to solve mainly two problems:

- Nodes are often spatially dense with some road segment shorter than 20 meters: if all the nodes contained in a path connecting two interesting areas were to be kept then the resulting subgraph will count too many nodes (the ideal order is around one or two hundreds)
- G_L is weakly connected in its entirety but not restricting to the subset of the road segments after selection (more than one connected component)

To deal with the first problem an approach of graph simplification could be adopted but most methods rely on arbitrary decision combining notions of centralities and geometric considerations in the initial topology. Moreover, they are found to be very dependent on the specific road network, thus an implementation ad-hoc and not generalizable would have been required. The second problem is linked to the first one and it needs careful thinking on how to find a connection between the components.
A simple approach, aimed at solving both questions, is to assign to each edge of G_L (not a road segment) a weight based on the free flow travel time (fftt) of the two nodes (road segments) that the edge connects. For example, given two connected nodes, roadA and roadB, the corresponding edge is defined by a tuple (roadA, roadB). Its weight is calculated by summing the fftt of roadA and roadB, with fftt(road) being calculated as length(road)/ffs(road). This quantity is calculated for each of the 746888 edges of G_L and it is considered as a weight, with an added penalty if it connects a node not belonging to the selected ones. Then, shortest paths are calculated between representative nodes of the areas in figure 4 (one per area) using Dijkstra algorithm and weights as defined before. A subset of these shortest paths, based on how many non-selected nodes they contain, is chosen to build a

subgraph G_s. For each of these shortest paths, nodes are progressively added if they satisfy one of the following conditions:

- They are representative nodes
- They belong to intersections with other shortest paths
- They are farther than a threshold distance from the last added node.

If they do not meet at least one of the conditions nodes are skipped, otherwise they are added and an edge is consequently added between the node and the one added right before it. By setting the threshold distance to 700m the resulting subgraph, directed and weakly connected, counted 180 nodes and 189 edges, in line with the desired size and connecting most of the areas of interest. The topology and representation on map are shown in figure 6.

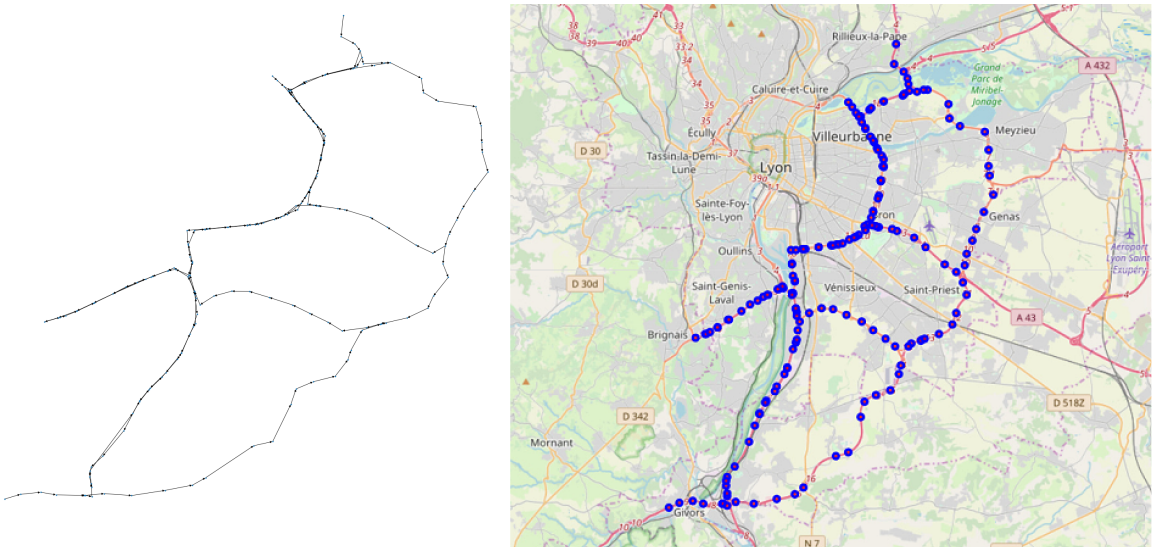

Figure 6: The subgraph G_s, its topology (left) and representation on map (right).

3 Forecasting

Traffic speed forecasting of a road segment aims at predicting, given *information* up to a certain point in time t, the speed measured on the road segment **at/up to** time $t+T$ with the highest *accuracy*, where T is called forecast horizon.
This definition requires further clarifications: given information could have a wide range of meanings, for example the measured speed on the road segment of interest, the measured speed on its surroundings, or other external notions, like road features and time context. Also, accuracy does not have a unique interpretation, various error metrics exist to give different notions of performance of the forecasting algorithm. The forecasting output depends on the choice between the prepositions in bold in the previous paragraph: time series are discrete in time since they come from an aggregation process and the forecast horizon may not correspond to a single time-step. In this case, if we forecast **at** time $t+T$ the algorithm returns a scalar, corresponding to the speed at time $t+T$, possibly n time-steps after t. On the contrary if we forecast **up to** time $t+T$ the algorithm returns multiple values (a vector), corresponding to the speed from time $t+1$ to time $t+T$. While discussing the implementation of various data-driven algorithms it is important to cast away any ambiguity on the choice in our definition of forecasting. In the present study when referring to accuracy, the following metrics are adopted:

- **MAE**, mean absolute error:

$$\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

. It measures the average absolute deviation of forecasted values $\hat{y}_i$ from the true ones y_i.

- **MSE**, mean squared error:

$$\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

. It measures the average squared deviation of forecasted values. Compared to MAE it penalizes more large errors.

- **MAPE**, mean absolute percentage error:

$$\frac{1}{n}\sum_{1}^{n}\frac{|y_i - \hat{y}_i|}{yi}$$

. It measures the percentage of the average absolute error.

As for the information exploited for the prediction and the output form of the algorithms, they will be specified when needed. The present study focuses on short-term forecasting with a maximum horizon of 1 hour, equivalent to 20 3-minute time steps.
To have an estimate of the improvements brought by a particular algorithm it is important to have the performances of commonly implemented techniques as references, i.e. baselines. In this context, in line with the literature, we selected data-driven algorithms belonging to a more classical time-series analysis approach, like ARIMA (AutoRegressive Integrated Moving Average) and VAR (Vector AutoRegression) models, and others belonging to the machine-learning area, such as FNN (Feedforward Neural Network) and LSTM (Long-Short Term Memory). Moreover, naive approaches, like naive forecasting and historical average, are considered, since they are readily available and provide useful indicators of the hardness of the prediction task.

3.1 Data for experiments

Our study was conducted by considering data from working days of October and November 2017. The decision was led by a compromise between data availability, the more data we use the more a deep learning approach is preferable, and computational resources, the more data we use the higher the computational burden. Moreover, October and November were considered rather homogeneous with respect to the underlying traffic phenomena. The various algorithms were trained and validated using data from October and the first half of November while performances were evaluated using data from the second half of November.

3.2 Naive approaches

Two basic methods to make a prediction are historical average and naive forecasting. Historical average is calculated by averaging over all training days the speed measured for each time slot. The process is run independently for each road segment. For example, the speed prediction of 5 AM on road A, is obtained by averaging the speed values of 5 AM on road A over all the training days. A difference was not made between different days of the week since it gave overall worse performances (This may differ by selecting other time intervals for training and test). As a result, we have the prediction that is independent of the actual traffic conditions but relies only on a hypothetical daily regularity of traffic. If good performances are achieved with this method, then adopting complex forecasting algorithms may not be necessary. The naive forecast method predicts the speed at time $t + T$ to be equal to the current speed, at time t. In other words, we predict the future speed to be equal to the speed we are currently measuring. As the horizon increases the performances will necessarily worsen, but it is

important to know the trend for all the metrics. In figure 7 the forecasting error metrics for historical average and naive forecasting, averaged over the 180 nodes of G_s, as a function of the horizon T, in table 2 numerical values for selected horizons. The errors are calculated considering only the difference between the real value at time $t+T$ and the corresponding forecast, not the time slots in-between. (A complete table with all the presented methods is presented in the Results section) When focusing on particular areas, those trends will vary, and will be presented when needed.

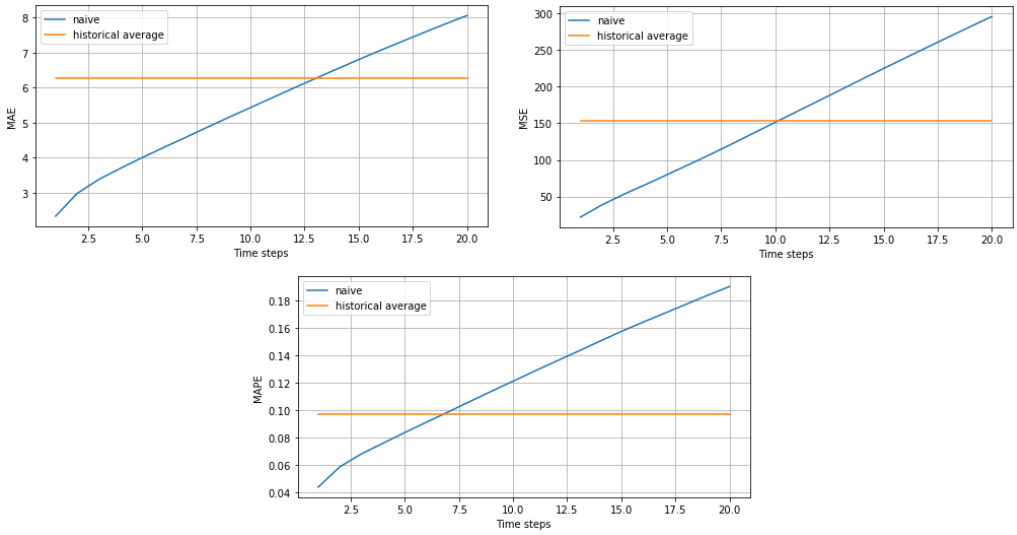

Figure 7: Error metrics for historical average and naive forecasting, averaged over the 180 nodes of G_s. The historical average does not depend on the horizon and it is calculated by averaging along the working days in the training set.

	15min	30min	45min	60min		**Hist Avg**
Naive MAE	4.01	5.43	6.81	8.06		6.27
Naive MSE	79.83	151.48	224.93	295.93		152.83
Naive MAPE	8.37%	12.11%	15.73%	19.00%		9.70%

Table 2: Naive forecasting and historical average performances

3.3 Traditional statistical approaches

3.3.1 ARIMA

ARIMA (AutoRegressive Integrated Moving Average) is a class of models suitable for the forecasting of a stationary time series (or that can be made stationary via differencing). An ARIMA model is defined by 3 integers (p, d, q) where p is the number of autoregressive terms, d is the number of differences needed for stationarity, and q is the number of lagged forecast errors in the prediction equation. It is a generalization of an ARMA (AutoRegressive Moving Average) which, on the contrary, is defined by only p and q.
The forecasting equation for an ARMA(p,q) model is defined as:

$$\hat{y}_t = c + \phi_1 y_{t-1} + ...\phi_p y_{t-p} + \theta_1 e_{t-1} + ... + \theta_q e_{t-q} \tag{1}$$

An ARIMA model has the same equation, with a different meaning for y_t: while in ARMA it represents the speed value at time t, in ARIMA(p,q,d) it represents the value at time t of the d-differenced time series. The model is fitted accordingly and the actual speed value is retrieved after forecasting. To implement the ARIMA algorithm the python library `statsmodels` was used and missing values were imputed via linear interpolation. For the choice of the parameters (p, d, q) we selected via grid-search

(p:[0,1,2,3] d:[0,1] q:[0,1,2]) the model with the lowest AIC (Akaike information criterion [5]) after training on the training set. The selected model was then evaluated on the test set by feeding data up to time t to predict the speed at time $t+T$, via successive one time step ahead predictions[2]. The procedure was run independently for each of the 180 nodes, for different forecasting horizons.

3.3.2 VAR

VAR (Vector AutoRegression) is a class of models aimed at capturing linear interdependencies among multiple time series. It is a generalization of AR(p), or ARIMA(p,0,0), where p is the order of the model.
The forecasting equation for a VAR(p) model is defined as:

$$\hat{Y}_t = C + \Phi_1 Y_{t-1} + ... + \Phi_p Y_{t-p} \tag{2}$$

Differently from ARIMA, the forecasted quantities are vectors $\hat{Y}_t$ corresponding to predicted speed for the 180 nodes of G_s. The Φ terms in the previous formula are matrices of size (180,180) and describe the linear dependencies among the different road segments. In the simplest model, VAR(1), the speed value for a road segment at time t is forecasted as a linear combination of the speed values of every road segment at time $t-1$. To implement the VAR algorithm the python library `statsmodels` was used and missing values were imputed via linear interpolation. VAR models of different orders were trained using the training set. The evaluation was conducted on the test set, by feeding data up to time t to predict the speed at time $t+T$, via successive one time step ahead predictions.

3.4 Machine learning approaches

All the methods introduced in this subsection were implemented using the `Tensorflow`, a popular machine learning library [6], and the deep learning API `Keras`.

3.4.1 Machine Learning dataset format

The methods presented hereafter are different classes of neural networks. Deep learning frameworks require data to be organized in a standard way, therefore, in order to avoid ambiguities, we briefly describe its structure.
In machine learning, a dataset is a collection of instances, each characterized by an input $\mathbf{x}$ and an output y. A machine learning model is defined and trained to best match the inputs to the corresponding outputs. In our case the inputs and outputs have a well-defined structure: in general the input of an instance is a vector of length 20, containing speed values of an hour on a specific road segment; the output is a scalar representing the speed value on the same road segment, n steps ahead[3].
These definitions imply that for different road segments and horizons T we have different datasets, hence different models, for the same method: a model will be trained to predict 5 steps ahead, another to predict 10 steps ahead, etc... . This is repeated for every road segment of interest.
The choice was led by the need of removing ambiguities on the forecasting part and provided an easy solution on how to deal with missing values in a coherent way. Every dataset is built such that, for each instance, the output y is a measured value. The corresponding input is allowed to have missing values and they are imputed through linear interpolation. A major downside is the computational power needed for training numerous models.
A few exceptions can occur in the definition of a data instance, see appendix A for a more in-depth discussion.

[2]For example, to predict the speed at horizon 3, I predict the speed at horizon 1 according to equation 1 (adapted to the ARIMA case), then, to predict the speed at horizon 2, we still employ equation 1 where the predicted value for the speed at horizon at horizon 1 is inserted as the term y_{t-1} in the right hand side, instead of the unknown true value. The same procedure is then repeated for calculating speed at horizon 3, but having the speed at horizon 2 as y_{t-1} and the speed at horizon 1 as y_{t-2}. The notion of successive one time step ahead prediction can be easily generalized to a generic horizon T and it is also shared by the VAR method.

[3]n = 1 corresponds to 3 minutes ahead, n = 5 to 15 minutes, n = 10 to 30 minutes, etc...

3.4.2 Feedforward Neural Network

FNN (Feedforward Neural Network) is the simplest deep learning architecture and one of the main building blocks for more complex models. It consists of a series of fully connected layers, each of which can be viewed as a function that maps a vector of dimension m to another one of dimension n. Each dimension is also called "neuron", from an analogy with the biological cell introduced with the early perceptron model.
Equation 3 shows the mathematical representation of a fully connected layer.

$$h_{W,b}(x) = \phi(W^T x + b) \tag{3}$$

with x input vector ($x \in \mathbb{R}^m$), W weight matrix ($W \in \mathbb{R}^{(m,n)}$), b bias term ($b \in \mathbb{R}^n$) and ϕ a nonlinear function (such as Rectified Linear Unit (ReLU), sigmoid, tanh, etc...), generating the output vector h ($h \in \mathbb{R}^n$).
In a FNN, the first layer is associated with the input data instances, the last layer with the outputs, while layers in-between are called hidden layers. In figure 8 a schema of a FNN with one hidden layer.

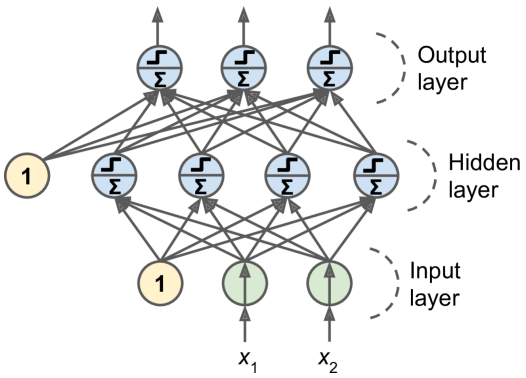

Figure 8: One layer FNN architecture, bias term is explicitly represented as a neuron of constant value 1. (Image taken from [7])

According to the universal approximation theorem [8] a FNN having just one hidden layer is capable of representing any function. In practice, since there is not a theoretically grounded criterion, the number of layers should be regarded as an hyperparameter, to be set according to the particular application[4].
The weight matrix W and the bias term b are parameters that are learned during the training of the model. The training is based on the notions of loss function, an error metric that the model will try to minimize (in an image classification setting the loss function could be the number of wrongly classified images) and backpropagation, an algorithm according to which the matrix and bias values are updated. Backpropagation, also called automatic differentiation, does that by computing the gradient of the loss function with respect to every learnable parameter of the neural network. A full description of backpropagation and gradient-descent like algorithms is not in the scope of the present work; also every major deep learning library does not require a manual implementation of the most common optimization procedures.
In our traffic speed prediction context, the input data is a vector of dimension 20, i.e. an hour-long time series, while the output is a scalar (a single neuron layer) corresponding to the speed at the prediction horizon. The error metric to be optimized is one the metrics listed before, such as MAE, MSE or MAPE.
Our FNN forecast, was implemented using an architecture having 2 hidden layers of 32 neurons each and optimizing the mean average error.

[4]State-of-the-art performances in image classification, object detection and other computer vision tasks are achieved with deep architectures, involving models more complex than a FNN.

3.4.3 Recurrent Neural Network

Recurrent Neural Network (RNN) is a class of deep learning models designed to process sequences. One of their many applications is in the field of Natural Language Processing (NLP) and machine translation.
They are relevant to our case scenario since traffic data consists in time series and RNNs are one of the main machine learning approaches used to model time dependencies.
In its simplest form an RNN layer is constituted by a cell that accepts as input the signal x at time t and the hidden state (that contains relevant information from previous time steps) from $t-1$, and returns an output y and the next hidden state[5]. Equation 4 describes its mathematical representation.

$$y_{(t)} = \phi(W^T[x_{(t)}; y_{(t-1)}] + b) \tag{4}$$

with $x_{(t)}$ input vector at time t ($x_{(t)} \in \mathbb{R}^m$), $y_{(t)}$ output vector at time $t-1$ ($y_{(t-1)} \in \mathbb{R}^n$), W weight matrix ($W \in \mathbb{R}^{(m+n,n)}$), b bias term ($b \in \mathbb{R}^n$) and ϕ a nonlinear function, generating the output vector y at time t ($y_{(t)} \in \mathbb{R}^n$)[6]. A single data instance is then composed by an input x having dimensions (sequence length, m) and an output y having dimensions (sequence length, n) or simply n, depending on the specific context.
Figure 9 provides a graphical description of an RNN layer.

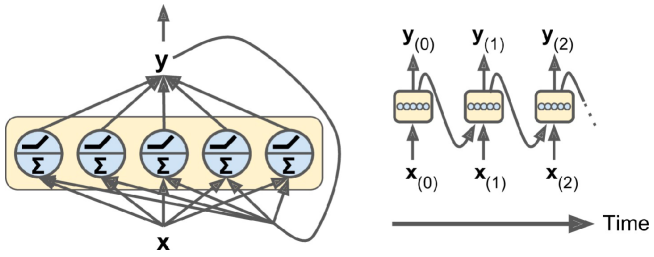

Figure 9: An RNN layer shown in its looped version (left) and unrolled through time (right). (Image taken from [7])

The same concepts of model training, loss function and backpropagation discussed for FNN still applies with RNN, whereas the technical implementation can differ.
In our study we have inputs of fixed sequence length (20 time steps) whereas the signal can be of dimension 1, if we consider just the traffic speed of a node of interest, or greater than 1, if we consider the traffic speed of the neighboring road segments as well. The model output is conceptually different from the output described in equation 4. The output of the RNN layer is to be intended as a hidden state, keeping track of the state of the traffic conditions, whereas the model output is the same as the one of the FNN architecture. To pass from the output of the RNN, of dimensionality n, to the model output, a dense layer is added after the RNN layer (dense means fully connected, but without an activation function).
In our forecast we did not employ the simple RNN structure, but a more sophisticated one, based on the Long Short Term Memory (LSTM) cell, known to overcome the issues of unstable gradients and short memory present in the original RNN implementation.

3.4.4 Long Short Term Memory

LSTM is an evolution of standard RNN cell, first introduced in 1997 [9]. It has two hidden states, namely c and h, allowing to capture both long and short term context respectively, hence the name. It is

[5] Often the output and the hidden state are the same vectors.
[6] Generally $n >> m$

the one of main models, along with the recently proposed transformers, to address machine translation tasks. The system of equations of the LSTM cell is described by equation 5.

$$\begin{cases} i_{(t)} &= \sigma(W_i^T[x_{(t)}; h_{(t-1)}] + b_i) \\ f_{(t)} &= \sigma(W_f^T[x_{(t)}; h_{(t-1)}] + b_f) \\ o_{(t)} &= \sigma(W_o^T[x_{(t)}; h_{(t-1)}] + b_o) \\ g_{(t)} &= \tanh(W_g^T[x_{(t)}; h_{(t-1)}] + b_g) \\ c_{(t)} &= f_{(t)} \odot c_{(t-1)} + i_{(t)} \odot g_{(t)} \\ y_{(t)} &= h_{(t)} = o_{(t)} \odot \tanh(c_{(t)} \end{cases} \tag{5}$$

where g is the main layer, meaning that if g alone was present, then we would have returned to the basic rnn cell definition (equation 4). f, i and g, standing for *forget*, *input*, *output*, act as gate controllers, since their values are in the (0, 1) interval, due to the applied sigmoid function, and they are employed to compute element-wise multiplication (denoted by the $\odot$ symbol). Their names suggest their functionalities, in plain words [7]:

- The *forget gate* determines which parts of c, the long-term state, should be erased.
- The *input gate* determines which parts of g should be added to c.
- The *output gate* determines which parts of c should be sent to h.

The dimensionalities are analogous with the standard cell, with the output being the vector h and as learnable parameters the weight matrices (W_i,W_f,W_o,W_g) and the biases (b_i,b_f,b_o,b_g).
Our implementation involved a single layer LSTM (32 units each hidden state vector) with a dense layer on top, mapping the output of the recurrent layer with the model output and optimizing the mean average error.

4 Graph approaches

All the methods presented up to now are graph-agnostic, i.e. they do not consider explicitly the structure of the road network. Novel deep learning methods have been introduced to address those problems, like our study case, where data lies on graphs by exploiting their underlying topology [10]. Such problems arise in many domains: in chemistry, molecules are made of atoms linked via chemical bonds, in e-commerce, costumers and products are linked via their consumer relationship.
These methods are generally referred as Graph Neural Networks (GNN) and they are typically implemented to solve problems such as node classification, graph classification or link prediction.

4.1 Graph Neural Networks, general concepts

GNNs are based on the notion of message a passing, meaning that the embedding/signal associated with a node is obtained by combining the embeddings/signals associated with its neighbors. Given a graph signal X ($X \in \mathbb{R}^{(N,P)}$ with N number of nodes and P signal dimensionality), the message passing can be implemented by premultiplying X with a $(N \times N)$ "support" matrix, containing information about the network. Examples of support matrices can be[7]:

- The *adjacency matrix* A, whose elements A_{ij} are ones if there is an edge from node i to node j, otherwise they are set to 0.
- The *laplacian matrix* $L = D - A$, where D is the diagonal degree matrix.
- Any matrix that contains topological information, relevant to the problem that is intended to solve.

[7]Typically support matrices are normalized according to the node degree, in such a way that graphs with a heterogeneous degree distribution are processed properly.

Due to this matrix multiplication, this kind of GNN are often called Graph Convolution Networks (GCN), to make an analogy with the standard Convolutional Neural Networks (CNN), that dealt with data represented in Euclidean space (e.g. images). One of the first GCN examples was introduced in 2016, with an application to a citation network case study [11].
We have focused our attention on a particular GNN, namely Diffusion Convolutional Recurrent Neural Network (DCRNN), that has already shown promising results in traffic speed forecasting settings [12].

4.2 Diffusion Convolutional Recurrent Neural Network

DCRNN is neural network architecture specifically aimed at tackling spatiotemporal forecasting tasks. It was described and evaluated on a traffic speed prediction problem in the Los Angeles area, USA, reporting state-of-the-art performances.
As anticipated in section 4.1, to conform to the GNN framework, speed data[8] is represented as graph signals X_t laying on the road network, in our case the directed graph G_s, where its N nodes are road segments.
In figure 10 a graphical representation of the DCRNN architecture.

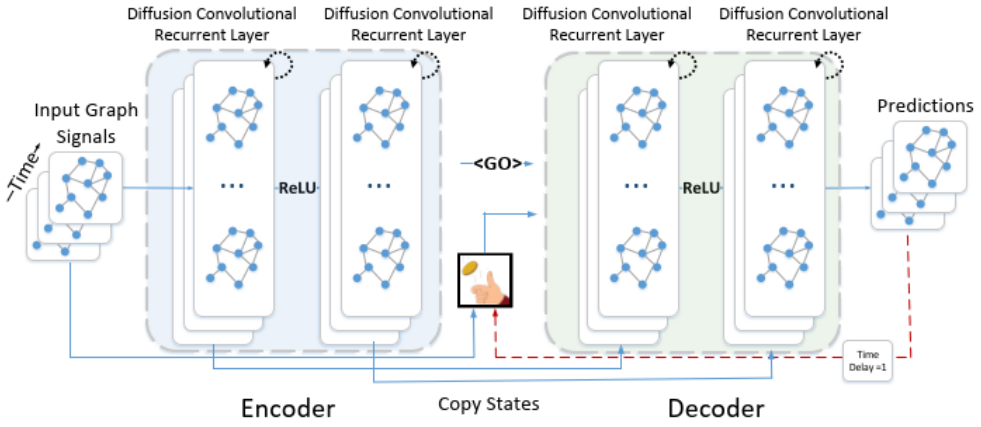

Figure 10: Model architecture for the Diffusion Convolutional Recurrent Neural Network [12], designed for spatiotemporal traffic forecasting. The speed time series are fed into an encoder whose final states are used to initialize the decoder. Scheduled sampling [13] was adopted to select decoder inputs. See appendix A.

4.2.1 Weighted adjacency matrix

Road segments differ in length, free flow speed and other features. Therefore, an edge of G_s represent a different connectivity depending on the nodes it connects and consequently, the standard adjacency matrix may not constitute the appropriate support matrix for message passing.
To address this problem, following the approach described by [12], we have built a weighted adjacency matrix W where each element W_{ij} is given by the thresholded gaussian kernel:

$$W_{ij} = \exp\left(-\frac{\overline{\text{dist}}(v_i, v_j)^2}{\sigma^2}\right) \text{ if } \overline{\text{dist}}(v_i, v_j) \leq \kappa, \text{ otherwise } 0 \tag{6}$$

where $\overline{\text{dist}}(v_i, v_j)$ is the adjusted distance between road segments v_i and v_j, σ is standard deviation of adjusted distances and κ is the threshold applied to the gaussian kernel. The adjusted distance is calculated starting from free flow travel time, retrieved with the same process described in subsection

[8] In [12] speed data was retrieved via road sensors.

2.5[9], and converted into a distance by fixing an arbitrary speed. In our case study we set this speed to 90 km/h and κ to 0.7.

4.2.2 Diffusion Convolution operation

Spatial dependency is modelled following an analogy with a diffusion process. Given the graph signal $X_t \in \mathbb{R}^{(N,P)}$ (if we consider as signal just the speed time series, then $P = 1$) and the diffusion transition matrices $D_O^{-1}W$, $D_I^{-1}W^T$ (D_O: out-degree diagonal matrix, D_I: in-degree diagonal matrix, W: weighted adjacency matrix of G_s) the convolution operation is defined as:

$$\boldsymbol{X}_{:,p} \star f_\theta = \sum_{k=0}^{K-1} (\theta_{k,1}(D_O^{-1}W)^k + \theta_{k,2}(D_I^{-1}W^T)^k)\boldsymbol{X}_{:,p} \text{ for } p \in \{1, ..., P\} \tag{7}$$

The filtered signal of a node i ($1 \leq i \leq N$) is the result of combinations of the signals of the neighbors in the graph up to a distance of K steps from the node, weighted by the parameters $\theta \in R^{K\times 2}$.

4.2.3 DCGRU cell

The diffusion convolution operation accounts for the spatial dynamics component of the problem and its implementation is directly connected with the temporal dynamics part.
The temporal dependency is modelled through a recurrent neural network variant, the gated recurrent unit (GRU) [14]. To combine spatial and temporal modelling each matrix multiplication operation is replaced by the the diffusion convolution operation, described in the previous equation. The resulting modified GRU cell, called DCGRU, can be defined by the following equations:

$$\begin{cases} r_{(t)} &= \sigma(\Theta_{r\star G}[X_{(t)}, H_{(t-1)}] + b_r) \\ u_{(t)} &= \sigma(\Theta_{u\star G}[X_{(t)}, H_{(t-1)}] + b_u) \\ C_{(t)} &= \tanh(\Theta_{C\star G}[X_{(t)}, (r_{(t)} \odot H_{(t-1)})] + b_c) \\ H_{(t)} &= u_{(t)} \odot H_{(t-1)} + (1 - u_{(t)}) \odot C_{(t)} \end{cases} \tag{8}$$

where $X^{(t)}$, $H^{(t)}$ are the input and output (or activation) at time t, $r^{(t)}$, $u^{(t)}$ are the reset gate and update gate, and $C^{(t)}$ is the candidate output, which contributes to the new output based on the value of the update gate $u^{(t)}$. An explanation of the roles of the different control gates can be found in the appendix B, along with a presentation of the standard GRU cell.

4.3 Reusable DCGRU

In order to allow future model developments, the DCGRU cell should be readily usable, independently from its original context. Starting from the original implementation, we have built the DCGRU cell using the latest version of `Tensorflow` in such a way that is compatible with `Keras` and does not rely on deprecated libraries.
Our goal is to provide code that allows for quick model upgrading and can be integrated with minimal effort. An attempt to meet this goal can be found at github.com/mensif/DCGRU_Tensorflow2, along with usage examples. See appendix C.
We have tested the newly implemented DCGRU layer on the prediction of a graph-based synthetic signal and compared it with other commonly used deep learning architectures. The DCGRU layer obtained the best performance, showing potential for employment in similar tasks.

[9] Without added penalties

5 Results

We present here the results obtained with the previously discussed methods. This section is divided into two categories: the first one is related to prediction on the whole subgraph G_s and the second one is focused on two subareas of interest. Due to computational constrains, deep learning baselines have been tested only in the second part. Nevertheless, we have obtained the DCRNN forecasts for both cases, by using its original implementation.

5.1 Subgraph forecasts

In table 3 the results obtained for all the road segments in the subgraph G_s (see figure 6).

T	Metric	Hist Avg.	Naive	ARIMA	VAR	DCRNN
15min	MAE	6.27	4.01	4.36	4.13	**3.23**
	MSE	152.83	79.83	85.84	69.33	**59.80**
	MAPE	9.70%	8.37%	9.63%	9.23%	**7.39%**
30min	MAE	6.27	5.43	5.78	5.05	**4.07**
	MSE	152.83	151.48	147.53	131.04	**101.82**
	MAPE	**9.70%**	12.11%	13.42%	12.07%	10.07%
45min	MAE	6.27	6.81	7.01	5.76	**4.74**
	MSE	152.83	224.93	204.57	190.09	**137.89**
	MAPE	**9.70%**	15.73%	16.64%	14.33%	12.52
60min	MAE	6.27	8.06	8.07	6.40	**5.29**
	MSE	**152.83**	295.93	256.92	232.44	167.13
	MAPE	**9.70%**	19.00%	19.35%	16.32%	14.54

Table 3: Forecasting performances comparison on the whole subgraph G_s

We can see that the DCRNN approach has the overall advantage against traditional graph-agnostic methods, especially for the MAE metric and for shorter time horizons. While for the metric the explanation may simply be that it was the error that the neural network was trying to optimize, the fact that historical average can be more accurate than DCRNN suggests that it may be fundamental to consider available information other than the previous hour speed data.

5.2 Limited area case studies

We have conducted a more in-depth study on two subareas, where at least one node per area was interested by heavy congestion phenomena. In total, 4 road segments per subarea were considered in the analysis, shown in figure 11.
In figures 12 and 13, the speed data for three consecutive days measured on two road segments, belonging to different subareas.
In tables 4 and 5 the results obtained for subarea 1 and subarea 2, respectively. The difference between LSTM and LSTM multi is that the second one considers as input at time t a vector containing the speed at time t of all the 4 road segments in the subarea. By referring to the RNN framework described in 3.4.3, for LSTM we have $x_{(t)} \in \mathbb{R}$, whereas for LSTM multi $x_{(t)} \in \mathbb{R}^4$.
The same conclusions derived from whole subgraph predictions still apply. Moreover, we can see that in the first subarea, the graph-agnostic LSTM multi often had the advantage over the DCRNN, suggesting that there could be improvements related to the message passing aspect of the DCRNN, strictly related with the weighted adjacency matrix employed in the method.

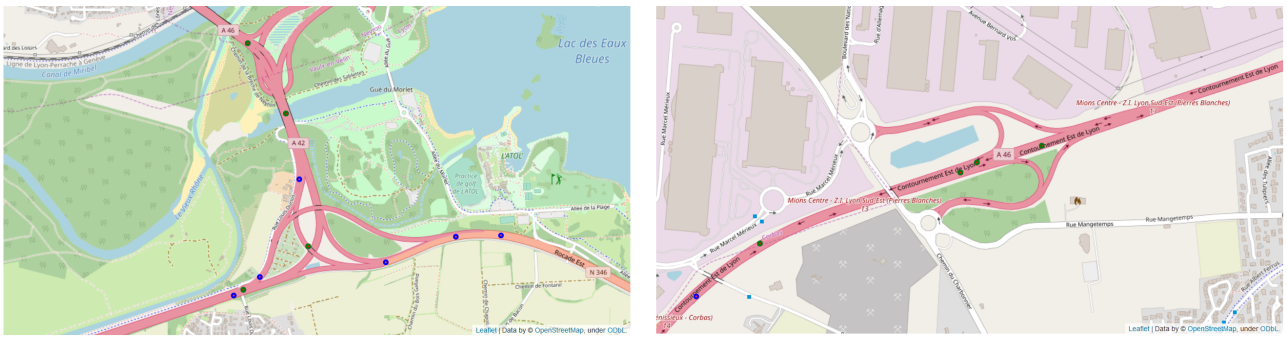

Figure 11: The two areas from which a sub sample of nodes of G_s have been selected (shown in green)

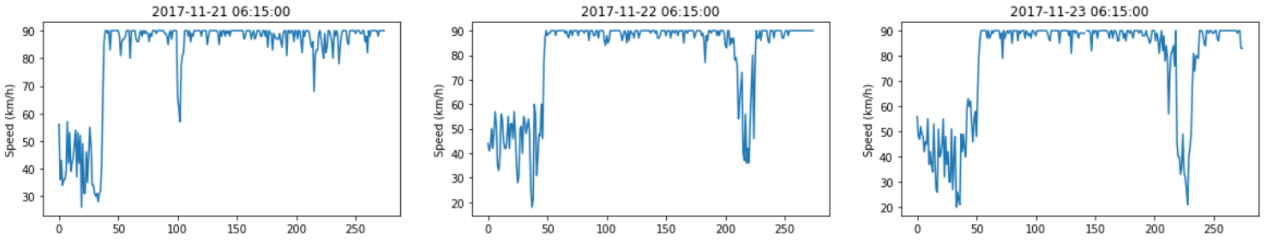

Figure 12: Speed data for three consecutive days, on the x axis the time instances, separated by 3 minutes. 0 := 6:15 and 275 := 19:57 for road 12500009382784 – length: 160 m, ffs: 90 km/h, area: Vaulx-en-Velin.

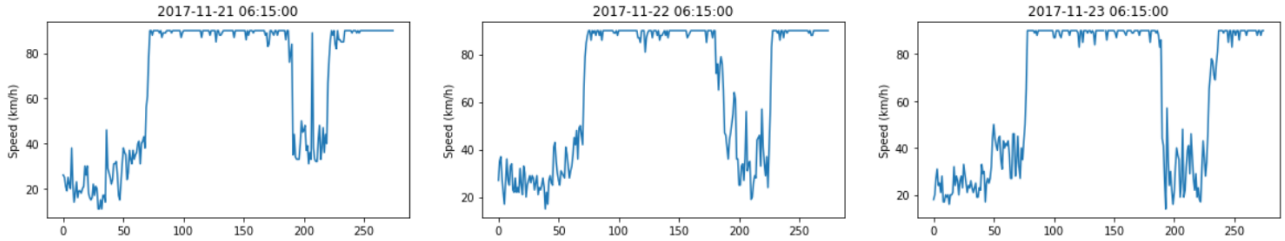

Figure 13: Speed data for three consecutive days, on the x axis the time instances, separated by 3 minutes. 0 := 6:15 and 275 := 19:57 for road 12500009363953 – length: 36 m, ffs: 90 km/h, area: Mions.

T	Metric	Hist Avg.	Naive	ARIMA	VAR	FNN	LSTM	LSTM multi	DCRNN
15min	MAE	7.92	4.70	5.06	4.84	4.43	4.54	4.16	**4.00**
	MSE	191.74	87.99	96.74	87.40	85.65	86.14	78.81	**72.89**
	MAPE	12.21%	8.68%	9.69%	9.27	9.29%	9.30%	8.56%	**7.49%**
30min	MAE	7.92	6.11	6.57	5.74	5.33	5.35	**4.80**	5.26
	MSE	191.74	157.45	161.41	139.74	127.01	118.39	**107.09**	135.20
	MAPE	12.21%	11.21%	12.68%	12.12	11.66%	11.39%	10.28%	**9.83%**
45min	MAE	7.92	7.51	8.00	7.34	5.86	6.17	**5.30**	6.51
	MSE	191.74	232.52	227.67	177.62	151.88	159.4	**129.14**	196.58
	MAPE	12.21%	13.64%	15.36%	13.64	13.61%	14.83%	**11.67%**	12.54%
60min	MAE	7.92	8.94	9.38	8.00	6.21	7.33	**5.63**	7.04
	MSE	191.74	314.54	295.20	254.85	176.13	257.19	**141.64**	228.17
	MAPE	**12.21%**	16.18%	17.82%	15.61	15.51%	20.34%	13.26%	14.39%

Table 4: Forecasting performances comparison on subarea 1

T	Metric	Hist Avg.	Naive	ARIMA	VAR	FNN	LSTM	LSTM multi	DCRNN
15min	MAE	8.25	3.58	4.39	3.98	3.46	3.66	3.35	**3.14**
	MSE	245.04	78.54	86.43	80.44	80.29	86.33	79.39	**77.69**
	MAPE	12.99%	8.12%	10.34%	9.82%	8.76%	9.04%	8.30%	**7.95%**
30min	MAE	8.25	6.04	6.29	5.43	4.71	4.85	4.60	**4.19**
	MSE	245.04	160.52	154.48	150.28	142.74	149.88	**140.15**	142.68
	MAPE	12.99%	11.11%	14.82%	13.57%	11.94%	12.25%	11.90%	**10.89%**
45min	MAE	8.25	6.50	7.93	6.57	5.85	6.37	5.75	**5.15**
	MSE	245.04	250.81	212.92	210.96	197.05	221.25	211.00	**198.70**
	MAPE	**12.99%**	13.87%	18.37%	16.12%	15.43%	16.15%	15.81%	13.70%
60min	MAE	8.25	8.02	9.24	7.38	6.58	7.22	6.70	**5.68**
	MSE	245.04	343.90	259.76	252.48	247.55	276.07	267.42	**235.12**
	MAPE	**12.99%**	16.42%	20.86%	17.45%	18.73%	20.36%	18.38%	15.54%

Table 5: Forecasting performances comparison on subarea 2

6 Related Works

Several deep learning methods have been implemented in a traffic forecasting setting, both on speed and flow prediction. Early attempts restricted to the forecast of roads of interests, in a way similar to the baselines described in section 3.4, and shown improvements over traditional techniques [15]. When dealing with larger scales, Convolutional Neural Networks (CNN) have been employed by converting the road network in a 2D grid [16] [17], resembling methods applied to computer vision tasks. Recent trends have found better performances in treating the road network as a graph by explicitly using adjacency-like matrices in their neural network architectures, hence GCNs [18] [19], to model spatial dependency. We decided to focus our attention on the DCRNN since it was one of the first attempts to model both spatial and temporal dependency, by combining RNNs with GCN computations [12] that led to state-of-the-art results on large partitioned road networks [20] and can be a starting point for further model developments.

7 Conclusion

To summarise, our work offers a detailed view of the traffic speed prediction problem, involving road network analysis, time series forecasting and implementation of recently proposed approaches, dealing with real-world data coming from GPS trajectories of Lyon traffic. The contribution is threefold:

1. The data processing part, from the initial floating car data to the construction of a relevant subgraph (from the prospective of traffic forecasting), provides a general approach to select interesting road segments with respect to their associated speed time series. This issue was tackled by taking into account data availability, non-stationarity and noisiness, with simple criteria that ensure reproducibility and extensibility to other case scenarios.

2. The applications of different forecasting methods, from traditional approaches to machine learning based techniques, gave an in-depth characterization of the traffic prediction problem on a relevant part of the Lyon road network. Such results could serve as baselines for the evaluation of future state-of-the-art methods, when applied to the same speed dataset.

3. The analysis of a recently proposed neural network architecture, designed to tackle spatiotemporal forecasting tasks, such as DCRNN, highlighted a way of combining recurrent neural networks and graph-based approaches. This study led to the reimplementation of the DCRNN fundamental constituent, i.e. its recurrent cell DCGRU, in the latest version of a common deep learning framework, drastically lowering the learning curve for model upgrading and reimplementation.

7.1 Future perspectives

Further efforts can be devolved in refining or modifying the current DCRNN architecture, to overcome possible limitations that emerged after the comparative results analysis. Such improvements could allow to incorporate information other than traffic speed or graph-based signals. For example non-graph related temporal data (in a traffic prediction context this could mean weather, holidays, weekday, etc...), static graph-related features (e.g. road structure [21]) or the adoption of attention mechanisms [22] in the diffusion convolution operation.
DCRNN-like models can be employed in online traffic systems, to deal with real-time traffic monitoring and congestion prediction tasks. Improvements in these areas would be beneficial to traffic operators in implementing congestion-avoidance systems, route planners and resilience monitoring.
Other work could consist in redefining the traffic speed prediction problem in order to meaningfully consider those nodes that have been excluded in the filtering step. Most urban road segments belonged to the filtered-out nodes, hence studies focused on urban areas would benefit from novel frameworks.

7.2 Personal thoughts on the internship

This internship has been a significant experience that allowed me to actively participate in a research environment and, being part of a collaboration between the LIRIS and LICIT laboratories, it offered me complementary views on how to approach the same problem. It was no short of challenges, both technical and theoretical, that in the end led to rewarding outcomes.
Despite the brevity to which the Covid-19 pandemic reduced my in-presence work at the labs, the project developments continued remotely at a constant pace, thanks to meaningful discussions with my supervisors, who gave me guidance and freedom to work independently and follow my intuition.
For these reasons, I consider the internship as a truly enriching experience that made me grow, both academically and personally.

References

[1] B. Shahsavari and P. Abbeel, "Short-term traffic forecasting: Modeling and learning spatio-temporal relations in transportation networks using graph neural networks," *University of California at Berkeley, Technical Report No. UCB/EECS-2015-243*, 2015.

[2] M. S. Ahmed and A. R. Cook, *Analysis of freeway traffic time-series data by using Box-Jenkins techniques*. No. 722, 1979.

[3] Y. Kamarianakis and P. Prastacos, "Forecasting traffic flow conditions in an urban network: Comparison of multivariate and univariate approaches," *Transportation Research Record*, vol. 1857, no. 1, pp. 74–84, 2003.

[4] D. Kwiatkowski, P. C. Phillips, P. Schmidt, Y. Shin, *et al.*, "Testing the null hypothesis of stationarity against the alternative of a unit root," *Journal of econometrics*, vol. 54, no. 1-3, pp. 159–178, 1992.

[5] H. Akaike, "Information theory and an extension of the maximum likelihood principle," in *Selected papers of hirotugu akaike*, pp. 199–213, Springer, 1998.

[6] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pp. 265–283, 2016.

[7] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2019.

[8] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural networks*, vol. 4, no. 2, pp. 251–257, 1991.

[9] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[10] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2008.

[11] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *CoRR*, vol. abs/1609.02907, 2016.

[12] Y. Li, R. Yu, C. Shahabi, and Y. Liu, "Diffusion convolutional recurrent neural network: Data-driven traffic forecasting," *arXiv preprint arXiv:1707.01926*, 2017.

[13] S. Bengio, O. Vinyals, N. Jaitly, and N. Shazeer, "Scheduled sampling for sequence prediction with recurrent neural networks," in *Advances in Neural Information Processing Systems*, pp. 1171–1179, 2015.

[14] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[15] Y. Jia, J. Wu, and Y. Du, "Traffic speed prediction using deep learning method," in *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*, pp. 1217–1222, IEEE, 2016.

[16] J. Zhang, Y. Zheng, D. Qi, R. Li, and X. Yi, "Dnn-based prediction model for spatio-temporal data," in *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pp. 1–4, 2016.

[17] J. Zhang, Y. Zheng, and D. Qi, "Deep spatio-temporal residual networks for citywide crowd flows prediction," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[18] B. Yu, H. Yin, and Z. Zhu, "Spatio-temporal graph convolutional networks: A deep learning framework for traffic forecasting," *arXiv preprint arXiv:1709.04875*, 2017.

[19] L. Zhao, Y. Song, C. Zhang, Y. Liu, P. Wang, T. Lin, M. Deng, and H. Li, "T-gcn: A temporal graph convolutional network for traffic prediction," *IEEE Transactions on Intelligent Transportation Systems*, 2019.

[20] T. Mallick, P. Balaprakash, E. Rask, and J. Macfarlane, "Graph-partitioning-based diffusion convolution recurrent neural network for large-scale traffic forecasting," *arXiv preprint arXiv:1909.11197*, 2019.

[21] Z. Xie, W. Lv, S. Huang, Z. Lu, B. Du, and R. Huang, "Sequential graph neural network for urban road traffic speed prediction," *IEEE Access*, vol. 8, pp. 63349–63358, 2019.

[22] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

Appendices

A Sequence to sequence models

At the beginning of the Forecast section we highlighted the difference between predicting **at**/**up to** a certain time horizon, and how different models may fall into first or second definition. The machine learning models that have been tested on the Lyon speed dataset predicted **at**, with the exception of DCRNN that predicted **up to**. We have already discussed about the reasons that led us to choose the first approach in section 3.4.1.
Sometimes the second approach is preferable, for example if we need a model that is not bounded by a fixed time horizon and returns a sequence of predicted future speed values. In the RNN framework, the standard machine learning approach to deal with sequential data, such models fall in the sequence to sequence (seq2seq) category, as both the inputs and outputs are sequences.
Usually seq2seq models are implemented with an Encoder-Decoder architecture, a two step model. The first part, the encoder, has a sequence as input and a vector, embedding information about the input sequence, as output. The second part, the decoder, takes the output vector from the encoder and returns the output sequence. In figure 14, a schema of an encoder-decoder model, adapted to our case with fully connected layers above the decoder RNN.

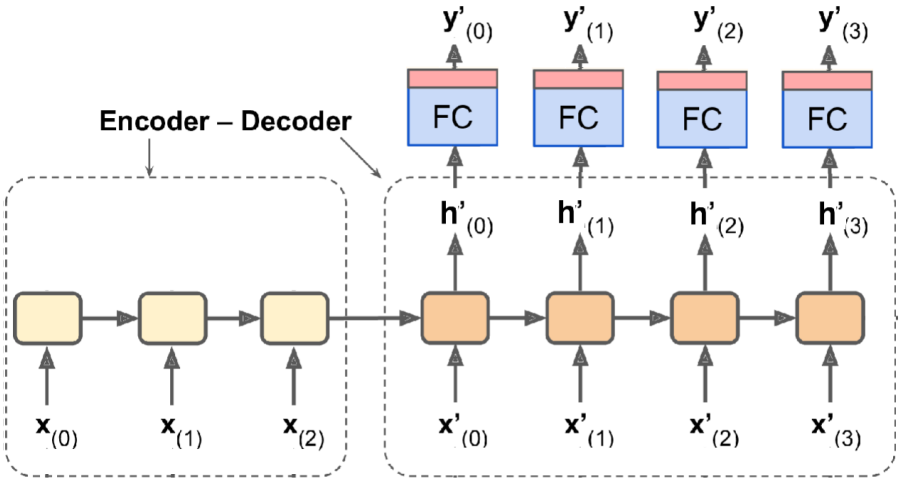

Figure 14: Recurrent Neural Network with an encoder-decoder architecture. Fully connected layers (FC) on top, to convert from hidden states to model outputs. (Image adapted from [7])

Encoder-decoder models have a great popularity in machine translation tasks whereas their advantage in a traffic speed or other predictive contexts may not be as evident. Nevertheless, if a single model for multiple time horizons is desirable, then they provide a viable solution.
Referring to figure 14, it is important to underline what the various inputs and outputs represent, especially for the decoder part. The notion of $x_{(t)}$ in the encoder is identical to the RNN models described in section 3.4.3, i.e. the speed value at time t. In the decoder, the sequence inputs $x'_{(t)}$s can have different meanings. During testing we have that $x'_{(t)} = y'_{(t-1)}$, i.e. the input at time t is the predicted output from $t-1$, with $x'_{(0)}$ set to be equal to the last input of the encoder sequence. During training it can be the same as testing or instead, the inputs can be the true speed values at time t, independently from predictions at previous time steps.
The choice of how the decoder training is implemented has important consequences on the model performance. In general, in a machine learning framework, it is recommended to have data instances from the validation set[10] to come from the same distribution as the test set. With this in mind, training and testing should be conducted in the same way. In practice, it leads to a poorly trained model. On

[10] Also the training set, but this constraint becomes softer as its size increases.

the contrary, if we feed true values as decoder inputs, then the before-mentioned issue arises. This kind of model training is also called "teacher forcing" and it usually gives subpar performances.
The DCRNN architecture falls into the encoder-decoder class of models and it overcomes the decoder training problem by implementing scheduled sampling [13]. Scheduled sampling consists in sampling the true speed values or the corresponding prediction based on a probability that depends on the training epoch: at the beginning it almost coincides with teacher forcing and as training progresses, the probability of sampling the true speed value goes to zero.

B Gated Recurrent Unit

In section 4.2.3 we have described the recurrent cell of the DCRNN architecture, the DCGRU cell, as a modified version of the GRU [14]. In this appendix section we present the standard GRU using a consistent naming convention.
The system of equations of the GRU cell is described by equation equation 9, whereas a more graphical description is given by figure 15:

$$\begin{cases} u_{(t)} &= \sigma(u_i^T[x_{(t)}; h_{(t-1)}] + b_u) \\ r_{(t)} &= \sigma(W_r^T[x_{(t)}; h_{(t-1)}] + b_r) \\ c_{(t)} &= \tanh(W_g^T[x_{(t)}; r_{(t)} \odot h_{(t-1)}] + b_c) \\ y_{(t)} &= h_{(t)} = u_{(t)} \odot h_{(t-1)} + (1 - u_{(t)}) \odot c_{(t)} \end{cases} \tag{9}$$

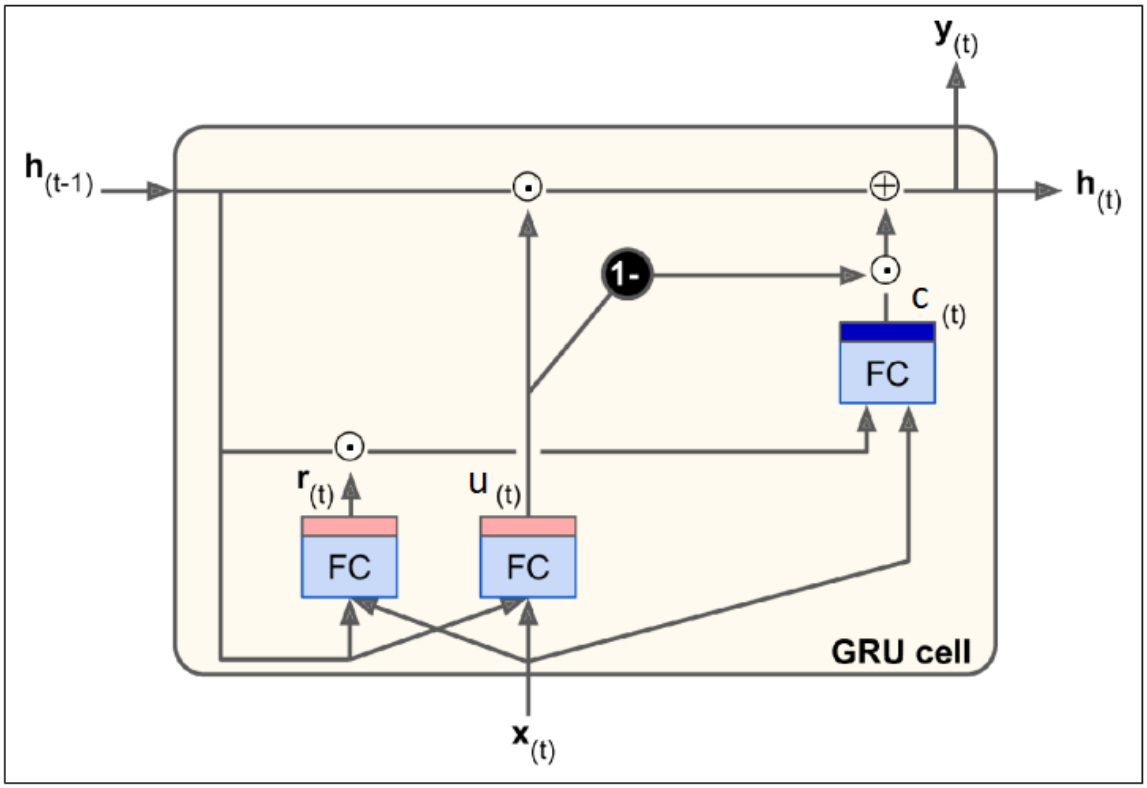

Figure 15: GRU cell schema. (Image adapted from [7])

The GRU cell can be viewed as a simplification of the LSTM, since in the GRU there is only one state vector h. The gate controller u, or *update gate*, determines how much of the previous state h is passed to successive time steps, whereas the gate r, or *reset gate*, determines how much of the previous state h to forget in the computation of the candidate output c.

C DCGRU, Tensorflow 2 code

In this section, the code for the alternative version of the DCGRU cell, introduced in section 4.3. It has been written following common `Keras` guidelines, making it readily resuable.
A github repository has been created containing the presented code and examples on its usage and testing, available at github.com/mensif/DCGRU_Tensorflow2.

```
"""
Alternative implementation of the DCGRU recurrent cell in Tensorflow 2
References
----------
Paper: https://arxiv.org/abs/1707.01926
Original implementation: https://github.com/liyaguang/DCRNN
"""


import tensorflow as tf
from tensorflow import keras
from tensorflow.python.keras import activations

from lib.matrix_calc import *


class DCGRUCell(keras.layers.Layer):
    def __init__(self, units, adj_mx, K_diffusion, num_nodes, filter_type, **kwargs):
        self.units = units
        self.state_size = units * num_nodes
        self.K_diffusion = K_diffusion
        self.num_nodes = num_nodes
        self.activation = activations.get('tanh')
        self.recurrent_activation = activations.get('sigmoid')
        super(DCGRUCell, self).__init__(**kwargs)
        self.supports = []
        supports = []
        # the formula describing the diffsuion convolution operation in the paper
        # corresponds to the filter "dual_random_walk"
        if filter_type == "laplacian":
            supports.append(calculate_scaled_laplacian(adj_mx, lambda_max=None))
        elif filter_type == "random_walk":
            supports.append(calculate_random_walk_matrix(adj_mx).T)
        elif filter_type == "dual_random_walk":
            supports.append(calculate_random_walk_matrix(adj_mx).T)
            supports.append(calculate_random_walk_matrix(adj_mx.T).T)
        else:
            supports.append(calculate_scaled_laplacian(adj_mx))
        for support in supports:
            self.supports.append(build_sparse_matrix(support))
            sup0 = support
            for k in range(2, self.K_diffusion + 1):
                sup0 = support.dot(sup0)                    # (original paper version)
                # sup0 = 2 * support.dot(sup0) - sup0       # (author's repository version)
                self.supports.append(build_sparse_matrix(sup0))

```

```
    def build(self, input_shape):
        """
        Defines kernels and biases of the DCGRU cell.
        To get the kernel dimension we need to know how many graph convolution
        operations will be executed per gate, hence the number of support matrices
        (+1 to account for the input signal itself).

        input_shape: (None, num_nodes, input_dim)
        """
        self.num_mx = 1 + len(self.supports)
        self.input_dim = input_shape[-1]
        self.rows_kernel = (input_shape[-1] + self.units) * self.num_mx

        self.r_kernel = self.add_weight(shape=(self.rows_kernel, self.units),
                                        initializer='glorot_uniform',
                                        name='r_kernel')
        self.r_bias = self.add_weight(shape=(self.units,),
                                      initializer='zeros',  # originally ones
                                      name='r_bias')

        self.u_kernel = self.add_weight(shape=(self.rows_kernel, self.units),
                                        initializer='glorot_uniform',
                                        name='u_kernel')
        self.u_bias = self.add_weight(shape=(self.units,),
                                      initializer='zeros',  # originally ones
                                      name='u_bias')

        self.c_kernel = self.add_weight(shape=(self.rows_kernel, self.units),
                                        initializer='glorot_uniform',
                                        name='c_kernel')
        self.c_bias = self.add_weight(shape=(self.units,),
                                      initializer='zeros',
                                      name='c_bias')

        self.built = True

    def call(self, inputs, states):
        """
        Modified GRU cell, to account for graph convolution operations.

        inputs: (batch_size, num_nodes, input_dim)
        states[0]: (batch_size, num_nodes * units)
        """
        h_prev = states[0]

        r = self.recurrent_activation(self.diff_conv(inputs, h_prev, 'reset'))
        u = self.recurrent_activation(self.diff_conv(inputs, h_prev, 'update'))
        c = self.activation(self.diff_conv(inputs, r * h_prev, 'candidate'))

        h = u * h_prev + (1 - u) * c

        return h, [h]
```

```
    def diff_conv(self, inputs, state, gate):
        """
        Graph convolution operation, based on the chosen support matrices

        inputs: (batch_size, num_nodes, input_dim)
        state: (batch_size, num_nodes * units)
        gate: "reset", "update", "candidate"
        """
        assert inputs.get_shape()[1] == self.num_nodes
        assert inputs.get_shape()[2] == self.input_dim
        state = tf.reshape(state, (-1, self.num_nodes, self.units))
        # (batch_size, num_nodes, units)

        # concatenate inputs and state
        inputs_and_state = tf.concat([inputs, state], axis=2)
        input_size = inputs_and_state.get_shape()[2]
        # (input_dim + units)

        x = inputs_and_state
        x0 = tf.transpose(x, perm=[1, 2, 0])
        # (num_nodes, input_dim + units, batch_size)

        x0 = tf.reshape(x0, shape=[self.num_nodes, -1])
        x = tf.expand_dims(x0, axis=0)

        for support in self.supports:
            # premultiply the concatened inputs and state with support matrices
            x_support = tf.sparse.sparse_dense_matmul(support, x0)
            x_support = tf.expand_dims(x_support, 0)
            # concatenate convolved signal
            x = tf.concat([x, x_support], axis=0)

        x = tf.reshape(x, shape=[self.num_mx, self.num_nodes, input_size, -1])
        x = tf.transpose(x, perm=[3, 1, 2, 0])
        # (batch_size, num_nodes, input_dim + units, order)

        x = tf.reshape(x, shape=[-1, input_size * self.num_mx])

        if gate == 'reset':
            x = tf.matmul(x, self.r_kernel)
            x = tf.nn.bias_add(x, self.r_bias)
        elif gate == 'update':
            x = tf.matmul(x, self.u_kernel)
            x = tf.nn.bias_add(x, self.u_bias)
        elif gate == 'candidate':
            x = tf.matmul(x, self.c_kernel)
            x = tf.nn.bias_add(x, self.c_bias)
        else:
            print('Error: Unknown gate')

        return tf.reshape(x, [-1, self.num_nodes * self.units])
        # (batch_size, num_nodes * units)
```