ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DEPARTMENT OF COMPUTER SCIENCE
AND ENGINEERING

ARTIFICIAL INTELLIGENCE

MASTER THESIS

in

Combinatorial Decision Making and Optimization

Multiple time series forecasting with Graph Neural Networks

CANDIDATE
Alessandro Lombardi

SUPERVISOR
Prof. Zeynep Kiziltan

Academic Year 2020/21
Session 3rd

Acknowledgements

I am extremely grateful to Professor Zeynep Kiziltan for giving me the opportunity to work on this stimulating project and for her invaluable guidance through each stage of the process. I would like to express my deepest appreciation to the University of Bologna, for its pioneering idea of proposing an innovative Master's Degree in Artificial Intelligence. Special thanks go to the Computer Science and Engineering Department staff, for providing the students with new computing facilities which are vital to this research field.

Abstract

Time series forecasting aims to predict future values to support organizations making strategic decisions. This problem has been studied for decades due to its relevance in almost all industries and areas, ranging from financial data to product demand. Recently, modern solutions based on deep learning have gained popularity among academia and industry, mainly due to the necessity to automatize the forecasting of multiple time series and exploit external explanatory variables. Considering the recent successes of Graph Neural Networks (GNNs) in modelling graph data, this study extends previous works based on time series forecasting from visibility graphs. In particular, in the first direction, the link prediction task, targeted by local random walks in the related work, is resolved by custom GNNs. In the second direction, a new strategy based on graph regression using GNNs is proposed to learn graph representations able to combine hidden historical patterns and external features. The M5 competition dataset is used to compare the proposed models to the related work and other traditional and machine learning benchmarks. Final results show promising performances on various higher levels of the M5 competition and delineate multiple limitations from the related work.

Contents

1 Introduction **1**
1.1 Context . . . 1
1.2 Motivations . . . 2
1.3 Original contribution . . . 4
1.4 Document structure . . . 5

2 Background **6**
2.1 Time series . . . 6
2.1.1 Taxonomy . . . 6
2.1.2 Properties . . . 8
2.2 Time series forecasting . . . 10
2.2.1 Traditional approaches . . . 10
2.2.2 Machine learning approaches . . . 13
2.2.3 Neural networks and deep learning approaches . . . 14
2.3 Graph neural networks . . . 17
2.3.1 Motivations . . . 18
2.3.2 Static graph models . . . 20
2.3.3 Dynamic graphs models . . . 24

3 Dataset **27**
3.1 M5 competition . . . 27
3.2 Data analysis . . . 30
3.2.1 Sales . . . 30
3.2.2 Exogenous features . . . 37

4 Time series forecasting with GNNs **39**
4.1 Related work . . . 39
4.2 Methodology . . . 42

4.2.1 Link prediction with GNNs . . . 43
4.2.2 Graph regression with GNNs . . . 45
4.3 Implementation details . . . 47
4.3.1 Software and repository . . . 47
4.3.2 Data transformation . . . 49
4.3.3 Link prediction with GNNs . . . 54
4.3.4 Graph regression with GNNs . . . 57

5 Experiments **60**
5.1 Metrics . . . 60
5.2 Experimental setup . . . 61
5.2.1 Link prediction with GNNs . . . 62
5.2.2 Graph regression with GNNs . . . 63
5.3 Hyperparameter tuning . . . 64
5.3.1 Link prediction with GNNs . . . 65
5.3.2 Graph regression with GNNs . . . 66
5.4 Final results . . . 68
5.5 Discussion . . . 72

6 Conclusions **76**
6.1 Contribution summary . . . 76
6.2 Future works . . . 78

Bibliography **79**

List of Figures

1.1 Venn diagram of the involved domains, methods and topics. 3

2.1 Time series in the demand forecasting context 7
2.2 Time series seasonal decomposition 9
2.3 Example of a message passing framework 19
2.4 RecGNNs vs ConvGNNs 22
2.5 GraphSAGE sample and aggregate 23

3.1 M5 hierarchical scheme 28
3.2 Number of time series by M5 level 28
3.3 Three time series samples from the 12th level. 31
3.4 Number of sales distributions at each level 32
3.5 Time series space-related aggregations 33
3.6 Time series product-related aggregations 34
3.7 Sales by department and state 35
3.8 Monthly sales by store and state 35
3.9 Seasonal plots 36
3.10 Prices distributions 38

4.1 Visibility graph generation 53
4.2 Visibility graphs from the 6th level 53
4.3 LpGnn methods UML class diagram 54
4.4 LpGnn static architecture 56
4.5 Link prediction with GNNs vs graph regression with GNNs datasets 58
4.6 RgGnn architecture 59

5.1 Training loss and validation score 71
5.2 Predictions examples 71

List of Tables

5.1 Hyperparameter tuning of LpGnn 65
5.2 Hyperparameter tuning of RgGnn 67
5.3 Final results on the evaluation data for all the levels 68
5.4 Final comparison 69

Dedicated to all the educators I have met. My family, whose sacrifices taught me to love. My professors, who transmitted their passions. And everyone else who was always there to help me realising that every moment of this journey was spectacular.

1 Introduction

Time series forecasting is a technique adopted to predict future events by analysing past observations and external factors. It plays a critical role in improving data driven decisions in various business and industrial areas such as retails, energy, transportation, supply chain optimization, hardware monitoring or finance [5, 2, 48]. When an organization establishes some objectives, it relies on the ability to predict future scenarios to understand the impact of future events. Time series forecasting is at the core of any informed decision about future strategic planning. When historical data is available and it is assumed to maintain some properties in the future, it becomes an interesting and well studied quantitative problem [22]. There are many time series forecasting scenarios, for instance predictions can have different time horizons or multiple time series can be measured and forecasted at the same time.

1.1 Context

The simplest time series models focus only on the historical evolution of the variable to predict, by observing its seasonalities, cycles and trends. Linear models, such as ARIMA or exponential smoothing are very common solutions and they have been considered as the standard approach to time series forecasting for many decades [7]. Other models have been proposed to deal with specific scenarios, for instance Croston [13] was proposed for intermittent time series, GARCH for non-linear time series and VAR for multivariate time series. Recently, machine learning models have drawn attention, becoming serious contenders to classical methods in time series forecasting [7, 34].

One of the most important reason behind this new direction is the necessity to deal with modern applications, where thousands or millions of related time series must be forecasted together. The majority of the consolidated forecasting models used today depend on traditional methods and theory developed in the setting of forecasting individual or small groups of time series [48]. Moreover, many classical models are based on statistical assumptions and analyses which require domain knowledge and experience difficult to scale and automatize [31, 55].

Only recent works [43, 48, 50, 38] have developed machine learning solutions to fill the gap between the traditional methods and the modern necessities. Extremely successful are gradient boosting decision trees models, such as XGboost or LightGBM [35, 54, 41] or recurrent and convolutional networks [31, 5]. On one side, a high number of related time series is difficult to handle and surely represents a new challenge, on the other, it also gives the opportunity to approach the problem using modern techniques based on machine learning and deep learning. In particular, the availability of great amounts of historical data allows to fit more complex models without overfitting and to attenuate the intensive manual feature engineering and model selection steps required by conventional techniques.

Among the recent developments in deep learning, a new class of methods called Graph Neural Networks (GNN) have recently gained a lot of attention for their natural ability to effectively handle data described by graphs. GNNs are commonly applied to numerous fields not directly related to time series forecasting, such as graph mining, modelling physical systems and chemical systems, knowledge graphs and recommendation systems [69]. Since graphs can incorporate temporal information, especially in certain applications, such as traffic prediction or social networks evolution, time series forecasting methods have been applied together with GNNs to model the temporal aspects [66, 45]. However, very recent studies started considering GNNs to specifically improve time series forecasting. In particular, since multivariate time series forecasting is not based uniquely on historical data but is assumes interdependencies among variables, GNNs have been proposed to learn latent complex dependencies between the variables [51]. For instance, the Multivariate Time Series Graph Neural Network (MTGNN) [63] automatically discovers the hidden associations interleaving graph convolution and temporal convolution modules to capture both spatial and temporal dependencies. Hence, the multivariate time series data modelling and the internal graph structure learning occur simultaneously in an end-to-end manner. Similarly, the Spectral Temporal Graph Neural Network (StemGNN) [8] automatically captures inter-series correlations and temporal dependencies jointly in the spectral domain.

1.2 Motivations

All the works mentioned above rely on the common representation of time series as temporally ordered sequences of values, keeping separated time and space dimensions or using graphs only to model the relationships among the time series or their values. Another trend of studies proposes to convert time series to graphs and to apply complex network theory to approach time series analysis. Extremely successful are numerous recent multidisciplinary works which exploit the properties of the extracted graphs and correlate them to specific phenomena in the

underlying observed dynamic systems [70]. Among the different algorithms proposed to convert time series to graphs, this work focuses on visibility graphs [27], since there are multiple models proposing time series forecasting from visibility graphs [9, 32, 36, 68, 67].

The strategy adopted by the related work, firstly converts time series into visibility graphs, then it performs link prediction, to anticipate the evolution of the original graphs as new nodes are added. To predict the edges they use graph heuristics, such as local random walks, while interpolation procedures are finally adopted to retrieve the associated future values from the complete graphs. The main contribution observed in the cited previous works is limited to refining the forecasts, rather than proposing new methods in alternative to the heuristics. Instead, this work proposes to adopt GNNs to deal with visibility graphs. The application of GNNs to time series forecasting using visibility graphs and external features represents an original strategy which is motivated by the mentioned successes of deep learning solutions, the recent explosion of GNNs models and their various integrations with time series. From another perspective, this work proposes a similar shift observed in time series forecasting: from traditional and statistical models to deep learning based techniques. The objective is still improving the performances on modern time series, which are often multiple and accompanied by external explanatory variables. Thus, this work aims at opening a new research direction which encompasses and unifies three different disciplines: time series forecasting, graph theory and deep learning. The white dotted area in the Venn diagram in 1.1 represents where this work is situated and highlight the difference with the related work, in purple, and the importance of GNNs, in yellow, which constitute the missing link between all the three mentioned domains.

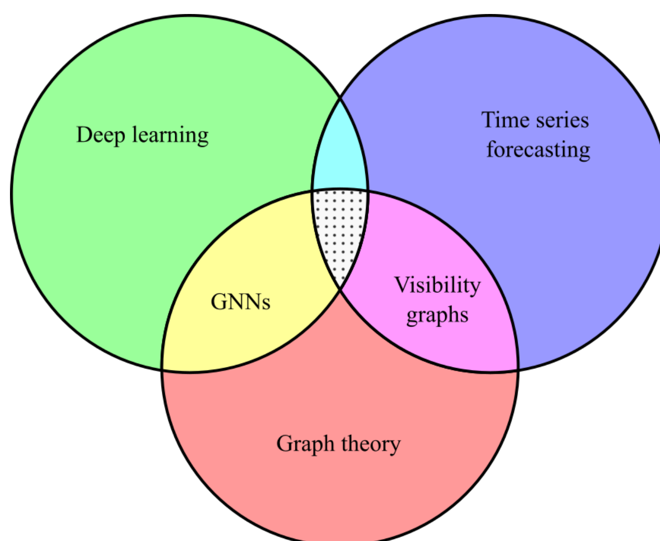


Figure 1.1: The Venn diagram representing the domains and the methods involved in this work. The white dotted area represents the approach followed by the current study.

Indeed, to the best of the author knowledge, there are only few works that have attempted to combine GNNs and time series to graphs algorithms. The Adaptive Visibility Graph (AVG) algorithm [65], proposed after the beginning of this work, maps time series into visibility graphs and it is involved in the training of a neural network for radio signal modulation classification. In [29], time series are converted to a variant of visibility graphs, called horizontal visibility graphs, and used to fed a GNN model for feature learning and faults diagnosis. Finally in [42], GNNs are deployed to provide model recommendations for time series forecasting. Despite their similar approach, these works target different time series tasks.

1.3 Original contribution

The first direction of this work studies the impact of replacing the heuristic methods used in the related work with GNNs to predict edges in visibility graphs, proposing two alternatives. In the first, standard link prediction methods based on GNNs are applied to independent visibility graphs generated from time series portions. In the second, a wider strategy is applied to the problem, considering visibility graphs as dynamic graphs and leveraging custom GNNs models specifically tailored for these scenarios. As a consequence, from this attempt a second direction, still based on GNNs and visibility graphs, is followed to overcome the limitations of the link prediction strategy. Specifically, a graph regression approach is proposed to use GNNs in order to extract meaningful representations from visibility graphs, which can be directly used to predict the associated future values via sequential neural networks architectures. To the best of the author knowledge, this is the first attempt to apply GNNs to visibility graphs in order to forecast time series data and exploit exogenous features and cross-learning over multiple time series. Indeed, the previous works proposing visibility graphs and GNNs focus on other time series tasks, while the other mentioned works do not even convert time series to graphs, leveraging both representations separately.

The new methods are presented and compared to the related work and several traditional forecasting methods using the data from the M5 competition. The M5 competition is a well known machine learning competition hosted in Kaggle in 2020 which involved more than 7 thousands participants. The competition objective is producing the most accurate point forecasts for 42840 time series representing the hierarchical unit sales of various Walmart stores in the United States [35]. By using the M5 dataset, this work enables the opportunity to compare the presented solutions to benchmark models prepared by academic experts and to measure itself with a real world modern dataset, which meets all the looked requirements. In conclusion, promising results and new interesting findings are shared and deeply discussed, giving the opportunity to suggest further improvements to the current limitations. In particular, long

horizons and the difficulty to relate link prediction results to the forecasts are found to be the main problems of the current and previous works. On the other hand, the new proposed method based on graph regression successfully reaches competitive results, following a similar trend along the M5 hierarchical levels which is observable also in the winning entries of the competition [35].

1.4 Document structure

Chapter 2 presents a deep introduction to the background theory related to time series forecasting and GNNs. The content of this chapter is essential to understand the common methodologies adopted to solve the presented problem and to introduce the methods involved in this work. In Chapter 3 the data used to evaluate the proposed models is described in details. The depth of the analysis is justified by the need to provide evidences on the reasons behind the choice of the M5 dataset and by the necessity to show how standard data exploration is performed in time series domain and its criticality in delivering better results. The following Chapter, 4, displays the proposed methods, starting from an higher and more abstract level, where the algorithms are shown and the differences among them are highlighted, concluding with a deeper description of the implementation details. Chapter 5 illustrates the experimentation methodology and the outcomes of the proposed solutions. The final Chapter, 6, discusses about the final considerations and the future works.

2 Background

In this chapter a literature overview of time series and graph neural networks is provided to the reader. The aim is to provide enough background to understand the terminology and the state of the art of both domains, in order to facilitate the comprehension of the subsequent chapters. Since this work proposes a method to forecast multiple time series, a wide overview of modern solutions is presented to the reader to familiarize to the challenges of this problem and underline the difficulties of more classical approaches.

2.1 Time series

A time series is a sequential set of data points, mathematically defined as a set of vectors $x(t)$, where t represents the time elapsed and $x(t)$ is treated as a random variable [2]. Time series are used in many fields, for instance in statistics, signal processing, pattern recognition, econometrics, mathematical finance, weather forecasting, biology, control engineering and astronomy. In general, the term **time series analysis** is used to describe any technique involved in the extraction of meaningful characteristics and statistics of the data. Indeed, time series can be involved in various common machine learning tasks, such as prediction, classification and clustering. Time series analysis is usually divided in *frequency-domain* methods and *time-domain* methods. The difference between the two classes relies on their different point of views: frequency-domain methods describe time series by decomposing them in a range of frequencies, while time-domain methods analyse how signals change over time. This work is prevalently focused on time-domain studies. Given the formal definition of time series, it is necessary to understand the common variations that can be encountered in real world and describe the most fundamental properties.

2.1.1 Taxonomy

There are different aspects to consider in a time series forecasting problem. Depending on the underlying data one of the greatest difference is between time series containing only one

temporal variable, called **univariate**, and time series which include more than one, called **multivariate**. Available data can be further categorized as **endogenous**, if it is affected by other variables in the system and the output variable depends on it, or **exogenous**, if it is independent from other variables in the system and the output variable depends upon it. Time series can be considered either **continuous** when observations are made continuously through time, or **discrete**, when observations are taken only at specific times, usually equally spaced. Another distinction can be made on the quality of a time series, which introduces the situation of **discontiguous** time series. For instance, they can be time series impacted by changes in the measurement activity or by corrupted data which show a non-uniform distributions of data over time. Depending on the data, the literature proposes specific categorizations of time series based on their field of application, for example in the demand forecasting it is possible to observe the time series families depicted in figure 2.1.

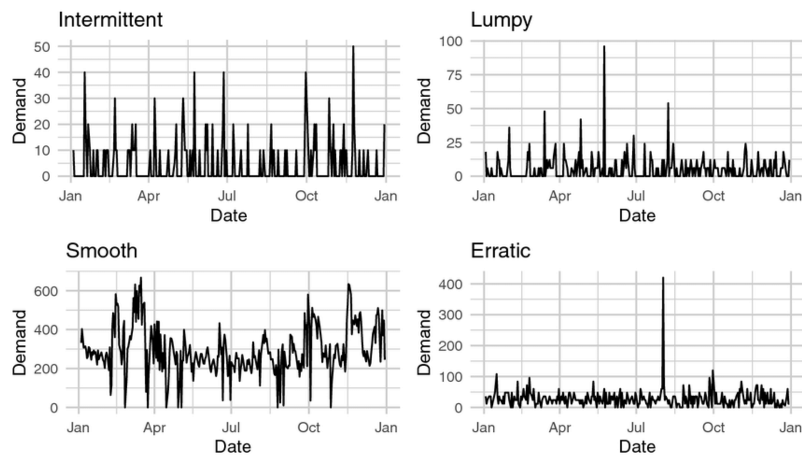


Figure 2.1: Examples and categorization of different time series common in demand forecasting from [53]

It is possible to have *multiple measurements*, hence time series, for the same period or in partially overlapping periods. Even if each time series should be considered on its own, this can be infeasible or not desirable when there is a vast number of time series and, intuitively it can be interesting to exploit the similarities between the multiple time series and build a single forecasting model. In particular, the literature treats the situations of **grouped** and **hierarchical time series**. It is common to deal with large collections of time series that can be aggregated and disaggregated by various attributes, maintaining a coherence across the aggregation structure [22]. If the structure can be divided in nested categories of time series, they are considered to have a hierarchical structure, on the other hand, if the aggregations are more complex and categories can be mixed together, the time series are considered grouped. Grouped time series

can be thought of as hierarchical time series that do not impose a unique hierarchical order. Forecasts for these kinds of time series require consistent adding ups, namely the sum of the forecasts of all the subcategories of a direct parent category must be equal to the forecast of the latter and the inverse must hold. Definitions of multivariate and multiple time series can be interchanged depending on the business problem, usually when there are multiple measurements, different from each other but recorded from the same source, they are usually considered as variables and are used to define a multivariate time series. On the other hand, if there are multiple independent measurements, for example from multiple instances of the same source, they can be considered as multiple experiments representing the same phenomenon. Nothing prevents from having multiple multivariate time series, or from considering multiple time series as a single multivariate time series, it depends on the circumstances and personal interpretations. Finally, forecasting problems are divided into **one-step** and **multi-step**, depending on whether they require the prediction of a single time step or more than one. The required number of time steps to forecast is called **horizon**, when this number is greater than the number of time steps predicted by a proposed model multiple predictions are executed. If a model uses its own forecasts to predict further values it is called **autoregressive**. The relationship between the size of the input and of the output time steps is arbitrary, in addition, both a univariate and a multivariate problem settings can be single-step or multi-step. In general, and especially in business solutions, it is always important to consider how the forecasting is performed at production time, in terms of how and when the data is available and whether the system is updated with fresh data and its rate. Unfortunately, it could be very easy to build solutions which cannot be used in production due to an erroneous ingestion of information which is not available at prediction time.

2.1.2 Properties

By observing time series it is usually possible to notice some important characteristics such as **trends**, **seasonal** patterns and **cycles** [22]. A trend can be detected as a variable's consistent upward or downward movement over time and can be seen as a long term movement in a time series. When a time series behaviour always repeats within a year, it is said to have a seasonality. Seasonality is usually caused by events which occur on monthly or weekly basis [37]. A cycle occurs when the data exhibits rises and falls that are not of a fixed frequency, usually these variations are caused by events that spans over longer periods, like for years, and have greater magnitudes than seasonal patterns [22]. Irregular or random variations in time series are caused by unpredictable and non-repeating patterns which are statistically not measurable. Sometimes time series plots show sudden changes in behaviour, which are referred

to *structural breaks*. These rapid and unexpected changes are usually associated to real events occurring outside the scope of the data, for example an economic crisis effecting the forecast of a product's sales, and can have a meaningful impact on the forecasting performances. Structural breaks can be observed visually and statistically tested using several methods. In general, time series can be described with the following four main components: trend, cyclical, seasonal and irregular components. Time series are assumed to follow probability models, which describe the joint distribution of the random variable $x(t)$. A sequence of observations is a sample of the **stochastic process** that produced it [2]. A time series is said to be **stationary** when its statistical properties do not depend upon time, namely, all transition probabilities from one state of the system to another are independent of time within the observation period [37]. Generally time series are not stationary due to trends and cyclical components, for this reason a less constrained definition of stationary, called **weak stationarity**, it is often considered valid. A stochastic process is defined weakly stationary when its mean function and its correlation function do not change by shifts in time.

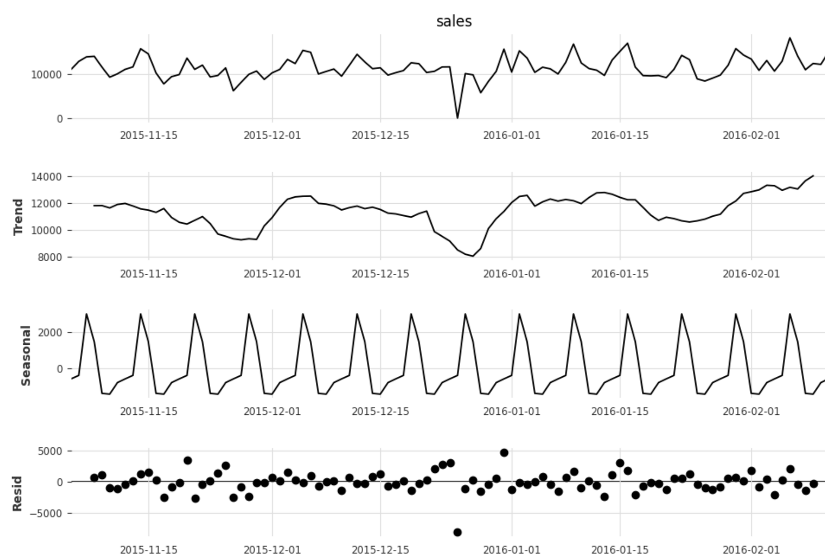


Figure 2.2: An additive seasonal decomposition using moving averages computed using the Python library statsmodels. In addition to trend and seasonalities, it is apparently possible to observe how a structural break might look like at the centre of the series.

The linear relationship between two variables is called *correlation*. When the values of two variables increase or decrease together they are said to have a positive correlation, otherwise if they show an opposite trend, they have a negative correlation. When the correlation is measured between a variable and its version in different points in time it is called *auto-correlation*.

More specifically, the auto-correlation is the correlation between time series observations and observations at previous time, or *lagged* time. An intuitive way to observe correlations is plotting the **auto-correlation function (ACF)** or *correlogram* and the **partial auto-correlation function (PACF)** of a time series [24]. ACF plots show the correlation between observations and previous lagged observations and the related confidence interval. ACF considers all the aforementioned time series components while finding correlations hence it is called a complete auto-correlation plot. Conversely, PACF computes correlations between observations at different time steps with the relationships of intervening observations removed. If ACF considers both direct and indirect correlations, PACF considers only direct ones, ignoring the effects of any correlations caused by observations in shorter lags. The autocorrelation plots of a time series play an important role in identifying its possible structures, supporting the formulation of an effective forecasting model. For instance, design choices on autoregressive-based models should be supported by strong auto-correlation values for the chosen parameters.

2.2 Time series forecasting

In time series forecasting, or time series prediction, past observations are collected and used to develop a suitable mathematical model which captures the underlying process that generated them, and thus, can be used to predict the future. In other words, time series forecasting is the task of predicting future values of a time series. Time series forecasting is a relatively old scientific topic originated at the beginning of the twentieth century. Since the early stages it has been continuously improved and included in other academic fields and industry. Forecasting models can be categorized considering different aspects, many of them are specific to particular types of time series, for example some models are used only for univariate or multivariate time series. An important distinction is between *linear* and *non-linear* models. Linear models consider each data point as a linear combination of external inputs and its previous output. Conversely, non-linear models consider more complex situations, where the relationship between future and past observations is modelled by non-linear functions. For example non-linear models are more suitable when data contains structural breaks or have a high volatility, like financial time series. This study majorly deal with non-linear models.

2.2.1 Traditional approaches

There is a plethora of models based on the assumption of a stochastic process representation. The simplest forecasting algorithm is called Naive, in which each time step is predicted using the value observed in the previous one. The most famous linear models are *Autoregressive*

(AR) and *Moving Average (MA)*, which are usually combined to form *Autoregressive Moving Average (ARMA)* and **Autoregressive Integrated Moving Average (ARIMA)** models [2, 37]. ARIMA is one of the most widely used method for univariate time series forecasting, its greatest limitation is not supporting seasonal components, for this reason the extension **SARIMA** has been proposed. ARIMA can be described by its three components:

- **Autoregression (AR)**, exploits the relationship between current and lagged observations of a variable. Namely, an autoregression model uses the observations from previous time steps as input to a regression equation to forecast the next steps.
- **Integrated (I)**, *differencing* subtracts each value of the time series with previous lagged values to make the time series stationary.
- **Moving average (MA)**, exploits the dependency between observations and residual errors from a moving average model applied to lagged observations. A moving average model is a Naive forecasting method which computes the average of the prior n observations as a forecast.

ARIMA is an extension of the ARMA model which supports also non-stationary time series. An ARIMA instance is defined as ARIMA(p, d, q) where the three parameters represent:

- p: the number of autoregressive lags.
- d: the number of non-seasonal differences needed for stationarity.
- q: the number of moving average lags.

In particular, the AR(p) model is described with the equation 2.1 while the MA(q) model with the equation 2.2, where y_t is the value of the time series at time t, ϵ is a random variable called *white noise*, μ is the expectation of y_t and $\phi_1, \dots, \phi_p$ and $\theta_1, \dots, \theta_q$ are parameters. ARMA is obtained from the sum of these two equations, while ARIMA is ARMA on an integrated time series, so an ARIMA with $d > 0$ predicts the difference between time steps rather than the time series itself.

$$y_t = \epsilon_t + \sum_{i=1}^{p}(\phi_i \cdot y_{t-i}) + (\mu + \sum_{i=1}^{p}(-\phi_i \cdot \mu)) \tag{2.1}$$

$$y_t = \epsilon_t + \sum_{i=1}^{q}\theta_i \cdot \epsilon_{t-i} \tag{2.2}$$

Fitting an ARIMA model means choosing p, q and d and then estimating the coefficients. ARIMA models can be fitted using maximum likelihood estimation, while parameters p and

q can be specifically estimated using information criteria metrics like Akaike's Information Criterion (AIC) or Bayesian Information Criterion (BIC) [22]. SARIMA involves the same parameters and includes other similar ones to represent also the seasonal component. A formal method to build an ARIMA model able to accurately fit a time series is the *Box-Jenkins methodology*. The Box-Jenkins method iteratively follows three steps: model identification, parameter estimation and statistical model checking. In the first step the time series are studied and stationarity and seasonality are found, in order to choose which components should be added to the model. The parameter estimation uses optimizations algorithms to fit the proposed model and the last stage performs some tests to assess if the model is good enough or a better one should be proposed, thus restarting the algorithm.

Another family of methods for univariate data is *Exponential Smoothing (ES) Methods*, which similarly to ARIMA predicts using past observations, but apply also an exponential weighting factor to give more importance to the recent observations. The basic exponential smoothing can be obtained from an ARIMA(0,1,1) model without constant. There are different extensions of the basic exponential smoothing to model trends and seasonality [37].

A model developed to deal with multivariate time series is *Vector Autoregressive Model (VAR)*, while for non-linear time series a famous model is *Generalized ARCH (GARCH)*. The reference model for intermittent time series forecasting is Croston [13]. Croston uses ES to forecast the non-zero demand size z_t and the inter-demand intervals p_t decoupling the final forecasting as their combination as stated in equation 2.3.

$$y_t = \frac{z_t}{p_t} \tag{2.3}$$

Croston assumes that all periods are equally likely to exhibit demand, since z_t and p_t are only updated when demand occurs, preventing the model from actively adjust its metrics for arbitrary periods. The Croston method has been improved in the recent years and multiple variations have been proposed, a general overview of other classical forecasting models can be found in the appendix of [35].

In general, classical statistical methods are fast, have lower computational costs, have higher interpretability and better statistical guarantees. They are still widely adopted, and they have proven to outperform machine learning solutions in the past [34]. However, traditional approaches usually suffer from some limitations, such as: robustness to missing data, hand-crafted features, heavy preprocessing and presence of hard design choices which require experience and strong theoretical foundations. Even if there are many tools to help automatize the creation of this models, since they require strong theoretical background they are not really practical in predicting multiple time series.

2.2.2 Machine learning approaches

Studies to adapt machine learning techniques to the forecasting problem are not new, the literature proposes a vast set of methods from the classical supervised machine learning domain such as Support Vector Machines [56], K-Nearest Neighbor [7] and Gaussian Processes [34]. Decision trees and random forest are the preferred solutions in many domains, including time series forecasting. In particular, one of the most shining model is **eXtreme Gradient Boosting (XGBoost)** [10] due to its fame in the data science competition community. XGBoost, such as the more recent variations LightGBM and CatBoost, are highly optimized implementations of the gradient boosting decision trees algorithm (GBDT). Boosting refers to the ensembling technique of sequentially adding models, also called weak learners such as simple decisions trees, to correct the errors that the previous models have not solved. Model evolution adopts the gradient descent algorithm to minimize the loss when new models are added. XGBoost is fast and scalable, it allows regularization to prevent overfitting, it can ingest missing data or inputs of different scales and it can optimize multiple objective functions. One of the greatest weakness of decision tree-based models is their lack of extrapolation capability. Decision trees learn how to create splits on the features of the data, therefore they extract useful tests to infer the value of a target. Which implies that trees are not able to predict outside the ranges of values seen during training, making time series predictions not reliable, especially when there are unobserved seasonalities or strong trends. In general, decision trees usually outperform linear models when there are high non-linear relationships between features, they are able to deal with different data types and can be easily interpreted. In [55] the authors propose a model to perform automatic forecasting of multiple time series having different characteristics without requiring the intervention of experts analysts, which can be hard to scale when there are tons of time series to control. The proposed model is called **Prophet**, it works by decomposing the time series in its trend, seasonality and holiday components and then handling the forecasting problem as a curve fitting problem. In such way, the points in the time series are not required to be equally spaced, fitting is fast and the model is highly interpretable. The model consist of three different statistical based modules to fit each component, while the analyst is only required to provide the information regarding the values forecasted, such as volume bounds, change points, holidays and seasonalities. There are some parameters which can be used as knobs to regulate the importance of previous seasonalities and trends. The results clearly show an improvement to the results obtained with traditional time series forecasting model. Machine learning solutions have been successfully deployed in many contexts obtaining distinct results. They have demonstrated to be robust to problems such as non-linearity and non-stationarity. The following part provides references for some works where the machine learning approach

has been chosen as the best among traditional and deep learning methods.

2.2.3 Neural networks and deep learning approaches

There are multiple advantages in using deep neural networks for time series forecasting, namely, they are able to learn complex non-linear mappings and they natively support multiple inputs and outputs. In addition, they can automatically extract the features and learn the hidden relations between them, automatizing preprocessing and feature engineering. Due to the great success in many fields deep learning has gained a lot of attention in recent years. There are multiple kinds of neural network architectures [31]. **Multi-layer Perceptrons (MLP)**, called also feed-forward neural networks because the flow of the inputs moves in a single direction, are the most basic form of artificial neural networks. MLPs consist of layers, each layer is defined as a set of nodes applying affine transformations followed by a non-linear activation. In its simplest form a MLP contains only a single layer. During training, the nodes receive all the same inputs but they learn different ways of processing them, namely each node has its own weight, allowing the representation of different functions and allowing complex representations. During training MLPs learn the relationships between the input features and the targets. The non-linear activation is used to constrain the output in a specific range, while the number of layers and nodes can be adapted to solve different tasks. In particular, the number of nodes in the last layer plays an important role on the final result. In the context of time series forecasting MLPs have been used for a while, and they represent the easiest attempt to use neural networks. One great advantage over the machine learning models is the possibility to directly predict multiple values. The literature defines different approaches such as Time Lagged Neural Networks (TLNN) and Seasonal Artificial Neural Networks (SANN) [2]. In general MLPs have many limitations, they cannot adapt to different input and output sizes and they do not exploit the structure of the input.

Convolutional Neural Networks (CNN) are a special class of neural networks designed for inputs having an ordinal structure such as images and time series [5], in addition, they have a strong connections with filtering in digital signal processing. As their name suggest, their main operator is the convolution. In this context, the convolution refers to the process of computing weighted sums by sliding a kernel over the input data. Convolutional kernels are discrete and contain learnable weights which are trained to extract meaningful information from the input features. The main advantage from MLPs resides in the possibility to share learnable parameters across multiple parts of the input, which reduces the complexity of the model and leads to better performances. In time series forecasting, CNNs are commonly one-dimensional and learn time-invariant relationships by limiting the processing on only past observations inside

their settable receptive field, thus, they are usually called causal. An advanced and promising model using CNNs is the **Temporal Convolutional Network (TCN)**, proposed in [28]. TCN has a encoder-decoder structure where the length of the input is the same of the output. The model consist of a stack of one-dimensional causal CNNs, where paddings are introduced to force each layer to assume the same length of the input. In order to allow the model to observe a long range of past time steps and learn long-term dependencies, dilated convolutions are applied. Another interesting example is WaveNet initially developed for speech generation in [58], it also uses causal convolutions and dilated convolutions, but introduces a neural block similar to those present in recurrent networks, with gates and skip connections.

Recurrent Neural Networks (RNN) are the neural networks deliberately designed to deal with sequential data in applications such as time series, natural language processing and speech recognition. The main idea behind RNNs is to deploy recurrent connections to connect neural networks hidden units back to themselves with a time delay [5]. This can be seen as a dynamic memory where at each step the RNN receive the external input and the hidden states from the previous steps. Nowadays, vanilla RNNs have been replaced by more sophisticated alternatives such as **Long Short-Term Memory (LSTM)** [21] and **Gated Recurrent Units (GRU)** [11] models. These implementations have been proposed to mitigate some of the common problems in RNNs, such as vanishing or exploding gradients. Specifically, LSTM proposes a more complex architecture made of cells and multiple gates (input, output and forget). The cell is used as a long-term memory carrying information over arbitrary time intervals and the gates regulate the flow of information into and out of the cell. The LSTM unit produces two values, the output and the cell state. While the outputs depend on the cell state and undergo many operations, the cell states are less influenced by each iteration and they can carry long-term information and reduce the typical problems of the RNN architecture. GRU is a more computationally efficient and yet comparable alternative to LSTM.

Attention mechanism and in particular **transformers** are currently obtaining state-of-the-art results in multiple natural language processing tasks. Attention helps models to focus on important features and learn regime-specific temporal dynamics. For example transformers have been used in [61] to forecast influenza-like illness. Even if it is possible to directly use these blocks to obtain decent results many techniques combine them, like in the CNN-LSTMs, or uses them to create more complex structures, like sequence-to-sequence (Seq2Seq) models, which are generally preferred for multi-step forecasting. Deep learning methods can either be used to directly predict the future targets or probabilistic estimates of them. In the latter, the network is used to learn parameters of a known distribution. Another great advantage of deep learning solutions is the possibility to train global models to forecast multiple time series. This requirement it is generally not considered in traditional forecasting models, which are not easily

scalable to large data-sets with millions of time-series, requiring individual training. Moreover, they cannot benefit from shared temporal patterns in the whole dataset [50]. The following list provides a list of very recent methods using deep learning for multiple time series forecasting.

- **Deep State Space Models** proposed in [43] fuses deep learning with State Space Models (SSM). SSMs [14] provide a principled framework for modelling and learning time series patterns such as trend and seasonality. In this work, a RNN is used to parametrize a linear SSM and it is trained from the entire dataset of raw time series and associated covariates, allowing the model to automatically extract features and learn complex temporal patterns. SSM parameters can also be used to inspect each time series independently and interpret the solution.

- In [48] the authors propose a probabilistic model called **DeepAR**. DeepAR is an autoregressive recurrent neural network able to globally forecast hundreds of time series in a dataset. The model is able to deal with multiple time series having different scales, exploit relations between them and provide reliable forecast for new time series having smaller historical data, compute quantile estimates from the probabilistic forecasts. The model uses an LSTM to process historical values and covariates (the model automatically adds some calendar information from the date) related to the time step to forecast. The initial value of the LSTM module is given by the embedding output of an ”encoder” network, which in this architecture is the LSTM model itself, trained on the whole historical data available. The initial values of the ”encoder” network are zero vectors. The outputs of the recurrent network are used in a customizable likelihood function. The authors propose a Gaussian likelihood for real-valued data and negative-binomial likelihood for positive count data. During training all time series are involved and multiple sequences are extracted from them at different starting points, allowing the model to learn the behaviour of new time series taking into account all other available features. It follows a **teacher forcing approach** where the model uses the true data as regressive inputs only during training.

- Also the work proposed in [50] deal with the problem of forecasting multiple time series. The authors propose **DeepGLO**, a deep forecasting model which ”thinks globally and acts locally”. In particular, DeepGLO is a hybrid model that combines a global matrix factorization model regularized by a TCN, along with another TCN that can capture local properties of each time-series and associated covariates. The main idea of this model is to use global data not only during training, by looking at all the time series, but also include global information for the prediction of each time series. Global information are captured by a matrix factorization model regularized by a TCN. Matrix factorization

allow to represent the global information in a low rank matrix which can be thought of to be comprised of k basis time-series that capture the global temporal patterns in the whole data-set made of $n \gg k$ time series. At the end, DeepGLO takes the predictions from the global TCN-MF model and uses them as covariates for a separate TCN.

- **N-BEATS** proposed in [38] is an interpretable architecture consisting of an ensembled feed forward networks with stacked residual blocks of forecasts and backcasts. The basic block is able to predict the future values and its reverse, called backcast, using multiple fully connected layer ending with two heads. Multiple basic blocks are arranged to follow the double residual stacking principle. In particular, the residual system allows to input to every block a vector which is made up of element wise subtraction of previous block's backcast output and input. This residual approach can be interpreted as a form of forget mechanism similar to the one proposed by LSTMs. Finally, the outputs of every stack are summed to obtain the final global forecast.

The superiority of deep learning models is still being questioned, for instance the work described in [15] compares deep learning solutions and machine learning ones, bringing evidences on why well configured machine learning approaches should not be dismissed. It is possible to find another study in [54] where the authors compare ARIMA, XGBoost and LSTM-based models, demonstrating the great capabilities of classical machine learning approaches over deep learning. One of the possible causes behind the poor results is the trend of deep learning methods to overfit due to the large difference between the size of the training data and the parameters to fit. An interesting research direction is on hybrid solutions able to incorporate domain knowledge, a perfect example of these type of models is the M4 winner ES-RNN, described in [52], which uses ES to capture non-stationary trends and learns additional effects with a RNN.

2.3 Graph neural networks

This section reviews the state of the art of Graph Neural Networks (GNNs), including the basic concepts behind them, their applications and their different categories. At the end of this part the reader has the required knowledge to understand the proposed methods. In **graph theory** graphs are described as mathematical structures used to model pairwise relations between objects. A graph is represented as $G = (V, E)$ where V is the set of *nodes* or vertices and E is the set of *edges* or links. Let $v_i \in V$ to denote a node and $e_{ij} = (v_i, v_j) \in E$ to denote an edge between v_j and v_i. When an edge connects the same node, namely it starts and ends in the same node $v_i = v_j$, it is called *loop*. Graphs are called **undirected** when edges link vertices

symmetrically or **directed** when edges link vertices asymmetrically. In directed graphs, edges create an asymmetric relationship between connected nodes, the node where the edge originates is called *origin*, while the node which receives the edge is called *destination*. For undirected graphs this distinction does not exist and roles are interchangeable. Edges of a graph can be represented with two lists having the same length representing the connected nodes. In certain formal definitions and software designs it is common to see undirected edges being modelled as pairs of directed and opposite edges. The **neighbourhood** of a node v is defined as the set of nodes $N(v) = u \in V|(v, u) \in E$, the number of edges incident to a node is called **degree**. There are multiple ways to represent a graph, an *adjacency matrix* A is a $n \times n$ matrix with $A_{ij} = 1$ if $e_{ij} \in E$ and $A_{ij} = 0$ if $e_{ij} \in E$. *Adjacency lists* represent an alternative method to represent graphs, it consists of a list of lists where the indexes of the outer list represent the nodes and the inner lists contain the neighbours of each node. A graph may have node features X, where $X \in R^{n \times d}$ is a node feature matrix with $x_v \in R^d$ representing the feature vector of a node v. Similarly, a graph may have edge features X^e, where $X^e \in R^{m \times c}$ is an edge feature matrix with $x^e_{v,u} \in R^c$ representing the feature vector of an edge (v, u). A very common type of edge feature is a simple number, usually called weight, which can be used to represent for instance the intensity, the cost or the length of the connection. Since weights are very common, graphs whose edges are associated to a numerical value are just called **weighted**, while the others are called **unweighted**. Graphs are called **complete** when every pair of nodes is connected by an edge and are called *bipartite* when their vertices can be separated into two disjoint sets, such that every edge runs from a vertex of the first group to a vertex in the second one. Graphs are called **connected** when there is a path between every pair of nodes, if there are no cycles, the graphs can be called trees. Nodes and edges in **homogeneous graphs** must have the same types, while nodes and edges can have different types in **heterogeneous graphs**. Heterogeneous graphs are used to represent multiple types of relationships between different objects, during this work only homogeneous graphs are took in consideration. In the context of GNN graphs can be considered either **static** or **dynamic**. Static graphs are just classical graphs, while dynamic, or temporal, graphs are used to represent evolving phenomena such as social networks or particle physics interactions. Dynamic graphs are expected to change frequently and usually they have ad-hoc representations which include temporal information.

2.3.1 Motivations

Traditional deep learning models such as CNNs or RNNs are successfully used to work with different types of *Euclidean* data structures, but they fail to handle non-Euclidean data such as graphs. While images can be plotted in n-dimensional linear space, graphs cannot be projected

into a space without loosing parts of the original information. Graph Neural Networks (GNNs) are proposed to combine the feature information and the graph structure to learn better representations on graphs via feature propagation and aggregation. Due to its convincing performance and high interpretability, GNNs have recently become a widely applied graph analysis tool. GNNs are closely related to **network embedding**. Network embedding refers to the approach of learning latent low-dimensional feature representations for the nodes or links in a network. In this way, the originated vector representations can be easily used by any traditional machine learning method. Usually, the nodes in the network are encoded such that the similarity in the embedding space reflects the similarity in the network. Well known network embedding methods are Spectral Clustering, DeepWalk, Large-scale Information Network Embedding (LINE) and node2vec [3]. Many of the embedding models propose sophisticated and scalable methods to traverse the graphs, while building representations and optimizing objective functions able to preserve the relationships between the nodes in the latent space. Embedding complete graphs can be achieved in multiple manners, some of the methods exploit node embedding, either averaging them or considering the graph as a virtual node. While node embeddings perform the same task in providing useful representations, GNNs are a group of neural network models which are designed to solve various tasks in an end-to-end fashion. The main characteristic of a GNN relies in its application of a **neural message passing** framework, in which, vector messages are exchanged between nodes in the graph and updated using neural networks. At the iteration $k+1$ the embedding $h_v^{(k+1)}$ of the node v is obtained following the equation 2.4

$$h_v^{(k+1)} = UPDATE^{(k)}(h_v^{(k)}, AGGREGATE^{(k)}(\{h_u^{(k)}, \forall u \in N(v)\})) \tag{2.4}$$

where $UPDATE$ and $AGGREGATE$ are differentiable functions such as neural networks. In this context the message is given by the output of the $AGGREGATE$ function. Figure 2.3 gives an example of the mechanism.

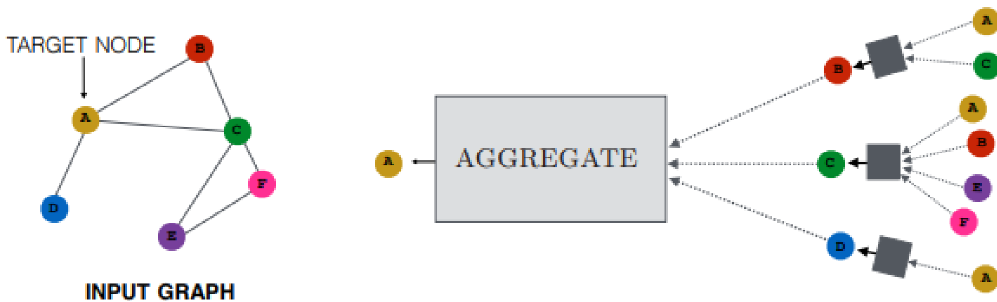


Figure 2.3: An example of a two-layer message passing model where the nodes B, C and D aggregate information from their neighbours and are used to update A.

At the kth iteration every node obtains information from its kth hop neighbourhood. It is expected to observe that each node obtains both topological and feature information from their neighbourhood

The concept of GNN was initially proposed in [49] and [18]. The vanilla GNN explained by [49] extends traditional neural networks to deal with homogeneous undirected graphs, where both nodes and edges can have their features. Node states are defined by equation 2.5, while output approximations by equation 2.6, where $E(v)$ are the edges in the node v, f is called local transition function and g local output function. Both f and g are parametrized functions.

$$h_v = f(x_v, x_{E(v)}, h_{N(v)}, x_{N(v)}) \tag{2.5}$$

$$o_v = g(h_v, x_v) \tag{2.6}$$

Node states depend on the node features x_v, neighbours features $x_{N(v)}$, edge features $x_{E(v)}$ and the neighbors nodes states $h_{N(v)}$. Training occurs in a supervised fashion, in particular, at each iteration the node states are computed based on the results of the previous iterations, until certain requirements are met. As in traditional neural networks, real values and predicted values, computed by g, are used to calculate the loss and optimize the objective function with gradient descent and backpropagation. During this process, the parameters of the functions are optimized.

In conclusion, a simplified implementation of the vanilla GNN can be obtained by proposing a neural network based message passing method, which aggregates summing the messages incoming from the neighbours, and combines the neighborhood information with the node's previous embedding using a non-linear fully connected network.

2.3.2 Static graph models

From a practical perspective, static graphs are usually considered the standard and the majority of GNN architectures, if not differently specified, are not expected to properly work with dynamic graphs. However, many models described in this part can be adapted or used to develop methods for dynamic graphs, so this part can be considered as a essential overview of GNN models. GNNs use graphs topological structures and features from nodes and links to solve problems at different levels.

- **Node level** tasks focus on nodes, including node classification, node regression and node clustering. Node classification tries to categorize nodes into several classes, and node regression predicts a continuous value for each node. Node clustering aims to partition

the nodes into several disjoint groups, where similar nodes should be in the same group. Node level tasks are very common because they can be used to generate inputs for more complex problems, thus, many GNNs architectures learn node representations and fed them to a final custom layer, imitating the objective of node embedding techniques.

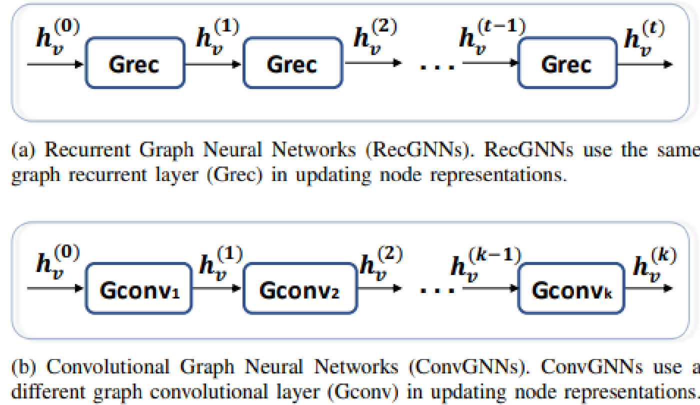
- **Link level** tasks are edge classification, which require the model to classify edge types or predict whether there is an edge existing between two given nodes. Another task is edge prediction, where edge features are predicted. A typical link prediction approach is using pairs of nodes states to represent edges.

- **Graph level** tasks include graph classification, graph regression, and graph matching, all of which need the model to learn graph representations. In graph regression, a model learns how to predict one or more numerical values associated to a graph.

The type of task influences the decision of the loss function used at training time. Graph learning tasks can either be **supervised**, when the data is labelled, **unsupervised**, when data labels are not available as in clustering, or **semi-supervised** when there is a small amount of labelled nodes and a large amount of unlabeled nodes for training. GNNs have many applications across different domains [62], for instance, they are used in computer vision to recognize relationships between objects in images or generate images from semantic representations. GNNs are used also in 3D computer vision to process point clouds or in natural language processing to extract relationships among documents and words, which can be used to classify text, extract knowledge or generate text. Another important domain is chemistry, where GNNs are involved in predicting molecular properties, inferring protein interfaces and synthesizing chemical compounds.

GNNs can be categorized into **recurrent graph neural networks (RecGNNs)**, which are historically the first proposed models, **convolutional graph neural networks (ConvGNNs)**, whose development was encouraged by the success of CNNs in the computer vision domain, and **graph autoencoders (GAEs)** which are among the most recent methods. RecGNNs learn node representations by applying the same set of parameters recurrently over the nodes of a graph. Vanilla GNN is considered a RecGNN method which iteratively alternates an information diffusion algorithm and a parameter gradient computation until a stable equilibrium is reached. A more recent model, the Gated Graph Neural Network (GGNN) [30] employs a GRU as a recurrent function, which fixes the convergence to a specific number of steps. Every node state is updated accordingly to the previous values as well as the neighbouring states.

Instead of iterating the nodes states, ConvGNNs propose an architectural solution based on a fixed number of layers with different parameters. The figure 2.4 shows the conceptual difference between RecGNNs and ConvGNNs.



(a) Recurrent Graph Neural Networks (RecGNNs). RecGNNs use the same graph recurrent layer (Grec) in updating node representations.

(b) Convolutional Graph Neural Networks (ConvGNNs). ConvGNNs use a different graph convolutional layer (Gconv) in updating node representations.

Figure 2.4: The difference between RecGNNs and ConvGNNs from [62].

ConvGNNs are divided in two groups: **spectral-based** and **spatial-based**. The methods from the former group are inspired by signal processing theory, on the other hand, the methods from the second follow a similar information propagation approach to the one proposed by RecGNNs. As the name suggests, spectral approaches work with a spectral representation of the graphs and define the convolution operator in the spectral domain [69]. They first compute the normalized graph Laplacian, then they eigendecompose it to obtain its matrix of eigenvectors. Finally, the eigenvectors are processed, using a convolutional operation defined in the Fourier domain. This operation results in potentially intense computations and non-spatially localized filters [33]. At training time the sets of parameters of the convolutional operators are iteratively updated. The most common spectral models are the Spectral Network, ChebNet, Graph Convolutional Network (GCN), Adaptive Graph Convolutional Network (AGCN) and the Dual Graph Convolutional Network (DGCN) [69]. Similarly to CNNs, spatial methods update the central node's representation by convolving it with its neighbours' representations. From another perspective, this is the same message passing approach of RecGNNs [62]. There are several spatial-based ConvGNN models, such as the Diffusion Convolutional Neural Network (DCNN), Learnable Graph Convolutional Network (LGCN), GraphSAGE, Graph Attention Networks and many others listed in [69] and [62]. **GraphSAGE** [20] is a general framework for inductive node embedding. The principal breakthrough of the proposed method is the possibility to train an embedding function able to generalize to unseen nodes. In particular, a node embedding is obtained by repeating a sampling and aggregating routine shown in figure 2.5.

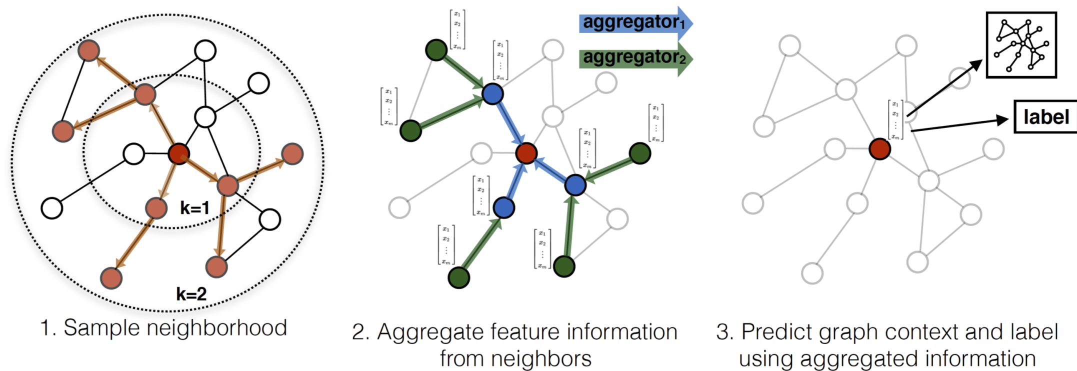


Figure 2.5: GraphSAGE sample and aggregate mechanism from [20]. An important detail in the image is the usage of different colours to underline that the model uses two distinct aggregators. Each aggregator learns its own set of parameters and can be applied to each node, including unseen ones.

At each iteration, a fixed-size neighbourhood is sampled for each node, the features of the neighbors are aggregated and are concatenated with the result from the previous iteration. Finally, the aggregation is passed to a non-linear fully connected network which outputs the final result of the iteration. During training the aggregators and the weights of the fully connected layers are optimized following standard gradient descent and backpropagation. GraphSAGE uses a graph-based loss function which encourages nearby nodes to have similar representations, while enforcing that the representations of disparate nodes are highly distinct. The authors propose different aggregator functions, such as LSTM, mean and max-pooling. GraphSAGE does not learn embeddings for each node, it learns a function that generates embeddings by sampling and aggregating neighbours' features, which allows it to generalize to new nodes. Instead of comparing all neighbours equally, **GAT** [59] introduces an attention mechanism into the propagation step. In particular, given a node it computes the normalized attention coefficients with its neighbours using a graph attentional layer. The graph attentional layer passes the nodes representation to a shared linear layer, then it concatenates the results and weights them, finally passing the resulting vectors to a LeakyReLU activation layer. Since the model applies a multi-head attention mechanism, it computes for every neighbour multiple independent normalized attention coefficients which are finally concatenated or averaged to obtain the final representations. **Graph Isomorphism Network (GIN)** [64] is a simple yet powerful GNN. It was proposed as a special case of spatial GNN suitable for graph classification tasks. Aggregation and combination are expressed in equation 2.7.

$$h_v^{(k)} = MLP^{(k)}((1 + \epsilon^{(k)}) \cdot h_v^{(k-1)} + \sum_{u \in N(v)} h_u^{(k-1)}) \tag{2.7}$$

where $\epsilon^{(k)}$ is a learnable parameter and MLP is a fully connected non-linear neural network.

GAEs learn how to map graphs into latent feature spaces and how to decode them back, thus, they can be used to create entire network embeddings or to parametrize graph generative distributions. Graph embeddings can be trained through the reconstruction of graph structural information like the graph adjacency matrix. Graph generation can occur sequentially, by proposing nodes and edges every step, or all at once, generating entire graphs [62].

GNNs are not only made by the computational modules described above, recent successes stand over advancements in **sampling** and **pooling** methods. Sampling techniques, such as the one proposed by GraphSAGE, aim to alleviate the "neighbour explosion" effect caused by message passing models which, by definition, blindly consider all neighbors at different depths, exponentially increasing the number of considered nodes. There are three different sampling layers families [69]: node sampling, layer sampling, subgraph sampling. While node sampling just select a subgroup of neighbouring nodes, layer sampling retains a small set of nodes for aggregation in each layer to control the expansion factor, and subgraph sampling limits the nodes only inside the sampled subgraphs. Similarly to computer vision, pooling layers are applied in GNNs to extract more general features from complex embedding structures. Direct pooling layers work at node level, while hierarchical pooling layers learn graph representations by layers [69].

2.3.3 Dynamic graphs models

Learning on dynamic graphs is a relatively recent task and most works are limited to the setting of discrete-time dynamic graphs represented as a sequence of snapshots of the graph, called *discrete-time dynamic graphs (DTDG)*. Alternatively, dynamic graphs can be also referred to *continuous-time dynamic graphs (CTDG)* when they can be seen as timed lists of events, which may include edge addition or deletion, node addition or deletion and node or edge feature transformations. As previously mentioned, there are multiple methods to perform representation learning on graphs based on the encoder-decoder principle. Some encoders adapts common static graph encoding techniques to dynamic graphs by performing aggregations of the graph snapshots over time into a single graph, or aggregations of the encoded snapshots. Other approaches based on decompositions methods, regularizations and random walks can be found in [26] and are widely adopted for DTDG. **EvolveGCN**, proposed in [39], uses a RNN to evolve the parameters of a GCN along the temporal dimension of a sequence of consecutive

mutating graphs. EvolveGCN aims to overcome the limitation of previous works where spatial and temporal aspects are modelled separately, requiring to know the nodes over the whole time span. An EvolveGCN is very similar to a RNN, only the parameters of the RNN are optimized, while the parameters of the GCN are computed by the RNN. In particular, the RNN inside the EvolveGCN model is used to predict the parameters of the GCN, while the GCN is ultimately used to create embeddings. Two versions are proposed, the -H treats the GCN parameters as hidden states and the node embeddings as input of a recurrent architecture, while the -O considers the GCN parameters as input and outputs of a recurrent architecture. CTDG are less explored and the related work usually involves ideas and models from RNNs and GNNs as in the case of Temporal Graph Networks (TGN) [46].

Spatio-temporal graphs are specific cases of dynamic graphs where the topology of the graph is fixed, a classic application is traffic prediction. Traffic prediction can be involved in multiple applications, such as route planning and order dispatching for taxi services or business location to improve the life quality in smart cities. Generally there are three kinds of traffic prediction problems as summed by [66]:

- Traffic classification, aims to classify traffic data, for instance classify among different types of transportations or detect anomalous behaviours.
- Traffic generation, is used to generate traffic data to perform data augmentation for deep learning models or improving testing environments.
- Traffic forecasting, is involved in the prediction of any traffic data, such as traffic speed, traffic flows, travel demand and time. In this scenario the data can either represent multiple regions or specific road networks.

Other very commons applications can be found in epidemiology, meteorology and medicine. In these domains the data is usually referred as spatio-temporal as it is collected across both space and time and it describes a phenomenon in a particular location and period of time. Spatio-temporal data can assume a huge variety of forms as they can describe events, trajectories, recordings on moving or fixed objects, which can be translated into different instances types, including time series. It is possible to distinguish time series for every spatial unit or multi-dimensional time series comprising all the spatial identifiers, [4] gives a more detailed description of the different types of spatio-temporal data and methods. Recently, GNNs have been extensively used in the traffic prediction context, replacing the older and more traditional time series forecasting methods like ARIMA, Hidden Markov Models and KNN, and more specifically the CNN portion of deep recurrent approaches. An extensive list of GNNs solutions can be found in [45]. Although many models perform quite well, the majority of them

solve separately the spatial and temporal problem, exploiting GNNs mainly for the spatial side of the problem, while only few recent works have attempted to unify the two sides [47].

3 Dataset

This chapter introduces the reader to the data involved in the evaluation of the methods proposed in this work. After a specific overview of the unique characteristics of the dataset, a deeper data analysis is presented to provide strong foundations on future interpretations and analyse the complexity, the advantages and the main challenges of the chosen dataset. This phase represents the first contact with the data and it aims to acquire as much knowledge as possible to develop a valuable set of intuitions and hypothesis to assess in the posterior modelling phase. Moreover, a concise review of the most interesting aspects of the competition, including the proposed metrics, is also considered.

3.1 M5 competition

The dataset used in this work comes from the M5 competition, a time series forecasting challenge hosted by the platform Kaggle[1] in 2020. The M5 competition is the most recent Makridakis Competition (M Competition). In the first editions, started in the 1980s, the majority of the participants were domain experts and machine learning solutions were really discouraged. The previous edition, the M4, concluded with ambiguous considerations on the advantages of machine learning and deep learning solutions over traditional methods. The M5 aimed to investigate on this direction and to include not only domain experts but also machine learning practitioners. The M5 competition was a success, attracting almost 6000 participants and teams and receiving a considerable interest by various scientific communities. Differently from the previous editions, it is the first dealing with sales forecasting with such a complex hierarchical structure and display of intermittency. The competition's objective is to produce the most accurate point forecasts for 42840 time series that represent the hierarchical daily unit sales of the largest retail company in the world, Walmart. The amount of time series and the complex nature of the data posed a real challenging environment for traditional methods encouraging new methods. The M5 competition gave the opportunity for everyone to understand how complex

[1] https://www.kaggle.com/c/m5-forecasting-accuracy

certain time series forecasting problems are in industrial applications.

In the first stage of the competition, called *validation* the participants were required to forecast 4 weeks (28 days) given the data of the 1913 days before. After the end of this stage, the days to forecast in the validation were made available and the participants were asked to perform a final submission, in a new stage called *evaluation*, consisting of 28 days of sales, following the previous ones, and used to determine the winning methods.

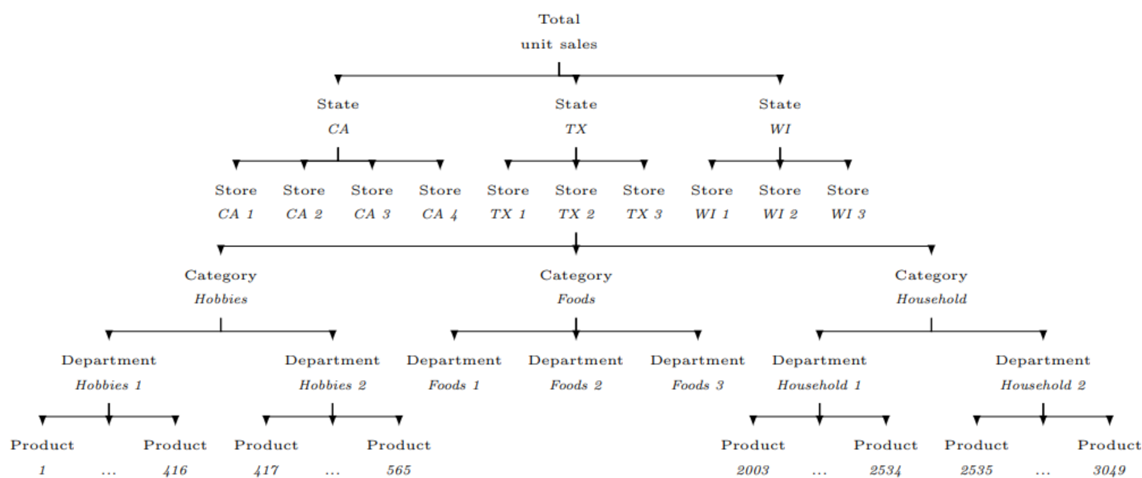


Figure 3.1: The hierarchical representation of the M5 data from [35]

Level id	Level description	Aggregation level	Number of series
1	Unit sales of all products, aggregated for all stores/states	Total	1
2	Unit sales of all products, aggregated for each State	State	3
3	Unit sales of all products, aggregated for each store	Store	10
4	Unit sales of all products, aggregated for each category	Category	3
5	Unit sales of all products, aggregated for each department	Department	7
6	Unit sales of all products, aggregated for each State and category	State-Category	9
7	Unit sales of all products, aggregated for each State and department	State-Department	21
8	Unit sales of all products, aggregated for each store and category	Store-Category	30
9	Unit sales of all products, aggregated for each store and department	Store-Department	70
10	Unit sales of product i, aggregated for all stores/states	Product	3,049
11	Unit sales of product i, aggregated for each State	Product-State	9,147
12	Unit sales of product i, aggregated for each store	Product-Store	30,490
Total			42,840

Figure 3.2: The number of M5 series per aggregation level from [35]

The dataset involves the unit sales of 3049 products, classified in 3 product categories (Hobbies, Foods, and Household), and 7 product departments in which the above-mentioned categories

are disaggregated. The products are sold across 10 stores, located in 3 States: California "CA", Texas "TX" and Wisconsin "WI". The figures 3.1 3.2 clearly show how the data is distributed across the different levels and the cardinality of each of them. In this work, levels near the 12th are referred as *lower/last levels*, because their granularity is smaller than those at the *higher/first levels*.

In addition to the historical data, the competition also includes selling prices and calendar information. In particular, for each combination of products and stores it is reported the averages of the selling prices over each week, if no sells has occurred this value is not available. Special events and holidays are categorized into four classes, namely "Sporting", "Cultural", "National", and "Religious". For each day there can be up to two different events, both the name and the category of the events are reported. Each day specifies whether Supplemental Nutrition Assistance Program (SNAP)[2] activities are performed in each of the three available states.

The competition proposes an original evaluation metric called weighted root mean squared scaled error (WRMSSE). WRMSSE is a weighted version of the root mean squared scaled error (RMSSE), a variation of MASE [23] and defined in equation 3.1, where y_t is the real value at time t, $\hat{y}_t$ its forecast, n the number of training samples and h the horizon.

$$RMSSE = \sqrt{\frac{\frac{1}{h}\sum_{t=n+1}^{n+h}(y_t - \hat{y}_t)^2}{\frac{1}{n-1}\sum_{t=2}^{n}(y_t - y_{t-1})^2}} \tag{3.1}$$

The denominator of the RMSSE, which is the in-sample, one-step-ahead mean squared error of the Naive method, is computed only for the periods during which the examined products are sold [35]. Similarly to MASE the RMSSE is scale independent and penalizes equally positive and negative forecast errors. Finally, the total WRMSSE scores for each level are computed by weighting the RMSSE of each forecast, the overall WRMSSE is the average of all the WRMSSE at each level, since each of them has the same importance. Equation 3.2 shows how the WRMSSE of a single level, having k time series is computed, w_i is computed based on the last 28 observations of the training sample of the dataset, specifically as the sum of units sold multiplied by their respective price [35].

$$WRMSSE = \sum_{i=0}^{k} w_i * RMSSE_i \tag{3.2}$$

In conclusion, the following list shows the multiple reasons behind the choice of the M5 competition data as a dataset to benchmark the methods proposed in this work.

[2] In the United States, SNAP formerly yet still commonly known as the Food Stamp Program, is a federal program that provides food-purchasing assistance for low- and no-income people [12].

- One of the main motivations is the presence of exogenous data. Traditional time series datasets usually lack of additional features, while in the M5 competition the features are explicitly added to facilitate the proposal of machine learning based solutions. For instance, all the previous M Competitions do not provide exogenous features.
- It is a multiple time series forecasting dataset. This is a general motivation to adopt machine learning solutions instead of traditional ones.
- It is a hierarchical time series dataset. This allows to expose the proposed methods to different types of time series, from the intermittent ones at the lower levels to more traditional ones at the higher levels.
- Discussions and results are publicly available. Participants and organizers have provided through the competition hosting website Kaggle and journal publications numerous insights and comments on the competition. Previous solutions help avoid common mistakes and suggest data preparation procedures. Organizers publications clearly explain the problem, the metrics and the winning solutions, providing also benchmarks results from traditional time series prediction models [35]. Results obtained from the organizers can be used to avoid a tedious evaluation work and ensure that they are obtained from the work of a team of domain experts.
- It comes with an agreed metric used to evaluate the models and reflecting the business nature of the problem.

3.2 Data analysis

The most straightforward way to examine time series is plotting them. There are other methods to study the properties of time series. The already mentioned auto-correlation functions can be used to measure how the signal as a function of the time is correlated. Spectral analysis, trend estimation and time series decomposition are other advanced techniques which can be useful to study the properties of the observed time series. However, classical data exploration methods can become difficult to apply when dealing with a vast number of time series. In this section, the competition data is explored and studied in order to understand its complexity and make meaningful observations useful to analyse the methods and the results of this work.

3.2.1 Sales

The data analysis begins with the target variable: the number of daily sales. One common way to start an Exploratory Data Analysis (EDA) is observing some examples of time series at

different aggregation levels. The figure 3.3 show some examples of time series.

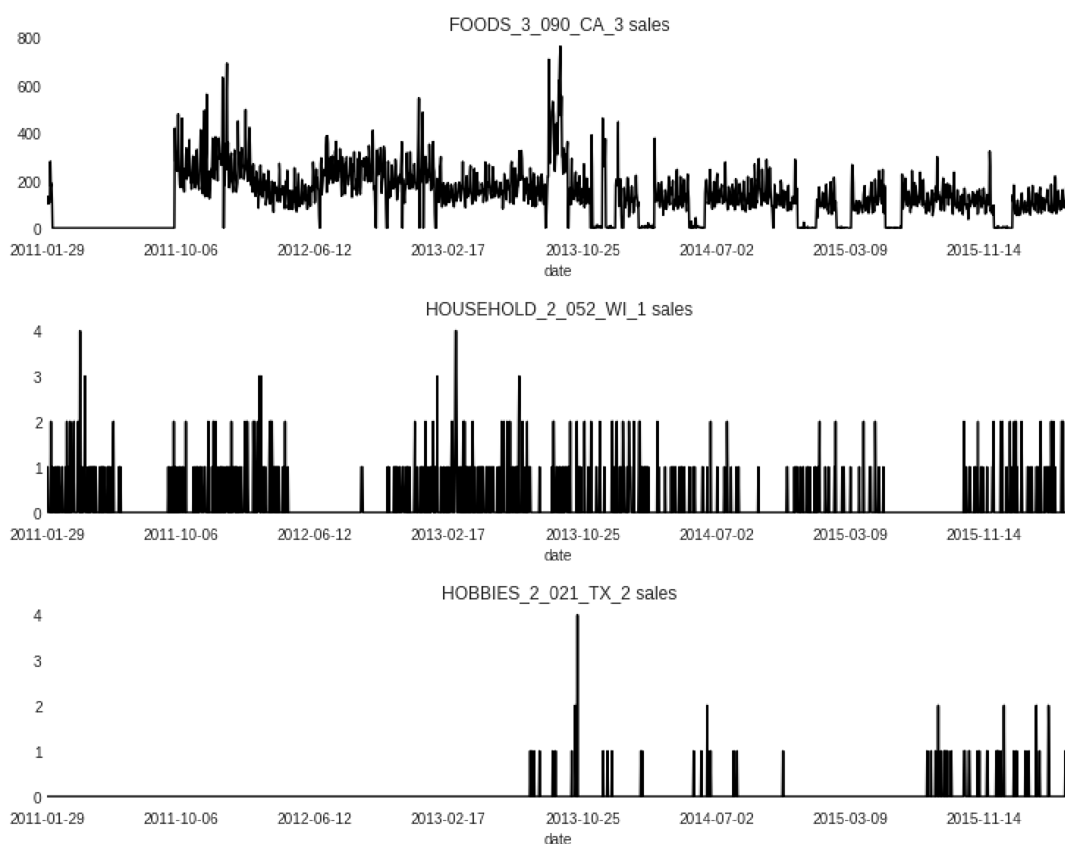


Figure 3.3: Three time series sampled from the the 12th level. The names of the time series can be read in following way: the first part denotes the product, for example ”FOODS_3_090” represents the product 90 of the department ”FOODS_3”, while the second part the store, for example ”CA_3” is the third store situated in California.

Time series represent intermittent count data, the scales between single products sold in specific stores may significantly vary. Some products are not sold for long periods and others seem to become available only later the start of the available training data. After 2016 almost all time series have registered at least a sale in the past, but it seems that some time series represent products launched just in 2016. A more formal and systematic way of observing the sales is plotting their distribution at each aggregation level, as shown in figure 3.4.

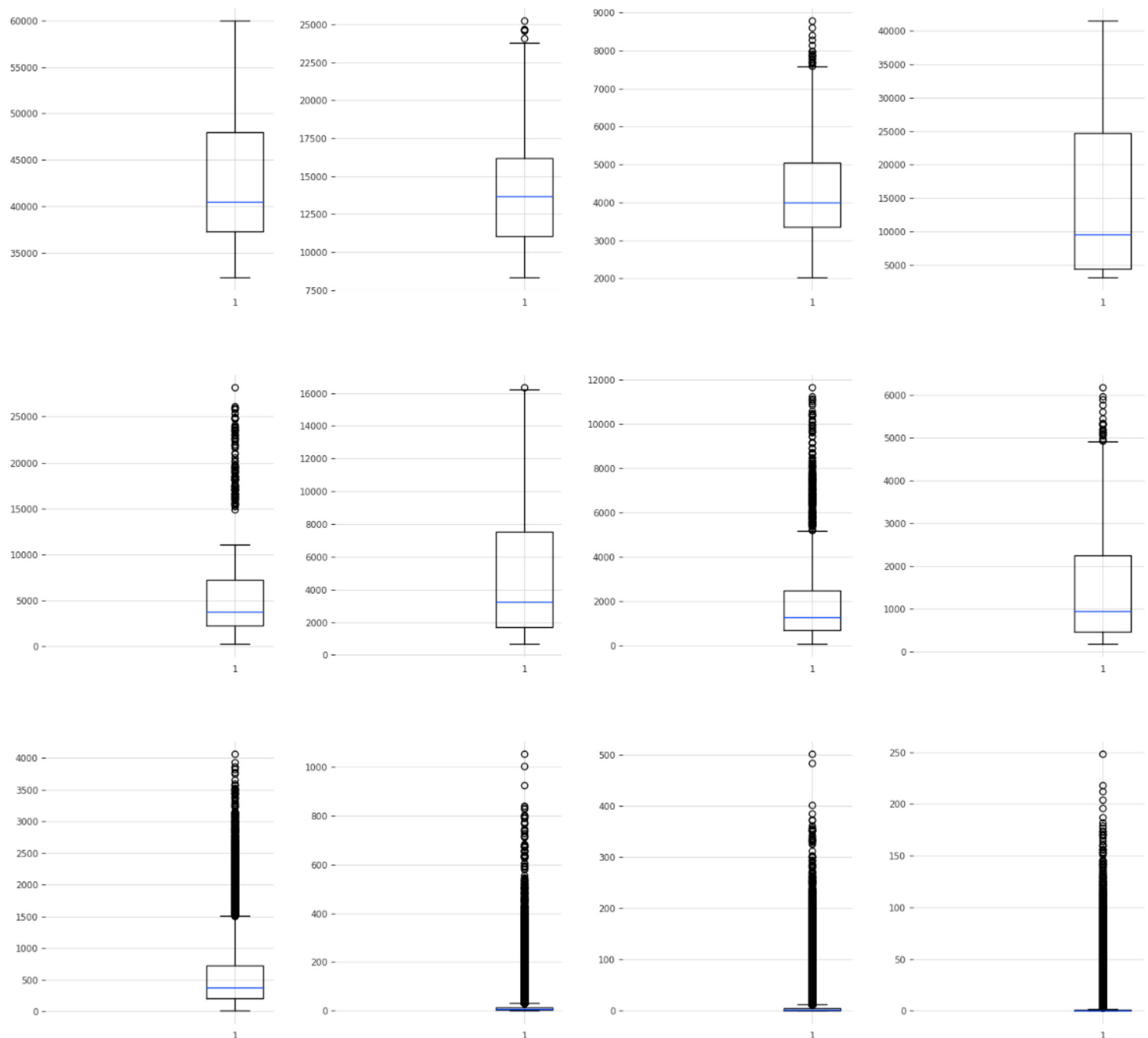


Figure 3.4: Boxplots representing the distribution of the sales sales taken from the last 100 points from each time series in each level. Upper left corner is the 1st level, where maximum sales reach 60K, while in the bottom right cornet the sales of the 12th level where an outlier reaches a maximum of 248.

A very interesting detail is the number of zeros. Among the data points of the 10th level 8.5% are zero, on the 11th level they are the 27% and in the 12th level they reach the 56%. This observation confirms the presence of intermittent time series starting from level 10. The other aggregations show an obvious decreasing of number of sales as the levels get closer to the 10th, but distributions are relatively similar. Only the 4th level shows a more accentuated tail distribution over larger values.

Another interesting way to visually explore the sales is observing the time series at higher aggregation levels by summing the daily sales and observe any patterns or anomalous behaviours. To enhance the visualization and deal with the high intermittency of the data, time series are smoothed, in particular each point of the plots 3.5 and 3.6 represents the average of the sales in the previous 7 days. In general, it is noticeable an upward trend and an annually seasonality particularly visible at Christmas where dips are clear caused by the commercial activities being closed. All the curves suggest other forms of seasonalities occurring in smaller periods of time, as weekly or monthly.

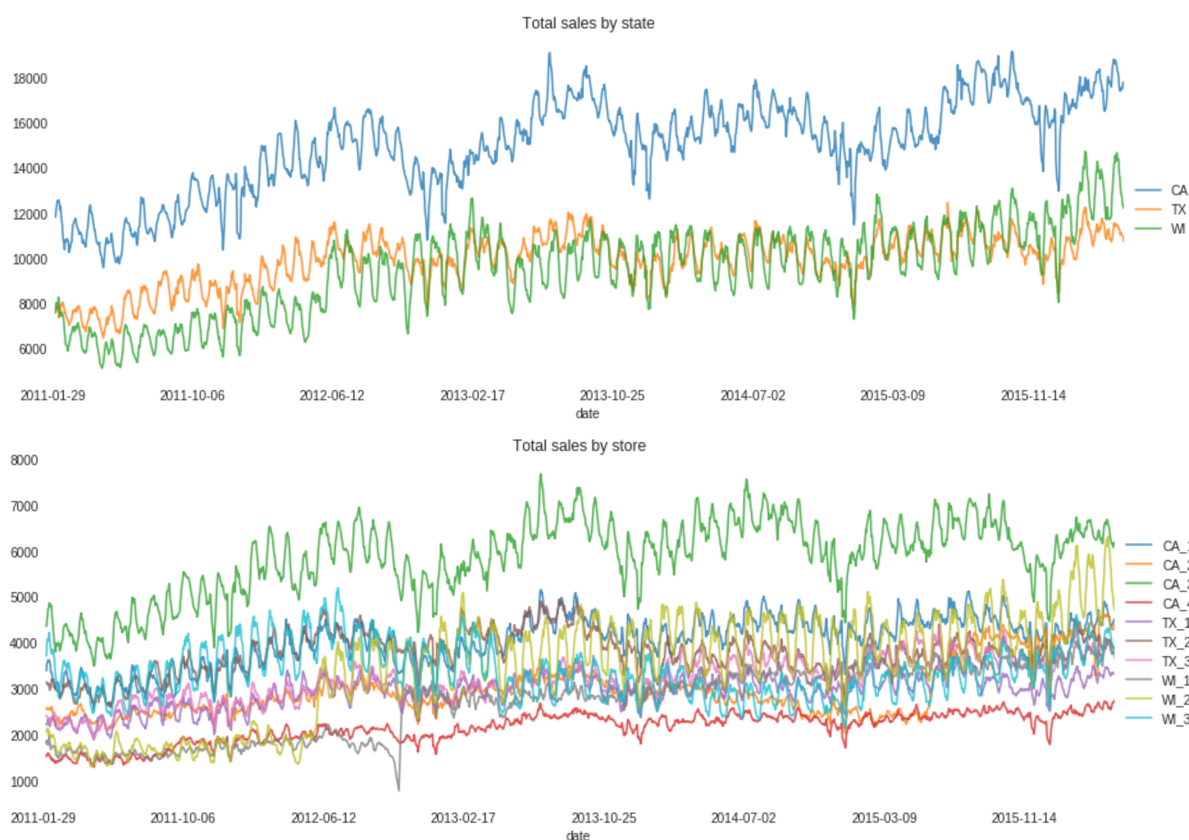


Figure 3.5: Time series space-related aggregations

The states plots in 3.5 suggest that the state may play an important role, since in California the numbers of sales are evidently higher than in the other states. Unfortunately, there is no visible difference between Wisconsin and Texas sales, and thus, from this perspective it is difficult to distinguish them. However, the stores plot in 3.5 confirms the presence of different scales and provides more detailed information which cannot be extracted only from the state level. In particular, Texas stores have quite similar sales volumes, with "TX_3" rising from "TX_1" level to "TX_2". The Wisconsin stores "WI_1" and "WI_2" show a curious jump in sales in

2012, while "WI_3" shows a prolonged decrease spanned over several years. The California stores are relatively well separated in sales volumes, denoting internal differences inside each state. In particular, as expected one Californian store, the "CA_3", has the highest total amount of sales, surprisingly, this pattern is not maintained all over the stores in California, as "CA_4" has one of the lowest volumes.

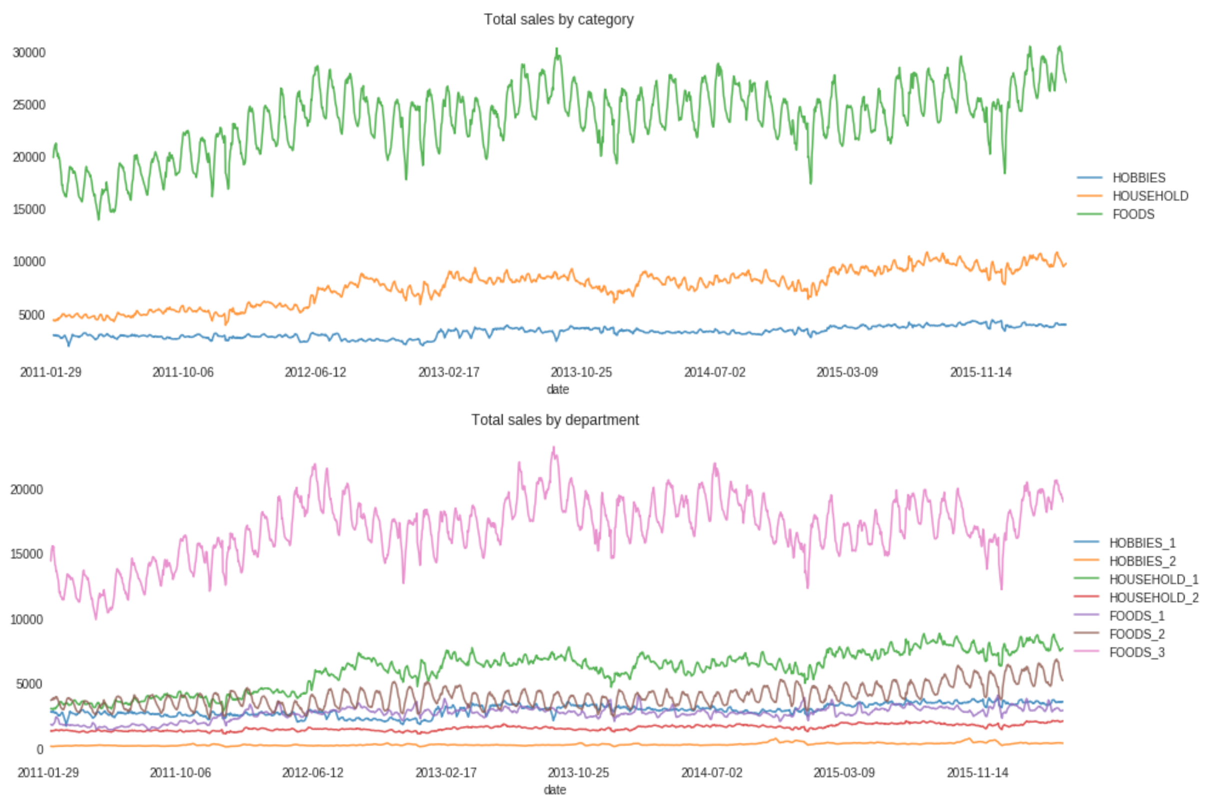


Figure 3.6: Time series product-related aggregations

Similarly, each category in 3.5 occupies a different scale of daily sales, namely "Foods" are the most sold items, they are followed by "Household" and "Hobbies". The departments sales in 3.5 confirms the previous observations, "FOODS_3" items are the most sold, they are followed by "HOUSEHOLD_1" and "FOODS_2". It is interesting to notice that the most sold departments show also the strongest variability, if the average number of "FOODS_1" items sold is around 17000, in ten months they can go from 14000 to 23000 sales per day with very large oscillations. On the other hand, other departments such as "HOBBIES_2" remains steady all over the observed years.

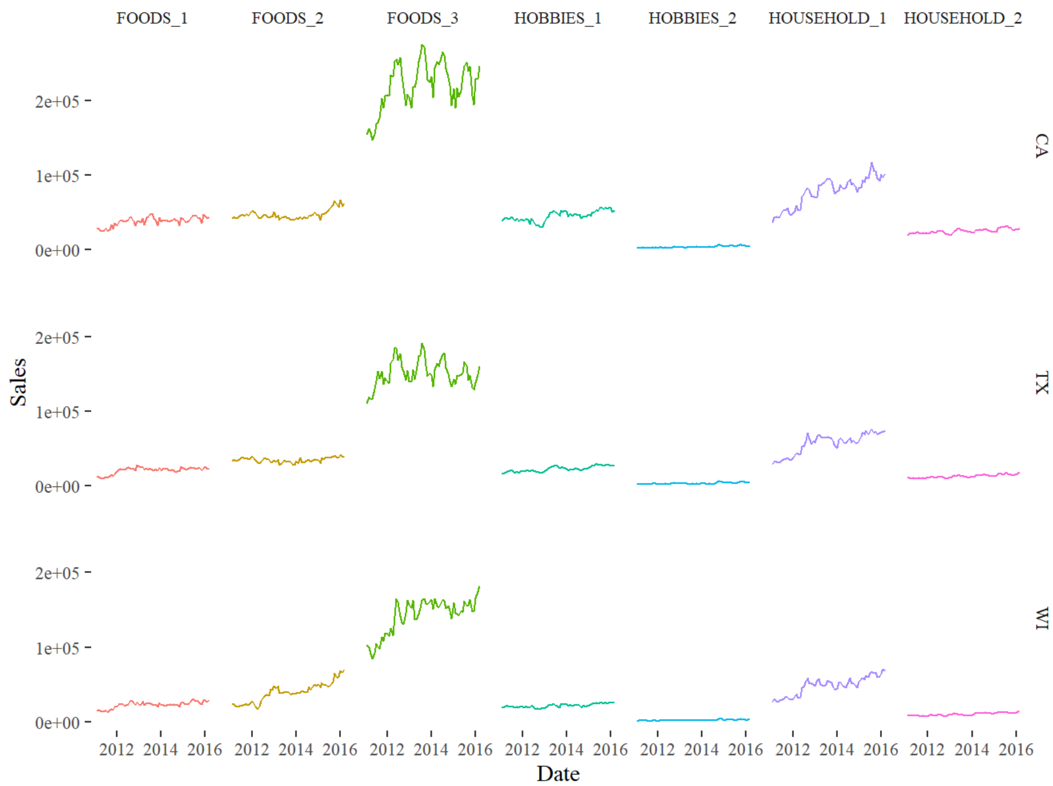


Figure 3.7: An insightful overview of the monthly sales aggregated by department and state from [1]. This represents also a view of the level 7 of the M5 hierarchy.

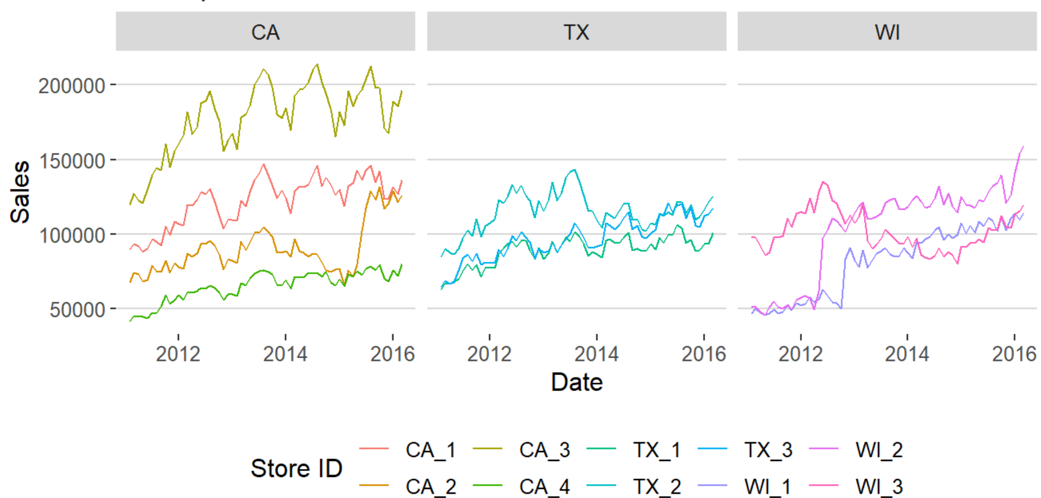


Figure 3.8: An interesting overview of the monthly sales aggregated by store and state from [1]

It is possible to better observe the seasonalities by temporally aggregating the data to create the **seasonal plots** [22] shown in image 3.9.

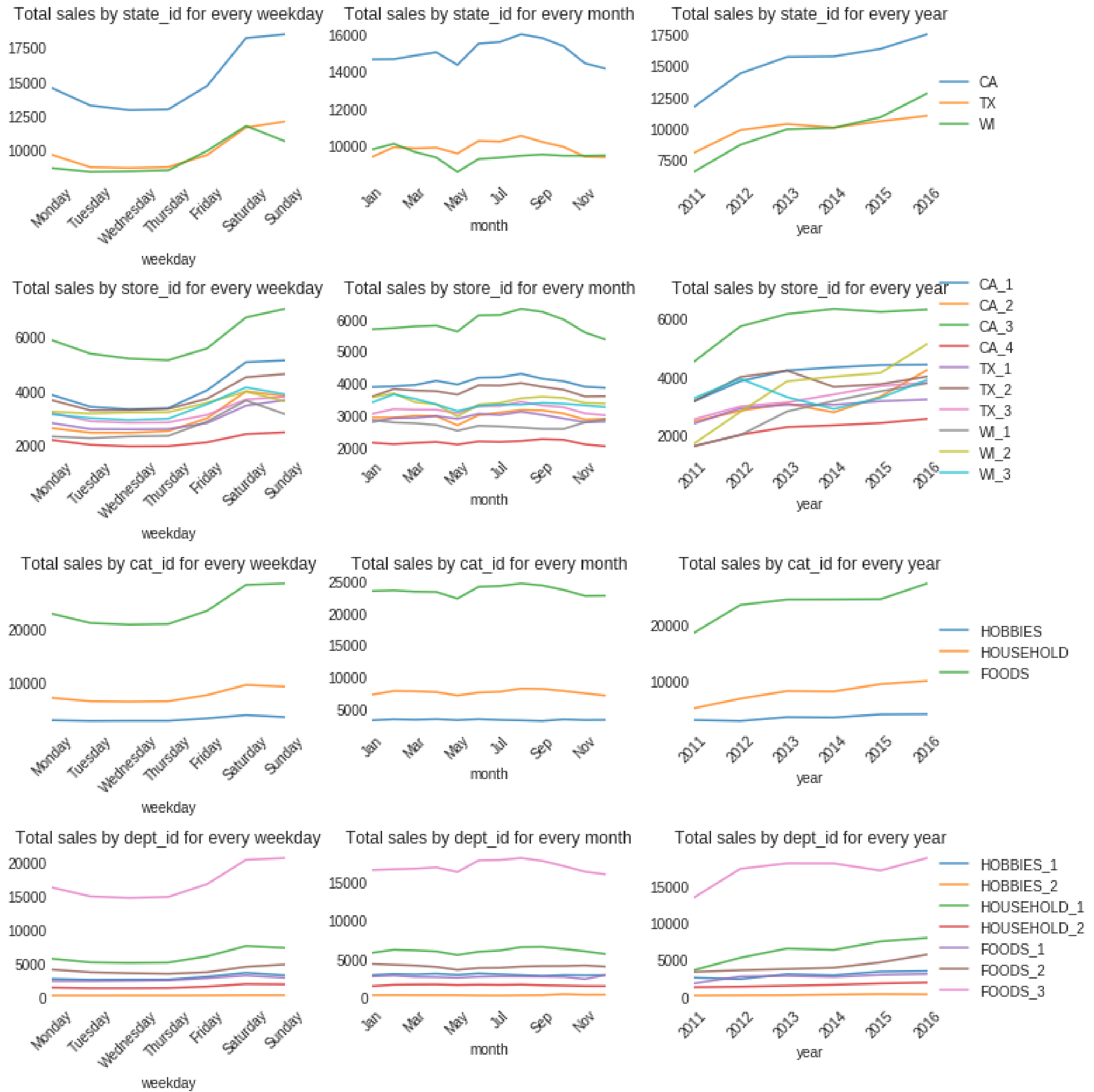


Figure 3.9: Seasonal plots for weeks, months and years.

The difference between weekdays and weekend is very similar for all three states, except for an interesting downturn in Sunday sales in Wisconsin, which is confirmed in all the categories. The monthly plots show a common decrease around the months May, November and December. Weekly seasonalities are maintained similar across different levels, while monthly show more

different patterns, suggesting the existence of multiple underlying processes. The ”Hobbies” category does not show any interesting pattern, suggesting a low and constant number of sales over time. In conclusion, it is worth considering information about the time series aggregation level or categorization, because it can provide meaningful indications on average volumes and trends observed over time.

3.2.2 Exogenous features

The M5 Dataset comes with various explanatory variables which can be used alongside the number of daily sales as covariates to improve the power of machine learning solutions. Each day contains the information for up to two possible events and the relative type, even if the number of two contemporaneous events is insignificant. Approximately 8% of days are associated to a special event, almost one third of them are ”Religious”, the other third are ”National” holidays and the remaining are split between ”Cultural” and ”Sporting” events. The amount of SNAP days is the same for all the three states, and they cover about 33% of the total days in the dataset. SNAP days seem to follow a monthly regular pattern and they recur only in the first half of the month and no more than 10 days every month. The impact of events on sales is noticeable and it follows some common beliefs. In general, events have a negative impact on the different categories of products, however in some cases, ”Sporting” and ”Cultural” events, such as the Superbowl, are associated to an increase of the number of ”Foods” sales. ”Religious” and ”National” holidays have a more consistent negative effect on the number of sales and the ”Hobbies” products show the most significant drop. Moreover, the number of sales tends to increase on SNAP days, especially for the ”Foods” items, which are included in the SNAP promotions, and more notably for the Wisconsin state.

As shown in figure 3.10, weekly prices are distributed similarly between each state, ”Foods” items are generally cheaper while ”Household” items are more expensive. The distributions of the ”Foods” departments are very similar while the two ”Household” departments show a more evident discrepancy. Interestingly, ”Hobbies” prices show a wider bimodal distribution which is less common in the other categories. ”Foods” category prices show an overall larger number of increasing changes over time, while ”Household” items seem to decrease their prices, although in general, the prices tend to increase for all states. There are some cases where prices and sales show a correlation, for instance, after long periods of zero sales, a change of price occurs and the sales restart. It can be supposed that this is the effect of some kinds of discount strategies or marketing campaigns which are meant to boost the sales of specific items. In conclusion, both events and prices show interesting patterns which, combined with geographical information, can be useful to estimate the volumes of sales.

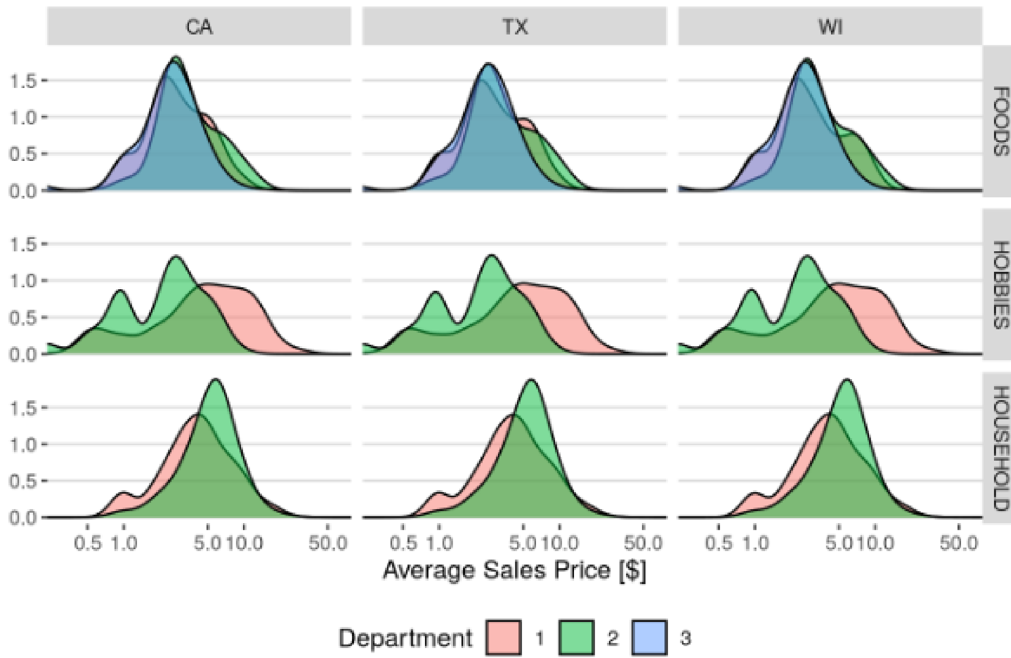


Figure 3.10: Prices distributions by state category and department from [1].

4 Time series forecasting with GNNs

In this chapter the original work is explained to the reader. At the beginning, a wider description of the related work presents the initial ideas and inspirations behind this study, successively it follows a more precise description of the proposed methods, including the specific algorithms and the conceptual differences between each of them. In conclusion, a deeper overview of the implementation is depicted, including the software involved, the data pipelines and the technical details of the proposed methods.

4.1 Related work

Subsection 2.3.3 gives an overview on how graphs can include a temporal component and which applications usually requires this kind of combination. In this section, an even more specific research field is presented, which introduces the idea of transforming time series into graphs. While in the first case, the goal is to mix solutions from the two disciplines, in order to solve problems which require both perspectives, in the second, the aim is to approach time series analysis from a completely different point of view. In other words, the methods described in subsection 2.3.3 can be seen as using time series analysis to model graphs' temporal components, while in this section graph theory is exploited to solve time series problems.

Even if the terms "graph" and "network" are often interchanged they assume a more meaningful distinction in two different scientific fields: graph theory and network theory. While graph theory studies graphs as abstract mathematical entities, complex theory studies **complex networks** as graphs with non-trivial topological features, typical of networks representing real systems. Considering that complex networks can be seen as the representations of complex systems and time series as trajectories of stochastic processes. If a stochastic process is regarded as a complex system, then the associated time series can be regarded as its state sequence. The Big Data era is challenging researchers from different research fields to work together to develop modern solutions to improve pattern extraction from data of very high volume, high speed or high variety. Techniques used to capture hidden information usually combine machine learn-

ing, statistics and data mining techniques. The aforementioned intuition gives the opportunity to approach time series analysis from a complex network theory perspective and to exploit all the developed methods used to perform data mining and machine learning on graphs. Even if there are multiple seminal studies on this approach, a big part of the literature addresses the utilization of complex network methods for the characterization of dynamical systems based on time series [70].

In order to exploit the methods from complex network theory, it is necessary to formulate methods to transform time series to graphs, [70] divides such methods in three main families based on different rationales:

- **Proximity networks** based on mutual statistical similarity or metric proximity between different segments of a time series, including approaches such as cycle networks, correlation networks, and phase space networks based on a certain definition of nearest neighbours. Especially important type of proximity networks are recurrence networks. Based on the concept of recurrences in phase space, recurrent networks are defined on recurrence matrix of a time series, which can be interpreted as the adjacency matrix of an associated complex network which links different points in time if the evolution of the considered states is very similar.

- **Visibility graphs** derives from computational geometry as a method to analyse mutual visibility relationships between points and obstacles in two-dimensional landscapes. There are multiple visibility graphs algorithms, but in general they share the same approach of creating graphs where the nodes represent the time series points and the edges the 'visibility relations' between the points based on their values.

- **Transition networks** are based on transition probabilities between discrete states. In particular the nodes of a transition network represent discrete states or patterns, they are linked by directed edges whenever it is possible to observe a greater than zero probability to move from a state to the following one in the trajectory of the system under observation. Transition networks can be viewed as Markov chains with given transition probabilities. There are various ways to define the nodes of a transition network, especially when the original time series is complex and it is not possible to represent all its possible values as nodes. This usually requires to first transform the time series into a simplified representation, with a limited number of discrete states, and then build the Markov chain to obtain a compressed or simplified representation of the original dynamics.

These representations are successfully involved in multiple studies of complex systems. In particular, the works analyse the properties of the graphs and correlate them to specific phenomenas in the respective dynamic systems. Global and local clustering coefficients, transitivity,

average path lengths of recurrent networks are behind successful researches on climate change, fluid dynamics and earth science [70]. While visibility graph's motif structures, topological measures like the diameter, average path length, modularity, clustering coefficient, density and hierarchical organizations of networks are also involved in medicine and climatology studies [70]. Even if time series forecasting is a crucial component in time series analysis, the majority of the works involving complex networks are not focused on this task, but instead, they exploit the networks to depict the latent properties of the original stochastic processes. To the best of the author knowledge, only visibility graphs are significantly involved in various studies for time series forecasting in industrial scenarios. The following temporally ordered list cites some of them:

- [9] transforms time series to visibility networks. At prediction time a new time step is added as a node and link prediction with local random walks is performed to find the edges between the old nodes and the new node. After the graph is completed with the new edges, it uses linear interpolation to forecast the value of the new added node. The model is tested on a the Taiwan stock Exchange Capitalization Weighted Stock Index values in 2012 and compared to ARIMA using RMSE, obtaining better results with low amount of data.

- [60] uses Data Fluctuation Networks Predictive Models (DFNPM) to transform the time series, then it extracts fluctuation features of time series accordingly to the topological structure of the data networks, and finally it constructs models with the extracted useful information to predict time series. The model is compared to traditional forecasting methods on oil prices prediction.

- [32] similarly to [9], it uses visibility graphs to represent time series but it proposes a faster method to measure the similarity between nodes. A fixed number of nodes which are the most similar to the last node are determined, and weights are calculated to make the final prediction. The model is tested on the Construction cost index time series.

- [36] improves the forecasting part of [9] proposing a weighting mechanism based on time distance. Model is tested on the Construction cost index time series.

- [68] uses a faster variation of the visibility graph algorithm to perform the time series transformation. At forecasting time, it uses a more sophisticated technique considering the last node and all the previous nodes connected to the last one to compute their slope and a preliminary prediction. Afterward, the predictions are revised, considering the last node value, and finally different weighting schemes based on nodes degree and time

distance are used to perform a weighted sum of the multiple forecasted values. The model is compared to traditional models in multiple human activity related datasets.

- [67] reduces forecasting errors from previous models based on link prediction over visibility graphs considering all the nodes in the graph and proposing a novel weighting method.

Visibility graphs are preferred because they are conceptually and computationally simpler than the other methods, they also have multiple variants and they are able to maintain multiple properties of the original time series. A more detailed description on how visibility graphs are created and their properties can be found in subsection 4.3.2. The above works present interesting results in small datasets outperforming some traditional methods, but it is important to mention that they rely on an approximate conversion from the graph representation to the original time series. When time series are converted to graphs, some information are lost, and using only the visibility graph topology does not allow to recover the exact values of the original time series. Despite that, the mentioned works, especially those following [9], propose different and more sophisticated methods to solve this problem. The remaining part of this document addresses the step of converting graphs back to time series as **interpolation**, subsection 4.3.3 is completely dedicated to the description of some of those approaches.

4.2 Methodology

The current study considers the related work on time series forecasting using visibility graphs and proposes two main extensions. The first extension targets specifically the link prediction task of the methods discussed above, proposing the adoption of GNNs as a more sophisticated and modern technique. While the second extension overcomes the problems of the link prediction approach proposing a more direct method, based on graph regression using GNNs. The introduction of GNNs adds another interesting source of ideas to the approaches proposed by the related work, giving the possibility to combine multiple disciplines and adapt methods from one application to another. This study does not propose any innovation on the time series to graph conversion mechanism, following the visibility graphs algorithm proposed by the previous works, but instead it introduces novel techniques based on GNNs to learn over graph data structures. This choice is also supported by the recent works on multivariate time series forecasting using GNNs [63, 51, 8] and time series classification using GNNs [42, 29, 65], which indicate a similar direction from academia on adopting GNNs for similar contexts. There are multiple expected advantages from using GNNs for graph-based time series forecasting.

- Almost all the related work performs link prediction using graph heuristics and none of them present any solution where exogenous variables are used as features for a machine learning approach. This recalls the general difference between traditional and machine learning/deep learning time series forecasting models explained in section 2.2. Modern datasets tend to be multidimensional and the advancements on machine learning follow this trend, proposing methods able to deal with hundreds of features. In this work, visibility graphs are used and generated as described in the related work, but their nodes are enriched with exogenous features which can be used by GNNs.
- Many of the latest proposed approaches based on visibility graphs do not improve the performances on forecasting horizons larger than a single time step, nor directly neither regressively. In particular, the recent related work proposes methods to improve the quality of the forecasts without mentioning how edges are effectively updated when multiple nodes (time steps to forecast) are added to the graph.
- Introducing a new machine learning method to solve multiple time series forecasting problems with cross-learning between different time series. This can have a huge impact in many fields, such as retails, transportation and internet of things, where it is possible to observe thousands of time series.
- Proposing original methods inspired by other applications. In particular, related work does not contemplate the possibility to consider graphs as dynamic, even if it seems to be the case. As mentioned in subsection 2.3.3, GNNs have been successfully applied to dynamic graphs.
- Opening the possibility to create a unified framework based on GNNs for time series forecasting, clustering and classification.

4.2.1 Link prediction with GNNs

The remaining part of this document names respectively the family of methods belonging to the first extension of the related work **Link prediction with GNNs (LpGnn)** and the model proposed in [9] **LpHeu**. Two LpGnn methods are proposed, the first considers the graphs as static, while the second as dynamic. Both methods follow the same procedure illustrated in algorithm 1, which is identical to the one proposed by LpHeu, apart from the usage of GNNs and exogenous features in the visibility graphs. Once the GNNs are trained to predict the links, a portion from the historical time series preceding the horizon to forecast is extracted to initiate the process and feed the algorithm with an initial input. At prediction time, the target variable is

only known in the historical time series and it is used to convert the extracted time series into a visibility graph, but it is not added as a feature to the nodes. By contrast, the exogenous features are always visible for both past and future time steps, and thus, they are used to populate the node features. At each iteration, an arbitrary number of nodes representing the oldest time steps is removed from the visibility graph, while the same number of new nodes, representing the future time steps to forecast are added as a chain of nodes to the temporally most recent node of the graph. The nodes are chained and connected to the most recent node because in visibility graphs each pair of consecutive time steps are always mapped to connected nodes. In other words, the objective of the GNN or of the heuristic method, is to reconstruct at each iteration the remaining portion of the visibility graph predicting all the remaining links between old and new nodes. Once the graph is completed, interpolation methods proposed by the related work are used to combine the predicted links and the known historical target values to forecast the future target values. At testing time, the complexity of the algorithm is linearly proportional

Data: n historical time series (*H*), n time series containing the features of the time steps to forecast (*F*), *splitlen*, *forecaster*, *horizon*
Result: future values predictions
consider only the last *splitlen* time steps of each time series in *H*;
for *every time series h in H* **do**
    transform *h* to a visibility graph *g* of *splitlen* nodes removing the target variable from the features;
    get the associated time series *f* in *F*;
    for *every time series portion p of length horizon in f* **do**
        remove the oldest *horizon* nodes from *g*;
        use *p* to create a chain of *horizon* nodes to add to the last node of *g*;
        use the *forecaster* forecast method to apply node prediction between the new *horizon* added nodes and the previous nodes in *g*;
        use the *forecaster* forecast method to interpolate the predictions from *g*;
        update the new added nodes of *g* with the predictions (autoregression);
        store the predictions;
    end
end

Algorithm 1: The LpGnn algorithm. For each time series a graph of *splitlen* nodes is iteratively updated removing and adding the same amount of nodes. The *horizon* represents how many time steps are forecasted by the model in a single step and not the total number of points to forecast in a specific application.

to the number of time series n and it is influenced by the chosen forecasting horizon size and the number of time steps to predict $total_horizon$. In general, larger horizons improve the speed of the model since fewer forward steps are performed and more edges are predicted at each step with less accuracy. Assuming that common horizons are not significantly big and the code is highly optimized, the time spent by the GNN to perform the link prediction can be reduced to a constant factor lp and the interpolation to int, the final computational complexity is $\mathcal{O}(n \cdot lp \cdot int \cdot \frac{total_horizon}{horizon})$. The main difference between static and dynamic methods occurs at training time, where in static graphs the link prediction occurs on multiple independent graphs representing portions of transformed historical time series, while in dynamic graphs link prediction is computed on sequences of related graphs representing the temporal evolution of initial visibility graphs. It follows a different GNN architecture between the two methods: the dynamic model uses an EvolveGCN, while the static model uses different GNN architectures to generate node embeddings. Both methods combine the node embeddings to create edge embeddings which are finally used to generate edge scores which are thresholded to distinguish existing from non-existing edges between pairs of nodes. Links tend to be generated between similar nodes, where the degrees of similarity are learnt by the model, during training. More details on the implementation, including the choice and the parameters of the GNNs and of the interpolation methods can be found in 4.3.3

4.2.2 Graph regression with GNNs

The second method is called **Regression with GNNs (RgGnn)** and it is well described in algorithm 2. In this case for each time series to forecast, it is created a visibility graph from the most recent window of historical time steps using both the target and the exogenous features as nodes attributes. Graphs are passed as inputs to a GNN which learns how to create nodes embeddings which are finally passed as a temporally ordered sequence to a regressor head, like a RNN, to forecast the future values. The generic algorithm is open to direct multi-step forecasts, but it is worth mentioning that in this work the final implemented RgGnn models predict only a single time step. In this scenario, no interpolation is needed to retrieve the time series from the graph, since only historical data is transformed into graphs and forecasts are already in a sequential format. Similarly to LpGnn, the complexity depends on the number of time series n, the $total_horizon$ and the forecasting $horizon$ size. Since no interpolation is required, the RgGnn is faster and its complexity can be assumed to be $\mathcal{O}(n \cdot reg \cdot \frac{total_horizon}{horizon})$, where reg is the constant time required by the GNN to predict $horizon$ future steps from a given input graph. The difference between the time to predict links or perform graph regression depends on the complexity of the chosen architectures, but can be considered on an equivalent

Data: n historical time series (*H*), n time series containing the features of the time steps to forecast (*F*), *splitlen*, *forecaster*, *horizon*
Result: future values predictions
consider only the last *splitlen* time steps of each time series in *H*;
for *every time series h in H* **do**
  get the associated time series *f* in *F*;
  for *every* i^{th} *(from 0 to number of portions - 1) horizon portion in f* **do**
    create a time series *t*, with length *splitlen*, from concatenating *h[i * horizon:]* and *f[:i * horizon]*;
    transform *t* to a graph *g* of *splitlen* nodes;
    use the *forecaster* forecast method to predict the next *horizon* values from *g*;
    update the first *horizon* time steps of *f* with predicted values (autoregression);
    store the predictions;
  end
end

Algorithm 2: The RgGnn algorithm. The notation *l[:k]* is used to express the first *k* elements in *l* and *[k:]* to express the elements following the first *k* in *l*. In the first iteration the concatenation results in the time series *h* itself, as new points are forecasted, larger portions from *h* are replaced by values from *f*. At each iteration, a new graph containing a window of preceding time steps data is generated and used to forecast the following *horizon* steps.

scale. More technical details on the architecture can be found in section 4.3.4 In conclusion, the differences between each method are resumed in the following list.

- LpGnn and LpHeu are all based on link prediction, while RgGnn on graph regression. LpGnn is an improvement of LpHeu which uses GNNs instead of heuristic methods. Moreover, also RgGnn uses GNNs.
- LpGnn and RgGnn use graphs containing exogenous features while LpHeu uses plain graphs.
- LpGnn and LpHeu use graphs containing both some historical time steps and the time steps to forecast, requiring to exclude from the features the target variable. On the other hand, the RgGnn uses graphs containing only historical data and thus it can use the target variable as a feature.
- The LpGnn algorithm updates the same graph adding and removing nodes representing time steps. Whereas, the RgGnn creates a new graph at every iteration containing recent

historical time steps.

- LpGnn methods are two-step approaches, they require to convert back graphs to time series using interpolation methods, while RgGnn are direct.
- Both LpGnn and RgGnn can predict multiple time steps at a time. When the predicted steps are not enough to cover the entire horizon an autoregressive approach is followed. RgGnn's GNN uses directly the predicted values while LpGnn uses the predicted values only in the interpolation updating the attributes of the graph nodes.
- The main difference between LpGnn static and dynamic is only noticeable at training time, in the way graphs are represented and edges are predicted. LpGnn static uses static graphs while LpGnn dynamic learns over sequences of snapshots, the former learn to predict generic edges, while the latter only the edges linking old and new nodes at each sequential step. More information can be found in the experimental setup in 5.2.1. At test time both static and dynamic methods are applied at the same manner.

4.3 Implementation details

This section goes deeper into the technical description of this work. After introducing the software used to implement and experiment the proposed solutions, it concisely describes the most important steps of the data pipeline and components of the methods implementation, providing an essential guide to any further investigation or improvement to this work.

4.3.1 Software and repository

The project is developed using the version 3.9 of the programming language **Python**. Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Due to its fast edit-test-debug cycle and intuitive syntax it is widely used in multiple domains. The Python Package Index[1] lists thousands of third party modules available for the Python environment. Among them, many provide scientific and numeric computing functionalities, such as *SciPy* and *Pandas*, which are used in this project. The Stack Overflow 2021 Developer Survey[2], the 2021 Kaggle Machine Learning & Data Science Survey[3] and the 2020 Python Developers Survey[4] clearly show the increasing trend of Python. According to these surveys,

[1]https://pypi.org
[2]https://insights.stackoverflow.com/survey/2021
[3]https://www.kaggle.com/c/kaggle-survey-2021
[4]https://www.jetbrains.com/lp/python-developers-survey-2020/

Python is among the three most popular programming languages and it is the most used by the Kaggle community, Data Analysis and Machine Learning are the most common activities and Python is generally preferred by students and new graduates. There are other languages specialized in scientific computing, for instance the programming language R has extensive facilities for analysing time series data. Although the two languages have many similarities, R is mainly used for statistical analysis, while Python provides a more general approach to data wrangling. Above advantages comes at a cost in computational efficiency, Python is generally slower and more memory intensive than other languages, as C or C++. A common solution is deploying Python as a friendly API between the developers and the libraries' core parts written in CUDA and C/C++. Some examples are the image processing library OpenCV, and the deep learning frameworks TensorFlow and **PyTorch**.

Working with time series data can be difficult and tedious. Hence, this work leverages one of the existing Python library developed for time series analysis called **Darts** [57]. Darts provides many functionalities to manipulate and forecast time series, supporting both univariate and multivariate time series and models. Backtesting, metrics and methods to combine multiple methods are provided. Another Python package used in this project is **visibility-graph** [17]. visibility-graph provides an implementation of the algorithm described in [27]. In practice, it converts a series of values to a **NetworkX** graph. NetworkX [19] is a Python package used for exploration and analysis of networks and network algorithms. The library supports many types of graphs, in addition, nodes and edges can contain any hashable data, allowing NetworkX to represent networks from many different scientific fields. Moreover, many graph algorithms are implemented for calculating network properties, structure measures and generate visualizations. The Python library used to develop and train GNNs is **PyTorch Geometric (PyG)** [16], version 2.0.1. PyG is a library built upon PyTorch, consisting of various implementations of deep learning published works on graphs and other irregular structures, also known as *geometric deep learning*. PyTorch [40] is an open source deep learning framework, based on the Torch library. Among other alternatives, such as Keras and TensorFlow, PyTorch is generally preferred in academia due to its flexibility, performances and debugging capabilities. As a side effect, PyTorch verbosity and complexity is generally higher, making it less appealing to industrial applications. Regarding the current study, choosing PyTorch has been mainly influenced by the choice of PyG among other less mature GNNs libraries. This work uses PyTorch version 1.8.2. Finally, the Python library **DeepSnap** [44] is used to simplify and support GNNs manipulation and training pipelines. Developing a machine learning application requires multiple iterative processes, where multiple aspects are involved, including algorithms, models, pipelines, data and hyperparameters. Versioning plays an important role in maintaining order inside a machine learning project and tracking evolutions and results. In this work, the version

control software Git is used to track the code and the configurations, while the free service made available by **Weights and Biases (W&B)** and its Python API [6] is used to record experiments. W&B is an experiment tracking tool for machine learning, it can be used to log various data formats in the cloud and to generate nice visualizations and reports. More details on the experiment process can be found in chapter 5. This work is supported by the computing resources made available by the Computer Science and Engineering department of the University of Bologna. In particular, a 10-node HPC cluster, where each node is equipped with an Intel Xeon quad-core CPU with 44 GB of RAM and a GeForce RTX 2080 Ti graphics card[5] is used to train and test the proposed models. Cluster computational resources are accessible by submitting requests to a SLURM service. SLURM is an open source, fault-tolerant, and highly scalable cluster management and job scheduling system for large and small Linux clusters.

The project repository[6] is structured in the following folders:

- *data* contains the code related to data downloading, preparation and processing. It also contains configurations and definitions strictly related to the M5 competition data, such as hierarchical levels and forecasts evaluation.
- *gnn* contains the PyG models and datasets.
- *methods* contains the proposed solutions, including models, training and testing functions.
- *nn* contains PyTorch helpers and models.
- *test* contains executable main functions.
- *utils* contains functions to manipulate time series and graphs, reduce memory occupation and visualize data and results.

It is preferable to install the Python packages in a isolated virtual environment following the instructions in the repository. Some packages may differ their version depending on whether GPU support is enabled or not.

4.3.2 Data transformation

Data transformation from its original form can be divided in two steps: **time series transformation**, where the time series and their covariates features are extracted from raw data, and **visibility graph transformation**, where time series are transformed into graphs. As in many

[5]https://disi.unibo.it/it/dipartimento/servizi-tecnici-e-amministrativi/servizi-informatici/utilizzo-cluster-hpc
[6]https://github.com/alomb/MasterThesis

data-driven applications, the data transformation phase, although preliminary to the real objective of the work, represents a crucial and complex moment. In this work this is particularly true for several reasons:

- The amount of data is considerably big, hence, efficiency plays an important role. This represent the main price to pay to enjoy the benefits of using the M5 competition data described in 3.1.
- A phase of the transformation process, where time series are converted into graphs, is actually the fundamental idea behind this work.
- GNNs, as other deep learning methods, require a suitable and meticulous data preprocessing step in order to properly work.

Time series transformation

In this context, the time series transformation consists of two consecutive steps in the data pipeline: **time series preparation** and **time series preprocessing**. Time series preparation mixes the different raw data sources to create a unique optimized and readable file. The preprocessing phase extracts from the result of the preparation step the time series belonging to one of the twelve M5 hierarchical levels and transform the time series into their final form. Raw data is publicly available from the official GitHub repository [7]. It consists of multiple comma-separated files: a pair of train and test files for both validation and evaluation competition phases, a file containing calendar information and a final one containing selling prices. Train and test tables contain in each row the daily sales of a specific time series at the 12th level, namely, the sales of a specific product in one of the available stores. This original table is transformed using Pandas into a new one, where each row contains the sales for a specific date and a specific time series. This format, even if memory inefficient, it is necessary to create a unique table containing both the sales and the additional calendar and sell prices information, which depend on time and thus, require a time-based indexing. This approach is widely adopted in machine learning based solutions, the stacked complete time series rows are transformed to single data points for every date and time series, which can be easily used as input samples for a global machine learning model. In this new representation, the original time series are still identifiable by the *hierarchical columns*, namely the columns containing the information of the product, department, category, store and state. Since the amount of data involved in the preparation may become very large, causing memory errors and extremely long execution times, multiple data reduction techniques are added. String type is replaced with categorical

[7] https://github.com/Mcompetitions/M5-methods

type whenever it is possible, numerical data types are reduced to their minimal form, not used columns such as the names of the events are immediately removed and a limit of days to consider is introduced. For instance, considering the last 1000 training days of each time series, the first step creates an initial table of 30490000 rows (30490 is the number of time series at the lowest level and 1000 is the length of each one) and 1.8 GiB of memory occupation. After adding the week and year column and applying memory reduction, the resulting table becomes a 30490000×10 of size 641 MiB. The final merge between the temporary tables results in a 30490000×16 table of size 844 MiB after appropriate reductions.

The result obtained at this stage is yet not usable by aggregations levels different from the 12th. The preprocess step groups the data to get higher hierarchical levels and performs feature engineering. In particular, the sales obtained from the preparation phase are summed at the desired hierarchical level, the 1st level creates a unique time series, while the others the number of time series specified in the image 3.2. The calendar information is not influenced by the aggregation because it is related to the time index, while sell prices can be combined in different ways, for example producing different descriptive statistics, such as mean, maximum, minimum and standard deviation. Finally, the feature engineering is guided by a configuration class which contains the desired operations to apply and it is saved along with the final result to keep track of the feature configuration. The following list show the possible configurations:

- Include or not the SNAP boolean features.
- Include or not the one-hot encoded events information.
- Include or exclude specific feature columns.
- Sales input feature scaler.
- Sales lag features, specifying the shift and the lagging intervals. Shifting a time series means moving each value a number of steps behind, while the lag intervals are the indexes of past values considered as features.
- Sales rolling windows features, specifying the shift, the windows sizes and the statistic operations: mean, standard deviation, minimum and maximum. Windows are subsets of past values used to create descriptive statistics.
- Enable or not the sales output feature scaling. If set, also the sales to forecast are scaled accordingly.
- Temporal features transformer function.

- Temporal features scaler which can be used in the temporal transformer function.
- Sell prices transformer function.
- Sell prices scaler which can be used in the sell prices transformer function.

In lagging and rolling windows features time series are first shifted to avoid using information which is not available at prediction time. For example the M5 the horizon has a length of 28 days, using lagged values within the last 28 days is not feasible because it requires knowing in advance the future values. This is a common mistake during time series preprocessing. Scalers are used to refer to transformer classes available in scikit-learn preprocessing package to change raw feature vectors into representations that are more suitable for machine learning estimators. Common classes are StandardScaler, which standardize features by removing the mean and scaling to unit variance, MinMaxScaler to scale to a given range, PowerTransformer to apply Box-Cox, or FunctionTransformer to apply any function, such as logarithm transformation. Transformer functions allow more customizable transformations, for example dates can be used to extract additional temporal information. An interesting available option for temporal features is using sine and cosine functions to encode the cyclical categorical features, such as the day of the week or the month. Using both sine and cosine of the signal doubles the number of features but leaves all categorical entries unique and helps the model learning the inherent cyclical structure, instead of adopting linear structures which provides misleading distances or one-hot encoding which dramatically increases the number of features.

Visibility graphs transformation

The transformations described in 4.3.2 are part of a generic data transformation process which can be shared among other conventional machine learning time series forecasting approaches. In this work, the preprocessed time series are split into multiple parts and transformed into graphs, using the visibility graph algorithm implemented in the Python package visibility-graph. Commonly, this stage ends up in creating multiple graphs which are used to create a PyG dataset. The details of the datasets may vary between each proposed method, but all of them share the same time series transformation and graph conversion logic. Thus, the details of the datasets are reported in the next sections.

The algorithm proposed in [27] creates graphs from univariate time series, where each node is related to a point of the time series and the edges represent their visibility relationships. The algorithm first represents time series values as vertical bars, each point of a time series is mapped to a vertical line with height proportional to the value of the point. The space between each pair of consecutive bars is the same. Consequently, the algorithm link every bar with all

those that can be seen from the top of the considered one, obtaining the associated graph. More formally, two arbitrary points y_{t1} and t_{t2} have visibility if any other time step y_{t3} between them satisfy the equation 4.1. Figure 4.1 shows an example of the process.

$$y_{t3} < y_{t2} + (y_{t1} - y_{t2}) + \frac{t2-t3}{t2-t1} \tag{4.1}$$

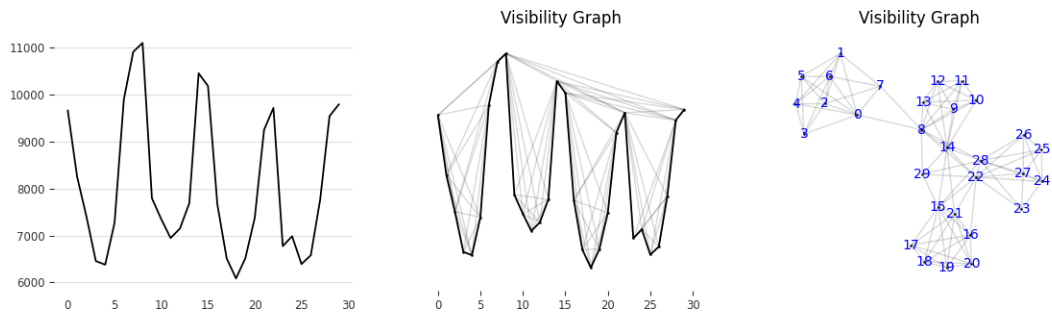


Figure 4.1: The visibility graph obtained from the cumulative sales in the department HOUSEHOLD_1, on the 5th level.

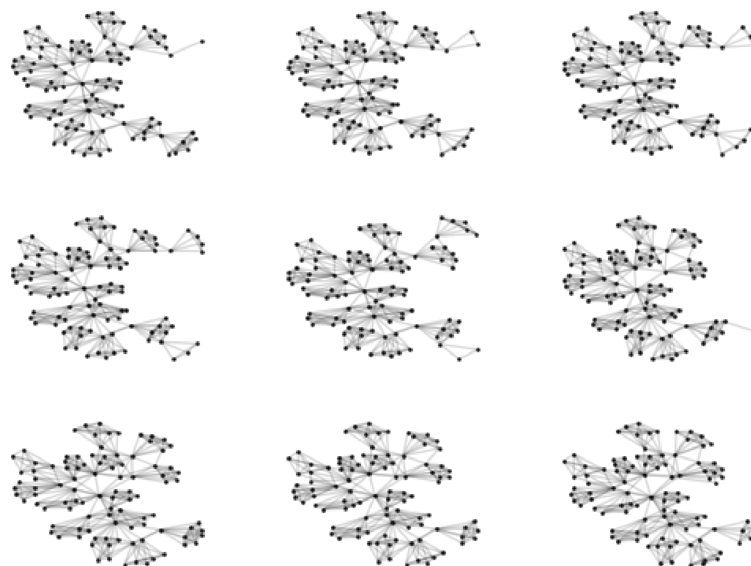


Figure 4.2: The visibility graphs obtained from the last 100 points of the time series at the 6th level.

Visibility graphs are always connected, undirected and invariant under affine transformations. The visibility criterion is invariant under rescaling of both horizontal and vertical axes, under horizontal and vertical translations and under addition of a linear trend to the data. The degree of the resulting graphs reflects the nature of the time series, for instance periodic time series are mapped to regular graphs, while random series into random graphs.

4.3.3 Link prediction with GNNs

As mentioned in 4.2, two GNNs families are involved in the link prediction task: the first considers the graph as static while the second as dynamic. Both static and dynamic methods consist of a PyG **dataset**, a configurable **forecaster** model containing a trainable GNN and a **training loop**. All the forecaster models extend the same `LinkPredictionForecaster` interface, including the implementation of the heuristic method explained in [9]. The interface provides separately the interpolation methods described below, leaving the possibility to personalize the forecasting. Finally, the `LinkPredictionForecaster`'s forecast method merges the link prediction performed by the GNN to the chosen interpolation algorithm. Generic functionalities to forecast at testing time and performing all the required transformations are also shared among the two different methods and included, maximizing the code reuse. Figure 4.3 briefly explain the described design choice.

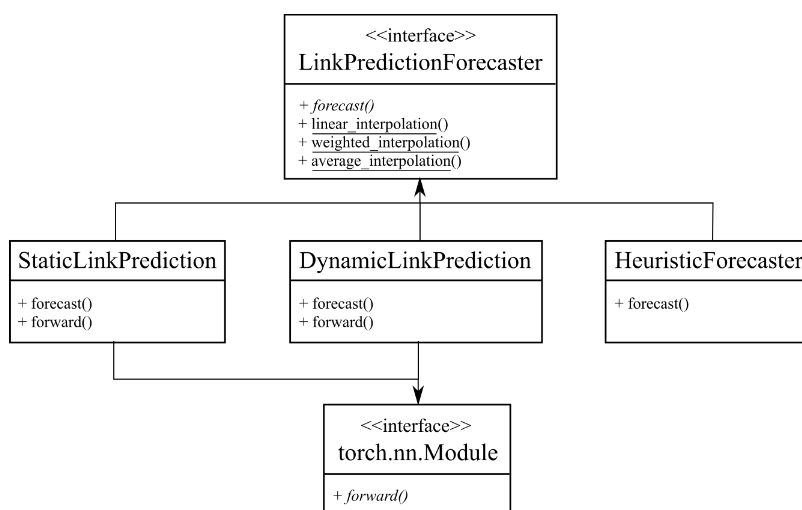


Figure 4.3: The main LpGnn classes represented in a UML class diagram.

Interpolation methods

Link prediction alone does not provide the final forecast. Once the links between the old and the new nodes are predicted, the graph must be converted back to the time series containing

the target variable, obtaining the forecasts. Old nodes can be simply mapped to their original values, while new nodes are interpolated using the similarity between the nodes and the known values of the previous nodes. This work implements the following methods:

- **Linear interpolation**, used in [9], it initially creates pairs of time steps where the first element correspond to a node linked to the target node and the second element is the node representing the most recent time step which is linked to the first element. Every pair is used to compute a linear interpolation and the minimum value is chosen. Since the last known time step node does not have a linked node to a later time step, excluding the target time step itself, it cannot be considered as the first element of a pair.

- **Weighted interpolation**, used in [36], the value of the last known time step and its most similar previous time step are used to interpolate the future value. In addition, the distances between the different time steps are used to compute weights. Thus, the final value is the weighted sum between the resulting interpolation and the value of the last known time step.

- **Average interpolation** proposed in this work, simply performs a weighted average of the values of the previous nodes connected to the target using the similarity scores as weights.

None of this method guarantee an exact solution, but with correct similarities scores they are able to perform acceptable approximations.

LpGnn with static graphs

Graphs are created from the time series as described above and are used to create a PyG dataset. PyG allows to create custom datasets by extending specific library classes. Each dataset receives the location of a root folder which indicates where the dataset should be stored and some functions are free to be implemented in order to specify how to download and process the raw data. It follows that, all the operations described in 4.3.2 are managed and organized in a custom PyG dataset called `M5Dataset`. After graphs are obtained from time series they are transformed to PyG `Data` objects, where nodes features and labels are represented as PyTorch tensors. The link prediction task does not require a label associated to each node, such as in node classification, because they are associated to edges. Edge labelling is not performed in the dataset creation and it is described in chapter 5. In conclusion, `Data` objects are populated with the features from each node, which are the same features associated to the points of the original time series. Since link prediction is based on node similarity, the number of sales is

excluded from the features, because it is not possible to know in advance the target variable in the nodes added during the forecast at testing time.

The class `StaticLinkPrediction` extends both the `LinkPredictionForecaster` and the basic PyTorch class `Module` providing an unified and self contained class to learn link prediction and perform forecasting. During training links are passed and the model is asked to classify them as real or non-existent edges. Training is performed in an end-to-end fashion, pairs of node embeddings associated to the given edges and generated by a customizable GNN are passed to a **link embedding method** and the resulting vector representation is passed to a group of fully connected layers which returns a single value. The resulting value is used to determine the existence of the link or not. The GNN is expected to learn how to generate similar embeddings between linked nodes, the proposed architectures are GraphSAGE, GIN and GCN. The following list shows the different link embedding methods:

- Inner product between the two vector representations. It already computes a single value, so the final fully connected layers are not used.
- Absolute value of the difference between the two vector representations (L1 norm).
- Exponentiation with base two of the difference between the two vector representations (L2 norm).
- Average between the two vector representations.
- Concatenation between the two vector representations.

The number and the size of layers as well as the type of activation function in the final classification network are all customizable.

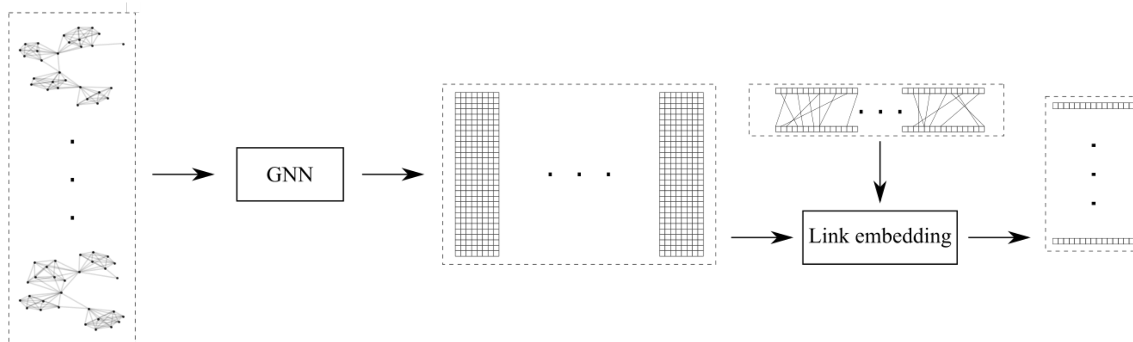


Figure 4.4: The representation of the LpGnn static architecture neural architecture. The link embedding receives the node embeddings and the edges to consider, assigning a score to every edge.

LpGnn with dynamic graphs

As anticipated in 2.3.3 and 4.2, dynamic graphs expect topological evolution over time. To learn predicting the changes in the graph, dynamic approaches usually works with sequences of graphs taken at consecutive time steps. This is opposed to the static approach, where link prediction is learnt from multiple not associated graphs having the same number of nodes. As the static method, the method can be described in its main components: the dataset, the forecaster and the training loop.

The dataset is called `M5TemporalDataset` and it is implemented extending native PyG classes, similarly to the `M5Dataset`. Instead of using the PyG class `Data`, a custom class called `TemporalData` is specifically developed to store the graph data. This method does not strictly follow the standards imposed by PyG, especially regarding how all the graphs data are collated together. In particular, for each time series a split is extracted and it is used to retrieve multiple portions of increasing length called *snapshots*. Snapshots represent the evolution of the graph over time and together they can be used to represent discrete-time dynamic graphs. As the name suggests, the `M5TemporalDataset` adds an additional dimension into the data: the time. Figure 4.5 schematically shows the difference with the other methods.

The forecaster class, called `DynamicLinkPrediction`, extends the LpGnn base class `LinkPredictionForecaster` and can either use the version -H or -O of the EvolveGCN model proposed in [39] and described in subsection 2.3.3. Similarly to the static model the GNN component of the EvolveGCN creates node embeddings for each pair of nodes, which are concatenated and passed to a fully connected neural network with customizable layers and activation functions. Even if the forecasting follows the same algorithm 1, training is performed differently and it is explained in chapter 5.

4.3.4 Graph regression with GNNs

The implementation is similar to the one described for LpGnn, except for the forecasting part which is substantially different. Also the graph regression method requires a custom PyG dataset, a configurable forecaster model containing a trainable GNN and a training loop. Following the same design principles of the LpGnn approach, a new interface is defined for this new purpose and it is named `RegressionForecaster`. The simpler interface does not contain any interpolation code and just defines the length of the horizon, leaving the specializing classes to implement the forecast method. In this case, the forecasting algorithm follows the classical autoregressive approach of using predicted values as inputs. As the prediction of new values continues, a new graph is generated from the concatenation of the starting time series, took from the historical time series, and the time series to forecast, where all the features are

known, except for the sales.

Regarding the PyG dataset, the graph regression uses the same `M5Dataset` class described in 4.3.3, with two differences. During time series extraction, each split is associated to a temporally consecutive additional split containing the values to forecast, called *horizon split*. The target variable of each graph is set to the associated extracted horizon split. The dataset is used to yield graphs with a custom number of nodes and a target vector representing the future time steps to forecast. This format is ready to be ingested into a graph regression training and testing routine. Figure 4.5 shows the difference between the data pipeline involving the `M5Dataset` in link prediction mode and in regression mode.

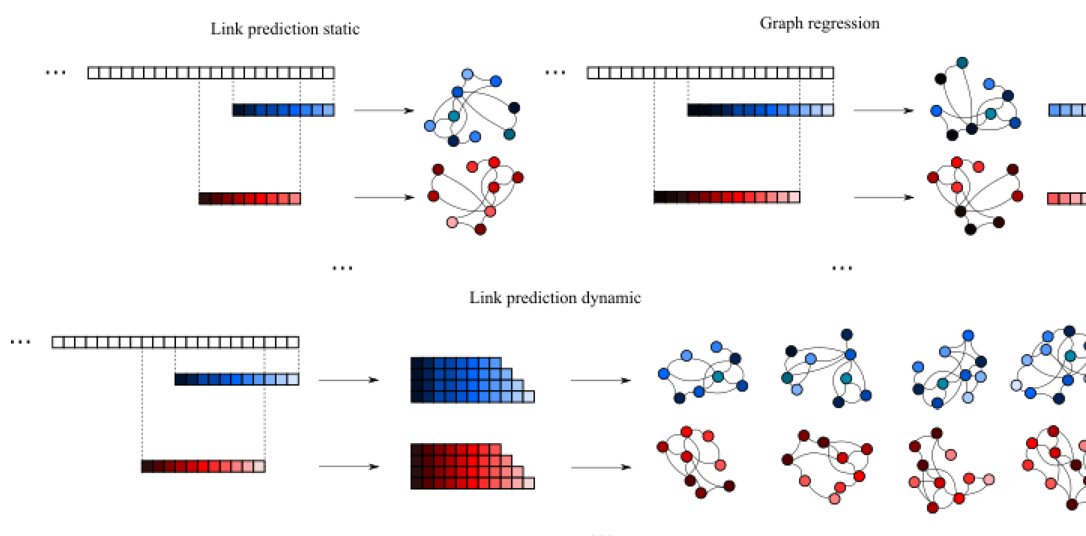


Figure 4.5: The difference between the splits created in the `M5Dataset` for the LpGnn static, LpGnn dynamic and RgGnn approaches.

The class `DirectRegression` extends both `RegressionForecaster` and the PyTorch class `Module`. During training the inner GNN model is asked to generate node representations, which are concatenated as a sequence following the same temporal order in the original time series and fed to the to either a recurrent neural network, a transformer or a TCN. Available GNN models are GraphSAGE, GIN and GCN. The head of the model is highly customizable, for instance the recurrent neural network option allows choosing between the LSTM or GRU architectures and setting the number of stacked layers and directions. For the recurrent networks, the final output is obtained selecting only the hidden state from the last layer, summing together the possible directions and passing the final vector to a fully connected layer followed by a ReLU activation function. In the Transformer case, a simplification of the original transformer architecture is proposed. The encoder combines the sequential output of the GNN with a temporal encoding and consists of a stack of fully connected layers with self-attention, while

the decoder is modelled with a single fully connected layer. The simpler decoder is motivated by the choice of using mainly the RgGnn approach for single-step forecasts, even if multiple steps can be easily integrated. The number of heads, the hidden size of the fully connected layers and the number of layers of the transformers can be personalized. Finally, the option to use a standard TCN is implemented and made fully customizable, for instance the choice of the kernel size, the number of blocks or channels can be modified.

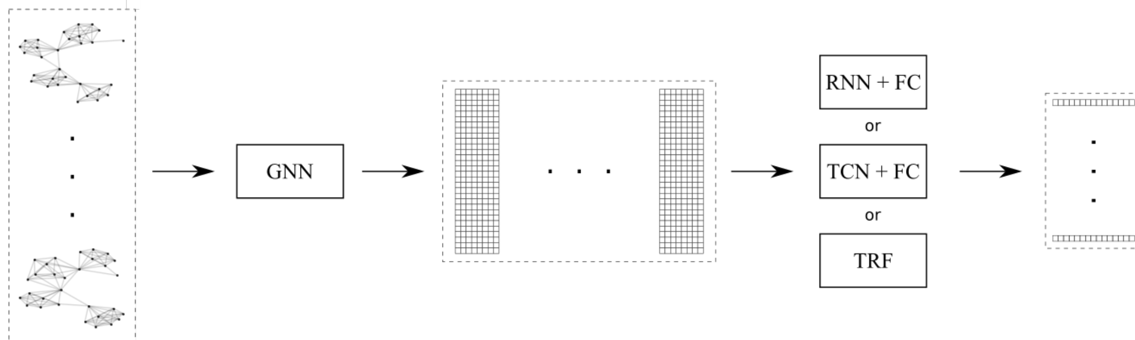


Figure 4.6: The representation of the RgGnn neural architecture. Each node is encoded to a vector which is passed to a sequence model such as a RNN, transformer (TRF) or TCN.

5 Experiments

This chapter describes the evaluation procedure applied to the proposed methods. The details on the training and testing of the models are disclosed, together with the description of the involved metrics. Exhaustive descriptions on the tuning of the final models and discussions on the results are also provided. The aim is to compare and measure the strengths and weaknesses of different configurations and methods.

5.1 Metrics

The time series forecasting literature proposes different metrics to measure the quality of a forecast with respect to the real observations. Forecast *errors* are different from *residuals*, the former are related to the test set and can involve multi-step forecast, while the latter are computed on the training set and can be only based on one-step forecasts [22]. There are different families of metrics. *Scale dependent errors* are on the same scale as the data, therefore they cannot be used to compare series with different measuring units. They are usually the most straightforward and common used metrics, especially for simple scenarios with few time series. **Mean absolute error (MAE)**, equation 5.1, is easier to interpret and it is more suitable to study how the model differs from the median of the actual data, instead the mean squared error (MSE), or its squared root (RMSE), can be used to address the mean forecasting. This work prefers the linearity of MAE over MSE, using it as an additional metric to compare the different methods at the same level and show an intuitive estimate of the error scale.

$$\frac{1}{n}\sum_{t=1}^{n}|e_t| \tag{5.1}$$

Forecasts are represented as $\hat{y}_t$, actual data as y_t and the error as $e_t = y_t - \hat{y}_t$. *Percentage errors* metrics are used to obtain unit-free performances, allowing the comparison of different units. On the other hand, percentage errors have several disadvantages. They assume undefined or infinite values when y_t is zero or close to it, in addition they assume that the unit of measurement has a meaningful zero [22]. For instance the mean absolute percentage error (MAPE) is

an asymmetric metric and it gives an heavier penalty on forecasts higher than the actual values, as a result, MAPE favours models that under-forecast rather than over-forecast. In addition, it cannot be used if there are zero values, which is really common in the sparser metrics of the M5 competition data, because there would be a division by zero. Although MAPE is easy to interpret and compute, its many drawbacks have excluded its adoption is this work. Finally, *Scale free errors* represents a good alternative to percentage errors. For example the mean absolute scaled error (MASE) [23] can be used to have a unit-free measurement of how the model perform against an average forecasting model computed on the training data, represented by its error in the denominator. In this work the already described metrics RMSSE and WRMSSE, which are similar to MASE, are mainly used to evaluate the methods, including also the business perspective.

The **R2 Score** or coefficient of determination, equation 5.2, is a common statistical measure of how well the regression predictions approximate the real observations and it represents the proportion of the variation in the dependent variable that is predictable from the independent variable. Intuitively, R2 score is not preferred for time series forecasting because time series models are evaluated not in how well do they fit past data but rather in how well do they predicts future values. In this work the R2 score is used to evaluate the general performances of the machine learning based graph regression task, namely to score the differences between predicted and real values.

$$1 - \frac{\sum_t (y_t - \hat{y}_t)^2}{\sum_t (y_t - \overline{y})^2} \tag{5.2}$$

On the other hand, since the link prediction task can be seen as a classification task, where edges are classified between existing and non-existing, the **Area Under the Receiver Operating Characteristic Curve (ROC AUC)** is used to evaluate the models during training. The ROC AUC denotes the extent of how a model is capable of distinguishing between classes.

5.2 Experimental setup

Usually when a machine learning model is applied to solve a particular problem the available data is split in two, or better three, portions to respectively train, configure the hyperparameters and evaluate on unseen data. In time series forecasting the splits are not sampled from all over the dataset, but are extracted as temporal consecutive portions of data. In this work the M5 validation and evaluation phases are exploited to retrieve all the three required data portions. In order to improve the effectiveness of the proposed methods and provide more materials to discuss, a model is trained and tested for each hierarchical level. During hyperparameter tuning, models are trained on the data of the validation phase while the final results are computed on

the data of the evaluation phase, as in the official competition. At testing time, the models are trained considering also the 28 days of the validation test data from the validation phase. Even if similar, each approach has its own peculiar experimental setup.

5.2.1 Link prediction with GNNs

Link prediction is commonly solved as a self-supervised task where labels are initially unknown and they must be created before training, in this work, both static and dynamic methods automatically label the edges. In particular, labelling a graph in the link prediction task means creating a set of positive edges, which contains real existing edges of the graph, and a set of negative edges, which contains edges which do not exist. When the two sets are generated the link prediction task can be seen as a binary classification problem, where a model is trained to classify positive and negative edges. In addition to these two sets, some works propose the creation of a separate group of edges involved only in message passing operations and not used as training samples for the model. Labelling in the dynamic method is simpler because at each snapshot, a node is added and the existing edges between the old nodes and the new are labelled as positive, while the remaining combinations of pairs of nodes are just used to sample the negative edges. In the static method positive and negative edges are selected randomly between pair of nodes inside visibility graphs. This choice represents the main difference between the two approaches: the dynamic method is closer to the final problem setting, imitating the graph evolution, while the static model tackles the link prediction task from a very generic perspective. In the static method the library DeepSNAP is leveraged to generate two balanced sets of edges. The ratio between positive and negative edges is set to one, all training edges are also used to pass messages during training and negative edges are not resampled at each iteration.

Training a LpGnn method means training its GNN, namely its link predictor. In order to evaluate the ability of this component, a split of edges is extracted from the data and used to compute the ROC AUC at each training epoch. As the ROC AUC score on the test split overcomes the preceding best one, the model is saved. The forecasting abilities are evaluated lately, when the link predictor component is trained and ready. In general, there are two possible ways of splitting between training and testing graph data. In **transductive** learning all the graphs are included in each split, but nodes or edges are split depending on the task, while in **inductive** learning, a dataset containing multiple graphs must be used because each split include distinct graphs. The static method uses 80% of the edges of each graph to train and the remaining 20% to create the test split, adopting a transductive strategy. On the other hand, the dynamic method splits the snapshots of each evolving graphs in two consecutive sets, the first 80% of available snapshots are used for training, while the remaining 20% for testing,

following an inductive strategy.

Both models are trained using Adam optimizer with adjustable learning rate and weight decay, moreover, early stopping and learning rate scheduler are provided. The binary cross entropy (BCE) function is chosen as the most suitable loss function, as commonly adopted in many binary classification problems. Batching differs between static and dynamic methods, static follows standard machine learning procedures, simply creating batches of arbitrary graphs, while dynamic batching is ”spatial”, aggregating multiple graphs at the same snapshot time, instead of multiple snapshots of the same evolving graph. After the models are trained, forecasts and error computations are performed on the horizon days of the validation or evaluation phase. In conclusion, the test split at training time is used to measure the performances of the link prediction task using the ROC AUC, while the validation horizon is used to measure the performances of the overall forecasting method and tune some parameters of the interpolation method which are not related to the link prediction. At the end, the final results are obtained on the data of the evaluation phase.

5.2.2 Graph regression with GNNs

The experimental setup for the regression approach recalls more classical machine learning settings because the target variables are already created during the dataset generation. At training time the model is trained to generate graph representations from each input graph and learn how to use them to predict future values. In particular, after the data has undergone all the processing steps, the training dataset is converted to a DeepSNAP dataset. The DeepSNAP dataset provides some utility functions to inductively split the graphs in a train split, containing the 80% of available graphs, and a test split, containing the remaining 20%. The test split, similarly to the LpGnn, is used to evaluate the graph regression task and compute the R2 score at each training epoch. The R2 score on the test split is used to decide whether the model has improved from the previous epochs and it should be saved. Once the model is trained, it can be completely evaluated in either the validation or evaluation phase by computing the forecasting errors. The validation phase can be used to tune the predictive horizon of the model, which is not strictly related to the graph regression task.

Models are trained using Adam optimizer with tunable learning rate and weight decay, in addition, also the early stopping and the reduce learning rate on plateau services are provided. As in many regression problems, the model is trained to optimize the MSE loss function computed between each value in the prediction and real observations. Even batching follows standard machine learning procedures. In conclusion, the model is trained to get the best score on the validation phase data and it is finally tested on the evaluation phase data.

5.3 Hyperparameter tuning

Since the number of models, methods and possible configurations combined together does not allow a complete manual hyperparameter tuning some strategies are used to restrict the search space and explore possible solutions. First, a **random hyperparameter search** is applied for each method and levels between the 1st and the 9th, to select the most promising configurations. Tables 5.1 and 5.2 show the scheme followed by the random searches: two searches are applied over the LpGnn static and dynamic respectively, while the other two are used to compare RNNs and transformers in the RgGnn approach. For each combination, more than 700 experiments are run. Lower levels 11th and 12th are completely excluded from the search because the amount of data is too big for any ad-hoc and extensive random search, since their distributions of the target variable are closer to the one at the 10th level, a smaller random search is applied at level 10 and used to define the models for the last two levels. Even if it is possible to perform random searches on these specific levels by sampling smaller datasets and considering limited amount of data, this work avoid this strategy because it becomes difficult to decide the representative samples and because testing the trained models on the whole dataset it is still to slow to perform. Each run of a random search trains a model from scratch, sampling random parameters from a specified configuration, forecasts on the validation phase data and store the obtained metrics.

During the random search the data configuration is fixed for all combinations, the only difference is on the number of graphs extracted from each time series, which depends on the number of time series on the level, and it is set to create datasets of approximately 1500 time series. The customization of the dataset, namely the features to include, the sizes, the transformations, are obtained after a long trial and error phase to reach a decent compromise between the different levels. This choice restricts the search only on the model parameters. During random search the target is scaled between 0 and 1, temporal features are not scaled but transformed using sine and cosine functions and one week of window statistics and lags of the target variable, starting 28 days before, is provided. At levels 10, 11 and 12 the configuration is the same apart from the normalization of the number of sales which is avoided, because values are very small and there might be some peaks in the training data which completely shrink the majority of the values. At level 12 any creation of additional features from the weekly prices or any other features it is not feasible because no aggregations are performed since the data is at its finest form. At this stage, trainings are meant to be fast and overfit, almost all the features are exploited when possible, higher learning rates are considered and small graphs are preferred. In general, the models are fast to train and they quickly achieve good performances, but loading data and forecasting both represent time and memory bottlenecks which are not easy to overcome.

The results from the random search are used to write a leaderboard of the proposed models and to study the most and least favourite configurations. Among the best configurations selected by the random search, **grid searches** can be applied considering a restricted number of parameters. For instance, this second step is an extremely effective way to tune the LpGnns thresholds and horizons, in addition, also LpHeu parameters are tuned on the validation set using grid searches on its only two parameters: the length of the split and the number of steps in the random walk. Finally, whenever it is possible and recommended by the results, the best models are regularized to get the most from the models and understand which are the strongest contributors to the final results. In this final stage, modifications on the data, especially on the quantity, are considered. Namely, bigger graphs are created, larger lags are involved and exogenous features are both switched on or off and preprocessed differently.

5.3.1 Link prediction with GNNs

Table 5.1 shows the best results between static and dynamic LpGnn in terms of AUC. The static model outperforms the dynamic in almost all levels. Apparently, the only speculation which can be done on the benefits of using the dynamic model is based on the results of the last levels, where it is possible to observe an improvement as the number of time series increases. But regarding the comparison, it should not be excluded a simple degradation of performances of the static approach. However, a possible explanation of the improvement of the dynamic model is given by the increasing number of available sequences of snapshots as the number of time series to forecast increase, while the static model is always trained on the same amount of graphs. Even if the sequences are reduced in length in the lower levels, the dynamic method may benefit from the higher variety.

		1	2	3	4	5	6	7	8	9
AUC	Static	**0.9669**	**0.9531**	**0.8638**	0.9312	**0.9452**	**0.8675**	**0.9656**	0.5169	0.5876
	Dynamic	0.4569	0.6030	0.8432	**0.9455**	0.6500	0.5202	0.6725	**0.8754**	**0.9282**

Table 5.1: Results from the hyperparameter tuning of LpGnn on validation data for all the levels.

Additional experiments on the MAE scores reveal that the dynamic model is still a good alternative, outperforming in some cases the static approach. In addition, the dynamic method has less parameters and it is faster, but it may suffer from training on an unequal number of positive and negative edges. The dynamic method does not guarantee the creation of a balanced dataset, as it imitates the real evolution of a visibility graph. The approach taken by this work

decides to compare static and dynamic models over the link prediction only, excluding the final forecasting abilities, because theoretically the difference between the two methods relies only on the link prediction. As further explained, the results of the LpGnn are really influenced by decisions on the threshold to add the edges and the horizon, hence it is even more difficult to derive conclusions on the comparison between static and dynamic models in the forecasting step, limiting an evaluation only on the link prediction, where static models are generally better. From the configurations of the LpGnn hyperparameter tuning some final observations are drawn:

- In the static approach the GIN is one of the preferred GNN convolutional layer, followed by GCN and then GraphSAGE. Three layers and embedding size of 16 seems to be the one of the preferred architectures. Some levels score better results with larger embedding sizes and lower number of levels.
- In the static models, the L2 link embedding dominates the experiments, with an exception only at level 9 where L1 and inner product seem to perform better.
- The static model in general prefers small fully connected layers in the head, for instance with a single hidden layer of size 8. Activation functions vary a lot between models, but in general ReLU alternatives are very appreciated, such as SELU, ELU and Leaky ReLU. Similarly the dynamic experiments show a tendency over single layered heads, with size 16, and Leaky ReLU and ReLU6 activation functions.
- Dynamic models are generally faster to train and they are more lightweight and less customizable, but still performing comparably well. Experiments deploy the version -O of the EvolveGCN because the version -H implements some transformations on the input data which might be not feasible if the number of features is small.
- On the forecasting stage, both static and dynamic prefers the linear interpolation over the weighted and average methods. The static method performs better with higher thresholds, close to the maximum value, and smaller horizons, while the dynamic works relatively good with more equilibrated threshold values.

5.3.2 Graph regression with GNNs

Table 5.2 shows the best results for the RgGnn approach. In this case even the models are compared on either the R2 score and the MAE, because the majority of the R2 scores are not significantly different as the differences are minimal. Unfortunately, also the MAE scores do not provide a clear distinction between the two design choices, so at the end both RNN

architectures and transformers are equally considered valid and effective methods. Due to their novelty, transformers are chosen as the best model. This choice is also supported by the generic advantages of transformers over RNNs, namely, they can process directly entire sequences and thus, they are faster to train and more resistant to long dependency issues.

		1	2	3	4	5	6	7	8	9
R2 score	RNN	0.7118	0.8603	**0.8822**	0.963	0.9848	0.9826	0.9849	0.9739	0.9796
	TRF	**0.7336**	0.8540	0.8655	**0.9854**	0.9788	0.9740	0.9816	0.9739	0.9859
MAE	RNN	**4449.11**	818.10	**311.11**	607.77	388.8	375.43	**190.87**	**131.52**	76.09
	TRF	4893.68	**720.70**	325.02	**598.21**	**386.27**	**365.20**	205.62	135.18	**75.42**

Table 5.2: Results from the hyperparameter tuning of RgGnn on validation data for all the levels. TRF is used to indicate transformers.

In addition also results from the RgGnn hyperparameter tuning provide some interesting observations:

- The GraphSAGE model is definitely the best performing GNN architecture over GCN and GIN for both transformers and RNNs.
- Differently from the LpGnn, the best performing GNN architectures show an increasing complexity as the level number increase. Layers with 64 hidden units are preferred and but up to 128 hidden units and 3 layers can be found in the latest levels, such as the 9th.
- Regarding the complexity of the transformer, the number of layers increase along with the level number, while the number of heads does not usually overcome 2 and the hidden state size varies differently at each level. Regarding the RNN models GRUs clearly outperform LSTMs and models tend to have smaller hidden state sizes and stack up to 3 layers. Bidirectional recurrent architectures are not always beneficial. It is also noticeable an increasing model complexity which follows the number of different time series.
- Generally small batches of 16 or 32 graphs work good and very small learning rates deliver the best results.

In general, for both LpGnn and RgGnn, a good data preprocessing represents the most crucial and fundamental step to start obtaining discrete results. Random searches help selecting among the multiple well performing solutions the more suitable and discover generic patterns to investigate. For all the models execution time varies by level, and thus, by the number of time series.

Until level 9 the overall execution time, including the time series transformation, the creation of the graphs from the time series splits, the training, the forecasting and the evaluation do not generally exceed 15 minutes on the DISI cluster. This timing motivates the choice of adopting extensive hyperparameter tunings in those levels. On the 10th level time starts increasing dramatically reaching a few hours, a single run on level 11 may take up to 5 hours while on the 12 level more than 12 hours. It is important to consider that, the current implementation it is not optimized and thought for a production environment, but it is rather built around the idea of facilitating the systematic exploration of the different methods in the different hierarchical levels of the M5 data.

5.4 Final results

Table 5.3 compares the related work and the two different approaches proposed in this work on the evaluation data. LpGnn static and LpHeu obtain similar results, although the LpGnn performs better in multiple higher levels excluding the 4th. In the levels between the 6th and the 12th the LpGnn does not show any particular improvement, obtaining the same results of LpHeu. On the other hand, the RgGnn method obtains remarkably different and better results, beating all the other approaches in all the first 10 levels. Unfortunately, on the last levels RgGnn results are not as good as in the previous, but overall results suggest that the RgGnn model performs much better than LpHeu and LpGnn.

		1	2	3	4	5	6	7	8	9	10	11	12	Avg
WRMSSE	LpHeu	1.354	1.751	1.727	1.333	1.756	1.724	1.727	1.651	1.624	1.487	**1.366**	**1.259**	1.563
	LpGnn	1.199	1.314	1.583	1.695	1.600	1.729	1.726	1.652	1.624	1.487	**1.366**	**1.259**	1.520
	RgGnn	**0.311**	**0.448**	**0.584**	**0.451**	**0.717**	**0.541**	**0.718**	**0.762**	**0.989**	**1.484**	1.392	1.465	**0.821**
MAE	LpHeu	7186.29	3698.67	1163.18	2478.10	1540.07	1244.66	561.06	398.83	181.16	7.28	**3.37**	**1.47**	1538.67
	LpGnn	5827.00	2511.50	994.41	3338.25	1406.14	1251.14	560.88	398.94	181.15	7.28	**3.37**	**1.47**	1373.46
	RgGnn	**1422.00**	**809.20**	**344.04**	**812.08**	**611.23**	**361.16**	**215.40**	**163.80**	**101.93**	**7.15**	**3.57**	1.63	**404.43**

Table 5.3: Final results on the evaluation data for all the levels

It is interesting to compare current results with traditional time series forecasting models. Table 5.4 shows the results of both RgGnn and the M5 competition benchmarks. The initials "tb" stand for the hierarchical forecasting top-down strategy and "bu" for the bottom-up strategy. In bottom-up a model forecasts the values at the lowest level and the higher levels predictions are constructed by summing up the predictions at the lower levels. Oppositely, the top-down strategy disaggregates the predictions at higher levels to forecast the lower levels. ESX and ARIMAX are the traditional models described in 2.2.1 but with top-down strategy and support

	1	2	3	4	5	6	7	8	9	10	11	12	Avg
RgGnn	**0.311**	**0.448**	0.584	0.451	0.717	**0.541**	0.718	0.762	0.989	1.484	1.392	1.465	0.821
Naive	1.967	1.904	1.880	1.947	1.914	1.881	1.878	1.798	1.764	1.479	1.360	1.253	1.752
sNaive	0.560	0.673	0.718	0.623	0.708	0.760	0.829	0.801	0.888	1.223	1.205	1.176	0.847
SES	0.921	0.938	0.944	0.927	0.983	0.959	1.002	0.956	0.994	1.071	1.002	0.932	0.969
MA	0.891	0.918	0.931	0.900	0.960	0.940	0.986	0.944	0.988	1.070	1.006	0.938	0.956
CRO	0.900	0.915	0.923	0.909	0.971	0.941	0.987	0.940	0.983	1.083	1.002	0.926	0.957
optCRO	0.902	0.916	0.926	0.910	0.970	0.940	0.987	0.942	0.985	1.084	1.004	0.928	0.958
SBA	0.902	0.917	0.926	0.914	0.983	0.943	0.993	0.940	0.983	1.073	0.994	0.919	0.957
TSB	0.911	0.926	0.935	0.918	0.975	0.949	0.994	0.948	0.988	1.068	0.997	0.928	0.961
ADIDA	0.902	0.917	0.924	0.912	0.969	0.943	0.987	0.941	0.982	1.063	0.993	0.922	0.955
iMAPA	0.909	0.925	0.932	0.917	0.973	0.948	0.992	0.946	0.986	1.065	0.996	0.925	0.960
ES td	0.470	0.550	0.664	0.530	0.640	0.629	0.727	0.717	0.801	1.029	0.973	0.915	0.720
ES bu	0.426	0.514	**0.580**	0.478	**0.557**	0.577	**0.654**	**0.643**	**0.728**	1.012	0.969	0.915	**0.671**
ESX	0.350	0.494	0.627	**0.438**	0.567	0.590	0.692	0.692	0.779	1.026	0.974	0.917	0.679
ARIMA td	0.615	0.673	0.753	0.656	0.768	0.725	0.810	0.785	0.856	1.027	0.969	0.910	0.796
ARIMA bu	0.829	0.850	0.870	0.844	0.905	0.882	0.932	0.893	0.938	1.048	0.981	0.917	0.908
ARIMAX	0.374	0.514	0.638	0.459	0.606	0.604	0.707	0.700	0.787	1.019	0.968	0.912	0.691
MLP	0.892	0.942	0.974	0.910	0.972	0.965	1.016	0.984	1.026	1.084	1.014	0.943	0.977
RF	0.960	0.989	1.026	0.962	1.012	1.003	1.047	1.023	1.057	1.085	1.010	0.940	1.010
gMLP	0.882	0.914	0.919	0.923	0.996	0.967	1.013	0.953	0.997	1.063	0.991	0.920	0.961
gRF	1.062	1.073	1.081	1.071	1.108	1.096	1.116	1.075	1.089	1.078	1.001	0.932	1.065
Com b	0.522	0.591	0.644	0.561	0.641	0.647	0.718	0.696	0.771	1.012	0.963	0.907	0.723
Com t	0.517	0.592	0.693	0.571	0.688	0.661	0.755	0.738	0.819	1.026	0.970	0.912	0.745
Com tb	0.444	0.520	0.598	0.496	0.587	0.588	0.673	0.658	0.743	**1.008**	**0.960**	**0.905**	0.682
Com lg	0.886	0.922	0.936	0.898	0.959	0.948	0.989	0.948	0.986	1.058	0.989	0.921	0.953
Difference	-0.039	-0.046	+0.004	+0.027	+0.160	-0.036	+0.064	+0.119	+0.261	+0.476	+0.432	+0.560	+0.150
Rank	1st	1st	2nd	2nd	9th	1st	4th	8th	19th	25th	25th	25th	9th
Winner	0.199	0.310	0.400	0.277	0.365	0.390	0.474	0.480	0.573	0.966	0.929	0.884	0.520

Table 5.4: Comparison of the WRMSSE scores including the benchmarks of the competition from [35]. Positive differences represent the distance from the best benchmark, while negative differences represent the improvement from the best benchmark. Rank is the reached position while winner is the competition winner.

of explanatory variables. Com benchmarks are the combination of different models: Com tb averages predictions from the top-down and bottom-up ES, Com b combines ARIMA bu and ES bu and similarly Com t combines ARIMA tp and ES tp, while Com lg averages global (gMLP) and local MLPs. A global model, like gMLP or gRF, is a single model which learns from all the time series, while local models target each time series independently. During the initial experiments, the results of RgGnn on the first 9 levels were actually guided by a global common model which was performing discretely, but it was discarded in favour of a level wise approach. A detailed description of all the benchmarks can be found in [35]. In the first 8 levels the results obtained by RgGnn are promising: in the 1st, 2nd and 6th levels it beats all the other models and it is among the first 10 best performing models. The rank at level 9 is the most ambiguous, since RgGnn performs similarly to the majority of the other models, while on other levels, such as the firsts, even if results are not very different from the best benchmark, they stand out compared to the majority. On the last 4 levels, results progressively deteriorate and RgGnn starts placing last, by averaging the results from all the levels RgGnn places 9th out of 25 models. Differences from the closest better model show that in general the improvement is not large, but it is still robust as it is compared to more than 20 models, while the differences on the last level are much higher, evidencing a more consistent problem. Even though, final results are still far from the competition winner, which is reported for completeness.

Another important measure of the performances is the training and execution time of the proposed models. As anticipated in the hyperparameter tuning, the total execution time varies between levels, mainly due to the increasing number of time series to forecast. For instance, there is a common time overhead on data loading, which is shared among the GNN-based models and the Heuristic approaches and does not depend on the chosen model. Obviously, heuristic approaches do not require any training phase, and thus, they are always utterly faster than the proposed approaches. Results confirm that in general, the GNNs' training time speed is very fast. Plots in 5.1 show how the losses and the validation scores, observed during the training of different final models, converge to optimal values in a small number of iterations. It is clearly visible that the models obtain very good results in few epochs, even when the learning rates are small and dropouts are applied to regularize, such as in the shown plots. To provide a comparison on the forecasting time, the time to predict a 28 days horizon on a single time series is computed averaging several executions on multiple levels. LpHeu is the fastest, requiring usually 0.12 seconds, LpGnn spends around 0.70 seconds and RgGnn almost 1.12 seconds.

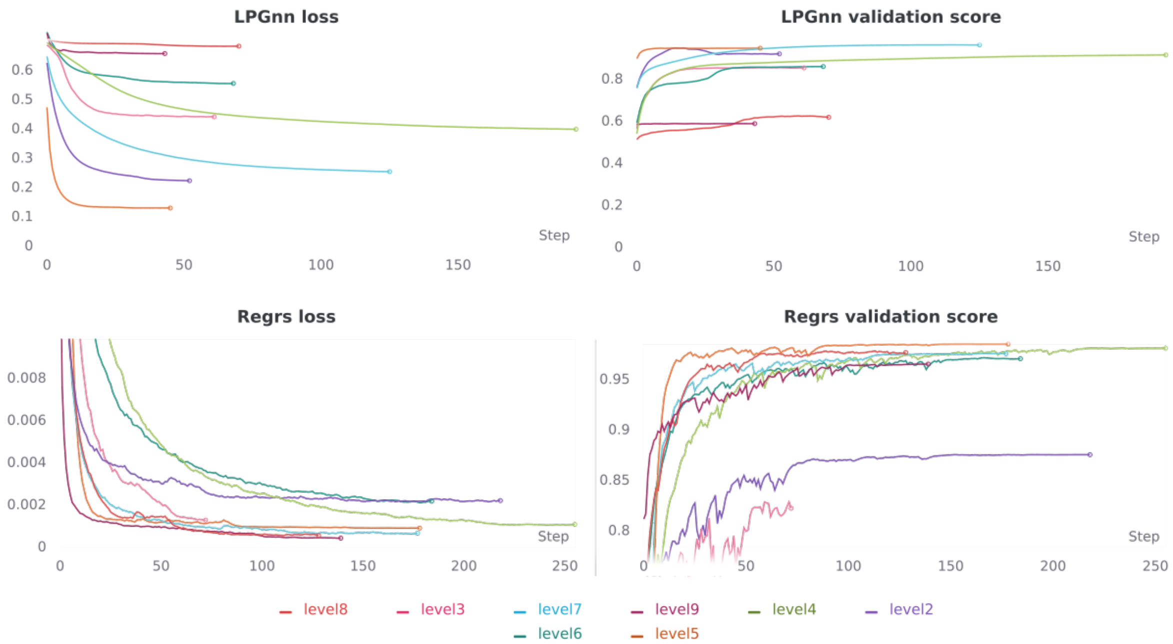


Figure 5.1: The evolution of the loss and the validation score of the LGnn static and RgGnn trained in the evaluation phase.

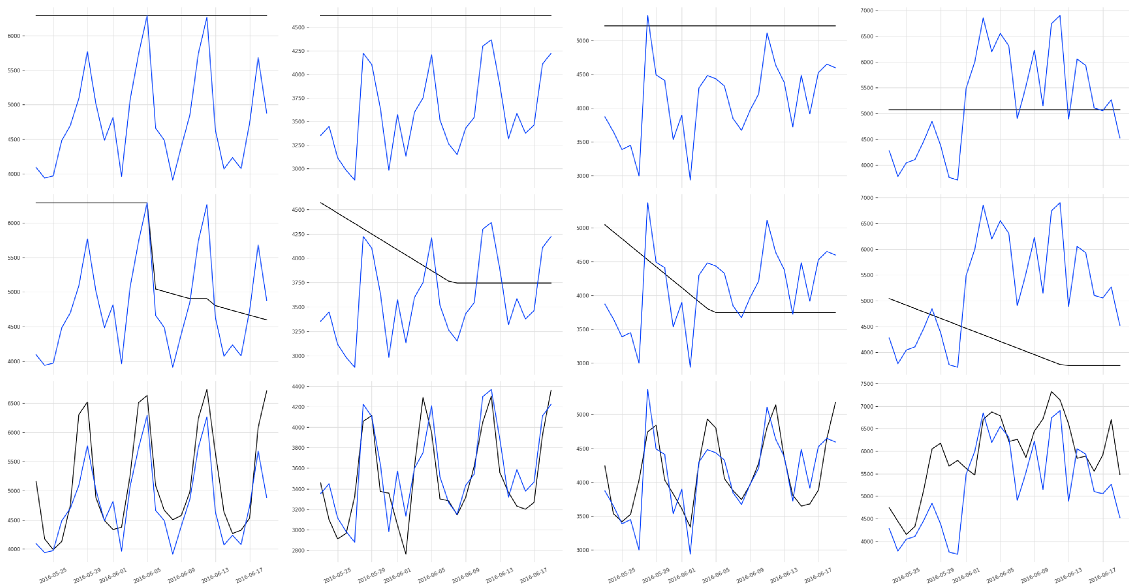


Figure 5.2: Some examples from the 6th level. In blue the real values and in black the predictions.

In the case of the RgGnn model, these observations and the obtained results, still confirm the beneficial impact of the proposed approach and its superiority over the related work. On the other hand, given the similar results to LpHeu and the additional complexity of the LpGnn models, it is not possible to consider the LpGnn approach as a valid alternative to LpHeu.

5.5 Discussion

The interpretation of the presented results can be supported by various considerations about the underlying data and the proposed methods.

The two-step curse First, link prediction methods clearly suffer from their two-step strategy. Table 5.1 shows the effectiveness of the GNN in the link prediction task during the validation phase. It is evident that the problem does not depend on this step, but on the consequent interpolation phase. Wrong interpolation settings may completely change the final outcomes, thus, their tuning represent a crucial step which requires a lot of effort. The risk of defining overfitted solutions, which are not able to generalize to future periods, is high. A clear symptom of lack of coordination between the two steps of the LpGnn is represented by the different trends between the ROC AUC scores and the MAE. Even though, also the R2 score of the RgGnn method is not always aligned with the MAE, the difference is not significant and there are no cases where bad R2 scores bring good time series forecasting performances. As stated before, the R2 score is not used to measure the forecasting skills of the model, but it represents a good and intuitive proxy, while the ROC AUC can be really misleading sometimes. It is common to observe during the hyperparameter tuning multiple models with low link prediction scores but strong forecasting results and vice versa.

An ineffective interpolation The two hyperparameters of the interpolation methods are the horizon size and the threshold used to add or not a new link. The algorithm 1 shows the possibility to forecast arbitrary horizons, but this is not theoretically correct. When multiple nodes are added to the graph, only edges between old and new nodes are predicted, hence, the edges between the new nodes are not considered. The possibility to set larger horizons is supported by some cases where it is possible to observe better results, remarking again the gap between the link prediction and the interpolation steps. Moreover, also the threshold to add new edges can be tuned to deliver better results, even if all the link prediction models are successfully trained passing the link prediction scores to a sigmoid function which maps them between 0 and 1. During training, it is possible to observe that positive edges are mapped to high scores and negative edges to very low scores, and thus, it should not be necessary to

increase the thresholds and apply a more conservative behaviour as it is done for some models. This suggests that predicting all the edges correctly may not really help the final forecasting, in particular, higher thresholds and fewer added edges can lead to better performances in certain cases.

An interesting observation, obtained by chance during testing, explains in another way the ineffectiveness of the link prediction method. In an erroneous implementation, graphs were not updated with the forecasted sales, instead real sales values were used to interpolate. In such scenario, the performances were far better than the reported ones, but there was a general tendency to generate results similar to a Naive method. This erroneous situation is interesting to analyse because it is completely independent from the link prediction, as it depends only on the interpolation, highlighting its limitations and sensitivity. In other words, with the same link prediction model the final forecasting results can change completely based solely on the interpolation. In visibility graphs, it is evident that when a node is added, its most similar node is the one representing the previous time step, which is always directly connected to it. Using the real value of the last known point allows the model to easily forecast by just adjusting the previous known point, obtaining results very similar to a Naive approach. A way to decrease this effect is using larger graphs, hoping to observe more complex relationships between the points, but it does not always work. This behaviour is also confirmed by the higher thresholds applied to the edge prediction of the LpGnn models, which push the model to assume more conservative choices in edge prediction and, in many cases, to only consider the similarity with the previous point. Many of these considerations can be applied to the LpHeu since the interpolation method is the same. In truth, also LpHeu obtains better results just observing the previous few node values, from the grid search results it is evident that there is no need to use very large graphs and explore long term similarities to obtain better results. It is also important to observe that this weakness is accentuated by the presence of a long horizon to forecast, for instance a single-step horizon would have led to the situation described in the erroneous implementation, and thus, to better results.

Similarity vs visibility Finally, an important consideration for the LpGnn methods goes to the concept of similarity between nodes of a visibility graph. Even if link prediction performances seem to work properly, it is important to investigate deeper its objective in this specific context. A common example of link prediction is discovering relationships in social networks, where it is usually more clear the concept of similarity, however in visibility graphs edges represent visibility and hence, the concept of similarity should match with the visibility one. For instance, in seasonal time series the presence of typical ”V” shapes, causes the creation of many densely connected clusters, where the nodes representing the points facing each other in

the two sides of the curve are all connected. Predicting edges in this types of graphs is very simple and it can be accomplished by just observing few examples, especially when the graphs are small. Since clusters must be linked together and intermittent time series generate more chaotic graphs, the problem of predicting links is still challenging and interesting, but from the current results it is difficult to delineate a strong relationship between effective link prediction results and good forecasting performances.

Data drives the results One of the main lessons learnt from traditional methods is exploiting as much as possible the previous observations, and the metrics on table 5.4 confirm this rule. Very simple models, such as sNaive and ES, are able to challenge more complex approaches like MLPs and RFs, by taking advantage of the evident and strong seasonalities observable in the highest levels. In link prediction, sales are added only as lagged features, but since the horizon is long 28 days, only sales older than 28 days are considered. Afterward the sales values, including the predictions, are just used in the interpolation, on the other hand, RgGnn approach allows to straightly include the sales among the node features. Since RgGnn does not adopt a two-step strategy, the learnt graph representations are directly used to forecast and train end-to-end all the neural components. In this scenario, the graphs only contain historical data and can be passed to GNNs to generate embeddings which include past target values and exogenous features in a compact way. Common approaches usually require a lot of work to properly include lags and various window-based statistics, which are difficult to update and maintain when long autoregressive forecasts are performed. This manual feature engineering phase can be replaced by training a GNN to extract meaningful features from powerful graph representations based on both topological and exogenous information. The architecture depicted in 4.6 clearly shows how the RgGnn architecture resembles typical neural network architectures composed by a feature extractor and a regressor head. A generic issue, more visible in RgGnn models, is caused by the normalization of the time series of a level using the maximum and minimum observed values among all the time series. Since the time series may have completely different volumes, some of them are normalized to values too closer to zero, disrupting the models ability to predict.

Two forecasting scenarios It is important to consider that the difference on the amount of time spent performing hyperparameter tuning on the first 10 levels compared to the last levels justifies the differences between the resulting scores. In addition, the levels before the 10th and the levels following it can be seen as two completely different forecasting scenarios, as shown during the data exploration. It is well known that forecasting in sparse and intermittent time series, as those in the last 3 levels, is a difficult task and for this reason specific models

have been developed. In particular, when data is sparse, seasonalities and trends disappear, thus, models relying on these aspects fail. The best models submitted in the competition are pure machine learning and deep learning models, especially successful are those based on ensembling of multiple LightGBM models trained with different parameters and complex cross validation approaches. The findings from the M5 competition confirm the same trend of results observed in this work, namely, the average improvement of the methods over the benchmark is 40% at level 1, which drops to about 23% at levels 5, 6, and 7, and reaches 3% at levels 10, 11 and 12 [35]. Therefore, similarly to this work, the gains of the best methods from the M5 competition mainly refer to the top and middle parts of the hierarchical levels.

6 Conclusions

This chapter provides a summarisation of the proposed work by briefly describing its motivations, main steps and results. At the end, a list of various ideas to improve and extend the current solutions is provided as a reference for future works.

6.1 Contribution summary

After presenting an extensive research on time series forecasting, it is clear that the recent direction taken by academia and industry aims at solving the challenge of automatically predicting and adapting to a high number of correlated time series and exploit external features. Taking into account the recent advancements on GNNs, this work considers interesting to propose a similar direction on a less known area of research, where time series are first converted to visibility graphs and then forecasted. In particular, the objective of this research is to replace the heuristic methods used in the previous studies with GNNs. In order to assess the performances of the new solutions the data and the benchmarks of the M5 competition are involved.

Two new methods based on GNNs are proposed to forecast multiple time series based on external features and visibility graphs. In the first, LpGnn static and dynamic models confirm that GNNs are a valid alternative to predict graph edges, compared to the heuristics used in the related work. In the second, the RgGnn model advances an original solution to leverage graph regression with GNNs to extract meaningful representations from historical data which are used to predict future values. This novel approach follows the intuition that visibility graphs may be more suitable to unveil hidden properties from topological properties, rather than generating an equivalent representation of the original time series data. Recent works adopting graph-based strategies for time series classification confirm this direction [42, 29, 65]. In addition, GNNs are able to combine both topological and node features enabling the possibility to easily integrate exogenous features, which in time series forecasting are not traditionally considered as crucial as in machine learning. Conclusive comments on the M5 competition confirm the importance of using exogenous features in multiple time series forecasting to effectively account for the

effects of external factors that can affect the values of the historical data. RgGnn demonstrates to overcome the results of the heuristic methods, without exceeding the reasonable trade-off imposed on computational costs and time during both training and inferencing. This becomes particularly relevant if the complexity of the M5 competition is considered, as well as the intricacy of the winning solutions which are intensively trained and manually adjusted.

By using the M5 competition data, several models are trained and tested to predict a grand total of 42840 time series. The M5 competition offers the opportunity to measure the results on a real world demand forecasting dataset and to compare the results with well designed benchmarks. Final comparisons shows that RgGnn obtains results comparable to the most common traditional and machine learning models. Even if the results are still far from the competition winner, the fast training time and the contained complexity of the RgGnn model confirms its ability to produce discrete forecasts on moderately sized time series datasets. There are no doubts that the M5 dataset poses a new challenging environment for many time series forecasting models, especially due to the sparsity of the data at the lower levels. This particular aspect is inherently difficult to handle, and the models proposed in this work are not meant to overcome this specific obstacle, but it is still interesting to study the behaviour of these methods in such scenarios. Unfortunately, since the majority of the time series are concentrated on the lowest levels, it is difficult to decouple multiple time series forecasting and intermittent time series forecasting, compromising the performances of the former. However, several experiments confirm the effectiveness of RgGnn using a single hyperparameter configuration on all the levels below 10.

Despite the good performances on link prediction, this work does not qualify LpGnn as a significant improvement to the initial works. The analysis of the results suggests that there are several drawbacks on this method, especially on the two-step approach and on how the interpolation strategies operate. The initial objective of enhancing the methods proposed in the related work, by solely adopting GNNs to build link prediction models, encountered an unexpected obstacle represented by the interpolation involved in the conversion from graphs to time series. Moreover, multiple observations drew the conclusion that link prediction capabilities are difficult to relate to the final forecasting performances, especially when longer horizons are involved, as in the M5 dataset. This consideration holds for any link prediction method cited in this work, indeed, none of the related work mention possible solutions to forecast on multi-step horizons and current interpolation methods clearly have difficulties on this side. By looking at the predictions and at the performances, it is possible to compare link prediction to the Naive method and RgGnn to the ES.

6.2 Future works

Following the previous observations, an interesting future direction aims to fill the gap between the link prediction and the interpolation steps, by proposing either a single machine learning method to reconstruct the time series or two separate machine learning components coupled together in an end-to-end training. Other improvements can be either predicting edges between the new nodes when the horizon is larger than one or developing a new concept of similarity which is more related to the visibility, even if, these changes are considered of minor impact. In the future, it could be interesting to reconsider the dynamic link prediction strategy as this work is limited only on a specific model and the obtained results are still interesting, since many dynamic models are better at forecasting than the static strategy. On the RgGnn side, it is interesting to explore how far the model can go by using a sequence-to-sequence method. Some experiments prove that using an encoder-decoder architecture, where the encoder takes the same input as the RgGnn and the decoder is fed with the exogenous features of the future time steps, not only benefits the results in general, but it allows to predict larger horizons in a single step, reducing the prediction time of the model. This represents an improvement over the current neural network architectures and settings, which suffer from regressive cumulative errors and latencies. In addition to the model enhancements, more analyses on the structural properties of visibility graphs and how they can be used to improve current results should be carried out accompanied by a deeper ablation study. Once results are more mature, GNNs and visibility graphs can be used together to develop a more general framework able to encompass time series forecasting, clustering and classification. Alternatively, a similar work proposed by Time2Vec [25] can be thought to produce general-purpose representation for time series data that can be potentially used in any architecture.

Bibliography

[1] Back to (predict) the future - interactive m5 eda. https://www.kaggle.com/headsortails/back-to-predict-the-future-interactive-m5-eda. Accessed: 2021-09-30.

[2] Ratnadip Adhikari and R. K. Agrawal. *An Introductory Study on Time Series Modeling and Forecasting*. 2013.

[3] Nino Arsov and Georgina Mirceva. Network embedding: An overview, 2019.

[4] Gowtham Atluri, Anuj Karpatne, and Vipin Kumar. Spatio-temporal data mining. *ACM Computing Surveys*, 51(4):1–41, September 2018.

[5] Konstantinos Benidis, Syama Sundar Rangapuram, Valentin Flunkert, Bernie Wang, Danielle C. Maddix, Ali Caner Türkmen, Jan Gasthaus, Michael Bohlke-Schneider, David Salinas, Lorenzo Stella, Laurent Callot, and Tim Januschowski. Neural forecasting: Introduction and literature overview. *CoRR*, abs/2004.10240, 2020.

[6] Lukas Biewald. Experiment tracking with weights and biases. https://www.wandb.com/, 2020. Software available from wandb.com.

[7] Gianluca Bontempi, Souhaib Ben Taieb, and Yann-Aël Le Borgne. Machine learning strategies for time series forecasting. In *Business Intelligence*, pages 62–77. Springer Berlin Heidelberg, 2013.

[8] Defu Cao, Yujing Wang, Juanyong Duan, Ce Zhang, Xia Zhu, Congrui Huang, Yunhai Tong, Bixiong Xu, Jing Bai, Jie Tong, and Qi Zhang. Spectral temporal graph neural network for multivariate time-series forecasting. *CoRR*, abs/2103.07719, 2021.

[9] S. Chen, X. Lan, Y. Hu, Q. Liu, and Y. Deng. The time series forecasting: from the aspect of network, 2014.

[10] Tianqi Chen and Carlos Guestrin. XGBoost. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, August 2016.

[11] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2014.

[12] Wikipedia contributors. Supplemental nutrition assistance program — Wikipedia, the free encyclopedia, 2021. [Online; accessed 10-April-2021].

[13] J. D. Croston. Forecasting and stock control for intermittent demands. *Journal of the Operational Research Society*, 23(3):289–303, September 1972.

[14] James Durbin and Siem Jan Koopman. *Time Series Analysis by State Space Methods*. Oxford University Press, 2 edition, 2012.

[15] Shereen Elsayed, Daniela Thyssens, Ahmed Rashed, Lars Schmidt-Thieme, and Hadi Samer Jomaa. Do we really need deep learning models for time series forecasting? *CoRR*, abs/2101.02118, 2021.

[16] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR 2019 Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[17] Rodrigo García-Herrera. visibility-graph. `https://github.com/rgarcia-herrera/visibility_graph`, 2021.

[18] M. Gori, G. Monfardini, and F. Scarselli. A new model for learning in graph domains. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.* IEEE, 2005.

[19] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[20] William L. Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M.

Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *NIPS*, pages 1024–1034, 2017.

[21] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, November 1997.

[22] R.J. Hyndman and G. Athanasopoulos. *Forecasting: principles and practice, 3rd edition*. OTexts: Melbourne, Australia, 2021.

[23] Rob Hyndman. Another look at forecast accuracy metrics for intermittent demand. *Foresight: The International Journal of Applied Forecasting*, 4:43–46, 01 2006.

[24] Peter J. Brockwell and Richard A. Davis. *Introduction to Time Series and Forecasting*. Springer-Verlag New York, 2002.

[25] Seyed Mehran Kazemi, Rishab Goel, Sepehr Eghbali, Janahan Ramanan, Jaspreet Sahota, Sanjay Thakur, Stella Wu, Cathal Smyth, Pascal Poupart, and Marcus Brubaker. Time2vec: Learning a vector representation of time. *CoRR*, abs/1907.05321, 2019.

[26] Seyed Mehran Kazemi, Rishab Goel, Kshitij Jain, Ivan Kobyzev, Akshay Sethi, Peter Forsyth, and Pascal Poupart. Representation learning for dynamic graphs: A survey. *J. Mach. Learn. Res.*, 21:70:1–70:73, 2020.

[27] Lucas Lacasa, Bartolo Luque, Fernando Ballesteros, Jordi Luque, and Juan Carlos Nuño. From time series to complex networks: The visibility graph. *Proceedings of the National Academy of Sciences*, 105(13):4972–4975, March 2008.

[28] Colin Lea, Michael D. Flynn, Rene Vidal, Austin Reiter, and Gregory D. Hager. Temporal convolutional networks for action segmentation and detection. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, July 2017.

[29] Chenyang Li, Lingfei Mo, and Ruqiang Yan. Rolling bearing fault diagnosis based on horizontal visibility graph and graph neural networks. In *2020 International Conference on Sensing, Measurement & Data Analytics in the era of Artificial Intelligence (ICSMD)*. IEEE, October 2020.

[30] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. 2015.

[31] Bryan Lim and Stefan Zohren. Time-series forecasting with deep learning: a survey. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 379(2194):20200209, February 2021.

[32] Fan Liu and Yong Deng. A fast algorithm for network forecasting time series. *IEEE Access*, 7:102554–102560, 2019.

[33] Zhiyuan Liu and Jie Zhou. Introduction to graph neural networks. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 14(2):1–127, March 2020.

[34] Spyros Makridakis, Evangelos Spiliotis, and Vassilios Assimakopoulos. Statistical and machine learning forecasting methods: Concerns and ways forward. *PLOS ONE*, 13(3):e0194889, March 2018.

[35] Spyros Makridakis, Evangelos Spiliotis, and Vassilios Assimakopoulos. The m5 competition: Background, organization, and implementation. *International Journal of Forecasting*, September 2021.

[36] Shengzhong Mao and Fuyuan Xiao. Time series forecasting based on complex network analysis. *IEEE Access*, 7:40220–40229, 2019.

[37] Douglas C. Montgomery, Cheryl L. Jennings, and Murat Kulahci. *Introduction to Time Series Analysis and Forecasting*. Wiley-Interscience, 2008.

[38] Boris N. Oreshkin, Dmitri Carpov, Nicolas Chapados, and Yoshua Bengio. N-BEATS: neural basis expansion analysis for interpretable time series forecasting. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[39] Aldo Pareja, Giacomo Domeniconi, Jie Chen, Tengfei Ma, Toyotaro Suzumura, Hiroki Kanezashi, Tim Kaler, Tao B. Schardl, and Charles E. Leiserson. Evolvegcn: Evolving graph convolutional networks for dynamic graphs. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 5363–5370. AAAI Press, 2020.

[40] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances*

in Neural Information Processing Systems 32, pages 8024–8035. Curran Associates, Inc., 2019.

[41] Bohdan Pavlyshenko. Machine-learning models for sales time series forecasting. *Data*, 4(1):15, January 2019.

[42] Aleksandr Pletnev, Rodrigo Rivera-Castro, and Evgeny Burnaev. Graph neural networks for model recommendation using time series data. In *2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, December 2020.

[43] Syama Sundar Rangapuram, Matthias Seeger, Jan Gasthaus, Lorenzo Stella, Yuyang Wang, and Tim Januschowski. Deep state space models for time series forecasting. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 7796–7805, Red Hook, NY, USA, 2018. Curran Associates Inc.

[44] Ying Rex, You Jiaxuan, Zhang Zecheng, He Xinwei, Sosic Rok, and Leskovec Jure. Deepsnap. `https://github.com/snap-stanford/deepsnap/`, 2020.

[45] João Rico, José Barateiro, and Arlindo Oliveira. Graph neural networks for traffic forecasting. *CoRR*, abs/2104.13096, 2021.

[46] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael M. Bronstein. Temporal graph networks for deep learning on dynamic graphs. *CoRR*, abs/2006.10637, 2020.

[47] Amit Roy, Kashob Kumar Roy, Amin Ahsan Ali, M Ashraful Amin, and A K M Mahbubur Rahman. Unified spatio-temporal modeling for traffic forecasting using graph neural network. In *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE, July 2021.

[48] David Salinas, Valentin Flunkert, Jan Gasthaus, and Tim Januschowski. Deepar: Probabilistic forecasting with autoregressive recurrent networks. *International Journal of Forecasting*, 36(3):1181–1191, 2020.

[49] F. Scarselli, M. Gori, Ah Chung Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, January 2009.

[50] Rajat Sen, Hsiang-Fu Yu, and Inderjit S. Dhillon. Think globally, act locally: A deep neural network approach to high-dimensional time series forecasting. In Hanna M. Wallach,

Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 4838–4847, 2019.

[51] Chao Shang, Jie Chen, and Jinbo Bi. Discrete graph structure learning for forecasting multiple time series. *CoRR*, abs/2101.06861, 2021.

[52] Slawek Smyl. A hybrid method of exponential smoothing and recurrent neural networks for time series forecasting. *International Journal of Forecasting*, 36(1):75–85, 2020.

[53] Evangelos Spiliotis, Spyros Makridakis, Artemios-Anargyros Semenoglou, and Vassilios Assimakopoulos. Comparison of statistical and machine learning methods for daily SKU demand forecasting. *Operational Research*, September 2020.

[54] Devendra Swami, Alay Dilipbhai Shah, and Subhrajeet K. B. Ray. Predicting future sales of retail products using machine learning. *CoRR*, abs/2008.07779, 2020.

[55] Sean J Taylor and Benjamin Letham. Forecasting at scale. PeerJ, September 2017.

[56] U Thissen, R van Brakel, A.P de Weijer, W.J Melssen, and L.M.C Buydens. Using support vector machines for time series prediction. *Chemometrics and Intelligent Laboratory Systems*, 69(1-2):35–49, November 2003.

[57] Unit8. Darts. `https://github.com/unit8co/darts/`, 2021.

[58] Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew W. Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. In *The 9th ISCA Speech Synthesis Workshop, Sunnyvale, CA, USA, 13-15 September 2016*, page 125. ISCA, 2016.

[59] Petar Velickovic, Guillem Cucurull, A. Casanova, A. Romero, P. Liò, and Yoshua Bengio. Graph attention networks. *ArXiv*, abs/1710.10903, 2018.

[60] Minggang Wang, Andre L. M. Vilela, Lixin Tian, Hua Xu, and Ruijin Du. A new time series prediction method based on complex network theory. In Jian-Yun Nie, Zoran Obradovic, Toyotaro Suzumura, Rumi Ghosh, Raghunath Nambiar, Chonggang Wang, Hui Zang, Ricardo Baeza-Yates, Xiaohua Hu, Jeremy Kepner, Alfredo Cuzzocrea, Jian Tang, and Masashi Toyoda, editors, *BigData*, pages 4170–4175. IEEE Computer Society, 2017.

[61] Neo Wu, Bradley Green, Xue Ben, and Shawn O'Banion. Deep transformer models for time series forecasting: The influenza prevalence case. *CoRR*, abs/2001.08317, 2020.

[62] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, January 2021.

[63] Zonghan Wu, Shirui Pan, Guodong Long, Jing Jiang, Xiaojun Chang, and Chengqi Zhang. Connecting the dots: Multivariate time series forecasting with graph neural networks. *CoRR*, abs/2005.11650, 2020.

[64] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2019.

[65] Qi Xuan, Kunfeng Qiu, Jinchao Zhou, Zhuangzhi Chen, Dongwei Xu, Shilian Zheng, and Xiaoniu Yang. Adaptive visibility graph neural network and its application in modulation classification. *CoRR*, abs/2106.08564, 2021.

[66] Haitao Yuan and Guoliang Li. A survey of traffic prediction: from spatio-temporal data to intelligent transportation. *Data Science and Engineering*, 6(1):63–85, January 2021.

[67] Tianxiang Zhan and Fuyuan Xiao. A novel weighted approach for time series forecasting based on visibility graph, 2021.

[68] Junyin Zhao, Hongming Mo, and Yong Deng. An efficient network method for time series forecasting based on the DC algorithm and visibility relation. *IEEE Access*, 8:7598–7608, 2020.

[69] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.

[70] Yong Zou, Reik V. Donner, Norbert Marwan, Jonathan F. Donges, and Jürgen Kurths. Complex network approaches to nonlinear time series analysis. *Physics Reports*, 787:1–97, January 2019.