GRAPH-TIME CONVOLUTIONAL NEURAL NETWORK

LEARNING FROM TIME-VARYING SIGNALS DEFINED ON GRAPHS

GRAPH-TIME CONVOLUTIONAL NEURAL NETWORK

LEARNING FROM TIME-VARYING SIGNALS DEFINED ON GRAPHS

Thesis

to obtain the degree of Master in Computer Science, Data Science and Technology track, at Delft University of Technology,

to defend in public on Thursday, July 23rd, 2020 at 10:00

by

Gabriele MAZZOLA
born in Vigevano, Italy.

Multimedia Computing Group,
Faculty EEMCS, Delft University of Technology,
Delft, The Netherlands.

Thesis committee:

Chair:	Dr. Odette Scharenborg, Faculty EEMCS, TU Delft
Daily Supervisor:	Dr. Elvin Isufi, Faculty EEMCS, TU Delft
Committee Member:	Dr. Ir. F.C. Vossepoel, Faculty CEG, TU Delft
Committee Member:	Prof. Dr. Ir. G.J.T. Leus, Faculty EEMCS, TU Delft

Keywords: time-varying graph signals; graph convolutional neural networks; graph signal processing; product graphs.

ACKNOWLEDGEMENTS

At the end of this journey, I would like to take some time to thank all those people who made this possible and, without whom, this project would have had a totally different flavour.

First, I want to thank my supervisor, Elvin, for all the discussions and critical feedback you gave me throughout these nine months of work. My technical writing improved so much thanks to your comments! Furthermore, I want to thank Odette Scharenborg, Femke Vossepoel and Geert Leus for accepting to be part of my thesis committee.

Second, my study buddies: Bianca, Jody, Colm, Matteo, Panos, Rafal, Zubeer, and Kyriakos. Thank you for all the daily coffee/tea breaks and the good times we've had together before the pandemic happened! You made the first half of this project a lot more fun. A special thanks to Bianca, who provided me invaluable feedback throughout these months, and who was always there to discuss any technical (and non) topic. I also want to thank Kate for the fantastic nights spent playing zombies on Call of Duty together!

I also want to thank my family for believing in me and for being there whenever I was in need of something. You really made this journey possible!

Last but not least, I want to give a special thanks to my girlfriend, Juliane. Thank you so much for being part of my life, and for making this whole isolation time feel wonderful.

SUMMARY

Time-varying network data are essential in several real-world applications, such as temperature forecasting and earthquake classification. Spatial and temporal dependencies characterize these data and, therefore, conventional machine learning tools often fail to learn these joint correlations from data. On the one hand, hybrid models to learn from time-varying network data combine several specialized models able to capture these dependencies separately, thus ignoring their joint spatio-temporal interactions. On the other hand, state-of-the-art approaches for jointly learning time-varying network data do not exploit the useful prior provided by a graph-time product graph. This prior structural knowledge can aid learning and help the models improve their performance. For this reason, we propose a novel neural network architecture to learn from time-varying network data using product graphs and graph convolutions, thus exploiting this prior during learning. In particular, our architecture (i) learns the most suitable graph-time product graph to represent the time-varying network data and model the graph-time interactions; (ii) performs graph convolutions over this product graph to learn the graph-time interactions from data; (iii) employs graph-time pooling to reduce the dimensionality over layers. To the best of our knowledge, no research has yet attempted to capture spatial and temporal dependencies using graph convolutions over product graphs. Results on synthetic and real-world data show that the proposed method is useful in learning from time-varying network data for both regression and classification tasks. For classification, we cure a real-world dataset for earthquake classification and compare the proposed approach with state-of-the-art models. Results indicate that graph-aware models outperform graph-unaware models on this task. For regression, we evaluate the proposed method on two real-world temperature datasets and compare our architecture with graph-aware and graph-unaware models. Results on the smaller dataset show that linear models outperform neural-network models. On the larger dataset, we find that prior knowledge about graph-time interactions seems to be less beneficial in case of abundance of data. For the synthetic experiments, we find that (i) learning the structure of the graph-time product graph from data improves the performance compared to adopting a fixed type of product graph; (ii) learning sparser graph-time product graphs further improves the performance; (iii) the proposed graph-time pooling technique contributes to the model's generalization capabilities.

CONTENTS

1

INTRODUCTION

1

Network data are important in several real-world applications such as temperature forecasting [1], traffic management [2], and earthquake detection [3], to name a few (see [4] for an extended survey). These network data often evolve over both space and time. As an example, consider a network of weather stations and links representing distances between pairs of nodes. The stations measure temperature over a temporal observation window, as shown in Figure 1.1. The resulting time-varying data possesses both spatial and temporal properties since the temperature recorded at a particular station correlates with previous values at that station, but also with the temperatures recorded at neighbouring stations. These combined dependencies over space and time make it challenging to capture the correlations in time-varying data. Therefore, conventional tools focusing either on spatial or temporal dependencies fail to learn these spatio-temporal correlations [2, 5]. To overcome this issue and successfully learn from time-varying network data, a learning framework is required which jointly exploits these correlations [6, 4, 5].

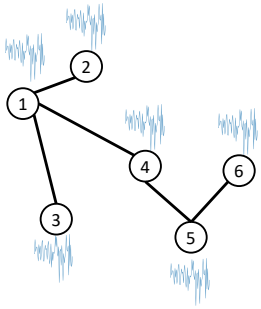


Figure 1.1: Time-varying network data. In this example, the network consists of six weather stations and each station records temperatures over a temporal observation window. The links between the stations encode the distance between them.

Graphs are a suitable and widespread tool to model complex relationships between the data and represent the underlying structure. The most appealing characteristic about graphs is their flexibility to model information living on irregular and complex structures. In the weather station example introduced above, the graph represents distances between stations, but a graph can also represent a road network or spatial dependencies in a seismic network. In general, graphs can serve as a relevant prior about the data structure, which is proven to be beneficial to learn meaningful representations [7].

Difficulties arise when learning from graph data as they do not live in the structured Euclidean domain. For example, standard Convolutional Neural Networks (CNN) cannot learn from graph data. The reason for this is the convolution operation assuming Euclidean data as input. To overcome this issue, researchers have worked on methods to apply learning techniques to unstructured data, giving life to the so-called Graph Convolutional Neural Networks (GCNNs) [6]. Analogously to CNNs performing standard convolutions over the Euclidean domain, GCNNs perform graph convolutions over unstructured data defined on a graph. GCNNs are often employed to capture the spatial

correlations from the data, while RNNs (and its variants) or CNNs are adopted to capture the temporal correlations [6]. GCNNs, however, ignore the temporal dependencies in time-varying data and likewise, the popular RNN models ignore spatial dependencies in network data.

Building upon GCNNs and RNNs, the number of works that try to learn spatio-temporal dependencies is steadily growing [6, 4]. This is because recent findings have shown the benefits of jointly processing time-varying network data [8, 2]. A first direction to capture spatio-temporal dependencies comprises hybrid model, exploiting GCNNs to learn the spatial dependency, and subsequently an RNN or CNN to learn the temporal dependency [6, 4, 9, 10, 5]. Conversely, other works achieve spatio-temporal learning by fusing the spatial and temporal information in a unified model, often modifying recurrent models to also take into account the topology of the data [7, 11, 3, 12, 5, 13]. We shall detail hybrid and fused models in Chapter 3. Our rationale is that combining several specialized models to capture graph-time dependencies misses some information in graph-time correlations since part of the pipeline focuses on learning the spatial dependency and part on the temporal dependency. Therefore, we follow the direction of the fused models and propose a unified graph-time learning framework.

We argue that the most natural way to model time-varying network data is to represent them on an extended graph that provides a structure for values over the graph and over time. This structure is common in graph signal processing literature, and it is known as product graph [14]. Therefore, product graphs provide a useful prior w.r.t. the graph-time interactions in the data. We show an example of modelling time-varying network data with product graphs in Figure 1.2. Such a representation allows graph convolutions to capture the joint correlations in the data. However, state-of-the-art models ignore this prior knowledge and do not exploit such information during learning. To the best of our knowledge, no research has yet attempted to capture spatial and temporal dependencies using graph convolutions over product graphs.

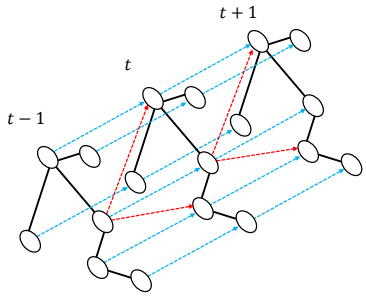


Figure 1.2: Product graph representation of the temperature time-varying network data over three timesteps. A static network data on this larger graph can now represent all the measurements recorded by the network in Figure 1.1 over three timesteps. In this example, blue links imply that the temperature at a particular station affects the subsequent temperature at the same station. Red links imply that the temperature at a particular station affects the subsequent temperatures at its neighbouring stations. For illustration purposes, we show the red links only for the central station in the network.

Therefore, in this thesis, we investigate how to learn from time-varying network data through a fully convolutional framework over product graphs, answering the research question:

(RQ) *"How to learn meaningful representations from time-varying network data by means of graph convolutions and product graphs?"*

We first use product graphs to represent time-varying network data as static data over this extended graph. There are different types of product graphs, each modelling the graph-time interactions differently [cf. Section 2.3.1]. Thus, we first ask the following question:

(RQ1) *"How to learn the correct product graph for modelling signal variations over graph and time domains?"*

To answer this question, we propose to use parametric product graphs [15] in the learning framework and learn the most suitable graph-time interactions for the task at hand. Once we represent the time-varying network data using product graphs, we ask the question:

(RQ2) *"How to develop a graph convolutional architecture to jointly process time-varying graph signals over parametric product graphs?"*

We propose a novel neural network architecture to perform graph convolutions over a learned parametric product graph. Graph convolutions are performed using the graph filters introduced in [16], which consist of polynomials in the shift operator. We also propose a new graph-time pooling module that reduces the dimensionality of the convolutional features both over the graph and the time domain. This pooling strategy is inspired by the so-called zero-padding pooling used in conventional GCNNs [17]. Finally, for the proposed architecture we ask the research question:

(RQ3) *"How to use the developed architecture for classification and regression tasks?"*

For classification, we predict the geographic location of an earthquake's epicentre given seismic measurements prior to the strike, relying on the data from the International Federation of Digital Seismograph Networks (FDSN) in New Zealand [18]. For regression, we forecast temperatures across several locations in two real-world datasets: the Molene [19] and the NOAA [20] datasets, providing temperature data across an area of France and the US, respectively.

More specifically, the answer to these research questions yielded the following contributions:

- A novel neural network architecture that can learn from time-varying network data (Chapter 4). Within this architecture, we propose:
  - a parametric graph-time convolutional layer, which uses product graphs to learn the graph-time interactions from data (Section 4.2). We also propose a regularization term to enforce sparsity in the learned product graphs.
  - a graph-time pooling layer, which builds upon the zero-padding pooling in [17] to reduce the dimensionality of the features also over time (Section 4.3).

- A new dataset for classifying seismic measurements, which relies on seismic wave measurements in New Zealand. We use this dataset to evaluate the proposed architecture for classification (Chapter 5).
- An evaluation of the proposed approach for temperature forecasting with the Molene and the NOAA datasets (Chapter 6).

This thesis is organized as follows. Chapter 2 introduces the background information. Chapter 3 contains the literature review related to spatio-temporal learning. Chapter 4 presents the proposed GTCNN architecture to learn from time-varying graph signals. Chapter 5 evaluates the proposed architecture for earthquake classification, based on measurements prior to the strike. Chapter 6 evaluates the proposed architecture for temperature forecasting. Finally, Chapter 7 concludes this thesis and lays down some future research directions.

2

BACKGROUND

In this chapter, we introduce the background information that we will leverage in the following chapters. The chapter is organized as follows. Section 2.1 introduces the building blocks of Graph Signal Processing. Section 2.2 presents the Graph Convolutional Neural Network model, including the graph convolutional layer and pooling. Section 2.3 introduces the concept of time over graphs and shows how we can represent time series on graphs using product graphs. Finally, Section 2.4 introduces graph-aware and graph-unaware models for time series modelling.

2.1. GRAPH SIGNAL PROCESSING

Graph Signal Processing (GSP) is a framework that extends concepts of signal processing such as Fourier transform and filters to unstructured domains such as graphs [21]. This section introduces the building blocks of GSP. Section 2.1.1 defines graphs as well as their matrix representations. Section 2.1.2 introduces graph signals and the Graph Fourier Transform. Section 2.1.3 defines the Graph Shift Operator, which is then used in Section 2.1.4 to formulate graph convolutional filters.

2.1.1. GRAPHS

A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ consists of a set of N nodes $\mathcal{V} = \{1, \dots, N\}$ and a set of edges $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ connecting the nodes. We denote by $|\mathcal{E}|$ the number of edges in the graph. If $(i, j) \in \mathcal{E}$, it means nodes i and j are connected, i.e., there exists an edge connecting nodes i and j. Graphs can be *undirected* or *directed*. For *undirected* graphs, if edge $(i, j) \in \mathcal{E}$ then also $(j, i) \in \mathcal{E}$; i.e., if node i is connected to node j, also node j is connected to node i. In *directed* graphs, edges have a direction associated with them; i.e., $(i, j) \in \mathcal{E}$ does not imply that $(j, i) \in \mathcal{E}$. We define $\mathcal{N}_i^k$ as the set of nodes that are reachable from node i in k hops or fewer, i.e., following a path with k edges or fewer. For directed graphs, we look at incoming edges when computing the neighbourhoods. Figure 2.1 shows two graphs of six nodes. In the undirected graph on the left, $\mathcal{N}_4^1 = \{1, 5\}$, while $\mathcal{N}_4^2 = \{1, 2, 3, 4, 5, 6\}$. In the directed graph on the right, $\mathcal{N}_4^1 = \{1\}$, while $\mathcal{N}_4^2 = \{1, 2\}$.

(a) Undirected graph.

(b) Directed graph.

Figure 2.1: Nodes one to six are represented with circles containing the indices. The lines connecting the nodes represent edges. An edge with the arrow pointing from node i to node j means $(j, i) \in \mathcal{E}$.

The *adjacency* matrix $\boldsymbol{A} \in \mathbb{R}^{N \times N}$ is one way to represent a graph $\mathcal{G}$. The entries of $\boldsymbol{A}$ are:

$$A_{ij} \geq 0 \text{ if } (i, j) \in \mathcal{E} \text{ and } A_{ij} = 0 \text{ otherwise.}$$

The scalar A_{ij} captures the strength of the connection between nodes i and j. If all edges $(i, j) \in \mathcal{E}$ have binary weights $A_{ij} \in \{0, 1\}$, then the graph is said *unweighted*. For undirected graphs, the adjacency matrix $\boldsymbol{A}$ is symmetric, i.e., $\boldsymbol{A}^\top = \boldsymbol{A}$. For directed graphs, we follow the notation adopted in [21]: if the entry A_{ij} is non-zero, it means there is an edge going from node j to node i, i.e., the i-th row of $\boldsymbol{A}$ indicates the in-edges of node i.

The *degree* matrix $\mathbf{D} \in \mathbb{R}^{N \times N}$ for undirected graphs is a diagonal matrix with diagonal elements $D_{ii} = \sum_{j=1}^{N} A_{ij}$. For unweighted graphs, the degree of a node is the number of

its edges. For weighted graphs, the degree of a node is the sum of the weights of its edges. We can also represent the graph with the *normalized adjacency* matrix $\boldsymbol{A}_{\mathrm{n}} = \boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{A}\boldsymbol{D}^{-\frac{1}{2}}$. The graph *Laplacian* matrix $\mathbf{L} \in \mathbb{R}^{N\times N}$ is another representation matrix of graph $\mathcal{G}$ and is defined as

$$\boldsymbol{L} = \boldsymbol{D} - \boldsymbol{A}$$

which is also symmetric. Likewise, the *normalized Laplacian* matrix is $\boldsymbol{L}_{\mathrm{n}} = \boldsymbol{D}^{-\frac{1}{2}}\boldsymbol{L}\boldsymbol{D}^{-\frac{1}{2}}$.

The adjacency matrix $\boldsymbol{A}$ can be used with both directed and undirected graphs, while the Laplacian matrix $\boldsymbol{L}$ can only be used with undirected graphs. If we need the eigenvalues of the matrix to be bounded, we should use the normalized counterpart. The choice of the graph representation matrix is, however, application dependent. For example, from a graph spectral perspective, the graph Laplacian matrix is a popular choice [21].

In general, we will consider a matrix $\boldsymbol{S}$ to represent the graph called the graph shift operator (GSO). The GSO is a square matrix $\boldsymbol{S} \in \mathbb{R}^{N\times N}$ having the same sparsity pattern of the graph. In other words, $S_{ij} \neq 0$ if and only if $(i, j) \in \mathcal{E}$. Valid choices for the GSO are the graph matrices introduced above, such as the adjacency matrix, the graph Laplacian, and their normalized counterparts.

2.1.2. GRAPH SIGNALS

A graph signal $\boldsymbol{x} = [x_1, x_2, \ldots, x_N]^\top$ is defined as a set of N values, where entry x_i is associated with node $i \in \mathcal{V}$ of the graph. The edges connect nodes, thus encoding pairwise relationships between signal values. For instance, a graph signal can represent measurements obtained from a network of N sensors: the graph topology $\boldsymbol{A} \in \mathbb{R}^{N\times N}$ encodes pairwise relationships between sensors (the distance between them or the communication network), while the graph signal $\boldsymbol{x}$ represents the values measured by the sensor network. Figure 2.2 illustrates an example of graph signal.

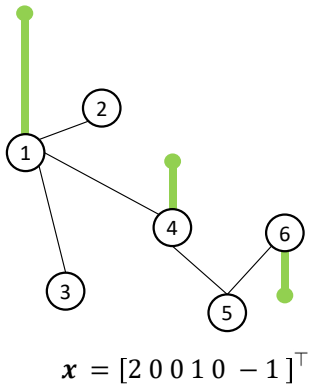


Figure 2.2: A graph signal $\boldsymbol{x}$ defined on top of a graph of six nodes. The green lines on top of the nodes represent the value of the graph signal associated with each node. The signal is zero on the nodes without a green line.

GRAPH FOURIER TRANSFORM

The Graph Fourier Transform (GFT) is the equivalent for graph signals of the standard Fourier Transform [16]. When the GSO allows for its eigendecomposition (such as the case where $\boldsymbol{S} = \boldsymbol{L}$), the eigenvectors represent the modes of the graph and the eigenvalues represent the frequencies [21]. More in detail, the eigendecomposition of the GSO $\boldsymbol{S}$ is written as $\boldsymbol{S} = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^{\mathsf{H}}$, where $\boldsymbol{U} = [\boldsymbol{u}_1, \ldots, \boldsymbol{u}_N] \in \mathbb{R}^{N \times N}$ contains the eigenvectors of $\boldsymbol{S}$ and $\boldsymbol{\Lambda} = \text{diag}(\lambda_1, \ldots, \lambda_N) \in \mathbb{R}^{N \times N}$ is a diagonal matrix that contains the i-th eigenvalue on the i-th diagonal entry. The eigenvalues can be ordered as $\lambda_1 <= \ldots <= \lambda_N$ without loss of generality. One attractive property of this decomposition is that, when $\boldsymbol{S} = \boldsymbol{L}$, the eigenvectors associated with smaller eigenvalues are smoother over the graph compared to eigenvectors associated with higher eigenvalues [22]. That is, the eigenvectors and eigenvalues obtained from the eigendecomposition of the Laplacian carry information about the graph frequencies.

The GFT of a graph signal $\boldsymbol{x}$ is then defined as

$$\hat{\boldsymbol{x}} = \boldsymbol{U}^{\mathsf{H}}\boldsymbol{x} \tag{2.1}$$

and provides an expansion of $\boldsymbol{x}$ in terms of the eigenvectors $\boldsymbol{u}_1, \ldots, \boldsymbol{u}_N$. The i-th entry of $\hat{\boldsymbol{x}}$ indicates the weight of eigenvector $\boldsymbol{u}_i$ in such expansion. The inverse Graph Fourier Transform (IGFT) is defined as

$$\boldsymbol{x} = \boldsymbol{U}\hat{\boldsymbol{x}}\,. \tag{2.2}$$

The GFT is useful because it allows analyzing the signal from a graph spectral perspective. This spectral equivalent will be critical to assess the operation of convolutions on the graph and subsequently to process times series on graphs.

2.1.3. GRAPH SIGNAL SHIFTING

Given a graph signal $\boldsymbol{x}^{(0)} = \boldsymbol{x}$, its graph-shifted version is

$$\boldsymbol{x}^{(1)} = \boldsymbol{S}\boldsymbol{x}^{(0)}. \tag{2.3}$$

By expanding (2.3) for the i-th component of $\boldsymbol{x}^{(1)}$ as

$$x_i^{(1)} = S_{i1}x_1^{(0)} + S_{i2}x_2^{(0)} + \ldots + S_{iN}x_N^{(0)}\,, \tag{2.4}$$

we can see the application of the GSO is a local operation that replaces the value at node i with a linear combination of the values at the neighbours of i [17]. If the shift operator is the adjacency matrix $\boldsymbol{A}$, i.e., $S_{ij} = A_{ij}$, the value $x_i^{(1)}$ is a weighted sum of the values measured at the neighbouring nodes of node i, weighted by the strength of the edges. For a directed graph, the nodes contributing to $x_i^{(1)}$ are those with an edge ending at node i, see Figure 2.1b. We can obtain a K-shifted version of $\boldsymbol{x}$ over the graph through successive applications of the GSO to signal $\boldsymbol{x}$:

$$\boldsymbol{x}^{(K)} = \boldsymbol{S}^K\boldsymbol{x}^{(0)} = \boldsymbol{S}(\boldsymbol{S}^{K-1}\boldsymbol{x}^{(0)}) = \boldsymbol{S}(\boldsymbol{x}^{(K-1)})\,. \tag{2.5}$$

For the example in Figure 2.3, the value at node four after two successive applications of the GSO is $x_4^{(2)} = A_{41}x_1^{(1)} + A_{45}x_5^{(1)} = 1 + 1$.

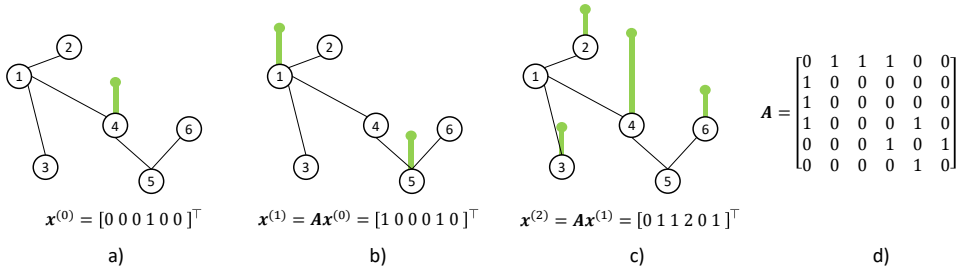


Figure 2.3: Applications of two successive shifts to a graph signal $\boldsymbol{x}$ over an *undirected* graph. The GSO is the adjacency matrix $\boldsymbol{A}$. In each shift, neighbours exchange values between them. **a)** The initial graph signal is zero everywhere except on node four. **b)** One shift of the signal $\boldsymbol{x}^{(0)}$: node four contributes to the new values on node one and five. **c)** A second shift to the signal: the nodes exchange their values again. Node four reaches the highest value, due to nodes one and five, both contributing to the new value on node four. **d)** The adjacency matrix of the graph.

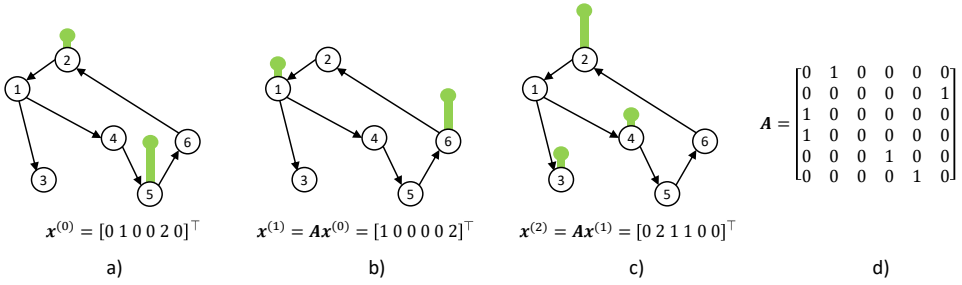


Figure 2.4: Applications of two successive shifts to a graph signal $\boldsymbol{x}$ over a *directed* graph. The GSO is the adjacency matrix $\boldsymbol{A}$. **a)** Initial signal. **b)** One shift is performed. **c)** A second shift is performed. **d)** The adjacency matrix of the graph. Recall the notation adopted; if the entry A_{ij} is non-zero it means there is an edge going from node j to node i.

Shifting the signal once over the graph amounts to a complexity of $\mathcal{O}(|\mathcal{E}|)$. This is because the GSO is a sparse matrix containing non-zero elements only in positions related to the edges. Thus, the graph signal shifting operation is computationally less expensive than a matrix-vector product of dense matrices [14]. The cost of computing the K-shifted version of $\boldsymbol{x}$ over the graph as per (2.5) is of order $\mathcal{O}(K|\mathcal{E}|)$, as it requires computing K times the operation in (2.3). Figures 2.3 and 2.4 illustrate successive applications of the GSO to a graph signal $\boldsymbol{x}$ over undirected and directed graphs, respectively.

To better understand how the application of the shift operator $\boldsymbol{S}$ is a natural extension to graphs of the standard time shifts for signals, it is useful to think about a directed cyclic graph $\mathcal{G}_c$ as support of a discrete periodic time signal $x[t]$ [21]. The adjacency matrix $\boldsymbol{C}_T$ of the directed cyclic graph has non-zero entries $[\boldsymbol{C}_T]_{i+1 \bmod T,i} = 1$, for $i = 0,\ldots,T-1$. For example, if $T = 4$, this formulation of $\boldsymbol{C}_T$ results in

$$\boldsymbol{C}_4 = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

Graph $\mathcal{G}_c$ has as many nodes as the values of the discrete time signal, see Figure 2.5. If

we consider $x[t]$ as a graph signal $\boldsymbol{x} = [x[1], ..., x[T]]^\top$ defined on top of graph $\mathcal{G}_c$, then applying the shift operator $\boldsymbol{S}_c = \boldsymbol{C}_T$ is equivalent to a shift in time of $x[t]$, i.e., the shifted graph signal $\boldsymbol{x}^{(1)} = \boldsymbol{S}_c\boldsymbol{x}$ is a circularly time-shifted version of the initial time signal $\boldsymbol{x}$: $\boldsymbol{x}^{(1)} = [\boldsymbol{x}_T, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_{T-1}]^\top$. As illustrated in Figure 2.5, the shift operation results in each node obtaining the value from its previous node.

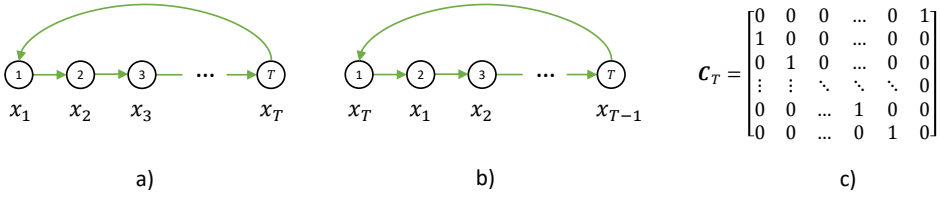


Figure 2.5: The directed cyclic graph $\mathcal{G}_c$ with T nodes can be thought of as the support of a time discrete signal having T values. **a)** Before shift. The graph signal shown is $\boldsymbol{x}$. **b)** After shift. The graph signal shown is $\boldsymbol{x}^{(1)} = \boldsymbol{S}_c\boldsymbol{x} = \boldsymbol{C}_T\boldsymbol{x}$. **c)** The adjacency matrix of the graph.

2.1.4. GRAPH CONVOLUTIONAL FILTERS

Let $x[t]$ be a discrete time signal and $h[k]$ a finite impulse response (FIR) filter of order K. Recall, the output $y[t]$ of a discrete convolution between $x[t]$ and the filter $h(\cdot)$ is

$$y[t] = (x * h)[t] = \sum_{k=0}^{K} h[k]x[t-k]\,, \tag{2.6}$$

where "$*$" indicates time convolution. The output $y[t]$ is the convolution of the filter $h(\cdot)$ with the input $x[t]$ and is computed as a weighted sum of *time-shifted* versions of the discrete input signal $x[t]$. Analogously, a *graph convolution* can be defined as a weighted sum of *graph-shifted* versions of the input graph signal.

We denote the input graph signal as $\boldsymbol{x}$, the GSO as $\boldsymbol{S}$, and as $\boldsymbol{h} = [h_0, h_1, \ldots, h_K]^\top$ a set of $K+1$ parameters. The output $\boldsymbol{y}$ of a graph convolution is defined as

$$\boldsymbol{y} = \boldsymbol{h} *_s \boldsymbol{x} = \sum_{k=0}^{K} h_k \boldsymbol{S}^k \boldsymbol{x} = \boldsymbol{H}(\boldsymbol{S})\boldsymbol{x}\,, \tag{2.7}$$

where "$*_s$" indicates that this convolution involves the GSO of the graph. That is, the GSO $\boldsymbol{S}$ performs signal shifting over the graph, and the output $\mathbf{y}$ is a weighted sum of shifted signals. Matrix $\boldsymbol{H}(\boldsymbol{S}) \in \mathbb{R}^{N\times N}$ is called a *graph filter* of order K and has the form:

$$\boldsymbol{H}(\boldsymbol{S}) = \sum_{k=0}^{K} h_k \boldsymbol{S}^k. \tag{2.8}$$

By comparing (2.6) with (2.7), we can see how the time shift in (2.6) is linked to the graph shift in (2.7), which is obtained through the application of the GSO. Therefore, the graph filter $\boldsymbol{H}(\boldsymbol{S})$ uses the signal shift [cf. (2.3)] to perform convolutions through K successive applications of the graph shift operator. When $k = 0$, $\boldsymbol{S}^0\boldsymbol{x} = \boldsymbol{I}\boldsymbol{x}$ is the original

input graph signal, $\boldsymbol{x}$. For $k > 0$, $\boldsymbol{S}^k\boldsymbol{x}$ represents the k-shifted version of $\boldsymbol{x}$ over the graph and gathers at each node information from its k-hop neighbourhood. Therefore, a graph filter with K filter taps aggregates at each node information from its K-hop neighbourhood. The complexity of (2.7) is $\mathcal{O}(K|\mathcal{E}|)$, since it requires the computation of K graph shifts as in (2.5). The graph convolution operation is illustrated in Figure 2.6.

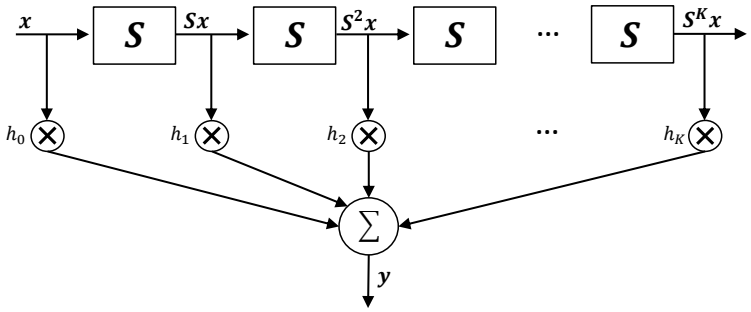


Figure 2.6: Illustration of the filtering process for graph signals. Each graph-shifted version $\boldsymbol{S}^k\boldsymbol{x}$ of the input signal $\boldsymbol{x}$ is weighted by coefficient h_k. The output $\boldsymbol{y}$ is the weighted sum of the graph-shifted versions of $\boldsymbol{x}$.

The graph filter formulation is given here from a vertex perspective, since we define graph convolutions as shift-and-sum operations using the GSO. However, graph filtering can also be defined as a pointwise multiplication in the spectral domain [23, 21]. That is, given the GFT $\hat{\boldsymbol{x}}$ of a graph signal $\boldsymbol{x}$, the output $\hat{\boldsymbol{y}} = [\hat{y}(\lambda_0), \hat{y}(\lambda_1), \ldots, \hat{y}(\lambda_{N-1})]^\top$ of the filtering operation is

$$\hat{\boldsymbol{y}} = h(\boldsymbol{\Lambda})\hat{\boldsymbol{x}}\,, \tag{2.9}$$

where $h(\boldsymbol{\Lambda}) = \text{diag}(h(\lambda_0), h(\lambda_1), \ldots, h(\lambda_{N-1}))$ is the graph filter frequency response. In other words, each graph frequency content $\hat{x}(\lambda_i)$ is multiplied by the corresponding filter coefficient $h(\lambda_i)$. Starting from a graph signal $\boldsymbol{x}$, the filtered (from a spectral perspective) output graph signal is obtained as:

$$\boldsymbol{y} = \boldsymbol{U}h(\boldsymbol{\Lambda})\boldsymbol{U}^\top\boldsymbol{x} = \boldsymbol{U}h(\boldsymbol{\Lambda})\hat{\boldsymbol{x}} = \boldsymbol{U}\hat{\boldsymbol{y}}\,. \tag{2.10}$$

2.2. GRAPH CONVOLUTIONAL NEURAL NETWORKS

Consider a *dataset* of pairs $(\boldsymbol{x}_i, \boldsymbol{y}_i)$, where $\boldsymbol{x}_i$ represents the input data, and $\boldsymbol{y}_i$ is the associated output. Depending on the type of application, these $(\boldsymbol{x}, \boldsymbol{y})$ pairs can arbitrarily be scalars or vectors. For example, in a dataset representing temperatures across N sensors, the input data $\boldsymbol{x}_i$ may consist of N values, each one associated to a sensor, while the output y_i may be a scalar representing the season during which such temperatures were recorded.

A Graph Convolutional Neural Network (GCNN) is a neural network which makes a prediction $\hat{\boldsymbol{y}}_i$ to match the associated output $\boldsymbol{y}_i$, given the input $\boldsymbol{x}_i$ and the graph structure (expressed by the GSO $\boldsymbol{S}$). The GCNN is composed of L layers, each consisting of a graph convolutional module, a graph pooling, and a pointwise non-linearity [17], [6].

The output of the GCNN is obtained from the last layer L. Depending on the application, a number of fully connected layers can be added after the L-th layer to form the final output $\hat{\boldsymbol{y}}$. Next, we detail the building blocks of GCNNs.

2

2.2.1. GRAPH CONVOLUTIONAL LAYER

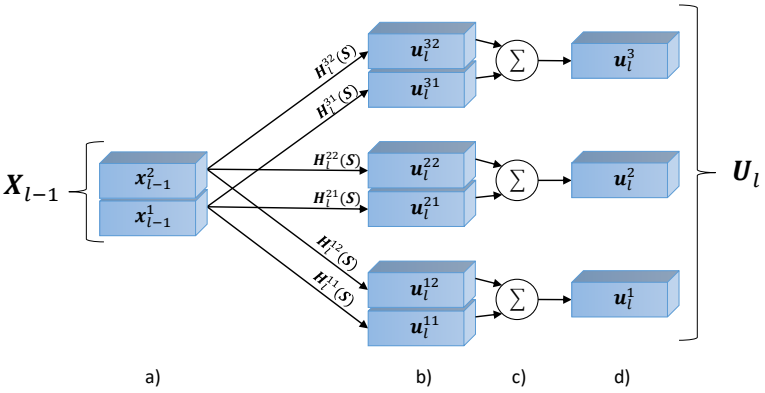


Figure 2.7: Convolutional features at layer l. **a)** The input $\boldsymbol{X}_{l-1} = [\boldsymbol{x}^1_{l-1}, \boldsymbol{x}^2_{l-1}]$ is composed of $F_{l-1} = 2$ features. **b)** The intermediate features $\boldsymbol{u}^{fg}_l$ in (2.11) are computed using the filters $\boldsymbol{H}^{fg}_l(\boldsymbol{S})$. For each of the $F_l = 3$ output features $\boldsymbol{u}^f_l$ we have $F_{l-1} = 2$ filters $\boldsymbol{H}^{fg}_l(\boldsymbol{S})$ involved. **c)** The intermediate features $\boldsymbol{u}^{fg}_l$ are summed as in (2.11). **d)** The F_l convolutional features $\boldsymbol{u}^f_l$ are obtained and form the output $\boldsymbol{U}_l = [\boldsymbol{u}^1_l, \boldsymbol{u}^2_l, \boldsymbol{u}^3_l]$.

The graph convolutional layer is the core of the GCNN. It performs graph convolutions as in (2.7), generating a pre-defined number of convolutional features. The input to layer l is a set of F_{l-1} features $\boldsymbol{X}_{l-1} = [\boldsymbol{x}^1_{l-1}, \boldsymbol{x}^2_{l-1}, \ldots, \boldsymbol{x}^{F_{l-1}}_{l-1}] \in \mathbb{R}^{N_{l-1} \times F_{l-1}}$ whose columns represent features (each of dimensionality N_{l-1}). At the input layer, the input $\boldsymbol{X}_0 = \boldsymbol{x}_0 \in \mathbb{R}^{|\mathcal{V}|}$ is a graph signal of dimensionality equal to the number of nodes. However, as we will see in Section 2.2.2, the dimensionality of these features is reduced at each layer because of pooling, i.e., $N_l < N_{l-1}$.

The output of layer l consists of a set of F_l *convolutional features* $\boldsymbol{U}_l = [\boldsymbol{u}^1_l, \boldsymbol{u}^2_l, \ldots, \boldsymbol{u}^{F_l}_l] \in \mathbb{R}^{N_{l-1} \times F_l}$ (each of dimensionality N_{l-1}), obtained as

$$\boldsymbol{u}^f_l = \sum_{g=1}^{F_{l-1}} \boldsymbol{u}^{fg}_l = \sum_{g=1}^{F_{l-1}} \boldsymbol{H}^{fg}_l(\boldsymbol{S})\, \boldsymbol{x}^g_{l-1} = \sum_{g=1}^{F_{l-1}} \sum_{k=0}^{K} h^{fg}_k \boldsymbol{S}^k\, \boldsymbol{x}^g_{l-1} \quad \text{for } f = 1, \ldots, F_l\,, \tag{2.11}$$

where $\boldsymbol{u}^{fg}_l \in \mathbb{R}^{N_{l-1}}$ are referred to as *intermediate features* and $\boldsymbol{H}^{fg}_l(\boldsymbol{S})$ is a K-th order linear shift invariant filter [cf. (2.8)]. Filter $\boldsymbol{H}^{fg}_l(\boldsymbol{S})$ is used to process the g-th input feature $\boldsymbol{x}^g_{l-1}$ when computing the f-th output feature $\boldsymbol{u}^f_l$. Figure 2.7 illustrates how the convolutional features $\boldsymbol{U}_l$ are computed from the input $\boldsymbol{X}_{l-1}$.

The input signals $\boldsymbol{x}_{l-1}^{g}$ in (2.11) can have dimensionality lower than N due to pooling. Therefore, it is impossible to compute the matrix-vector product $\boldsymbol{S}^{k}\boldsymbol{x}_{l-1}^{g}$ of (2.11), since the GSO $\boldsymbol{S}$ is of dimension $N \times N$. The solution to this dimensionality mismatch is achieved by padding the input with zero entries to bring the dimensionality of $\boldsymbol{x}_{l-1}^{g}$ to N, and then compute (2.11) with the GSO $\boldsymbol{S}$ without requiring any graph coarsening algorithm [17].

The number of learnable parameters to obtain $\boldsymbol{U}_l$ is $(K+1)F_{l-1}F_l$: for each of the F_l output features of layer l, there are F_{l-1} filters $\boldsymbol{H}^{fg}(\boldsymbol{S})$, each consisting of $K+1$ coefficients h_k^{fg}. The computational cost of computing the output $\boldsymbol{U}_l$ is of order $\mathcal{O}(|\mathcal{E}|(K+1)F_{l-1}F_l)$, since the cost of a single convolutional filtering operation is $\mathcal{O}(|\mathcal{E}|(K+1))$ and in (2.11) we repeat this operation $F_{l-1}F_l$ times. Notice we considered the filter order K fixed at each layer. However, in practice, the order may also be layer dependent; i.e., K_l.

2.2.2. GRAPH POOLING

Pooling reduces the dimensionality of the convolutional features $\boldsymbol{U}_l = [\boldsymbol{u}_l^1, \boldsymbol{u}_l^2, \ldots, \boldsymbol{u}_l^{F_l}]$ to reduce the overall number of parameters. In the context of this thesis, pooling is a *non-learnable* operation which summarizes the convolutional features $\boldsymbol{U}_l$ at layer l, reducing the dimensionality from $N_{l-1} \times F_l$ to $N_l \times F_l$, with $N_l < N_{l-1}$. In other words, each of the F_l features $\boldsymbol{u}_l^f$ is reduced from dimensionality N_{l-1} to N_l. To achieve this dimensionality reduction, there are two steps involved: *summarization* and *downsampling*. In the summarization step, illustrated in Figure 2.8 b), the value in each node is changed accordingly to a summarization criterion (such as *max pooling*), by considering its neighbouring nodes, up to a predefined α-hop neighbourhood. In the downsampling step, illustrated in Figure 2.8 c), we select only N_l *summarized* values per feature out of N_{l-1}.

The summarization step is implemented at layer l by applying the pooling operator $\rho(\cdot)$ on the convolutional features $\boldsymbol{u}_l^f$. The summarized features $\boldsymbol{V}_l = [\boldsymbol{v}_l^1, \boldsymbol{v}_l^2, \ldots, \boldsymbol{v}_l^{F_l}] \in \mathbb{R}^{N_{l-1} \times F_l}$ are defined as

$$\boldsymbol{v}_l^f = \rho(\boldsymbol{u}_l^f \,;\, \alpha, \mathcal{G}) \quad \text{for } f = 1, \ldots, F_l \,, \tag{2.12}$$

where $\rho(\cdot\,;\,\alpha,\mathcal{G})$ operates on the values reached by the α-neighbourhood of each node. That is, the *summarized* value at node i is given by applying $\rho(\cdot)$ to the values corresponding to the set of nodes $\mathcal{N}_i^{\alpha}$.

Downsampling poses a more difficult challenge. When decreasing the dimensionality of the generated features by pooling, the downsampled graph signal will not match the graph support. Adopting a zero-padding pooling implies activating or deactivating specific nodes in the nominal graph to always preserve the same dimensionality [17]. To be precise, denote by $\boldsymbol{C}_l$ the $N_l \times N_{l-1}$ sampling matrix at layer l and consider the reduced features $\boldsymbol{B}_l = [\boldsymbol{b}_l^1, \boldsymbol{b}_l^2, \ldots, \boldsymbol{b}_l^{F_l}] \in \mathbb{R}^{N_l \times F_l}$ computed as

$$\boldsymbol{b}_l^f = \boldsymbol{C}_l \boldsymbol{v}_l^f \quad \text{for } f = 1, \ldots, F_l \,. \tag{2.13}$$

Following [17], there are different sampling criteria for the active nodes. In this thesis, we consider matrix $\boldsymbol{C}_l$ is built to select the N_l nodes with highest degree from the available N_{l-1}.

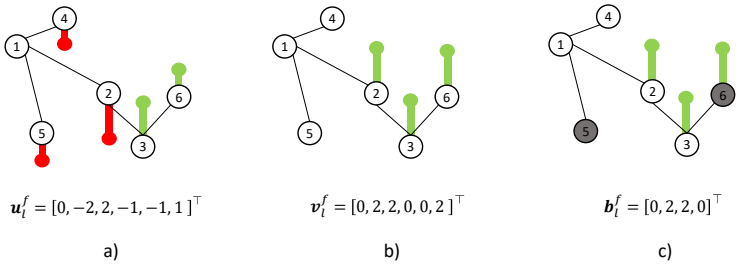


Figure 2.8: Max pooling at layer l (shown for one convolutional feature $\boldsymbol{u}_l^f$). In this example, we have $N_{l-1} = 6$, $N_l = 4$, and $\alpha = 1$, i.e., the pooling operation is performed over the one-hop neighbourhood of each node and only four out of six nodes are kept. **a)** Initial values of the *convolutional feature* $\boldsymbol{u}_l^f$ before pooling. **b)** Values of the *summarized feature* $\boldsymbol{v}_l^f$ after the summarization step. **c)** Values of the *reduced feature* $\boldsymbol{b}_l^f$ after the downsampling step.

The reduced features are fed to a pointwise non-linearity (activation function), such as ReLU, to obtain the output of layer l. Therefore, the output $\boldsymbol{X}_l = [\boldsymbol{x}_l^1, \boldsymbol{x}_l^2, \ldots, \boldsymbol{x}_l^{F_l}] \in \mathbb{R}^{N_l \times F_l}$ of layer l is

$$\boldsymbol{x}_l^f = \sigma(\boldsymbol{b}_l^f) \quad \text{for } f = 1, \ldots, F_l \,. \tag{2.14}$$

The output $\boldsymbol{X}_l$ is sent as input to layer $(l+1)$ and operations (2.11)-(2.14) are repeated for all layers $l = 1, \ldots, L$. If the network is composed of L layers, the output of the GCNN is the output $\boldsymbol{X}_L$ of the last layer:

$$\hat{\boldsymbol{y}} = \boldsymbol{X}_L \,. \tag{2.15}$$

Optionally, the output $\boldsymbol{X}_L$ of the last layer can be sent to a fully connected layer to obtain the final output $\hat{\boldsymbol{y}}$ of the GCNN as

$$\hat{\boldsymbol{y}} = \sigma(\boldsymbol{W}_{\text{FC}} \boldsymbol{X}_L) \,, \tag{2.16}$$

where $\boldsymbol{W}_{\text{FC}}$ is the $B \times N_L F_L$ weight matrix of this fully connected layer and B is the dimensionality of the output $\hat{\boldsymbol{y}}$.

Without considering the fully connected layer, the number of parameters at each layer of the GCNN is $(K_l + 1)F_{l-1}F_l$, as we discussed in Section 2.2.1. Thus, this number depends only on the length of the graph convolutional filters and the number of features at each layer, but not on the size of the graph N. The complexity of the GCNN is linear in the number of edges at each layer. Let us define by M_l the maximum number of edges at each layer, then the complexity is of order $\mathcal{O}(M_l(K_l+1)F_{l-1}F_l)$ at each layer.

2.2.3. Loss

Given the parametric nature of the network, we can represent the output as $\hat{\boldsymbol{y}} = f(\boldsymbol{x}; \boldsymbol{\theta})$ where $f(\cdot; \boldsymbol{\theta})$ represents the GCNN and $\boldsymbol{\theta}$ is a vector comprising all parameters: the $(K+1)F_{l-1}F_l$ weights at each layer l and the optional weights of the fully connected layer.

The network is trained to minimize a loss function that quantifies how far the output $\hat{\boldsymbol{y}}$ of the model is from the ground truth $\boldsymbol{y}$. That is, given a training set $\mathcal{T}$ of R samples, a model $f(\cdot\,;\boldsymbol{\theta})$, and a loss function $\mathscr{L}(\boldsymbol{y},\hat{\boldsymbol{y}})$, the objective of the training phase is to find the parameters which minimize the loss over $\mathcal{T}$ as

$$\arg\min_{\boldsymbol{\theta}} \mathscr{L}_{\mathcal{T}}(\boldsymbol{y},\hat{\boldsymbol{y}}) = \arg\min_{\boldsymbol{\theta}} \mathscr{L}_{\mathcal{T}}(\boldsymbol{y}, f(\boldsymbol{x}\,;\boldsymbol{\theta}))\,. \tag{2.17}$$

This training is achieved through backpropagation, since all operations are differentiable [24]. Backpropagation uses (stochastic) gradient descent (SGD) to learn the parameters of the network in a supervised learning setting. On a high level, backpropagation computes the gradient of the loss with respect to the weights $\boldsymbol{\theta}$. Then, it uses this gradient to update the weights and repeats the process for a fixed number of iterations.

The type of loss depends on the downstream task. For *regression*, we consider the mean squared error (MSE):

$$\text{MSE} = \frac{1}{R}\sum_{i\in\mathcal{T}} \left\|\hat{\boldsymbol{y}}_i - \boldsymbol{y}_i\right\|_2^2\,. \tag{2.18}$$

For *classification*, we adopt the cross-entropy loss (CE):

$$\text{CE} = \frac{1}{R}\sum_{i\in\mathcal{T}}\left(-\sum_{c=1}^{C} y_{ic}\,log(p_{ic})\right), \tag{2.19}$$

where y_{ic} is a binary value indicating whether the input $\boldsymbol{x}_i$ is assigned to the correct class c, and p_{ic} is the predicted probability of $\boldsymbol{x}_i$ being from class c (C classes in total).

2.3. GRAPH-TIME SIGNAL PROCESSING

In this section, we deal with graph signals that evolve over time. In Section 2.3.1, we present the concept of product graph, and in Section 2.3.2, we show how to use product graphs to model time-varying graph signals.

2.3.1. PRODUCT GRAPHS

A *product graph* $\mathcal{G}_\square = (\mathcal{V}_\square, \mathcal{E}_\square)$ is a graph which can be written as the product "$\square$" of smaller *factor graphs* $\mathcal{G}_0 = (\mathcal{V}_0, \mathcal{E}_0)$ and $\mathcal{G}_1 = (\mathcal{V}_1, \mathcal{E}_1)$ with N_0 and N_1 nodes, respectively. The product graph $\mathcal{G}_\square$ is written as

$$\mathcal{G}_\square = \mathcal{G}_0 \;\square\; \mathcal{G}_1\,, \tag{2.20}$$

where the vertex set $\mathcal{V}_\square$ is the Cartesian product of the vertex sets of the two graphs, i.e., $\mathcal{V}_\square = \mathcal{V}_0 \times \mathcal{V}_1$ consisting of $N_0 N_1$ nodes. The edge set of the product graph $\mathcal{E}_\square$ depends on the type of product operation. We indicate with $|\mathcal{E}_\square|$ the cardinality of the edge set $\mathcal{E}_\square$, i.e., the number of edges in the product graph. Next, we detail the most frequently used product graphs.

TYPES OF PRODUCT GRAPHS

There exist three main product graphs in the literature: the *Kronecker* product graph, the *Cartesian* product graph, and the *strong* product graph [25][14]. These product graphs

are shown in Figure 2.9. Starting from the GSOs of the factor graphs, we can derive the GSO $\boldsymbol{S}_{\square}$ of the product graph.

The *Kronecker* product graph $\mathcal{G}_{\otimes} = (\mathcal{V}_{\otimes}, \mathcal{E}_{\otimes})$ has the shift operator $\boldsymbol{S}_{\otimes}$:

$$\boldsymbol{S}_{\otimes} = \boldsymbol{S}_0 \otimes \boldsymbol{S}_1 \,, \tag{2.21}$$

where $\otimes$ indicates the Kronecker product [14], for which we recall in Appendix A. The number of edges of the Kronecker product graph $\mathcal{G}_{\otimes}$ is $|\mathcal{E}_{\otimes}| = |\mathcal{E}_0||\mathcal{E}_1|$. The *Cartesian* product graph $\mathcal{G}_{\times} = (\mathcal{V}_{\times}, \mathcal{E}_{\times})$ has the shift operator $\boldsymbol{S}_{\times}$:

$$\boldsymbol{S}_{\times} = \boldsymbol{S}_0 \otimes \boldsymbol{I}_{N_1} + \boldsymbol{I}_{N_0} \otimes \boldsymbol{S}_1 \,, \tag{2.22}$$

where $\boldsymbol{I}_N$ is the $N \times N$ identity matrix. The number of edges of the Cartesian product graph $\mathcal{G}_{\times}$ is $|\mathcal{E}_{\times}| = N_0|\mathcal{E}_1| + N_1|\mathcal{E}_0|$. The *strong* product graph $\mathcal{G}_{\boxtimes} = (\mathcal{V}_{\boxtimes}, \mathcal{E}_{\boxtimes})$ combines both the Cartesian and the Kronecker product graphs and has the shift operator $\boldsymbol{S}_{\boxtimes}$:

$$\boldsymbol{S}_{\boxtimes} = \boldsymbol{S}_{\otimes} + \boldsymbol{S}_{\times} = \boldsymbol{S}_0 \otimes \boldsymbol{S}_1 + \boldsymbol{S}_0 \otimes \boldsymbol{I}_{N_1} + \boldsymbol{I}_{N_0} \otimes \boldsymbol{S}_1 . \tag{2.23}$$

The number of edges of the strong product graph $\mathcal{G}_{\boxtimes}$ is $|\mathcal{E}_{\boxtimes}| = |\mathcal{E}_{\otimes}| + |\mathcal{E}_{\times}| = |\mathcal{E}_0||\mathcal{E}_1| + N_0|\mathcal{E}_1| + N_1|\mathcal{E}_0|$.

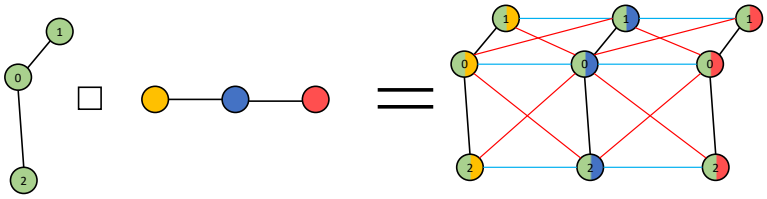


Figure 2.9: Kronecker, Cartesian, and Strong product graph. On the left, the two factor graphs. On the right, the resulting product graph. Node colours, together with the indices, indicate which two nodes in the factor graphs contribute to a specific node in the product graph. The *Kronecker* product graph leads to the edges colored in red, the Cartesian product graph leads to the edges colored in blue and black, while the strong product graph considers both sets of edges.

PARAMETRIC PRODUCT GRAPH

It is possible to represent all product graphs in a more compact form. Let $\boldsymbol{S}_{\diamond}$ be the GSO of the *parametric product graph* defined as

$$\boldsymbol{S}_{\diamond} = \sum_{i=0}^{1} \sum_{j=0}^{1} s_{ij} (\boldsymbol{S}_0^i \otimes \boldsymbol{S}_1^j) \,, \tag{2.24}$$

where $\boldsymbol{S}_0$ and $\boldsymbol{S}_1$ are GSOs of the two factor graphs $\mathcal{G}_0$ and $\mathcal{G}_1$, respectively, and s_{ij} are scalars. For specific parameters s_{ij}, (2.24) captures all the product graphs previously introduced, i.e., the Kronecker, the Cartesian, and the strong product [15]. If we allow the parameters s_{ij} to take on continuous values, we can also "weigh" the different building blocks in (2.24). To clarify, by expanding (2.24) as

$$\boldsymbol{S}_{\diamond} = s_{00}(\boldsymbol{I}_0 \otimes \boldsymbol{I}_1) + s_{01}(\boldsymbol{I}_0 \otimes \boldsymbol{S}_1) + s_{10}(\boldsymbol{S}_0 \otimes \boldsymbol{I}_1) + s_{11}(\boldsymbol{S}_0 \otimes \boldsymbol{S}_1) \,, \tag{2.25}$$

we see how it encompasses equations (2.21), (2.22), and (2.23). If $s_{00} = 0$ and $s_{ij} = 1$ for all remaining indices, we obtain the strong product graph, while if $s_{01} = s_{10} = 1$ and $s_{00} = s_{11} = 0$, we obtain the Cartesian product graph. We also see that (2.25) contains an additional term, $\boldsymbol{I}_0 \otimes \boldsymbol{I}_1$, which adds potential self-loops to all $N_0 N_1$ nodes of the product graph. Therefore, when all s_{ij} parameters are non-zero, the number of edges in $\mathcal{G}_\diamond$ is $|\mathcal{E}_\diamond| = |\mathcal{E}_\boxtimes| + N_0 N_1$, i.e, the number of edges in a strong product graph plus the $N_0 N_1$ self-loops.

From the next chapter onwards, we will leverage the flexibility of the parametric product graph to learn from the data the type of product.

2.3.2. TIME-VARYING GRAPH SIGNALS

So far, we considered the graph signal $\boldsymbol{x}$ to be time-invariant and saw how the GCNN can process it (Section 2.2). Here, we consider graph signals that evolve over time. We denote by $\boldsymbol{x}_t \in \mathbb{R}^N$ the time-varying graph signal over graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ at time t. Considering T time instances, we can describe the time-varying graph signal with matrix $\boldsymbol{X} = [\boldsymbol{x}_1 \;\; \boldsymbol{x}_2 \;\; \dots \;\; \boldsymbol{x}_T] \in \mathbb{R}^{N \times T}$. The i-th row of $\boldsymbol{X}$, $\boldsymbol{X}_{i:}$, contains the T values observed at node i, while the j-th column of $\boldsymbol{X}$, $\boldsymbol{X}_{:j}$, contains the graph signal $\boldsymbol{x}_j$ observed at time j. Figure 2.10 shows an example of a time-varying graph signal. An example of a time-varying graph signal consists of seismic waves over time across N stations (Chapter 5). At each timestep t, the N seismic measurements across the locations constitute a graph signal.

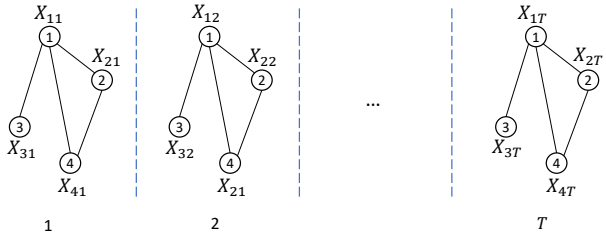


Figure 2.10: A time-varying graph signal over T timesteps on a graph $\mathcal{G}$ of four nodes. $X_{i\tau}$ indicates the value on node i at time τ.

Product graphs represent a structure that can capture the evolution of a time-varying graph signal over $\mathcal{G}$. Specifically, the two factor graphs are the nominal graph $\mathcal{G}$ and a graph $\mathcal{G}_t$, where each of the T nodes represents a time instant, see Figure 2.11. This graph is built in the same way as the directed cyclic graph in Section 2.1.2. We will refer to $\mathcal{G}_t$ as *time graph*.

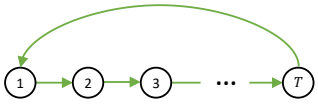


Figure 2.11: Directed cyclic graph representing the evolution of time over T timesteps. Node i represents the i-th time instant.

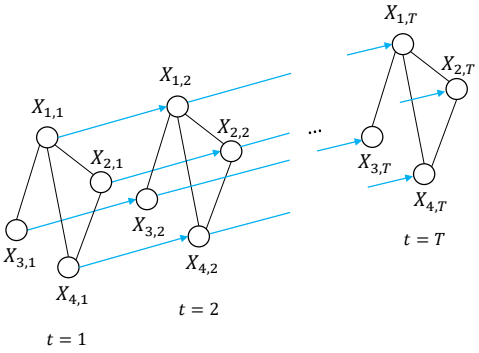


Figure 2.12: Cartesian product graph representation of a time-varying graph signal on four nodes over T timesteps.

As illustrated in Figure 2.12, we can look at the product graph $\mathcal{G}_\square$ as the graph combining T copies of the nominal graph $\mathcal{G}$ on which the time-varying graph signal $\boldsymbol{x}$ evolves over T timesteps. To refer to different nodes at different timesteps, we introduce here the following notation: node (i,t) indicates the node in $\mathcal{G}_\square$ which represents node i of $\mathcal{G}$ at timestep t, i.e., the i-th value of the graph signal $\boldsymbol{x}_t$. Moreover, set $\mathcal{N}_{i,t}^k$ represents the k-hop neighbourhood of node (i,t). For example, in Figure 2.12, value $X_{4,2}$ resides on top of node $(4,2)$.

The type of product graph plays a role in how to model time. For the Cartesian product graph, node (i,t) is connected to the nodes at timestep t which are its neighbours in the nominal graph $\mathcal{G}$, i.e., all nodes in $\mathcal{N}_i^1$ in $\mathcal{G}$. Moreover, node (i,t) is also connected to its previous copy in time in $\mathcal{G}_\square$, denoted as node $(i,t-1)$.

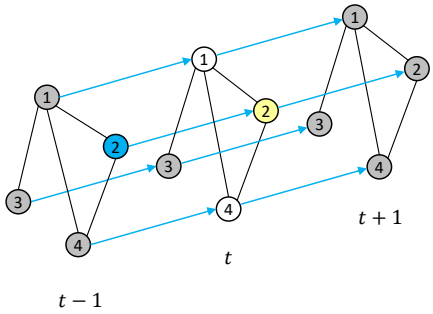


Figure 2.13: The 1-hop neighbourhood of node $(2,t)$ in a Cartesian product graph: $\mathcal{N}_{2,t}^1 = \{(1,t),(4,t),(2,t-1)\}$. We highlight in yellow the node for which we are looking at the neighbourhood, in blue the nodes included in the one-hop neighbourhood of the yellow node due to edges representing time, and in white the nodes included in the one-hop neighbourhood of the yellow nodes due to edges existing in the nominal graph $\mathcal{G}$. All remaining nodes are shown in grey.

If we build the product graph $\mathcal{G}_\square$ with the strong product, then we would increase the neighbourhood of each node. Node (i,t) would be connected to its neighbours in $\mathcal{G}$ both considering the copy at time t and the copy at time $t-1$, as shown in Figure 2.14.

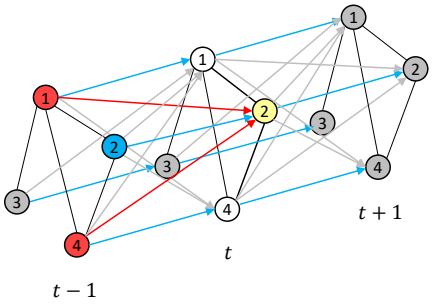


Figure 2.14: Reach of the one-hop neighbourhood of node $(2,t)$ in a Strong product graph: $\mathcal{N}^1_{2,t} = \{(1,t),(4,t),(2,t-1),(1,t-1),(4,t-1)\}$. We highlight in yellow the node for which we are looking at the neighbourhood, in blue the nodes included in the one-hop neighbourhood of the yellow node due to edges representing time, and in white the nodes included in the one-hop neighbourhood of the yellow nodes due to edges existing in the nominal graph $\mathcal{G}$. Additionally, we color in red the nodes included in the one-hop neighbourhood of the yellow node due to the Kronecker product graph. For visualization purposes, we show the edges of the Kronecker product graph for node $(2,t)$ in red, while the other Kronecker edges are colored in light grey. All remaining nodes are shown in grey.

We can also construct a product graph in a parametric fashion without constraining the representation to any specific products [cf. (2.24)], as illustrated in Section 2.3.1. That is, we can represent the structure of time-varying graph signal $\boldsymbol{x}_t$ employing the parametric product graph, thus allowing for a more flexible representation when learning the interactions from data. We denote as $\mathcal{G}_\diamond$ the parametric product graph between the directed time graph $\mathcal{G}_t$ and the graph $\mathcal{G}$, similarly to [26]. Specifically, the graph shift operator (GSO) of $\mathcal{G}_\diamond$ is

$$\boldsymbol{S}_\diamond = \sum_{i=0}^{1}\sum_{j=0}^{1} s_{ij}(\boldsymbol{C}_T^i \otimes \boldsymbol{S}^j)\,, \tag{2.26}$$

where $\boldsymbol{C}_T$ is the adjacency matrix of the directed cyclic graph and $\boldsymbol{S}$ is the GSO of the nominal graph $\mathcal{G}$. We can now represent the time-varying graph signal $\boldsymbol{x}_t$ as a (time-invariant) graph signal $\boldsymbol{x}_\diamond \in \mathbb{R}^{NT}$ defined on $\mathcal{G}_\diamond$. The new graph $\mathcal{G}_\diamond$ encodes now both the notions of space and time, providing a structure for all values of the time-varying graph signal. The vertex set $\mathcal{V}_\diamond$ consists of NT nodes, and the edge set $\mathcal{E}_\diamond$ will depend on the weights of the product graph [cf. (2.25)].

2.3.3. FILTERING ON PRODUCT GRAPHS

Consider a parametric product graph $\mathcal{G}_\diamond = (\mathcal{V}_\diamond, \mathcal{E}_\diamond)$ with $|\mathcal{V}_\diamond| = N_\diamond = NT$ nodes and shift operator $\boldsymbol{S}_\diamond$ obtained as in (2.26). Given a graph signal $\boldsymbol{x}_\diamond = [x_1, x_2, \ldots, x_{N_\diamond}] \in \mathbb{R}^{N_\diamond}$ representing a time-varying graph signal on the nominal graph $\mathcal{G}$ over this graph-time structure, a K-th order FIR graph filter on $\mathcal{G}_\diamond$ has the form:

$$\boldsymbol{H}(\boldsymbol{S}_\diamond) = \sum_{k=0}^{K} h_k \boldsymbol{S}_\diamond^k = \sum_{k=0}^{K} h_k \left(\sum_{i=0}^{1}\sum_{j=0}^{1} s_{ij}(\boldsymbol{C}_T^i \otimes \boldsymbol{S}^j)\right)^k\,, \tag{2.27}$$

where h_k denotes the coefficients of the graph filter [cf. (2.8)] and s_{ij} denotes the coefficients of the parametric product graph [cf.(2.25)]. Analogously to (2.7), the output $\boldsymbol{y}_\diamond \in \mathbb{R}^{N_\diamond}$ of the filtering operation is

$$\boldsymbol{y} = \boldsymbol{H}(\boldsymbol{S}_\diamond)\boldsymbol{x}_\diamond \,, \tag{2.28}$$

where the graph filtering operation is no longer performing shifts and sum over the nominal graph, but rather on the parametric product graph $\mathcal{G}_\diamond$. The definition of a parametric graph filter with a parametric shift operator allows for more flexibility than the graph filter counterpart working with the shift operator of a fixed product graph. Furthermore, recall the parametric product graph $\boldsymbol{S}_\diamond$ [cf. (2.24)] captures the Kronecker, the Cartesian, and the strong product graph. Therefore, it generalizes FIR filtering of the latter. Recall, we construct $\mathcal{G}_\diamond$ using a parametric product graph between the directed cyclic graph $\mathcal{G}_T$ (representing the time graph) and the nominal graph $\mathcal{G}$. Therefore, the cost of graph filtering in (2.28) is of order $\mathcal{O}(K|\mathcal{E}_\diamond|)$, where $|\mathcal{E}_\diamond| = NT + N|\mathcal{E}_T| + T|\mathcal{E}_\mathcal{G}| + |\mathcal{E}_T||\mathcal{E}_\mathcal{G}|$.

To provide further intuition on the convolution performed over this product graph, we illustrate the filtering process over the Cartesian and the strong product graph in the following example.

Example 1: Filtering on Graph-Time product graphs

In this example, we show the filtering process for the Cartesian and the strong product graphs. To be more precise, we aim at showing which values are aggregated at a particular node as we perform graph convolutions over the product graph, depending on the type of product graph.

We consider a time-varying graph signal evolving over $T = 3$ timesteps on a graph $\mathcal{G}$ consisting of $N = 3$ nodes (see Figure 2.15). In the following Figures (Figure 2.16 and 2.17) we adopt the following colour scheme. We depict as yellow the node for which we analyze the neighbourhood during filtering, as black the nodes reached through the (black) edges belonging to the nominal graph $\mathcal{G}$, in blue the nodes reached through the (blue) edges produced by the Cartesian product, and in red the nodes reached through the (red) edges produced by the Kronecker product.

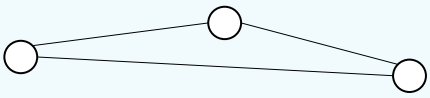


Figure 2.15: Nominal graph $\mathcal{G}$ on top of which the time-varying graph signal evolves over time.

Example 1: Filtering on Graph-Time product graphs

FILTERING ON CARTESIAN PRODUCT GRAPH

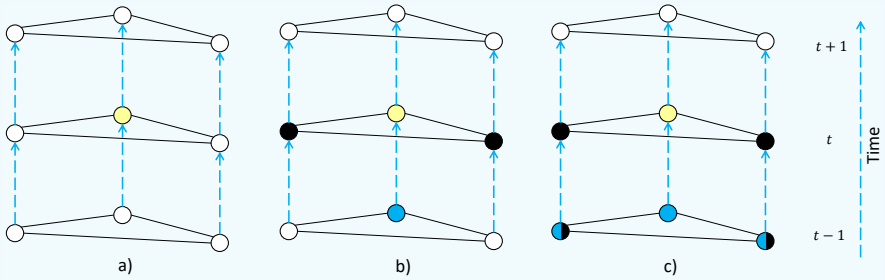


Figure 2.16: Cartesian product graph filtering.

Figure 2.16 a) shows the structure of the Cartesian product graph. Figure 2.16 b) shows that a graph convolution with $K = 1$ [cf. (2.7)] over the Cartesian product graph aggregates values at the yellow node from; (i) its spatial neighbours reached by the black edges, and (ii) its previous (if any) copy in time reached by the blue edges. Figure 2.16 c) shows that at least $K = 2$ is required to take into account also previous values of the spatial neighbours of the yellow node. These nodes are depicted half blue and half black, since they are reached through a path involving both black and blue edges.

FILTERING ON STRONG PRODUCT GRAPH

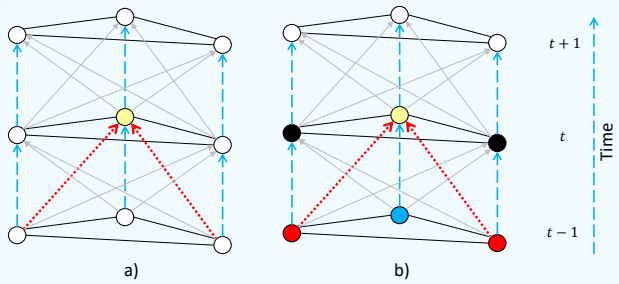


Figure 2.17: Strong product graph filtering.

Figure 2.17 a) shows the strong product graph (which we recall is the combination of both the Cartesian and the Kronecker products). To have a less cluttered figure, we highlight in red the Kronecker edges only for the yellow node, while in light grey the remaining Kronecker edges.
Figure 2.17 b) shows that a graph convolution with $K = 1$ [cf. (2.7)] over the strong product graph aggregates values at the yellow node from; (i) its spatial neighbours reached through the black edges; (ii) its previous (if any) copy in time reached by the blue edges; and (iii) the previous values of its spatial neighbours reached by the red edges.

The two examples of Figures 2.16 and 2.17 showed that performing filtering on this graph-time product graph aggregates information from both graph and time domains. Also, it shows how this filtering is different depending on the product graph adopted. This supports the choice of a parametric product graph to avoid constraining the architecture to a specific type of filtering.

JOINT GRAPH-TIME FOURIER TRANSFORM

We now present the joint graph-time Fourier transform (JFT), an extension of the GFT (Section 2.1.2) to analyze the frequencies of time-varying graph signals both along the time and graph dimensions. Although we present the JFT considering the Cartesian product graph, other product graphs formulations are possible [27].

Recall Section 2.3.2, we can describe a time-varying graph signal of T timesteps with matrix $\boldsymbol{X} = [\boldsymbol{x}_1 \ \boldsymbol{x}_2 \ \dots \ \boldsymbol{x}_T] \in \mathbb{R}^{N\times T}$. Furthermore, let us consider the vectorized form $\boldsymbol{x}_\diamond = \text{vec}(\boldsymbol{X}) \in \mathbb{R}^{NT}$, representing the time-varying graph signal over the graph-time product graph. Therefore, $\boldsymbol{x}_\diamond$ is obtained by stacking the columns of $\boldsymbol{X}$, i.e., the graph signals at each timestep t.

First, let us introduce the Discrete Fourier Transform (DFT) to analyze the frequency content of the rows of $\boldsymbol{X}$ along the temporal axis. Recall each row of $\boldsymbol{X}$ contains the T values observed at a certain node. We can apply the DFT to each row independently as [27, 22]

$$\hat{\boldsymbol{X}}_{\text{DFT}} = \boldsymbol{X}\boldsymbol{F}^* \,, \tag{2.29}$$

where $\boldsymbol{F}^*$ indicates the complex conjugate of the normalized DFT matrix defined as

$$\boldsymbol{F}_{t,k} = \frac{e^{j\omega_k t}}{\sqrt{T}} \,, \quad \text{with} \quad \omega_k = \frac{2\pi(k-1)}{T} \,, \tag{2.30}$$

where $t, k = 1, \dots, T$. Furthermore, we can apply the GFT independently to each column of $\boldsymbol{X}$ to analyze the graph frequency content of the graph signals at each timestep as

$$\hat{\boldsymbol{X}}_{\text{GFT}} = \boldsymbol{U}^{\text{H}}\boldsymbol{X} \,, \tag{2.31}$$

where $\boldsymbol{U}$ is the eigenvector matrix of the GSO of the nominal graph $\mathcal{G}$. Then, we can obtain the JFT $\hat{\boldsymbol{X}}_{\text{JFT}}$ by applying the DFT along the time domain (the rows of $\boldsymbol{X}$) and the GFT along the graph domain (the columns of $\boldsymbol{X}$) [27, 22]:

$$\hat{\boldsymbol{X}}_{\text{JFT}} = \boldsymbol{U}^{\text{H}}\boldsymbol{X}\boldsymbol{F}^* = \hat{\boldsymbol{X}}_{\text{GFT}}\boldsymbol{F}^* = \boldsymbol{U}^{\text{H}}\hat{\boldsymbol{X}}_{\text{DFT}} \,. \tag{2.32}$$

That is, we can obtain the JFT by applying first the GFT and then the DFT, or vice versa. We can also express (2.32) in a vectorized form as

$$\hat{\boldsymbol{x}_\diamond}_{\text{JFT}} = \text{vec}(\hat{\boldsymbol{X}}_{\text{JFT}}) = \text{vec}(\boldsymbol{U}^{\text{H}}\boldsymbol{X}\boldsymbol{F}^*) = (\boldsymbol{F}\otimes\boldsymbol{U})^{\text{H}}\text{vec}(\boldsymbol{X}) = \boldsymbol{U}_\diamond^{\text{H}}\boldsymbol{x}_\diamond \,, \tag{2.33}$$

where $\boldsymbol{U}_\diamond = \boldsymbol{F}\otimes\boldsymbol{U}$ is the $NT \times NT$ matrix obtained through the Kronecker product of the basis [27].

In Section 2.1.4, we introduced the spectral approach to graph filtering of graph signals. We now provide a formulation for graph-time filtering of a time-varying graph signals by leveraging the JFT introduced above. Graph-time filtering of the form of (2.7) over graph-time product graphs can be seen as a pointwise multiplication in the spectral domain that alters the graph-time frequency content of the input. To show this, let us write the output $\boldsymbol{y}_\diamond$ of the filtering process on the product graph as

$$\boldsymbol{y}_\diamond = \boldsymbol{H}(\boldsymbol{S}_\diamond)\boldsymbol{x}_\diamond = \sum_{k=0}^{K} h_k \boldsymbol{S}_\diamond^k \boldsymbol{x}_\diamond \,. \tag{2.34}$$

Then, by performing the eigendecomposition of the GSO $\boldsymbol{S}_\diamond$, we obtain

$$\boldsymbol{y}_\diamond = \sum_{k=0}^{K} h_k(\boldsymbol{U}_\diamond \boldsymbol{\Lambda}_\diamond^k \boldsymbol{U}_\diamond^{\mathsf{H}})\, \boldsymbol{x}_\diamond\,, \tag{2.35}$$

where $\boldsymbol{\Lambda}_\diamond = \boldsymbol{\Lambda}_1 \otimes \boldsymbol{I}_\mathrm{T} + \boldsymbol{\Lambda}_2 \otimes \boldsymbol{I}_\mathrm{N}$ is the $NT \times NT$ diagonal matrix whose diagonal entries contain the eigenvalues of the Cartesian product graph [22], while $\boldsymbol{\Lambda}_1$ and $\boldsymbol{\Lambda}_2$ are the eigenvalues matrices of the nominal and time graphs, respectively.

To obtain the JFT of the output, we then left-multiply both sides of (2.35) by $\boldsymbol{U}_\diamond^{\mathsf{H}}$ as in (2.33), yielding:

$$\hat{\boldsymbol{y}}_{\diamond\mathrm{JFT}} = \sum_{k=0}^{K} h_k \boldsymbol{\Lambda}_\diamond^k\, \boldsymbol{U}_\diamond^{\mathsf{H}}\, \boldsymbol{x}_\diamond = \sum_{k=0}^{K} h_k \boldsymbol{\Lambda}_\diamond^k\, \hat{\boldsymbol{x}}_{\diamond\mathrm{JFT}} = h(\boldsymbol{\Lambda}_\diamond)\, \hat{\boldsymbol{x}}_{\diamond\mathrm{JFT}}\,, \tag{2.36}$$

where $h(\boldsymbol{\Lambda}_\diamond) = \sum_{k=0}^{K} h_k \boldsymbol{\Lambda}_\diamond^k$ is an $NT \times NT$ diagonal matrix called joint time-vertex frequency response [7]. The i-th diagonal entry of $h(\boldsymbol{\Lambda}_\diamond)$, therefore, is a scalar altering the i-th graph-time frequency content of the input as a pointwise multiplication [27]. To summarize, given a time-varying graph signal over T timesteps, we represent it as a graph signal $\boldsymbol{x}_\diamond$ on the product graph $\mathcal{G}_\diamond$. Then, we can filter this graph signal (from a spectral perspective) as

$$\boldsymbol{y}_\diamond = \boldsymbol{U}_\diamond h(\boldsymbol{\Lambda}_\diamond)\boldsymbol{U}_\diamond^{\mathsf{H}}\boldsymbol{x}_\diamond = \boldsymbol{U}_\diamond h(\boldsymbol{\Lambda}_\diamond)\hat{\boldsymbol{x}}_{\diamond\mathrm{JFT}} = \boldsymbol{U}_\diamond \hat{\boldsymbol{y}}_{\diamond\mathrm{JFT}}\,. \tag{2.37}$$

2.4. TIME SERIES MODELING

In this section, we present other models capable of processing and learning of multivariate time series. In Section 2.4.1, we cover methods that process time series only by taking into account the evolution of the signals over time, but ignore a sparse pairwise relationship between data points, i.e., there is no graph involved. In Section 2.4.2, we discuss methods that take into account also the graph describing the relationships among the data points when processing multivariate series.

2.4.1. GRAPH-UNAWARE METHODS

In this section, we show two types of models to learn from multivariate series: a linear vector autoregressive model, and the recurrent neural networks.

VECTOR AUTOREGRESSIVE MOVING-AVERAGE

The Vector Autoregressive Moving-Average (VARMA) model is a linear model for multivariate time series [28]. Suppose we have N variables that vary over time. At timestep t, we have $\boldsymbol{y}_t = [y_{1t}, y_{2t}, \ldots, y_{Nt}]^\top$, consisting of values at timestep t. Then, according to the VARMA(P, Q) model we can write

$$\begin{aligned}\boldsymbol{y_t} &= \boldsymbol{A}_1\boldsymbol{y}_{t-1} + \ldots + \boldsymbol{A}_p\boldsymbol{y}_{t-p} + \boldsymbol{M}_0\boldsymbol{u}_t + \boldsymbol{M}_1\boldsymbol{u}_{t-1} + \ldots + \boldsymbol{M}_q\boldsymbol{u}_{t-q} \\ &= \sum_{p=1}^{P} \boldsymbol{A}_p\boldsymbol{y}_{t-p} + \sum_{q=0}^{Q} \boldsymbol{M}_q\boldsymbol{u}_{t-q}\,,\end{aligned} \tag{2.38}$$

where $\boldsymbol{u}_t$ is white noise with zero mean and covariance matrix $\boldsymbol{\Sigma}_u$. Matrices $\boldsymbol{A}_p$ ($p \in \{1,\dots,P\}$) are $N \times N$ and model the correlation between the N observations at time t and the past observations up to $t-p$. Matrices $\boldsymbol{M}_q$ ($q \in \{0,\dots,Q\}$) are also $N \times N$ and model the correlation between the N observations at time t and the past noise up to $t-q$. Matrices $\boldsymbol{A}_p$ constitute the autoregressive (VAR) part of the model, while matrices $\boldsymbol{M}_q$ constitute the moving-average (MA) part. The parameters of matrices $\boldsymbol{A}_p$ and $\boldsymbol{M}_q$ can be estimated through a maximum likelihood approach [29].

VARMA models have a long history of modeling multivariate time series and we will use them in Section 2.4.2 to show how they particularize to time-varying graph signals. Nevertheless, we should remark the number of parameters of the VARMA models is of order $\mathcal{O}(N^2)$, which becomes data demanding for large values of N.

RECURRENT NEURAL NETWORKS

Recurrent neural networks (RNNs) are neural networks that model series, including multivariate time-varying signals. Let us denote by $\boldsymbol{x}_t \in \mathbb{R}^N$ the input vector at timestep t, by $\boldsymbol{h}_t \in \mathbb{R}^H$ the *hidden state* at timestep t, and by $\boldsymbol{o}_t \in \mathbb{R}^O$ the output at timestep t. The equations governing the RNN model are

$$\begin{aligned} \boldsymbol{h}_t &= \sigma(\boldsymbol{W}_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{xh}\boldsymbol{x}_t + \boldsymbol{b}_h) && \text{(hidden state)} \\ \boldsymbol{o}_t &= \sigma(\boldsymbol{W}_{ho}\boldsymbol{h}_t + \boldsymbol{b}_o) && \text{(output)} \end{aligned} \tag{2.39}$$

where $\sigma(\cdot)$ is a non-linear activation function (such as tanh), $\boldsymbol{b}_h \in \mathbb{R}^H$ and $\boldsymbol{b}_o \in \mathbb{R}^O$ are optional bias vectors, and matrices $\boldsymbol{W}_{hh} \in \mathbb{R}^{H\times H}$, $\boldsymbol{W}_{xh} \in \mathbb{R}^{H\times N}$ and $\boldsymbol{W}_{ho} \in \mathbb{R}^{O\times H}$ represent the fully connected layers used to obtain the hidden state $\boldsymbol{h}_t$ and output $\boldsymbol{o}_t$ at timestep t. These matrices and bias vectors are learned during training using backpropagation [24]. Figure 2.18 illustrates the usage of an RNN to learn from a multivariate time series.

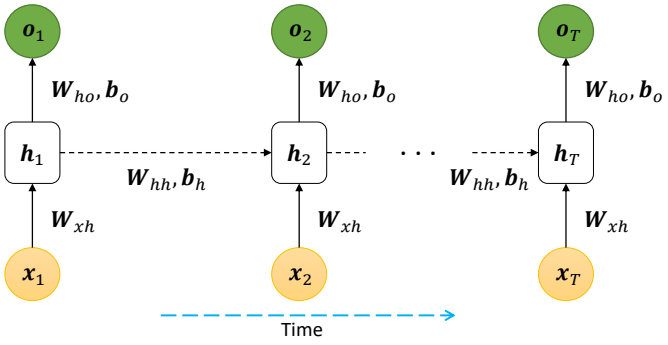


Figure 2.18: Unrolled RNN for a multivariate time series of length T. Notice how the weights of the RNN [cf. (2.39)] are shared across each timestep.

The RNNs in (2.39) suffer from vanishing and exploding gradients. We refer the reader to [30] for additional details on the matter. On a high level, this means that, although RNNs can virtually learn any temporal dependency, these models struggle in learning long-term dependencies [31].

This problem led to the development of a more complex recurrent architecture called Long Short Term Memory network (LSTM) that can learn both short and long term temporal dependencies from time series [32]. We show a representation of the LSTM in Figure 2.19. The LSTM consists of a *cell*, a memory unit able to maintain short and long term information. The information is allowed to enter/leave the cell only through two *gates*, namely the *input* and *forget* gates. Moreover, the *output* gate is used to compute the output of the LSTM. Next, we detail these building blocks of the LSTM and explain how the gating mechanism works. The equations explaining the LSTM are as follows:

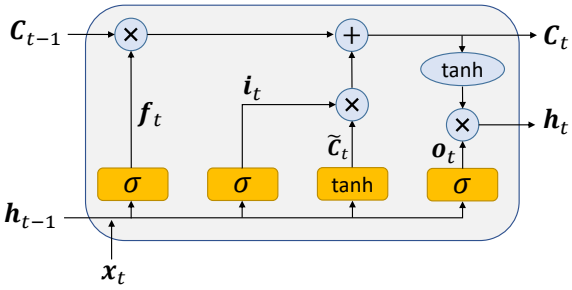


Figure 2.19: LSTM. The orange boxes indicate a layer followed by the specified non-linearity. Blue circles indicate pointwise operations. We denote the input vector by $\boldsymbol{x}_t$. We also denote the previous and current outputs by $\boldsymbol{h}_{t-1}$ and $\boldsymbol{h}_t$, respectively, and the previous and current cell state by $\boldsymbol{C}_{t-1}$ and $\boldsymbol{C}_t$, respectively.

$$\boldsymbol{f}_t = \sigma(\boldsymbol{W}_{xf}\boldsymbol{x}_t + \boldsymbol{W}_{hf}\boldsymbol{h}_{t-1} + \boldsymbol{b}_f) \tag{2.40a}$$

$$\boldsymbol{i}_t = \sigma(\boldsymbol{W}_{xi}\boldsymbol{x}_t + \boldsymbol{W}_{hi}\boldsymbol{h}_{t-1} + \boldsymbol{b}_i) \tag{2.40b}$$

$$\boldsymbol{o}_t = \sigma(\boldsymbol{W}_{xo}\boldsymbol{x}_t + \boldsymbol{W}_{ho}\boldsymbol{h}_{t-1} + \boldsymbol{b}_o) \tag{2.40c}$$

$$\tilde{\boldsymbol{C}}_t = \tanh(\boldsymbol{W}_{xc}\boldsymbol{x}_t + \boldsymbol{W}_{hc}\boldsymbol{h}_{t-1} + \boldsymbol{b}_c) \tag{2.40d}$$

$$\boldsymbol{C}_t = \boldsymbol{f}_t \circ \boldsymbol{C}_{t-1} + \boldsymbol{i}_t \circ \tilde{\boldsymbol{C}}_t \tag{2.40e}$$

$$\boldsymbol{h}_t = \boldsymbol{o}_t \circ \tanh(\boldsymbol{C}_t) \tag{2.40f}$$

where matrices $\boldsymbol{W}_{xf}$, $\boldsymbol{W}_{hf}$, $\boldsymbol{W}_{xi}$, $\boldsymbol{W}_{hi}$, $\boldsymbol{W}_{xo}$, $\boldsymbol{W}_{ho}$, $\boldsymbol{W}_{xc}$, $\boldsymbol{W}_{hc}$ and bias vectors $\boldsymbol{b}_f$, $\boldsymbol{b}_i$, $\boldsymbol{b}_o$, $\boldsymbol{b}_c$ are parameters learned during training, similarly to the RNN case. The operator "$\circ$" indicates element-wise multiplication and $\sigma(\cdot)$ is a sigmoid activation function.

We can see the forget, input, and output gates are obtained similarly and depend on the current input $\boldsymbol{x}_t$ and previous hidden state $\boldsymbol{h}_{t-1}$ [cf. (2.40a)-(2.40c)]. The forget gate $\boldsymbol{f}_t$ deletes part of the old information (first term of (2.40e)) and the input gate $\boldsymbol{i}_t$ adds some new information (second term of (2.40e)). This new information $\tilde{\boldsymbol{C}}_t$ is computed from the current input $\boldsymbol{x}_t$ and previous hidden state $\boldsymbol{h}_{t-1}$ [cf. (2.40d)]. Finally, the output gate $\boldsymbol{o}_t$ computes the current output $\boldsymbol{h}_t$ by transforming the newly updated cell state $\boldsymbol{C}_t$ [cf. (2.40f)].

In other words, the cell $\boldsymbol{C}_t$ maintains information that the model believes to be important. The sigmoid of the gates 'forces' the values between zero and one. Therefore,

successive pointwise multiplications translate into 'maintaining' information (when the value is one) and 'deleting' information (when the value is zero). On a high level, the input gate controls how much information from the current input $\boldsymbol{x}_t$ should be stored in the current cell state $\boldsymbol{C}_t$. The forget date controls how much of the information present in the previous cell state $\boldsymbol{C}_{t-1}$ should be discarded. Finally, the output gate controls what information of the current cell state $\boldsymbol{C}_t$ should be used to produce the current output $\boldsymbol{h}_t$.

This complex mechanism allows the network to store important pieces of information into the cell state and maintain it for as long as needed. That is, if the critical information is found at the beginning of a long sequence, the LSTM can update the cell state when this information is found, and then avoid overriding those cells for the rest of the sequence processing. This gating mechanism allows learning both long and short temporal dependencies [31].

2.4.2. GRAPH-AWARE METHODS

In this section, we look into models that take into account the graph describing pairwise relationships between data points when learning from multivariate time series. The main reason behind graph-aware methods is that, if the data points are described by a graph, a model exploiting this information can achieve good results while reducing the number of parameters effectively. This results in faster training procedures and better generalization capabilities.

G-VARMA AND GP-VAR

The VARMA model [cf. (2.38)] is a suitable choice to learn from multivariate sequences while keeping the complexity of the model low due to its linear form. In this section, we describe the method proposed in [7], where the VARMA model is extended to graphs to forecast time-varying graph signals. In Chapter 6, we will compare this model to our proposed approach for predicting temperatures across several locations.

The graph-VARMA (G-VARMA) model is defined as

$$\boldsymbol{x}_t = -\sum_{p=1}^{P} a_p(\boldsymbol{S})\boldsymbol{x}_{t-p} + \sum_{q=0}^{Q} b_q(\boldsymbol{S})\boldsymbol{\epsilon}_{t-q}\,, \tag{2.41}$$

where the $N\times N$ matrices $a_p(\boldsymbol{S})$ and $b_q(\boldsymbol{S})$ are graph filters of the form: $a_p(\boldsymbol{S}) = \boldsymbol{U}a_p(\boldsymbol{\Lambda})\boldsymbol{U}^\top$, $b_0(\boldsymbol{S}) = \boldsymbol{I}_N$, and $b_q(\boldsymbol{S}) = \boldsymbol{U}b_q(\boldsymbol{\Lambda})\boldsymbol{U}^\top$ (see Section 2.1.4). The term $\boldsymbol{\epsilon}_t$ is a random vector with zero-mean and covariance matrix $\boldsymbol{\Sigma}_\epsilon$ [7]. Since matrix $\boldsymbol{\Lambda}$ is an $N\times N$ diagonal matrix, the G-VARMA is essentially learning N ARMA models of $P+Q$ parameters, each learned w.r.t. one of the N frequencies of the graph. Therefore, this approach drastically reduces the number of parameters compared to the VARMA model, i.e., $N(P+Q)$ parameters instead of $N^2(P+Q)$.

The second model proposed in [7] is obtained from (2.41) by writing the graph filters $a_p(\boldsymbol{S})$ as polynomials in the GSO ($a_p(\boldsymbol{S}) = \boldsymbol{H}_p(\boldsymbol{S}) = \sum_{k=0}^{K} h_{pk}\boldsymbol{S}^k$) and by dropping the MA part of the G-VARMA model, i.e., $Q=0$. Thus, we obtain the graph polynomial-VAR (GP-VAR) model:

$$\boldsymbol{x}_t = -\sum_{p=1}^{P} \boldsymbol{H}_p(\boldsymbol{S})\boldsymbol{x}_{t-p} + \boldsymbol{\epsilon}_t = -\sum_{p=1}^{P}\sum_{k=0}^{K} h_{pk}\boldsymbol{S}^k\boldsymbol{x}_{t-p} + \boldsymbol{\epsilon}_t\,, \tag{2.42}$$

where the scalars h_{pk} are the coefficients of the graph filters. The GP-VAR has $P(K+1)$ parameters which do not depend on the size of the graph N, differently from the VARMA and the G-VARMA. Additionally, since the graph filter in the GP-VAR model is a polynomial in the GSO, it no longer requires the eigendecomposition of the GSO. This choice for the graph filters makes the GP-VAR computationally less expensive than the G-VARMA.

For details about the fitting process adopted to learn the parameters of the G-VARMA and GP-VAR models, we refer to [7].

GGRNN

Lastly, we describe the Gated Graph Recurrent Neural Network (GGRNN) proposed in [3]. The GGRNN is an extension to graph data of the RNN model introduced in Section 2.4.1, where, instead of combining the current hidden state $\boldsymbol{h}_t$ through fully connected layers, we consider graph convolutions.

The dimensionality of the hidden state $\boldsymbol{h}_t$ is set to the number of nodes in the graph N. Then, we update the hidden state and compute the output $\boldsymbol{o}_t$ as

$$\begin{aligned} \boldsymbol{h}_t &= \sigma(\boldsymbol{H}_{xh}(\boldsymbol{S})\boldsymbol{x}_t + \boldsymbol{H}_{hh}(\boldsymbol{S})\boldsymbol{h}_{t-1}) \qquad \text{(hidden state)} \\ \boldsymbol{o}_t &= \sigma(\boldsymbol{H}_{ho}(\boldsymbol{S})\boldsymbol{h}_t) \qquad \text{(output)} \end{aligned} \tag{2.43}$$

where $\boldsymbol{S}$ is the GSO of the graph, $\sigma(\cdot)$ is a non-linearity, and the $N \times N$ matrices $\boldsymbol{H}_{xh}(\boldsymbol{S})$, $\boldsymbol{H}_{hh}(\boldsymbol{S})$, and $\boldsymbol{H}_{ho}(\boldsymbol{S})$ are graph convolutional filters of the form of (2.8). By comparing (2.43) to the RNN case [cf. (2.39)], we see that the hidden state and the output are updated in substantially the same way, with the fully connected layers of the RNN now replaced by graph convolutions. Thus, the number of parameters is reduced and, more importantly, is no longer dependent on the size of the graph.

From (2.43), it is possible to see why it is necessary to set the dimensionality of the hidden state to the number of nodes in the graph: the graph filter $\boldsymbol{H}_{hh}(\boldsymbol{S})$ is an $N \times N$ matrix and, for the product to be defined, it is required that the hidden state is of dimensionality N. However, since the size of the hidden state is an essential hyperparameter for recurrent neural networks, this choice limits the flexibility of the model. The authors in [3], therefore, consider each one of the entries of $\boldsymbol{h}_t$ no longer as a scalar, but rather as a vector of dimensionality G. Thus, the dimensionality of the hidden state is transformed from N to $N \times G$. This modification then requires the computation of the new hidden state to be redefined to account for the additional dimension (see Eq. 7 and 8 of [3]).

Since this model is an extension of the RNN architecture to graphs, it suffers from the same problem of exploding/vanishing gradients (see Section 2.4.1). To tackle this, the authors in [3] provide the GGRNN with a gating mechanism (hence the name 'Gated GRNN'). This gating mechanism controls how much the hidden state should be updated with new information and how much of the old information should be instead deleted. This approach is the same we discussed in the LSTM model.

2.5. CONCLUSION

In this chapter, we presented background material, including the building blocks of GSP, Graph Convolutional Neural Networks, and product graphs for time-varying graph

signals. We will use this material in the remainder of this thesis. In the next chapter, we review current literature to showcase different approaches adopted for learning from time-varying graph signals.

3

Literature Review

Time-varying graph signals often exhibit both spatial and temporal dependencies [2]. In this chapter, we will discuss different approaches exploiting such dependencies, both separately and jointly. The chapter is structured as follows. Section 3.1 discusses graph convolutional neural networks for learning the spatial dependency. Section 3.2 presents methods to capture the temporal dependency. Then, Section 3.3 introduces recent works that jointly capture both spatial and temporal dependencies, which is the topic of interest for this thesis. Section 3.4 discusses methods for earthquake prediction, both with and without neural networks. We conclude the chapter in Section 3.5.

3.1. LEARNING SPATIAL DEPENDENCY THROUGH GCNNS

Graph Convolutional Neural Networks are very effective in learning spatial dependencies [6, 5]. The general idea behind Graph Convolutional Networks (GCNNs) is to extend the approach of Convolutional Neural Networks (CNNs) [33] to unstructured domains, such as graphs [34]. CNNs have gained popularity over the last decade due to their ability to effectively extract meaningful local features while reducing the complexity of the model. However, they can be applied only to Euclidean domains, such as images which are represented as a regular grid [6]. A GCNN stacks several graph convolutional layers to generate features and capture the spatial correlations in the data. Each of these layers generally aggregates node features from their respective neighbourhoods through graph filtering operations [6]. Figure 3.1 illustrates one of the possible GCNN architectures.

There are two main categories of Graph Convolutional Neural Networks: spectral-based and spatial-based [35, 6]. The former group looks at the graph convolution operation from a graph signal processing perspective, i.e., a convolution is obtained by a multiplication in the graph spectral domain [23, 36, 37, 38]. The latter category, instead, looks at graph convolutions as propagating node information along edges [6], i.e., the convolution operation essentially aggregates the information from the neighbours of each node. The work in [37] shows the two approaches are indeed similar and based on the same graph signal processing theory. That is, the spatial-based convolution can be seen as a convolution between the graph signal and a graph filter having a polynomial transfer function. This work also provides a unified framework under which state-of-the-art GCNN models can be seen as specific cases. Such a framework encompasses not only the GCNNs presented above but also attention-based graph neural networks, such as Graph Attention Networks (GATs) [39].

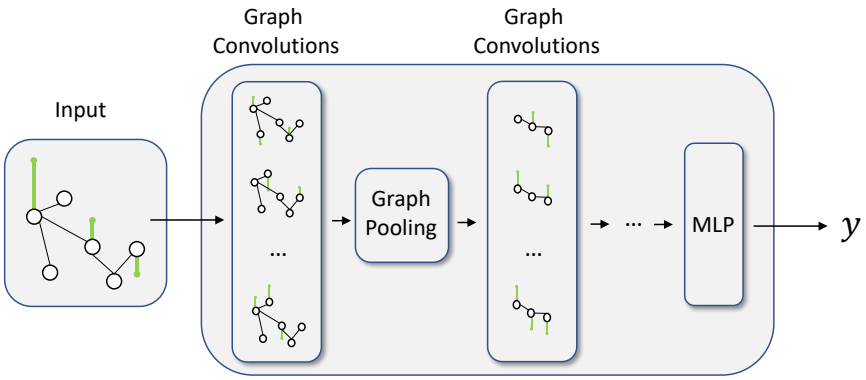


Figure 3.1: Example of a GCNN architecture. The model represented is the one introduced in [36]. Image adapted from [6].

POOLING

As in standard GCNNs, pooling is necessary to reduce the parameters in the network. Generalizing pooling to graphs is not straightforward, and led several works on GCNNs

to avoid performing pooling at all [17]. The complexity resides in identifying the correct coarsened graph to represent the pooled features and that, since graph clustering algorithms are NP-hard, approximations need to be employed [36]. In general, a possible way to categorize different pooling techniques for GCNNs can be whether or not they adopt graph coarsening techniques.

In [36], the authors propose an efficient algorithm to reduce the computational complexity of graph coarsening-based pooling. Their pooling approach employs a greedy algorithm which roughly reduces by a factor of two the number of nodes at each pooling layer. In [40] and [41], the methods employed hierarchical clustering techniques to apply average and max pooling. As stated in [40], however, finding the right clustering approach for a given graph is still an open area of research.

Differently from the pooling methods discussed above, the work in [17] proposes a pooling approach that bypasses the creation of a coarsened graph. Such an approach makes use of zero-padding to "activate" and "deactivate" nodes while still preserving the dimensionality match between the input graph and the generated features at each layer. By doing this, the approach eliminates the need for creating a new smaller graph at each pooling layer while obtaining dimensionality reduction. This method is easier and faster than graph-coarsening approaches and does not require approximations. For these reasons, we will devise in this thesis a graph-time pooling approach which borrows ideas from the zero-padding pooling introduced above.

3.2. LEARNING THE TEMPORAL DEPENDENCY

Learning temporal dependencies is key to time series regression and classification. In this section, we look at those models who are employed to learn temporal dependency but who do not employ any spatial structure in their inner-working mechanisms. Models that fit into this domain can be divided into categories according to whether or not they employ neural networks.

Many different models can be used to learn temporal dependencies without neural networks. An example is the class of the ARMA model and its extensions, such as seasonal ARIMA [42] and VARMA [28], to name a few. Although these models are relatively simple and fast to learn, their simplicity comes at the cost of not being able to learn complex non-linear relationships in the data [43]. Therefore, other methods able to learn more complex relationships are often applied, such as Kalman filters [44] or SVMs [45, 46].

The second group of models that can be used to learn temporal dependencies consists of neural network-based models. Neural networks models can, in principle, learn arbitrarily complex functions [47]. In particular, recurrent neural networks (RNNs) represent the most promising family of neural network models for time series and are effective in learning temporal dependency from the data [48, 49, 5]. Vanilla RNNs are the first type of recurrent neural networks. These models, although being simple and powerful, are challenging to train due to the problem of exploding and vanishing gradients [50]. LSTMs [32] and GRUs [51], instead, tackle these problems by using input, output, and forget gates and seem to be better at learning long and short term temporal dependencies [43].

The neural network-based approaches discussed above all involve the use of recur-

rent networks. However, empirical evidence has shown that CNN-based models can yield even better results when applied to time series modelling, such as the TCN model presented and evaluated in [52]. This model adopts causal dilated convolutions along the time axis to learn both short and long term temporal dependencies. Causal dilated convolutions are introduced to (*i*) avoid leakage from future data and (*ii*) increase the receptive field of the model exponentially with the number of stacked layers, rather than linearly.

Indeed, although most of the research related to time series modelling involves recurrent neural networks, the authors in [52] concluded that convolutional architectures should be looked at as the "natural starting point for sequence modelling". Following this line of thought, our approach will be fully convolutional and will not involve the use of recurrent neural networks.

3.3. LEARNING SPATIAL AND TEMPORAL DEPENDENCIES

Having shown how to capture spatial and temporal dependencies separately, we now turn our attention into methods that jointly capture such dependencies, which is also the focus of this thesis. These methods are of interest for us since we are interested in learning from time-varying graph signals, which jointly vary over the graph and along the time axis [22]. As we will see later in this thesis, our proposed method fits into this category. We will divide these approaches based on how they combine spatial and temporal information to process and learn from time-varying graph signals. First, we will discuss methods that combine existing domain-specific models. We refer to such models as *hybrid* models. Then, we discuss methods where existing models are *fused* to capture the graph topology and the temporal information in a unified framework.

Last but not least, we highlight that there is an entire branch of literature which deals with time-varying signals defined on regular domains (grids) [2, 53, 54, 55]. Although similar to the topic of interest of this thesis, these methods do not apply to irregular domains such as graphs. Therefore, we will not discuss them any further.

3.3.1. HYBRID MODELS

Hybrid models tackle the problem of learning time-varying graph signals by combining existing models known to be effective in learning either graph signals or time series. There is a subcategorization that can be made in this section: whether the models combined to obtain the hybrid model include recurrent neural networks, or whether the models are fully convolutional.

RNN-BASED HYBRID MODELS

Many researchers decide to adopt models from the family of recurrent neural networks because of their ability to capture long and short temporal dependencies from sequential data [10]. The general idea is first to extract spatial features exploiting the graph topology through Graph Convolutional Networks and then treat the newly extracted features as time series to feed into a recurrent neural network. The idea behind this approach is that GCNNs are effective in capturing spatial correlations, while recurrent neural networks are effective in capturing temporal correlations. Thus, combining these two domain-specific architectures allows these hybrid models to learn both temporal and

spatial correlations from time-varying graph signals. Figure 3.2 shows an example of an RNN-based hybrid model.

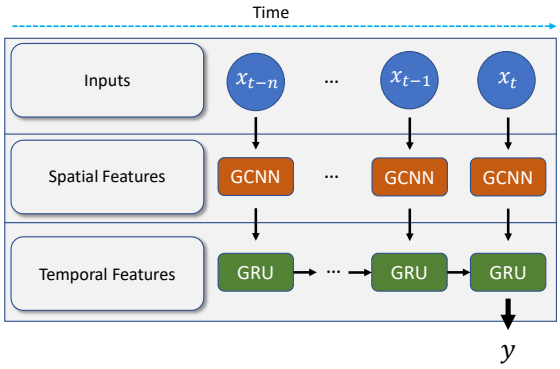


Figure 3.2: An example of hybrid model combining Graph Convolutional Networks and recurrent neural networks (GRUs). The model was proposed in [2]. At each timestep, a GCNN extracts *spatial features* which are then fed into a GRU to generate *temporal features*. The last hidden state of the GRU model is used to output the prediction. Image adapted from [2].

As said in Section 3.2, the most common types of recurrent neural networks adopted in the literature are vanilla RNNs or LSTMs and GRUs, which aim to combat the issue of vanishing and exploding gradients to capture both long and short-term dependencies [2]. In [56], the temporal information is also taken into account when constructing the graph instead of solely relying on geographical information, thus obtaining a graph constructed on both geographic and long-term temporal similarity between nodes. Then, such a graph is used for graph convolutions, and the extracted features are fed into an LSTM. In [9], the authors adopt a GCNN to process the node features at each timestep and then use an LSTM for each vertex to process the extracted convolutional features. The work in [57] fuses multiple graphs, each obtained using different types of measurements, and then exploits this new graph employing a GCNN followed by an LSTM.

The dual version of the above methods consists of swapping the order of GCNNs and RNNs to extract temporal features at the node level first, and then include the topological information through graph convolutions [58]. That is, the time series "seen" by a specific node over time is fed into a recurrent neural network to extract temporal features. Then, these temporal features at each node are processed using graph convolutional networks.

By making use of recurrent neural networks, the models introduced in this section might lead to time-consuming training as well as problems with vanishing/exploding gradients, especially when using RNNs. For these reasons, several authors experimented with non-recurrent fully convolutional models, which lead to higher parallelization and stable gradients [6]. Similarly, in this thesis, we will work towards a solution which does not make use of recurrent neural networks.

CNN-BASED HYBRID MODELS

Instead of adopting recurrent neural networks along the time axis to capture the temporal dependencies, another approach is to obtain hybrid models by combining only fully convolutional models. That is, using GCNNs over the graph domain to learn the spatial dependencies, and 1D convolutions over the time domain to learn the temporal one [59, 60, 35, 61]. Reasons to move from RNN-based hybrid models to CNN-based can be summarized into (*i*) easier training, (*ii*) higher parallelization, (*iii*) stable gradients and (*iv*) fewer parameters [62, 52]. However, a problem with CNN-based solutions is that, when adopting 1D convolutions along the time axis, many layers are required to capture long-term correlations because the receptive field grows linearly with the number of layers[35].

To tackle this problem, in [60] and [35], instead of using standard convolutions along the time axis, the authors employ dilated causal convolutions ([63, 52]) to exponentially increase the receptive field of the model on the time axis, thus increasing the ability to learn long-term temporal dependency. In [64], a multi-resolution temporal module is introduced which stacks together different layers of dilated causal convolutions, each one with a different dilation rate. By doing this, the goal is to capture both short and long-term dependencies through dilated convolutions. Similarly, the authors in [65] propose a fully convolutional model that extracts multi-scale temporal features using a stack of temporal convolutions, and multi-scale spatial features using a stack of GCNNs. Then, a third network fuses the extracted spatial and temporal information to forecast the level of air pollution. In [66] and [62], the authors employ a spatial-temporal convolutional block consisting of a "sandwich" structure made of a graph convolutional layer (capturing the spatial dependency) between two gated temporal convolutions (capturing the temporal dependency). The gated temporal convolutions are simply convolutions followed by a GLU non-linearity and have been found successful in language modelling, where temporal dependency plays a vital role [58, 67].

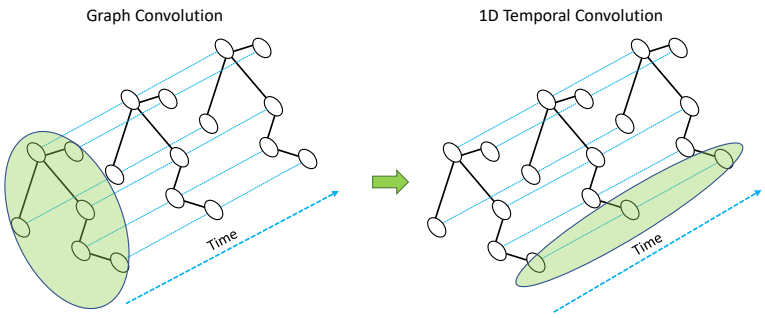


Figure 3.3: Model proposed in [59], obtained combining graph convolutions and 1-D convolutions. The spatial-temporal dependencies are captured by performing a graph-convolution over the graph dimension to capture spatial dependencies, and then performing a 1-dimensional convolution along the time axis to capture the temporal dependencies. Image adapted from [59].

In this section, we discussed hybrid models that are obtained by combining existing models which aim to capture a specific type of dependency, either temporal or spatial.

Now, let us take a look at models obtained by fusing different architectures, and that can capture both types of dependencies in a unified framework.

3.3.2. FUSED MODELS

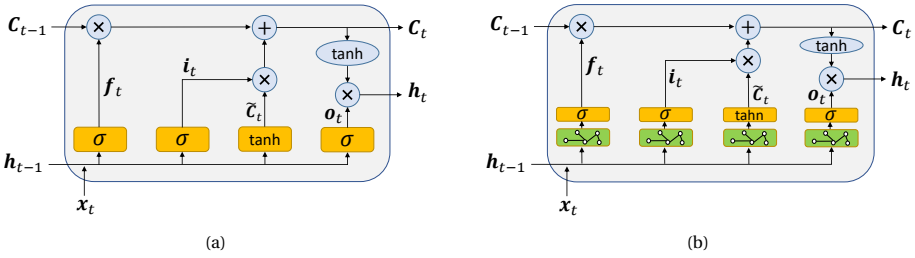


Figure 3.4: Illustration of the graph convolutional LSTM proposed in [68]. **a)** Standard LSTM network. **b)** Graph convolutional LSTM. The main difference between the graph convolutional LSTM (on the right) and the standard LSTM (on the left) is that the matrix multiplications for the input $\boldsymbol{x}_t$ and the hidden state $\boldsymbol{h}_{t-1}$ are now replaced by graph convolutions (green boxes).

In this category, we cover those methods where existing models are modified to include the graph topology information into their inner-working mechanisms, while also learning the temporal dependency. In other words, we no longer combine existing models which process the signals on the graph and time domains separately but instead fuse them into a new unified model.

A first category consists of models that do not involve neural networks. Such models capture spatial and time dependencies extending some more standard statistical methods. Specifically, in [7], the authors extend VAR and VARMA models [28] to model and forecast time-varying graph signals utilizing graph filters. The G-VARMA model is a natural extension of the VARMA model to graphs employing spectral graph filters, while the GP-VAR model is obtained by writing the coefficient matrices of the G-VARMA model as polynomials of the graph Laplacian. The two models proposed in [7], called G-VARMA and GP-VAR, will be used for comparison in this thesis when dealing with regression of time-varying graph signals. These models are simple and do not have many parameters that need to be learned. However, if the relationships in the data are complex and non-linear, such linear models are not able to learn them.

As explained before in this literature review, neural networks can be used to capture arbitrarily complex relationships in the data, provided enough data to learn from. An example of this approach is the graph convolutional LSTM introduced in [11], where graph convolutions replace the matrix multiplications of the standard LSTM model. That is, this architecture reduces the number of parameters necessary to compute the input, forget, and output gates by adding the graph prior to their computation. The same approach was followed in [68], which we illustrate in Figure 3.4. However, this approach is not limited to LSTMs only and can be applied to any recurrent neural network [11]. For example, in [3], the authors propose the GGRNN model by adopting an RNN structure where the dimensionality of the hidden state is set to match the number of nodes in the graph, and then update the hidden state values using graph convolutions. By doing

this, they obtain interpretability of the hidden state (for example, it can be interpreted as a graph signal, allowing for frequency interpretations) while capturing both spatial and time dependencies.

A different approach is the one adopted in [69], where the authors model the spatial dependency as a diffusion process using random walks with restarts. Once obtained the corresponding *diffusion convolutional* layer, the matrix multiplications in a GRU model are replaced with such diffusion convolutions, similarly to the model represented in Figure 3.4b. The same approach of fusing GRUs and graph convolutions was employed in [12], although their graph convolutional layer is built using attention mechanisms. To the best of our knowledge, the work closest to the approach we propose in this thesis is the one of [70], where the authors develop a spatial-temporal convolutional module by extending graph convolutions to take into account also nodes' values over time, considering multiple copies of the graph over time. As we will see, this can be seen as a specific case of the method proposed in this thesis, i.e., when the learned parametric product graph is a Cartesian product graph [cf. Section 2.24].

3.4. EARTHQUAKE PREDICTION

The task of predicting when and where an earthquake will happen is generally difficult because of its intrinsic random nature [71]. However, due to the increase in earthquake activity over the last decade, more effort is being put into finding reliable ways to predict such phenomena in advance [72]. There are two main approaches for earthquake detection: trend-based (also referred to as statistical methods) and precursors-based (also referred to as geophysical methods) [71]. The trend-based category tries to identify periodicity in the occurrences of earthquakes without relying on seismic measurements. Such methods are often used for long-term risk assessment (years) and cannot be used to predict upcoming earthquakes in the short-term [73]. Precursor-based methods, instead, are more suitable for short-term prediction and rely on analyzing phenomena thought to be correlated with the earthquakes activity, such as the velocity of seismic waves, gas emissions and temperature, to name a few.

Standard methods for earthquake prediction which do not involve the use of neural networks are waveform similarity using correlation, principal component analysis, FAST, etc [72]. However, since it is still not known how to predict earthquakes effectively, the end-to-end learning capabilities of deep learning-based methods seem promising, also due to the increase of available data concerning earthquakes. The works in [74], [75], and [71] adopt multilayer perceptrons and LSTM to perform trend-based earthquake prediction. Instead, [72] uses a CNN to process multi-dimensional wave velocity measurements and outputs a probability distribution of the earthquake location. Although obtaining promising results, this approach takes into account wave measurements at a single geographic location instead of considering a graph involving many different locations. Therefore, if the considered area for the prediction of the earthquake location grows too large and involves areas far away from the considered measurement location, such a method does not scale well. In [3], a graph is constructed using several seismic stations, and graph recurrent neural networks are employed to predict the location of the coming earthquake. That is, the experimental setting does not aim to predict whether the earthquake will happen, but it instead predicts where the earthquake is going to hap-

pen out of many clustered location. In this thesis, we will have a similar setting in the experiments concerning earthquakes.

3.5. DISCUSSION

In this chapter, we looked at different approaches to learn spatial and temporal dependencies, both separately and jointly. In Section 3.1, we saw how GCNNs can be used to capture the spatial dependency while keeping the model complexity low, while in Section 3.2 we presented methods for learning the temporal dependency with and without involving the use of neural networks. Then, we moved towards methods to jointly capture such dependencies (Section 3.3) and provided a categorization between *hybrid* and *fused* models. Finally, we provided in Section 3.4 an overview of different approaches for earthquake prediction, which constitutes a significant part in our experiments.

The methods presented in Sections 3.1 and 3.2 serve mainly as building blocks for the following sections, as in this thesis we are interested in jointly learning such dependencies. It is also clear from this review that when there are complex non-linear relationships in the data, neural networks are among the most popular models adopted in the literature, especially when there is an abundance of data. Moreover, although hybrid models are said to learn temporal and spatial dependencies jointly, we argue that they do so in a somewhat separated fashion, with part of the pipeline focusing on spatial dependency, and part of the pipeline focusing on the temporal one. Differently, the approach of fused models (Section 3.3.2) is the one we decide to follow in this work, since we believe it to be the approach to truly learn such dependencies in a joint fashion. A valuable insight from the literature involves recurrent neural networks, which appear to be harder to train and more complex than convolutional networks. This motivated researchers to pursue the direction of devising fully convolutional approaches, similar to the one we propose in this work.

In this thesis, we will propose a new approach which uses GCNNs to learn from time-varying graph signals. Our method, called GTCNN, can be categorized as fully convolutional since it does not make use of recurrent neural networks. However, our method is different from the hybrid methods above in the sense that it does not combine different models operating either on the graph or the time domain. Instead, it performs standard graph convolutions over a new graph which is obtained employing parametric product graphs, learned during training. Such graph carries both the notion of space and time. Therefore, we can categorize our approach into the fused models category.

4

GRAPH-TIME CONVOLUTIONAL NEURAL NETWORK

In this chapter, we introduce our proposed approach for modelling and learning from time-varying graph signals. The main contribution is a new deep learning architecture which performs graph convolutions over parametric product graphs. These product graphs provide a natural structure to represent time-varying graph signals. In the next two chapters, we will use this model for earthquakes classification and temperature forecasting.

The chapter is structured as follows. Section 4.1 presents a high-level introduction to the Graph-Time Convolutional Neural Network (GTCNN). Sections 4.2 and 4.3 present the two building blocks of the GTCNN, namely the graph-time convolutional layer and the graph-time pooling layer. We provide a summary of the overall architecture in Section 4.4 and, in Section 4.5, we introduce an optional regularization term to enforce sparsity in the learned product graphs. Section 4.6 investigates the proposed GTCNN on a synthetic dataset to interpret its different components. We conclude the chapter in Section 4.7.

4.1. OVERVIEW

The GTCNN follows the same layered structure of the GCNN in Section 2.2 with two key differences that we detail in this chapter. First, the graph convolutional module [cf. Section 2.2.1] is replaced with a parametric graph-time convolutional module which also takes into account the evolution of the signal over time. Second, the zero-padding pooling of the GCNN [cf. Section 2.2.2] is redesigned to also pool values across timesteps. This approach reduces the dimensionality not only over the graph dimension but also over the time dimension. The output of the GTCNN is again the output of the last layer or, optionally, the output of fully connected layers added at the end of the graph-time convolutional part.

4.2. PARAMETRIC GRAPH-TIME CONVOLUTIONAL LAYER

The parametric graph-time convolutional layer constitutes the core of the GTCNN architecture. It constructs a parametric product graph and generates graph-time convolutional features by performing convolutions on it (see Section 2.3.3 for details about convolutions on the parametric product graph).

At layer l, we denote as T_{l-1} the temporal window of observation, i.e., how many timesteps of the time-varying graph signal are considered as input. We indicate with F_{l-1} the number of features of the input. For each input feature $g = 1,\dots,F_{l-1}$, we denote as $\boldsymbol{X}_l^g = [\boldsymbol{x}_1^g, \boldsymbol{x}_2^g, \dots, \boldsymbol{x}_{T_{l-1}}^g] \in \mathbb{R}^{N_{l-1} \times T_{l-1}}$ the matrix containing the T_{l-1} consecutive feature-specific graph signals $\boldsymbol{x}_t^g \in \mathbb{R}^{N_{l-1}}$, for $t = 1,\dots,T_{l-1}$. The rows of $\boldsymbol{X}_l^g$ contain the feature-specific values over time for a certain node, while the columns of $\boldsymbol{X}_l^g$ contain the feature-specific graph signals for each timestep.

PARAMETRIC PRODUCT GRAPH

First, this layer computes a parametric product graph $\mathcal{G}_l = (\mathcal{V}_l, \mathcal{E}_l)$ following the approach presented in Section 2.3.2: a graph of T_{l-1} nodes representing time with shift operator $\boldsymbol{C}_{T_{l-1}}$ and the nominal graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ of N nodes with shift operator $\boldsymbol{S}$ are combined as

$$\boldsymbol{S}_l = \sum_{i=0}^{1} \sum_{j=0}^{1} s_{ij} (\boldsymbol{C}_{T_{l-1}}^i \otimes \boldsymbol{S}^j), \tag{4.1}$$

where $\boldsymbol{S}_l$ is the GSO of the parametric product graph $\mathcal{G}_l$ at layer l and s_{ij} are learnable parameters. Therefore, $\mathcal{G}_l$ is a graph consisting of NT_{l-1} nodes. The subscript l stresses the fact that the parametric product graph $\mathcal{G}_l$ is computed at each layer and does not need to have the same vertex set nor the same edge set across layers. This is due to the graph-time pooling layer which reduces the number of timesteps T_{l-1} as well as the number of active nodes $N_{l-1} \leq N$. Therefore, the resulting GSO $\boldsymbol{S}_l$ varies as we go deeper in the network. We will explain graph-time pooling in detail in Section 4.3.

Following our discussion in Section 2.3.2, the product graph in (4.1) is a suitable structure to represent a time-varying graph signal, since it captures its variations both along the time and graph dimensions. Once we obtain $\boldsymbol{S}_l$ at layer l, we can process the time-varying graph signal as a standard graph signal, following the same approach of the standard GCNN described in Section 2.2. To achieve this, we obtain the feature-specific

graph-time signal $\boldsymbol{x}^{g}_{l,\diamond} \in \mathbb{R}^{N_{l-1}T_{l-1}}$ for each input feature g as

$$\boldsymbol{x}^{g}_{l,\diamond} = \text{vec}(\boldsymbol{X}^{g}_{l}) = [x^{g}_{1,1}, x^{g}_{2,1}, \ldots, x^{g}_{N_{l-1},1}, x^{g}_{1,2}, \ldots, x^{g}_{N_{l-1},T_{l-1}}]^{\top}, \tag{4.2}$$

where $vec(\cdot)$ is the operation that transforms a matrix into a column vector by stacking the columns of the matrix. In other words, we obtain the graph signal $\boldsymbol{x}^{g}_{l,\diamond}$ by stacking the feature-specific graph signals for each timestep. We again highlight the fact that $\boldsymbol{x}^{g}_{l,\diamond}$ is a standard graph signal defined on the parametric product graph $\mathcal{G}_l$ at layer l.

CONVOLUTIONAL FEATURES

Next, we obtain the F_l output convolutional features $\boldsymbol{u}^{f}_{l,\diamond} \in \mathbb{R}^{N_{l-1}T_{l-1}}$ as

$$\boldsymbol{u}^{f}_{l,\diamond} = \sum_{g=1}^{F_{l-1}} \boldsymbol{u}^{fg}_{l,\diamond} = \sum_{g=1}^{F_{l-1}} \boldsymbol{H}^{fg}_{l}(\boldsymbol{S}_l)\, \boldsymbol{x}^{g}_{l,\diamond} = \sum_{g=1}^{F_{l-1}} \sum_{k=0}^{K} h^{fg}_{l,k} \boldsymbol{S}^{k}_{l}\, \boldsymbol{x}^{g}_{l,\diamond} \quad \text{for } f = 1, \ldots, F_l\,, \tag{4.3}$$

where $\boldsymbol{H}^{fg}_{l}(\boldsymbol{S}_l)$ is the K-th order linear shift invariant graph filte

$$\boldsymbol{u}^{f}_{l,\diamond} = \sum_{g=1}^{F_{l-1}} \boldsymbol{u}^{fg}_{l,\diamond} = \sum_{g=1}^{F_{l-1}} \boldsymbol{H}^{fg}_{l}(\boldsymbol{S}_l)\, \boldsymbol{x}^{g}_{l,\diamond} = \sum_{g=1}^{F_{l-1}} \sum_{k=0}^{K} h^{fg}_{l,k} \boldsymbol{S}^{k}_{l}\, \boldsymbol{x}^{g}_{l,\diamond} \quad \text{for } f = 1, \ldots, F_l\,, \tag{4.4}$$

r used to process the g-th input feature $\boldsymbol{x}^{g}_{l,\diamond}$ when computing the intermediate features $\boldsymbol{u}^{fg}_{l,\diamond}$ related to the f-th output feature $\boldsymbol{u}^{f}_{l,\diamond}$. This was illustrated in Figure 2.7 for the GCNN.

As we will see in the next section, the graph-time pooling reduces the dimensionality of the features. Therefore, when graph-time pooling is used, the computation of the intermediate features [cf. (4.4)] is not possible due to the dimensionality mismatch between the GSO $\boldsymbol{S}_l$ and the input graph signal $\boldsymbol{x}^{g}_{l,\diamond}$. That is, the matrix-vector multiplication between $\boldsymbol{S}_l \in \mathbb{R}^{NT_{l-1} \times NT_{l-1}}$ and $\boldsymbol{x}^{g}_{l,\diamond} \in \mathbb{R}^{N_{l-1}T_{l-1} \times N_{l-1}T_{l-1}}$ cannot be performed. This situation is equivalent to what we discussed for the GCNN case in Section 2.2.2. To overcome this, we adopt the same approach of the GCNN and employ zero-padding to resolve the dimensionality mismatch. That is, the input signal is zero-padded to match the dimensionality, as we discussed in Section 2.2.2. For additional information regarding the use of zero-padding, we refer the reader to Section III.A of [17].

CONNECTION TO 2D FILTERING

Notice, when the time window T in the parametric product graph grows large, the resulting filtering process becomes memory and computationally expensive. In such a case, we can substitute the filtering on the graph-time product graph with alternatives that bypass the construction of this bigger graph.

In GSP, the concept of 2D filters that capture time variations of graph signals was introduced in [8]. The authors introduced a graph-time FIR filter of the form:

$$\boldsymbol{y}_t = \sum_{k=0}^{K_g} \sum_{k'=0}^{K_t} h_{kk'} \boldsymbol{S}^{k} \boldsymbol{x}_{t-k'}\,, \tag{4.5}$$

where $\boldsymbol{x}_t$ is a graph signal, $\boldsymbol{S}$ is the GSO, and $h_{kk'}$ are parameters that need to be learned. In such a formulation, $\boldsymbol{S}$ is the GSO of the original graph and, thus, no product graph is constructed. Instead, this equation implements graph-shifting through $\boldsymbol{S}$ (up to K_g hops), and time-shifting by indexing past realizations of $\boldsymbol{x}_t$ (up to K_t past realizations).

Although a filtering as in (4.5) seems different from that of (2.28), the following proposition shows that filtering on product graphs can be written as the two-dimensional filtering in (4.5).

Proposition 1 (Connection to two-dimensional filtering) *Denote by $\boldsymbol{x}_t \in \mathbb{R}^N$ the t-th occurrence of a time-varying graph signals evolving for T timesteps over an N-node graph $\mathcal{G}$ with GSO $\boldsymbol{S}$. Denote by $\boldsymbol{x}_\diamond = vec([\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T]) \in \mathbb{R}^{NT}$ its representation as a standard graph signal on the product graph $\mathcal{G}_\diamond$ with GSO $\boldsymbol{S}_\diamond$ obtained as in (4.1). Performing graph filtering over $\mathcal{G}_\diamond$ using (2.27) as*

$$\boldsymbol{y}_\diamond = \boldsymbol{H}(\boldsymbol{S}_\diamond)\boldsymbol{x}_\diamond$$

with

$$\boldsymbol{H}(\boldsymbol{S}_\diamond) = \sum_{k=0}^{K} h_k \boldsymbol{S}_\diamond^k = \sum_{k=0}^{K} h_k \left(\sum_{i=0}^{1} \sum_{j=0}^{1} s_{ij} (\boldsymbol{C}_T^i \otimes \boldsymbol{S}^j) \right)^k$$

can be written as the two-dimensional filter introduced in [8], which we recall:

$$\boldsymbol{y}_t = \sum_{k=0}^{K_g} \sum_{k'=0}^{K_t} h_{kk'} \boldsymbol{S}^k \boldsymbol{x}_{t-k'} \, .$$

Proof of Prop. 1
As stated in [15], a filter of the form:

$$\begin{aligned} \boldsymbol{H}(\boldsymbol{S}_\diamond) &= \sum_{k=0}^{K} h_k \boldsymbol{S}_\diamond^k \\ &= \sum_{k=0}^{K} h_k \left(\sum_{i=0}^{1} \sum_{j=0}^{1} s_{ij} (\boldsymbol{C}_T^i \otimes \boldsymbol{S}^j) \right)^k \end{aligned}$$

can be written as a more general filter:

$$\boldsymbol{H}(\boldsymbol{S}_\diamond) = \sum_{k=0}^{K} \sum_{m=0}^{M} h_{km} \left(\boldsymbol{C}_T^m \otimes \boldsymbol{S}^k \right) \tag{4.6}$$

for some parameters h_{km} (cf. Eq (8) of [15]).
Let us use notation $x_{(i,t)}$ to indicate the scalar value on node i at timestep t. With this notation and the definition of $\boldsymbol{x}_\diamond$, we can see $\boldsymbol{x}_\diamond$ has the form:

$$\boldsymbol{x}_\diamond = [x_{(1,0)}, x_{(2,0)}, \ldots, x_{(N,0)}, x_{(1,1)}, \ldots, x_{(N,T-1)}]^\top = \begin{bmatrix} \boldsymbol{x}_0 \\ \boldsymbol{x}_1 \\ \vdots \\ \boldsymbol{x}_{T-1} \end{bmatrix} . \tag{4.7}$$

The output filtered signal $\boldsymbol{y}_\diamond \in \mathbb{R}^{NT}$ is then obtained as

$$\begin{aligned}\boldsymbol{y}_\diamond = \begin{bmatrix} \boldsymbol{y}_0 \\ \boldsymbol{y}_1 \\ \vdots \\ \boldsymbol{y}_{T-1} \end{bmatrix} &= \boldsymbol{H}(\boldsymbol{S}_\diamond)\, \boldsymbol{x}_\diamond \\ &= \sum_{k=0}^{K} \sum_{m=0}^{M} h_{km} \left(\boldsymbol{C}_T^m \otimes \boldsymbol{S}^k\right) \boldsymbol{x}_\diamond \,.\end{aligned} \tag{4.8}$$

We are interested in the values $\boldsymbol{y}_t \in \mathbb{R}^N$ rather than the entire filtered signal $\boldsymbol{y}_\diamond$. Consider an $N \times NT$ binary matrix $\boldsymbol{E}_t$ defined as

$$\begin{aligned}\boldsymbol{E}_t &= \boldsymbol{e}_t \otimes \boldsymbol{I}_N \\ &= \begin{bmatrix} 0 & \dots & 0 & 1 & 0 & \dots & 0 \end{bmatrix} \otimes \boldsymbol{I}_N \\ &= \begin{bmatrix} \boldsymbol{0}_N & \dots & \boldsymbol{0}_N & \boldsymbol{I}_N & \boldsymbol{0}_N & \dots & \boldsymbol{0}_N \end{bmatrix},\end{aligned} \tag{4.9}$$

where $\boldsymbol{e}_t \in \mathbb{R}^{1\times T}$ is a row vector with zero entries except the t-th entry and $\boldsymbol{I}_N \in \mathbb{R}^{N\times N}$ is the identity matrix. In other words, $\boldsymbol{E}_t$ is obtained by concatenating T square matrices of dimension N, where such matrices are the null matrices except the t-th matrix, which is $\boldsymbol{I}_N$.

Then, we can obtain $\boldsymbol{y}_t$ from $\boldsymbol{y}_\diamond$ [cf. (4.8)] as

$$\begin{aligned}\boldsymbol{y}_t &= \boldsymbol{E}_t\, \boldsymbol{y}_\diamond \\ &= (\boldsymbol{e}_t \otimes \boldsymbol{I}_N) \sum_{k=0}^{K} \sum_{m=0}^{M} h_{km} \left(\boldsymbol{C}_T^m \otimes \boldsymbol{S}^k\right) \boldsymbol{x}_\diamond \\ &= \sum_{k=0}^{K} \sum_{m=0}^{M} h_{km} \,(\boldsymbol{e}_t \otimes \boldsymbol{I}_N)\,(\boldsymbol{C}_T^m \otimes \boldsymbol{S}^k)\, \boldsymbol{x}_\diamond \\ &= \sum_{k=0}^{K} \sum_{m=0}^{M} h_{km} \left((\boldsymbol{e}_t \boldsymbol{C}_T^m) \otimes (\boldsymbol{I}_N \boldsymbol{S}^k)\right) \boldsymbol{x}_\diamond \qquad \text{(Mixed-product property (A.1))} \\ &= \sum_{k=0}^{K} \sum_{m=0}^{M} h_{km} \left((\boldsymbol{e}_t \boldsymbol{C}_T^m) \otimes \boldsymbol{S}^k\right) \boldsymbol{x}_\diamond \,.\end{aligned} \tag{4.10}$$

The product $(\boldsymbol{e}_t \boldsymbol{C}_T^m)$ selects the t-th row of $\boldsymbol{C}_T^m$, which we indicate with $[\boldsymbol{C}_T^m]_{t:} \in \mathbb{R}^{1\times T}$. Since t is used to index the rows of $\boldsymbol{C}_T^m$, admissible values are $t = 0, \dots, T-1$. Due to the definition of $\boldsymbol{C}_T$ (Section 2.1.2), it can be verified that $[\boldsymbol{C}_T^m]_{t:}$ is a $1 \times T$ row vector with all entries zeros except the entry indexed by $(t-m) mod (T)$. Therefore, the Kronecker product $([\boldsymbol{C}_T^m]_{t:} \otimes \boldsymbol{S}^k)$ yields a $N \times NT$ matrix containing zero matrices except for the $(t-m) mod (T)$-th matrix, which is $\boldsymbol{S}^k$ (similarly to the Kronecker product in (4.9)). Therefore, all the other graph signals $\boldsymbol{x}_j$, for $j \neq (t-m) mod (T)$, are multiplied by a zero

matrix $\mathbf{0}_N$, allowing us to write

$$((\boldsymbol{e}_t \boldsymbol{C}_T^m) \otimes \boldsymbol{S}^k)\, \boldsymbol{x}_\diamond = \begin{bmatrix} \mathbf{0}_N & \dots & \mathbf{0}_N & \boldsymbol{S}^k & \mathbf{0}_N & \dots & \mathbf{0}_N \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_0 \\ \boldsymbol{x}_1 \\ \vdots \\ \boldsymbol{x}_{T-1} \end{bmatrix} = \boldsymbol{S}^k\, \boldsymbol{x}_{(t-m)mod(T)}\,.$$

We can then rewrite (4.10) as

$$\boldsymbol{y}_t = \sum_{k=0}^{K} \sum_{m=0}^{M} h_{km} \boldsymbol{S}^k\, \boldsymbol{x}_{(t-m)mod(T)}\,. \tag{4.11}$$

For T going to infinity, we can drop the $mod(\cdot)$ operation and write $(t-m)mod(T)$ simply as $(t-m)$, yielding:

$$\boldsymbol{y}_t = \sum_{k=0}^{K} \sum_{m=0}^{M} h_{km} \boldsymbol{S}^k\, \boldsymbol{x}_{t-m}\,,$$

which was the objective of this proof. As such, we have shown how graph filtering over the parametric product graph defined by $\boldsymbol{S}_\diamond$ can be written as the two-dimensional filtering of (4.5). □

A key difference between the two approaches is that, by using the parametric product graph in (2.28), we gain interpretability over the type of product graphs learned by the GTCNN and provide a useful prior w.r.t the graph-time interactions in the data. Precisely, we can track the parameters s_{ij} [cf. (4.1)] during training and understand what parametric product graph is being learned by the GTCNN. We will provide a concrete example of this in the experiments in Section 4.6. Due to time constraints, in this thesis we experimented the GTCNN only with the product graphs-based implementation.

4.3. GRAPH-TIME POOLING

Graph-time pooling is a non-learnable operation whose effects are twofold. First, it reduces the dimensionality of the convolutional features $\boldsymbol{u}_{l,\diamond}^f$ [cf. Eq (4.4)] from $N_{l-1}T_{l-1}$ to $N_l T_l$, with the constraint that $N_l \leq N_{l-1}$ and $T_l \leq T_{l-1}$. That is, this module reduces the number of active nodes per timestep from N_{l-1} to N_l. Second, it reduces the number of timesteps from T_{l-1} to T_l. The *summarization* and *downsampling* steps, presented in Section 2.2 for the pooling of the GCNN are still employed. In addition, we consider an additional step, which we call *time slicing*.

SUMMARIZATION STEP

The first operation in pooling is the *summarization* step. Recall, from the graph-time convolutional layer [cf. Section 4.2], we have the F_l convolutional features $\boldsymbol{u}_{l,\diamond}^f \in \mathbb{R}^{N_{l-1}T_{l-1}}$, for $f = 1, \dots, F_l$. First, we introduce a pooling operator $\rho(\cdot\,;\ \alpha, \mathcal{G}_{l,\times})$, where we indicate with $\mathcal{G}_{l,\times}$ the Cartesian product graph between the directed cyclic graph representing T_{l-1} timesteps (Section 2.3.2) and the underlying graph $\mathcal{G}$[1]. Graph $\mathcal{G}_{l,\times}$ serves as

[1] Note that this support can be any valid product graph, such as the Kronecker, the Cartesian, the Strong, or the parametric product graph $\boldsymbol{S}_l$. However, we found that the parametric product graph and the Strong product

support over which we compute the α-hop neighbourhoods needed for pooling, which behaves in the same way as the pooling operator introduced for the GCNN in Section 2.2.2. The difference is that now it operates on neighbourhoods defined over both the graph and time, while for the GCNN neighbourhoods were defined only over the graph $\mathcal{G}$. Therefore, parameter α controls how many hops are taken into account when applying the pooling operator $\rho(\cdot)$ both with respect with time and graph. Analogously to (2.12), the summarized features $\boldsymbol{v}^f_{l,\diamond} \in \mathbb{R}^{N_{l-1}T_{l-1}}$ are obtained as

$$\boldsymbol{v}^f_{l,\diamond} = \rho(\boldsymbol{u}^f_{l,\diamond}\,;\,\alpha,\,\mathcal{G}_{l,\times}) \quad f = 1,\dots,F_l\,, \tag{4.12}$$

where $\rho(\cdot)$ can be the *max* operator, as illustrated in Figure 2.8 for the GCNN, but it can also be the *average* operator or any other function that operates on a set of real values. Since the neighbourhood of a node in $\mathcal{G}_{l,\times}$ includes also nodes in different timesteps, as shown in Figure 2.16, the values of $\boldsymbol{v}^f_{l,\diamond}$ are obtained through a summarization over both the graph and the time domain.

4

SLICING STEP

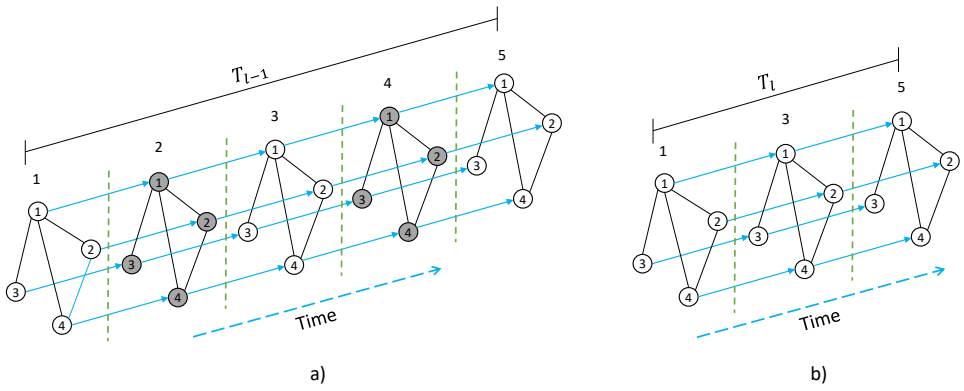


Figure 4.1: Slicing step. The number on top of each slice indicates the slice index. **a)** The input (all nodes) has $T_{l-1} = 5$ timesteps and $N_{l-1} = 4$ nodes per timestep. The slicing ratio is $R_l = 2$. Therefore, slices two and four are removed (grey nodes). **b)** The *sliced features* resulting from the slicing operation. The result has $T_l = \lceil \frac{5}{2} \rceil = 3$ slices and $N_{l-1} = 4$ nodes per timestep.

The second operation in pooling is *slicing*. Here, the summarized features $\boldsymbol{v}^f_{l,\diamond}$ are sliced across the time dimension. Consider a single feature $\boldsymbol{v}^f_{l,\diamond} \in \mathbb{R}^{N_{l-1}T_{l-1}}$ as a graph signal on the Cartesian product graph (Figure 2.12). We can see $\boldsymbol{v}^f_{l,\diamond}$ as T_{l-1} slices of features, where each slice has dimension N_{l-1}, as illustrated in Figure 4.1.

Given a *slicing ratio* R_l, we keep one every R_l slices out of the T_{l-1} available, thus resulting in $T_l = \lceil \frac{T_{l-1}}{R_l} \rceil$ output slices. For example, if $\boldsymbol{v}^f_{l,\diamond}$ consists of $T_{l-1} = 5$ timesteps and

graph lead to the summarization of features across areas that are too large, due to their higher number of edges compared to the Cartesian. This wide summarization leads to a loss of information since it flattens the variations of the features too much. Thus, we proceed with the Cartesian product graph for the graph-time pooling step.

the *slicing ratio* is $R_l = 2$, then we retain one every two slices, resulting in $\lceil 5/2 \rceil = 3$ output slices. That is, the graph-time pooling layer will preserve the values of the summarized features $\boldsymbol{v}^f_{l,\diamond}$ corresponding to the slices one, three, and five, discarding the values of slices two and four, as we show in Figure 4.1.

We denote as $h(\cdot) : \mathbb{R}^{N_{l-1}T_{l-1}} \mapsto \mathbb{R}^{N_{l-1}T_l}$ the slicing operation described above. Then, we obtain the *sliced features* $\boldsymbol{p}^f_{l,\diamond} \in \mathbb{R}^{N_{l-1}T_l}$ as

$$\boldsymbol{p}^f_{l,\diamond} = h(\boldsymbol{v}^f_{l,\diamond}\,;\, R_l) \quad \text{for } f = 1,\dots,F_l\,. \tag{4.13}$$

While it is possible to apply another summarization function (such as averaging) across time slices instead of discarding values, note that the summarization step in (4.12) already takes into account values across slices. Thus, we proceed with discarding slices, to keep the architecture complexity low.

4

DOWNSAMPLING STEP

The last step of the graph-time pooling layer is *downsampling*. We again use zero-padding (see Section 2.2.2) along the graph dimension to reduce the number of nodes per timestep from N_{l-1} to N_l. To do this, we use a sampling matrix $\boldsymbol{C}_l \in \mathbb{R}^{N_l T_l \times N_{l-1} T_l}$ to compute the reduced features $\boldsymbol{b}^f_{l,\diamond} \in \mathbb{R}^{N_l T_l}$ as

$$\boldsymbol{b}^f_{l,\diamond} = \boldsymbol{C}_l\, \boldsymbol{p}^f_{l,\diamond} \quad f = 1,\dots,F_l\,. \tag{4.14}$$

This procedure is illustrated in Figure 4.2. Analogously to the graph pooling in Section 2.2, the sampling matrix $\boldsymbol{C}_l$ selects the N_l nodes with highest degree out of the N_{l-1} nodes available *for each timestep*.

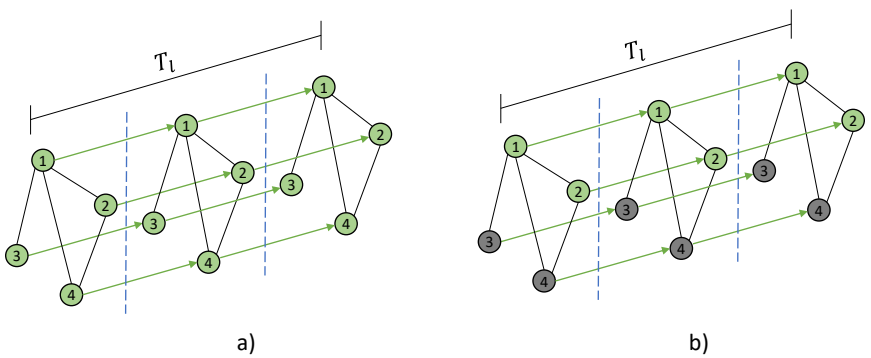


Figure 4.2: Downsampling step. **a)** The input (all nodes) has $T_l = 3$ timesteps and $N_{l-1} = 4$ nodes per timestep. **b)** The output (green nodes) of the downsampling step has $T_l = 3$ and $N_l = 2$ nodes per timestep. The values associated with the $N_l - N_{l-1}$ remaining grey nodes are discarded.

Let us summarize the steps of the pooling stage. The input to the graph-time pooling layer l consists of F_l convolutional features $\boldsymbol{u}^f_{l,\diamond} \in \mathbb{R}^{N_{l-1}T_{l-1}}$. We first obtain the summarized features $\boldsymbol{v}^f_{l,\diamond} \in \mathbb{R}^{N_{l-1}T_{l-1}}$ by means of a Cartesian product graph $\mathcal{G}_{l,\times}$ and a pooling

operator $\rho(\cdot)$ as in (4.12). Subsequently, we obtain the sliced features $\boldsymbol{p}^{f}_{l,\diamond} \in \mathbb{R}^{N_{l-1}T_l}$ by slicing the summarized features across the time dimension as in (4.13). The third step comprises obtaining the reduced features $\boldsymbol{b}^{f}_{l,\diamond} \in \mathbb{R}^{N_l T_l}$ by selecting only N_l values out of N_{l-1} available values for each timestep, as in (4.14). The output of the graph-time pooling layer consists of F_l features $\boldsymbol{b}^{f}_{l,\diamond} \in \mathbb{R}^{N_l T_l}$.

Following graph-time pooling, we obtain the output of layer l by applying a pointwise non-linearity $\sigma(\cdot)$, such as the ReLU function, to the reduced features $\boldsymbol{b}^{f}_{l,\diamond}$:

$$\boldsymbol{x}^{f}_{l+1,\diamond} = \sigma(\boldsymbol{b}^{f}_{l,\diamond}) \quad f = 1, \ldots, F_l \,, \tag{4.15}$$

which are then sent as input to the next layer.

4.4. OVERALL ARCHITECTURE

After having introduced the building blocks of our architecture in the previous sections, here we combine them and present the overall GTCNN architecture.

The input signal is a time-varying graph signal observed for T timesteps, as in Section 2.3.2. The GTCNN exploits information about the spatial graph $\mathcal{G}$, on top of which the time-varying graph signal evolves in time. This information is encoded through the spatial GSO $\boldsymbol{S}$. At the input layer, the signal has a window of observation $T_0 = T$ timesteps. For each timestep, the signal dimensionality equals the number of nodes in $\mathcal{G}$, i.e., $N_0 = N$.

At each layer, the GTCNN generates convolutional features through graph convolutions over the parametric product graph $\boldsymbol{S}_l$ as in (4.4). Thus, the GTCNN jointly learns both the $s_{ij,l}$ parameters of the parametric product graph and the $h^{fg}_{k,l}$ parameters of the graph convolutional filters [cf. (4.1)]. Then, the graph-time pooling module reduces the dimensionality of these features across the graph dimension (through downsampling) and time dimension (through slicing), see Section 4.3.

The output of each layer is sent as input for the next layer. If the network is composed of L layers, the output of the GTCNN is the output of the last layer:

$$\hat{\boldsymbol{y}} = [\boldsymbol{x}^{1}_{L,\diamond}, \ldots, \boldsymbol{x}^{F_L}_{L,\diamond}] \in \mathbb{R}^{N_L T_L \times F_L}. \tag{4.16}$$

Optionally, the output $\hat{\boldsymbol{y}}$ can be further sent to a number of fully connected layers to change the output dimensionality. Finally, the output $\hat{\boldsymbol{y}}$ is compared with the target labels to compute the loss function. We provide in Figure 4.3 an illustration of all the steps summarized above.

The number of parameters the network needs to learn at each layer l is $(K+1)F_{l-1}F_l$, plus the four additional parameters $s_{ij,l}$ necessary to compute the parametric product graph in (4.1). This number is unaffected by the size of the graph, which has NT_{l-1} nodes at layer l. The network needs to learn also the weights of the final fully connected layers, if present. Suppose we have C target classes, then the fully connected layer comprises $N_L T_L F_L * C$ parameters. The computational cost of layer l is the one of the parametric graph-time convolutional module, since the cost of the graph-time pooling operations is negligible. That is, this equals the cost of computing (4.4) and is of order $\mathcal{O}(|\mathcal{E}_l|(K+$

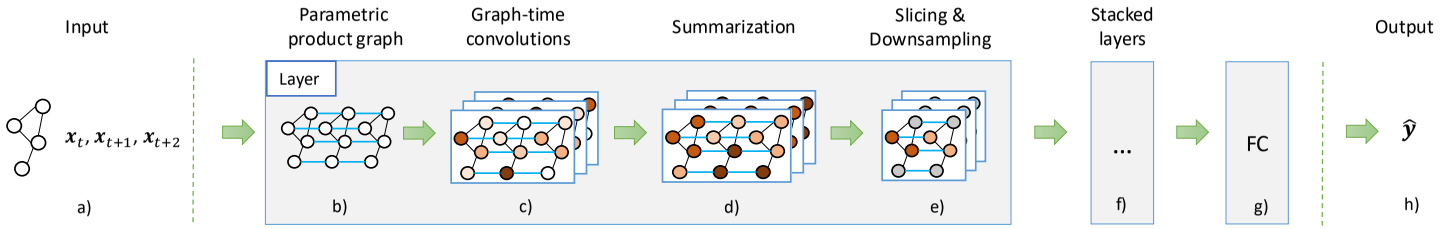


Figure 4.3: Overview of the GTCNN. The observation window is $T = 3$ and the graph has $N = 4$ nodes. **a)** The input to the network is the graph $\mathcal{G}$ and the graph signals $\boldsymbol{x}_t, \ldots, \boldsymbol{x}_{t+T-1}$. **b)** The parametric product graph $\mathcal{G}_l$ is built at layer l (Section 4.2). **c)** A number of convolutional features is generated by performing graph convolutions on the parametric product graph $\mathcal{G}_l$ as in (4.4). **d)** The convolutional features are summarized using the pooling operator $\rho(\cdot)$ as in (4.12). This step does not modify the dimensionality of the features but only substitutes the value at each node with its summarized value. **e)** The summarized features are sliced in time (only two out of three slices are kept) and downsampled (only two out of four nodes are kept for each slice) by means of zero-padding (grey nodes are discarded); see Section 4.3. **f)** A number of layers with the same structure of the previous layer are stacked in sequence. **g)** After the last layer, a fully-connected layer can be employed to match the dimensionality of the labels. **h)** The output $\hat{\boldsymbol{y}}$ of the GTCNN is the output of the final fully connected layer.

$1)F_{l-1}F_l)$, where $|\mathscr{E}_l|$ denotes the number of edges in the parametric product graph $\mathscr{G}_l$ [cf. Section 2.3.1].

In the next section, we introduce a regularization term that can be added to the loss function to encourage sparsity on the product graphs $\mathscr{G}_l$.

4.5. $\mathscr{L}_1$-NORM REGULARIZATION

We introduce a regularization term to achieve sparsity in the coefficients $s_{ij,l}$ the GTCNN learns when building the parametric product graph at each layer (see Section 4.2). This regularization will allow us to avoid overfitting and also learn sparse connections with the GTCNN. The notation $s_{ij,l}$ indicates the coefficient s_{ij} [cf. (4.1)] used to construct the parametric product graph $\boldsymbol{S}_l$ at layer l. To achieve sparsity, we employ the ℓ_1-norm; a known method to achieve sparse coefficients when training machine learning models [76]. We denote as $\mathbf{s} \in \mathbb{R}^{4L}$ the vector containing the $4L$ coefficients $s_{ij,l}$ (each layer l has four s_{ij} parameters), and we can define the regularization loss:

$$\mathscr{L}(\mathbf{s}) = \|\mathbf{s}\|_1 = \sum_{i,j,l} |s_{ij,l}| \,. \tag{4.17}$$

With this regularization term in place, the final loss of the GTCNN is

$$\mathscr{L}_{\text{GTCNN}} = \mathscr{L}(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \beta \mathscr{L}(\mathbf{s}) \,, \tag{4.18}$$

where β is a scalar value that controls the importance of the regularization term $\mathscr{L}(\mathbf{s})$, and $\mathscr{L}(\hat{\boldsymbol{y}}, \boldsymbol{y})$ is the loss computed between the true label $\boldsymbol{y}$ and the prediction $\hat{\boldsymbol{y}}$ (Section 2.2.3).

The goal of the regularization is twofold. On the one hand, it is common practice in machine learning to add a regularization term to reduce overfitting. Moreover, graphs are in nature inherently sparse [77] and, therefore, we want to encourage sparsity in the GTCNN learning process. On the other hand, sparser graphs at each layer reduce the computational cost, since the number of edges reduces.

4.6. NUMERICAL INSIGHTS: SOURCE LOCALIZATION

In this section, we use a synthetic dataset for the source localization problem [17, 37] to characterize the different components introduced in the previous sections. This includes analyzing the convolutions over the parametric product graph, the graph-time pooling, and the sparse regularizer.

PROBLEM FORMULATION

Consider a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with C communities, $\mathcal{C} = \{0, \dots, C-1\}$. At timestep $t = 0$, the graph signal is a vector with all entries set to zero except the entry indexed by i, denoted as $\boldsymbol{x}^{(0)} = \boldsymbol{\delta}_i \in \mathbb{R}^N$. The index i of the unique non-zero entry of $\boldsymbol{x}^{(0)}$ identifies the *source node*. Due to the community-like structure of the graph, the source node i belongs to community $c \in \mathcal{C}$, see Figure 4.4. As t increases (i.e., $t = 1, 2, \dots, T-1$), the graph signal $\boldsymbol{x}^{(t)}$ undergoes a diffusion process modelled through successive applications of the GSO as in (2.5). To avoid numerical instabilities, the GSO adopted for such a diffusion process is a normalized version of the adjacency matrix $\boldsymbol{A}_{\text{n}} = \frac{1}{\lambda_{\max}} \boldsymbol{A}$, where $\lambda_{\max}$ is the largest eigenvalue of $\boldsymbol{A}$. The problem of *Source Localization* is to predict the community $c \in \mathcal{C}$ from which the signal originated (the community source node i belongs to) given the diffused signal $\boldsymbol{x}^{(t)}$ at an *unknown* timestep t.

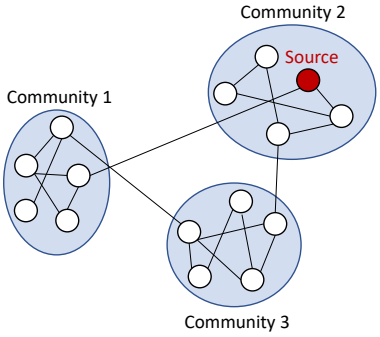


Figure 4.4: Source localization problem. The source node (depicted in red) belongs to community two. The graph signal shown is all zero except for the *source node i*, for which it has value one: $\boldsymbol{x}^{(0)} = \boldsymbol{\delta}_i \in \mathbb{R}^N$.

We consider as input all consecutive observations within a window T: $\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t+1)}, \dots, \boldsymbol{x}^{(t+T-1)}$. We formulate the *Source Localization* task as

$$[\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t+1)}, \dots, \boldsymbol{x}^{(t+T-1)}] \xrightarrow[\mathcal{G}]{f(\cdot)} c \in \mathcal{C} \,,$$

where $f(\cdot)$ is the function mapping a set of observations $\boldsymbol{x}^{(t)}$ to a community c, while exploiting the topology of $\mathcal{G}$. Our aim is to learn $f(\cdot)$, given a number of labeled examples.

EXPERIMENTAL SETTING

We generated a synthetic dataset using the Stochastic Block Model (SBM), a generative model for random graphs that creates graphs with communities. We built graphs

with five communities of 20 nodes each. The probability of an edge between nodes of the same community is 0.8, while for an edge between nodes of different communities is 0.2. Since SBM is a stochastic model, we generated ten different graphs, and for each one of them, we performed ten different splits. The results are then averaged across them. In each split, 80% of the data goes into the training set, 10% into the validation set, and the remaining 10% into the test set.

We investigated: (i) the GCNN baseline models; (ii) the use of non-parametric, i.e., fixed product graphs; (iii) the use of the parametric product graph; (iv) the effect of the sparse regularizer for the parametric product graph; (v) the impact of pooling. When adopting product graphs, we investigated modelling time as a *directed* graph [cf. Section 2.3.2] and as an *undirected* graph. Our rationale is that future time instants may carry meaningful information also about the past time instants. For the product graphs, we consider the Cartesian product graph, the strong product graph, and the parametric product graph. Each of these product graphs models graph-time interactions in a different manner. For each approach we investigated, we tested the models using observation windows T of two and three observations in time. For all experiments except (v), we set the number of features $F = 2$ and the filter taps $K = 2$. Furthermore, since graph-time pooling reduces the number of weights of the final fully connected layer and allows for wider and deeper architectures, we then experimented the complete GTCNN with a higher number of features than in the previous experiments.

The learning rate is 0.001 and we adopt the ADAM optimizer with decay rates $\beta_1 = 0.9$ and $\beta_2 = 0.999$ [78]. Since different observation windows mean a different size for the dataset, we adjust the number of epochs to train the networks with around 8000 training steps in all cases. Finally, we adopt a batch size of 100 samples.

EXPERIMENTS AND RESULTS

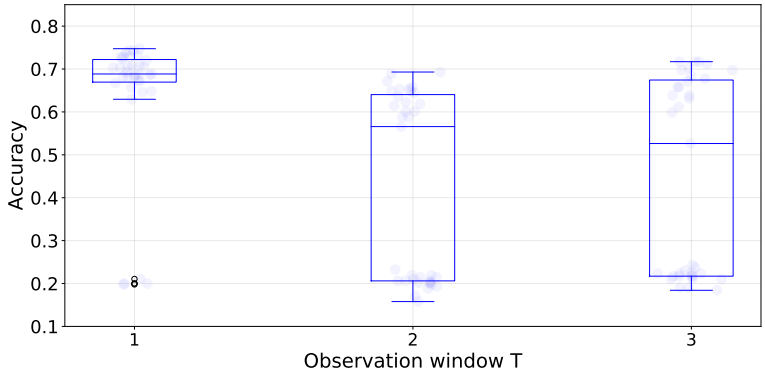


Figure 4.5: Baselines models. (Left) the GCNN proposed in [17], which does not take into account multiple observations in time but rather a single observation. (Middle-right) The same GCNN but the graph signals evolving over time are features of a multi-dimensional graph signal.

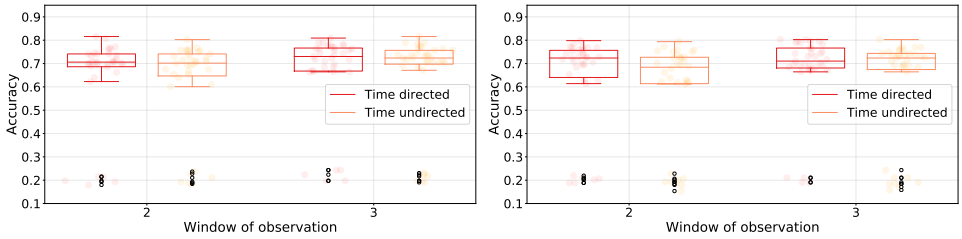
Baselines. We considered two baseline GCNNs. The first is a GCNN working with a single observation, where time is not taken into account as the one considered for source

localization in [17]. The second is a GCNN where different observations in time are regarded as features. In this second approach, each feature represents the value at a certain timestep t, thus modelling the time-varying graph signal $\boldsymbol{x}^{(t)}$ as a multi-dimensional graph signal with T features [3]. Both these cases are processed using the GCNN of Section 2.2. Figure 4.5 shows boxplots for each baseline method. First of all, it is worth noticing that the models do not always learn and behave similarly to a random classifier (20% accuracy) in several cases. However, this happens more frequently when the number of input features in the GCNN increases. We believe this is because of missing information about the relationships between data instances over time since they are just considered features.

We observe the approach proposed in [17] outperforms the other two baselines that consider the observations in time as features. In the approach of [17], time is not taken into account, and the classification is performed based on a single observation $\boldsymbol{x}^{(t)}$ at an unknown timestep t. This result shows that just considering observations in time as features of each node does not help the neural network improve its performance. On the contrary, it leads to worse performance. We believe this is because such an approach causes an increase in the input dimensionality, without providing information about the relationship between these additional features. This finding further supports our decision to explicitly model time through our parametric product graph architecture, which we investigate next.

Non-parametric product graphs. To investigate our approach of explicitly modelling time using a directed cyclic graph and graph products (see Section 4.2), we started with the case of a fixed (non-parametric) product graph. Recalling Section 2.3.1, valid options are the Kronecker, the Cartesian, and the strong product graphs. We experimented only with the Cartesian and the Strong product graphs since the Kronecker product graph does not retain the spatial edges of the nominal graph $\mathcal{G}$ [cf. Figure 2.9]. We also investigated whether modelling time as a directed or undirected cyclic graph led to a different performance. Figure 4.6 shows the boxplots of such models. Although we do not observe any significant difference in the performance of these models, the Cartesian product graph models, shown in Figure 4.6 a) seem to achieve slightly higher accuracy. Moreover, these results show that modelling time as directed rather than undirected graph leads to an increase in performance. A possible explanation for this result is that the strong product graph models more graph-time interactions than the Cartesian product graph, since it has a higher number of edges. This may lead to unnecessary connections and, therefore, worsen the performance.

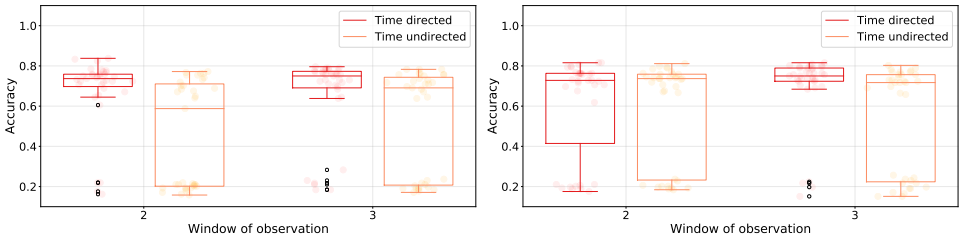
Parametric product graph. Next, we show the performance of the models which use a parametric product graph [cf. Section 4.2]. We investigate the behaviour of the models when initializing the s_{ij} coefficients [cf. (2.25)] to either one or a random number between 0 and 1. Moreover, we investigate the effect of modelling time as a directed or undirected cyclic graph. In Figure 4.7, we see the models perform better when modelling time as a directed graph. In fact, models which use an undirected graph fail to learn more often. A possible motivation for this behaviour is that by modelling time as a directed graph, the resulting graph-time product graph has fewer connections than in the case of time being modelled as an undirected path graph. Moreover, in the case of a directed time graph, the graph-time interactions modelled by the product graph follow



(a) Cartesian product graph.

(b) Strong product graph.

Figure 4.6: Non-parametric models. In both cases, we tested the models with observation windows of two and three timesteps and modelled time either as directed or undirected. **a)** Cartesian product graph. **b)** Strong product graph.



(a) s_{ij} coefficients initialized at one.

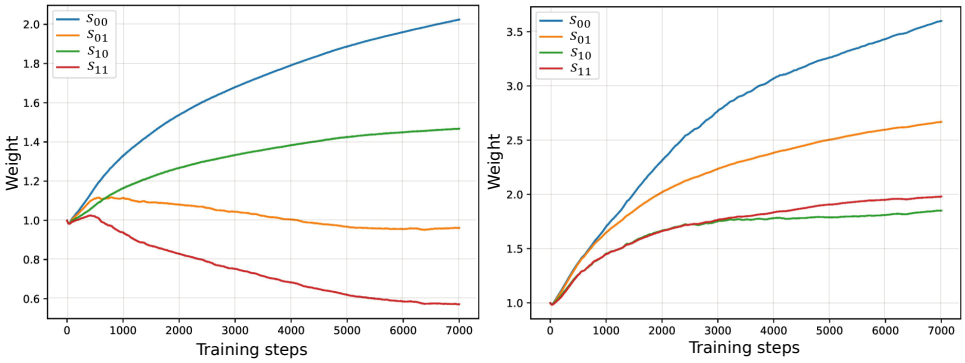
(b) s_{ij} coefficients initialized at random.

Figure 4.7: Parametric models. In both cases, we tested the models with observation windows of two and three timesteps and modelled time either as directed or undirected. **a)** s_{ij} coefficients initialized at one. **b)** s_{ij} coefficients initialized at random. Note that no sparse regularizer is adopted in these experiments, i.e., $\beta = 0$.

causality, and this is not true for the undirected case. Furthermore, we do not experience significant differences in the behaviour of the models when changing the initialization of the s_{ij} parameters. For the rest of this thesis, we will then consider time as a directed graph and initialize the s_{ij} coefficients to one, which implies the parametric product graph starts as a strong product graph with the addition of self-loops [cf. (2.25)].

With the parametric product graph, we can also track the learning of the four parameters s_{ij} (in each layer) weighting the different building blocks [cf. (2.25)]. By doing this, we can understand how the structure of the parametric product graph evolves over training. We show one of the most common situations when learning these coefficients in Figure 4.8. The model seems to give high importance to the self-loop parameter s_{00}, thus suggesting that the model maintains information about the initial value at a specific node when performing graph filtering. Also, in the first layer, a common situation is that parameter s_{11}, related to the edges of the Kronecker graph product, is the weight which is lowered the most during training. Finally, we see that the learned product graphs do not achieve sparsity w.r.t the s_{ij} parameters when the sparse regularizer [cf. Section 4.5] is not employed.

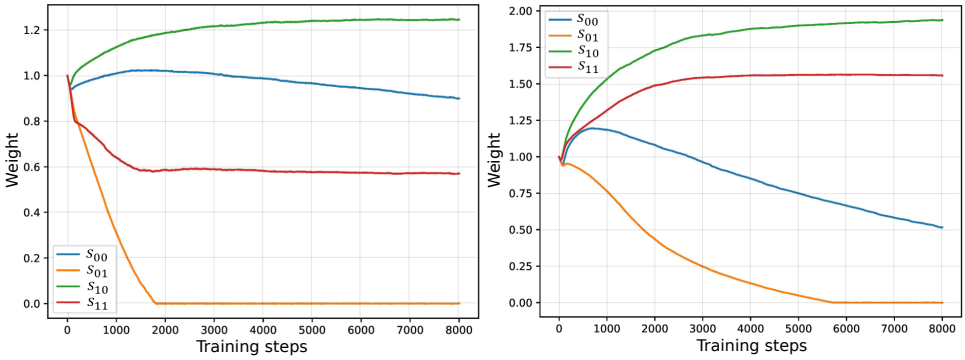
Sparse regularizer. Subsequently, we experiment with different values for the regularizer parameter β when adding the ℓ_1-norm regularization term in (4.18). Figure 4.10



(a) First layer's s_{ij} parameters.

(b) Second layer's s_{ij} parameters.

Figure 4.8: Parametric product graph learnable weights without the ℓ_1-norm regularization.



(a) First layer's s_{ij} parameters.

(b) Second layer's s_{ij} parameters.

Figure 4.9: Parametric product graph learnable weights with the ℓ_1-norm regularization. In this specific case, we choose $\beta = 0.0025$.

shows the result of applying this regularization technique to the best performing model obtained with previous experiments: parametric product graph with s_{ij} coefficients initialized to 1, modelling time as a directed path graph. When $\beta = 0$, the regularization is not present, and the loss of the GTCNN only compares the output $\hat{\boldsymbol{y}}$ with the label $\boldsymbol{y}$. As we increase β (the importance of the ℓ_1-norm regularization), the model seems to improve the accuracy on the test set, supporting our intuition about the effectiveness of learning a sparse parametric product graph. As expected, if β becomes too high, the accuracy of the model significantly drops as the model no longer minimizes the initial loss $\mathscr{L}(\hat{\boldsymbol{y}}, \boldsymbol{y})$, but rather the regularization loss $\mathscr{L}(\mathbf{s})$ [cf. (4.18)]. We provide additional details on this experiment in Table B.3 in Appendix B.

Finally, comparing Figure 4.9 with Figure 4.8, we notice coefficients s_{ij} differ when adopting the ℓ_1-norm regularization: while in the first case the self-loop building block

(s_{00}) received the highest weight in both layers, this is true for the s_{10} coefficient when using the sparse regularizer. Moreover, when using regularization, the coefficient s_{01} goes to zero in both layers, thus supporting the fact that a sparse parametric product graph can achieve higher performance. Moreover, we see that the sparse regularizer also promotes the convergence of the s_{ij} parameters. In fact, comparing Figure 4.9 with Figure 4.8, we see that the coefficients seem to converge faster when the regularizer is adopted. To conclude, we highlight that the resulting product graphs learned in Figures 4.8 and 4.9 are different from any product graph obtained using a fixed graph product (Kronecker, Cartesian, or Strong).

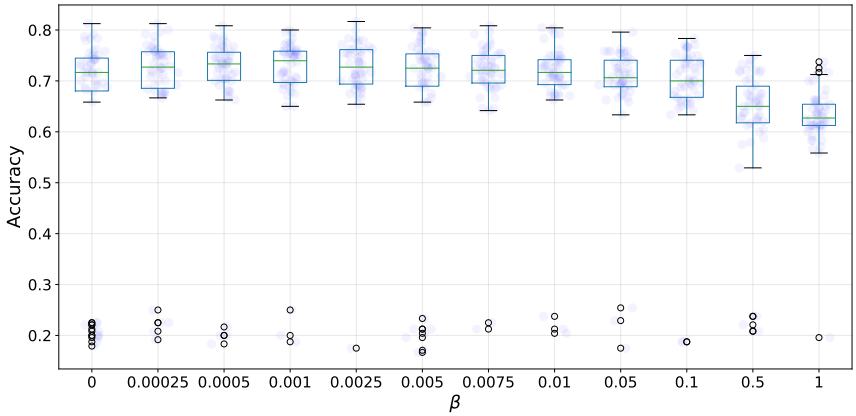


Figure 4.10: Best performing parametric model (s_{ij} coefficients initialized to one, time modelled as *directed* graph) with ℓ_1-norm regularization term. The values of β range from zero (no regularization) to one.

Impact of pooling. We now investigate the impact of the graph-time pooling component, which was absent in the previous models. We keep the number of layers of the GTCNN fixed at two, to offer a fair comparison with the models above. Given the higher number of hyperparameters of the GTCNN compared to the other models, we limit ourselves to experimenting with an observation window $T = 2$. We set the pooling slicing ratios to $R_1 = 1$, $R_2 = 2$. This means all the $T_0 = T$ time slices of the input are kept after the first convolutional layer, while only one slice out of two is kept after the second convolutional layer. We start with a GTCNN similar to the models previously introduced with $F_1 = 2$ and $N_1 = N = 100$ and assess the model by varying the number of features F_2 and the number of nodes N_2 kept at the second layer. To be more precise, we choose $F_2 \in \{2,4,16,32\}$ and $N_2 \in \{10,30,50\}$, representing 10%, 30%, and 50% of the total number of nodes. These results are shown in Figure 4.11. We see the performance is higher for $F_2 > 4$ and $N_2 \geq 30$. For these hyperparameters, we do not see any significant performance difference. Therefore, we choose $F_2 = 16$ and $N_2 = 30$, thus obtaining a model without too many parameters.

Next, we explore options for the hyperparameters F_1, N_1 of the first layer. Precisely, we evaluate $F_1 = \{2,4,16,32\}$ and $N_1 = \{30,75,100\}$, as we want to keep $N_1 \geq N_2$. These results are shown in Figure 4.12.

First, we see that the accuracy worsens when the number of features at the first layer grows. This is especially true for the case when $F_1 = 32$, suggesting that the number of features at the first layer should not be too high, and can be increased in deeper layers. Second, we see that the GTCNN achieves the best performance (both in terms of median and spread) for $F_1 = 2$ and $N_1 = 100$. The latter result suggests that the convolutional features generated by the GTCNN at the first layer are important for the majority of the nodes and, thus, the number of nodes N_1 kept at the first layer should be close to the total number of nodes N.

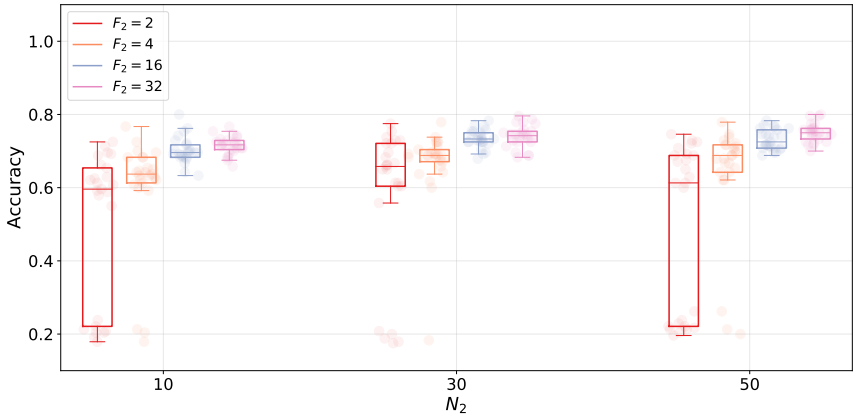


Figure 4.11: GTCNN experiments with varying F_2 and N_2. The other hyperparameters are fixed as $F_1 = 2$, $N_1 = 100$.

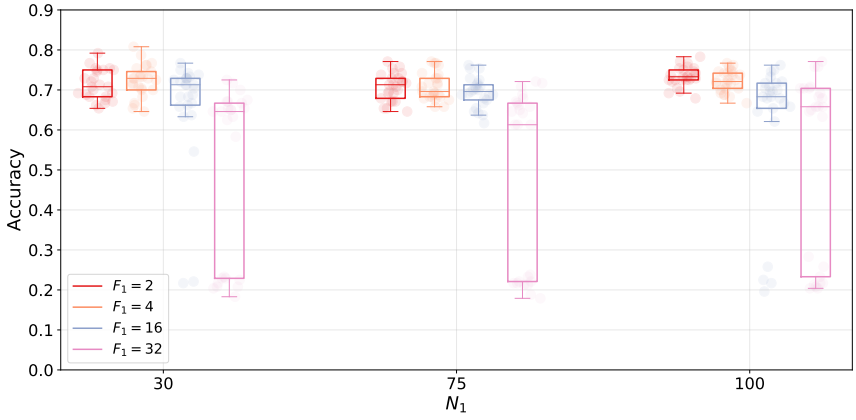


Figure 4.12: GTCNN experiments with varying F_1 and N_1. The other hyperparameters are fixed as $F_2 = 16$, $N_2 = 30$.

Overall comparison. We now compare the best performing model of each category (baselines, non-parametric product graphs, parametric product graphs, GTCNNs). For

specific categories, such as the non-parametric models, there is not an outstanding model to pick, as the performance is relatively similar across the different settings.

We report the mean, median, and standard deviation of the accuracy of these four models in Table 4.1. When adopting product graphs to model the time-varying graph signals, we notice we obtain a higher average accuracy than the best baseline. We also see an increase in performance when using a parametric product graph rather than a fixed product graph. More interestingly, we see the GTCNN model can achieve a significantly higher average accuracy while, at the same time, yielding a smaller standard deviation. This result means the GTCNN can perform well across different realizations of the SBM graph for the source localization, better than the other models.

Table 4.1: Average accuracy and its standard deviation of the best performing model for each category.

Model	Average Accuracy	Standard Deviation	Median Accuracy
Baseline	0.642	0.16	0.688
Non-parametric	0.67	0.174	0.711
Parametric	0.693	0.182	0.737
GTCNN	**0.734**	**0.024**	0.734

4.7. CONCLUSION

In this chapter, we presented our approach for learning from time-varying graph signal. This model, which we call GTCNN, consists of layers that perform graph-time convolutions (Section 4.2), graph-time pooling (Section 4.3), and apply a non-linearity function. We also introduced in Section 4.5 a regularization term based on ℓ_1-norm that can enforce sparsity in the learned product graph at each layer.

We validated the effectiveness of the different components of the GTCNN in Section 4.6 with a synthetic dataset. We found that the use of product graphs is beneficial for the model, yielding a higher accuracy compared to employing the conventional GCNN with these values as features. We also found that the GTCNN architecture further improved the accuracy while, at the same time, yielding a lower standard deviation across different initializations. Our rationale behind this result is that, due to graph-time pooling, we can drastically reduce the number of parameters of the model while retaining the capability of capturing graph-time correlations. Moreover, we showed how to interpret the product graphs learned at each layer and how these graphs are affected by the sparsity regularization term. Over the next chapters, we will investigate the GTCNN model on real-world datasets for earthquake classification and temperature prediction.

5

EARTHQUAKES CLASSIFICATION

In this chapter, we investigate the GTCNN model in the context of earthquake classification. The chapter is structured as follows. Section 5.1 sets the context for the experiments. Since we cure a new dataset for these experiments, we provide in Section 5.2 a detailed overview of the steps taken to create the dataset. Section 5.3 details the properties of the dataset. Next, Section 5.4 describes the experiments performed.

5.1. INTRODUCTION

As discussed in Section 3.4, there are two main approaches for earthquake detection: trend-based and precursor-based. In this thesis, since we deal with short-term predictions (the classification is performed using data recording up to 20 seconds before the strike), we decided to adopt a precursor-based approach. That is, the data that we use for the classification consists of measurements thought to correlate with earthquake activity.

For the experiments of this chapter, we cured a dataset consisting of earthquakes happening in New Zealand between 2016 and 2020. In total, the dataset contains 4633 earthquakes. For each of these earthquakes, the data we collected consists of weak motion (velocity) measurements across a sensor network of 58 seismic stations, recorded at 100Hz. The label of each earthquake is the station closest to its epicentre. Next, we detail all the steps taken to cure such a dataset.

5.2. DATASET

In this section, we explain the steps taken to build the earthquake dataset used in this chapter. Section 5.2.1 describes the area captured by the dataset and the stations providing the measurements. Section 5.2.2 describes the process of gathering the earthquakes. Section 5.2.3 provides details about the seismic waves measured by the stations. Section 5.2.4 describes the labelling process. Section 5.2.5 introduces some preprocessing steps to obtain a more balanced dataset. Finally, Section 5.2.6 describes the process to create and weigh the graph for the sensor network.

5.2.1. AREA OF INTEREST AND SEISMIC STATIONS

First of all, we defined a Bounding Box[1] to determine the geographic area for our dataset. A Bounding Box is uniquely determined given the coordinates (latitude and longitude) of the bottom-left and upper-right corners. The area captured by the Bounding Box of Table 5.1 is depicted in Figure 5.1.

Coordinate type	Value
Bottom-left longitude	166.104
Bottom-left latitude	-47.749
Top-right longitude	178.990
Top-right latitude	-33.779

Table 5.1: Coordinates specifying the Bounding Box for the New Zealand earthquake dataset.

Figure 5.1: Area defined by the Bounding Box in Table 5.1.

[1] https://wiki.openstreetmap.org/wiki/Bounding_Box

Next, we identified several stations that provide seismic wave measurements. All details of the available stations can be found on the website of the International Federation of Digital Seismograph Networks (FDSN) [18]. The details of the $N = 58$ stations we considered are provided in Table D.1 in Appendix D. To obtain information about such stations, we used the 'Station Service' provided by the FDSN website.

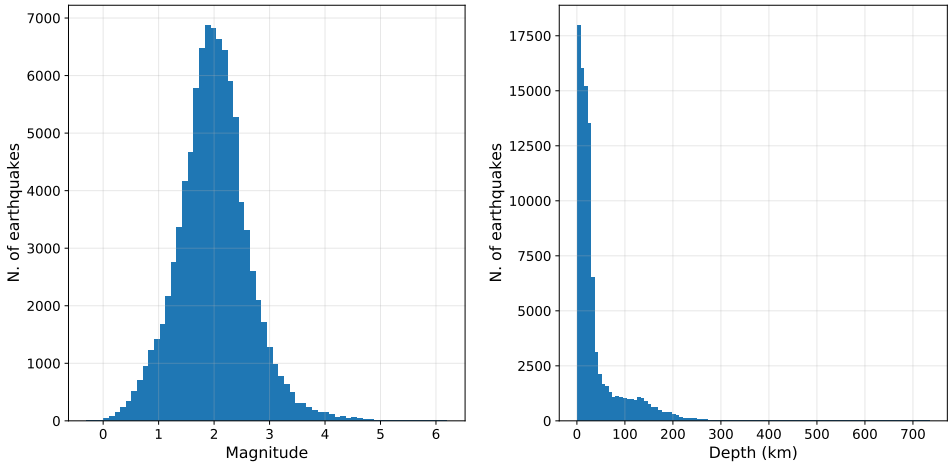
5.2.2. EARTHQUAKES

We then gathered a list of all the earthquakes that happened in the Bounding Box since the 1st of January, 2016[2]. This amounts to more than 90.000 earthquakes, which we show in Figure 5.2. For each earthquake, the available information consists of magnitude, coordinates (latitude and longitude) of the epicentre, and distance from the surface.

Figure 5.2: Earthquakes since 1st of January, 2016. Each yellow dot represents an earthquake in the dataset.

Initially, we investigated the distributions of magnitude and depth. We show the two respective histograms in Figures 5.3a and 5.3b. We can see that the majority of the earthquakes have magnitudes around two and a depth less than 200 kilometres. Therefore, to obtain a dataset consisting of earthquakes comparable between them, we retain earthquakes with a magnitude between one and three, and with depth less than 200. This choice is a trade-off between how many earthquakes we want to keep and how similar they are in the properties describing them, i.e., magnitude and depth. This filtering choice led us to approximately 87.000 earthquakes.

[2]We used the 'Event Service' [18] to obtain this information. This web service also provides data regarding previous years, but some of the stations were not active before 2016.



(a) Distribution of magnitude.

(b) Distribution of depth.

Figure 5.3: Histograms of magnitude and depth for the gathered earthquakes.

5.2.3. SEISMIC WAVES

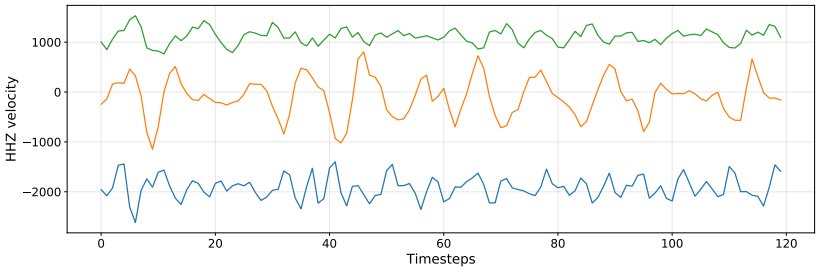
For each of the earthquakes, we adopted the Data Select Service [18] to obtain the waveform time series from 30 seconds before the earthquake timestamp until 30 seconds after it. We kept the default parameters for the query, i.e. *location* 10 and *channel* HHZ. This configuration implies the waveforms consist of weak motion (velocity) measured along the vertical axis and recorded at 100 Hz, i.e. 100 samples per second. We downsampled[3] the waveforms to obtain time series at 2 Hz, as in [3].

To summarize, for each earthquake, we collected 58 waveforms (one per station) each consisting of 120 samples (60 seconds of data). We discarded those data points for which the stations were not active in some timespans and retained only the earthquakes for which we had full data at all 58 stations. Figure 5.4 shows the waveforms for a randomly chosen earthquake. We note that different stations (different colours in the plot) have different ranges for the measurements. To account for the latter variability, we normalized the data station-wise.

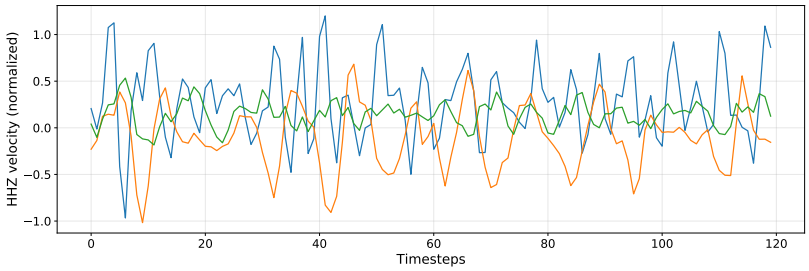
5.2.4. LABELLING PROCESS

Next, we labelled the earthquakes. We assigned each earthquake to its closest station based on the great-circle distance. We show in Figure 5.5 the distribution of such distance. Those earthquakes in the distribution tail whose distance from the closest station is high are the earthquakes that can be seen in the ocean, away from the coast, in Figure 5.2. We argue that a station label for such earthquakes is less meaningful than for those earthquakes happening near the stations. Therefore, we retain only the earthquakes whose distance from their closest station is below 75 kilometres. This choice leaves us with approximately 70.000 earthquakes. We show in Figure 5.6 two examples

[3] https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.resample.html



(a)



(b)

Figure 5.4: 60 seconds of data at 2 Hz. The earthquake is reportedly happening halfway through the recording. Each colour represents the waveform measured at a different seismic station. The waveforms shown in these plots belong to stations 1, 20, and 30. **a)** Raw measurements. Note how the mean of the waveforms differs for each station. **b)** Normalized measurements. The mean and standard deviation for normalization are computed per station.

of the result of our labelling process.

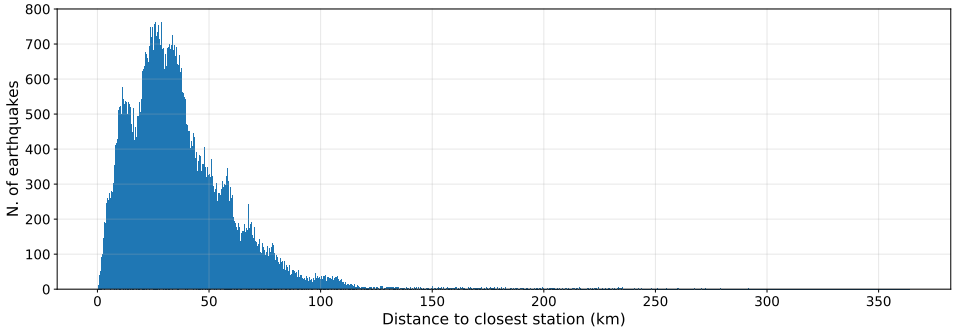


Figure 5.5: Histogram of the distance to the closest station for each earthquake.

5.2.5. FURTHER PREPROCESSING

The distribution of earthquakes is not uniform across New Zealand, as shown in Figure 5.2. Due to this, we observe a substantial imbalance among stations with regards to the number of assigned earthquakes. To be more precise, as we show in Figure 5.7, although only one station did not receive any earthquake, there are several stations which obtained less than 100 earthquakes. At the same time, 27 stations had more than 1000 earthquakes assigned. For a machine learning approach, imbalanced data is problematic since it can lead to poorer performance and additional hyperparameter tuning [79].

To mitigate the issue of imbalanced data, we performed two additional preprocessing steps. First, we discarded the stations (and the assigned earthquakes) with less than 150 earthquakes. Next, we randomly undersampled some earthquakes in the stations with a higher number of earthquakes. As a result, we obtained a more balanced dataset consisting of 45 station labels (out of the 58 stations available) and 4633 samples. We want to clarify that this filtering is performed only for the classes/labels. That is, the graph is still defined over $N = 58$ stations, but the final classes for the classification task will be only 45; those w.r.t. the stations for which we retained the data.

(a) Earthquake #16981 (b) Earthquake #17408

Figure 5.6: Labelling process for two earthquakes. The small pink dots represent stations. The yellow dot represents the earthquake of interest, while the red dot represents its closest station (the station chosen as label).

5.2.6. GRAPH CONSTRUCTION

There are many possible choices to create a graph for the sensor network. Since there is no common acknowledged approach to obtain such a graph, we decided to compute pairwise distances between the N available stations and find a threshold such that the average degree of the obtained graph is approximately ten. We adopted the *great-circle distance* [80] to measure the distance between the stations. This approach led to a threshold of 170.29 kilometres, obtaining the adjacency matrix shown in Figure 5.8a. In

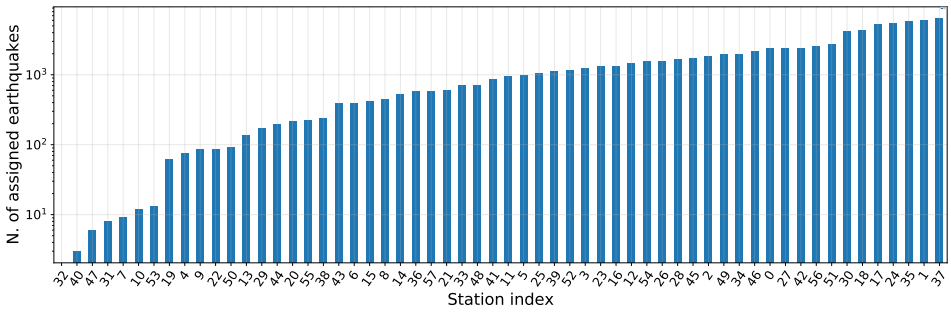
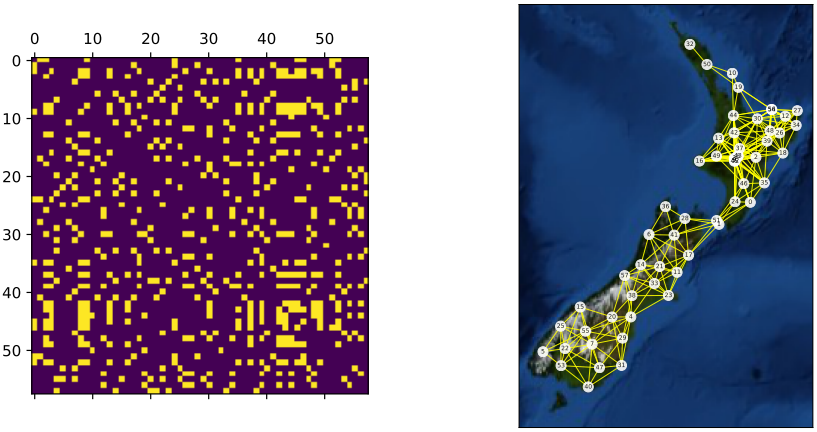


Figure 5.7: Count of assigned earthquakes for each station.

other words, there is an edge connecting two stations if their great-circle distance is below 170.29 kilometres. In Figure 5.8b, we show the corresponding graph, which provides a good qualitative representation of the area of interest. We again highlight that other choices are possible to construct the graph, such as using a nearest neighbour-based approach or choosing a different desired average degree.



(a) Adjacency matrix of the sensor network

(b) Sensor network

Figure 5.8: Graph representing the relationship between stations

Moreover, we decided to weigh the adjacency matrix following the approach adopted for the Molene dataset in Section 6.2. That is, we denote by $d(i, j)$ the great-circle distance between stations i and j, and by $\bar{d}$ the average great-circle distance among the connected stations (those stations whose great-circle distance is below the chosen threshold of 170.29 kilometres). The $N \times N$ weighted adjacency matrix $\boldsymbol{A}$ has entries of the form:

$$A_{ij} = e^{-d(i,j)/\bar{d}}.$$

Notice that $A_{ij} \in [0, 1]$, where smaller values indicate a higher distance between stations. Thus, our hypothesis is that the closer the stations, the stronger their connection.

5.3. DATASET PROPERTIES

The final dataset consists of 4633 data samples and 45 classes. We discuss in this section the properties of the obtained dataset. Figure 5.9 shows the dataset. We can see the graph structure together with the earthquakes resulting from the filtering and labelling process. We can also see that we still have many earthquakes happening in the ocean.However, they are not as far away from the coast as in Figure 5.2. Figure 5.10

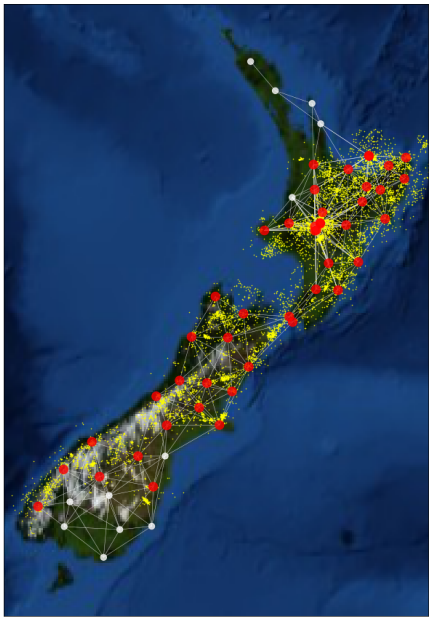


Figure 5.9: Summary of the earthquake data after filtering and labelling. Yellow dots represent earthquakes. White nodes of the graph represent stations whose labels have been discarded since their number of assigned earthquakes was below the selected threshold. Red nodes of the graph represent stations with enough earthquakes.

shows the number of earthquakes per year. Year 2017 is the one with the highest number of earthquakes. In Figures 5.11, 5.12, and 5.13, we show for each station the average magnitude, depth, and the average distance from the earthquakes to the station label. We infer from these plots that although there is no significant difference between classes with regards to the earthquake magnitudes, the average depth and the average distance from the labelled station exhibit significant difference among classes. This is because we mitigated this effect by filtering the earthquakes based on depth and magnitude in Section 5.2.2. In our opinion, these data distributions are satisfactory to run machine learning algorithms; further filtering would discard additional information which would lead to an even smaller dataset and hence affect the performance.

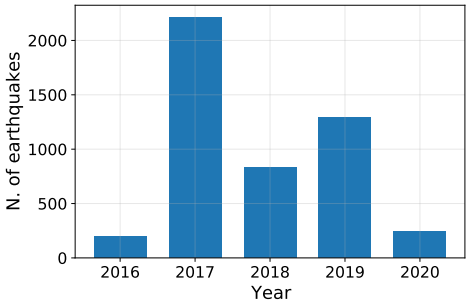


Figure 5.10: Number of earthquakes per year. The reason why fewer data samples were obtained from 2018 and 2019 is that many of the chosen stations were not active and we, therefore, discarded such incomplete samples.

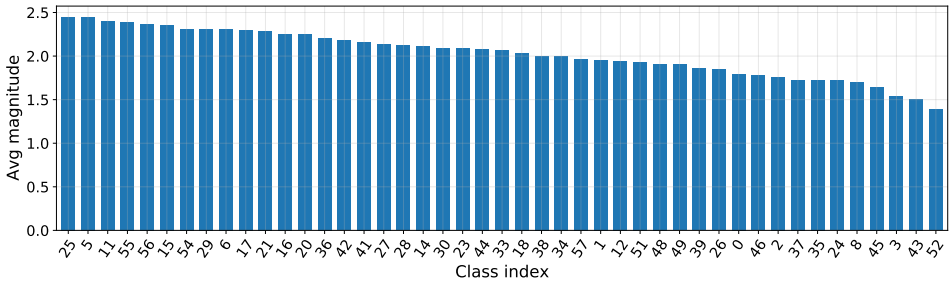


Figure 5.11: Average magnitude of earthquakes within each class.

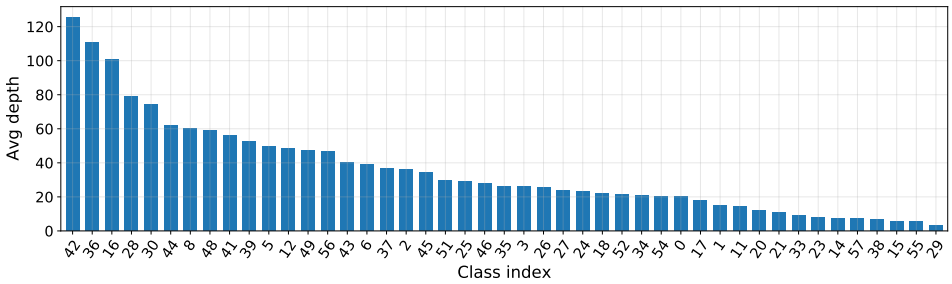


Figure 5.12: Average depth of earthquakes within each class.

5.4. EXPERIMENTS

Our objective is to investigate the GTCNN's capabilities for identifying the station closest to the earthquake's epicentre. This problem translates into a multi-class classification problem for the measured seismic wave. Specifically, we are interested in predicting the closest station given 10 seconds (20 timesteps) of the waveform before the reported earthquake timestamp.

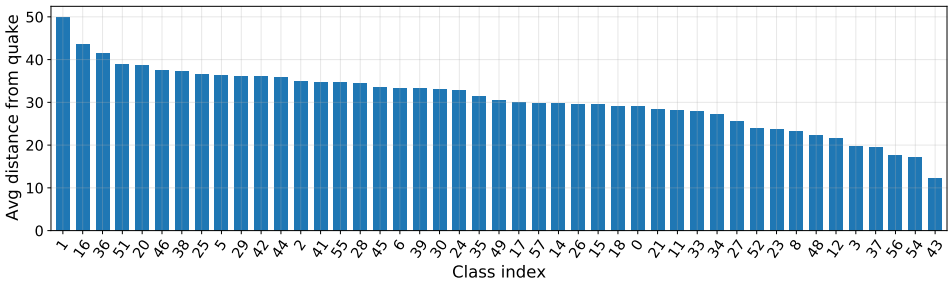


Figure 5.13: Average distance between earthquakes and corresponding label.

EXPERIMENTAL SETUP

We consider the weighted adjacency $\boldsymbol{A}$ to represent the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ between stations [cf. Section 5.2.6]. The input for the models consists of T graph signals $\boldsymbol{x}^{(t)}$, representing the 20 measurements prior to the strike. Each graph signal $\boldsymbol{x}^{(t)}$ has dimensionality $|\mathcal{V}| = N = 58$ since it contains the measurements across the sensor network at timestep t. In other words, for an earthquake happening at time T_e at station $c \in \{0, 1, \dots, 44\}$, we formulate the classification task as

$$[\boldsymbol{x}^{(T_e-20)}, \dots, \boldsymbol{x}^{(T_e-1)}] \xrightarrow[\mathcal{G}]{f(\cdot)} c\,,$$

where we stress the fact that the mapping $f(\cdot)$ learned by the GTCNN takes into account not only the input graph signals but also the graph $\mathcal{G}$.

Since our setup is a multi-class classification problem, a data sample is assigned to one station out of the possible 45. Such station is the one closest to the earthquake, according to the great-circle distance.

5.4.1. EVALUATION METRICS

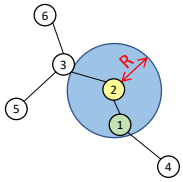
We measure the performance of the models through the accuracy, precision, recall, and f1 score. Due to the preprocessing steps [cf. Section 5.2], the dataset is not imbalanced. Therefore, we adopt the *macro-averaged* versions of these metrics. That is, each metric is computed for each class independently and then averaged to obtain the macro-averaged metric.

However, consider the multi-class problem and suppose the model predicts a station that is only a few kilometres away from the true station. Due to these stations being sufficiently close in space, one might still consider this prediction to be valid. Since the metrics mentioned above are not able to deal with this situation, we propose a type of evaluation metrics designed to this end.

RADIUS-BASED METRICS

We address this by considering as "correct" all the stations included in a circle of radius R, centered at the station of interest. More formally, let us consider the case of the i-th station. We can describe this set of stations as $\mathcal{R}_i^r = \{j : \text{dist}(i, j) < r\}$. Given the stations in $\mathcal{R}_i^r$, we can consider them as a single class and compute the standard

metrics w.r.t the new classes. For $r = 0$, the metrics defined w.r.t. $\mathscr{R}_i^0$ coincide with those for the 45 initial classes and do not take into account the radius-based approach. We show an example of this in Figure 5.14. Notice that although we achieved the desired



a)

True \ Predicted	1	2	3	4	5	6
1	4	2	2	1	1	1
2	2	5	1	1	1	1
3	3	2	7	0	1	2
4	0	0	0	5	2	3
5	0	1	1	1	6	4
6	2	1	2	1	2	8

b)

True \ Predicted	1 2	3	4	5	6
1 2	13	3	2	2	2
3	5	7	0	1	2
4	0	0	5	2	3
5	1	1	1	6	4
6	3	2	1	2	8

c)

Figure 5.14: Radius-based metrics. **a)** Example graph consisting of six stations. We select station two (yellow) and radius R. In this example, only station one (green) is included in the circle described by the radius R. **b)** The initial confusion matrix for a multi-class classification problem with six classes. **c)** The result of the confusion matrix transformation which takes into account a circle of radius R centered at station two. This can be seen as considering classes one and two as a new single class in the confusion matrix.

effect with the radius-based metrics, we can still have a different number of stations for a fixed radius R. In fact, dense areas, i.e., areas where stations are closer to each other, will include more stations in the circle compared to sparse areas. The only way to address this problem is to obtain a graph with stations uniformly distributed across the considered region. However, we shall not discuss this aspect further in this thesis and defer this analysis for future work.

5.4.2. MULTI-CLASS EXPERIMENTS AND RESULTS

EXPERIMENTAL SETTING

The input data for the models consists of 10 seconds (20 timesteps) of waveform data measured over the graph consisting of $N = 58$ seismic stations. We split the dataset into training, validation, and test sets using 60%, 20% and 20% of the data, respectively. We use the training set to learn the parameters, and the validation set to perform tuning and stop the training if the model's validation loss does not decrease for 20 epochs. The test set is used to compute the final results. We train the model for 100 epochs and we use a learning rate of 0.001. We considered the ADAM optimization algorithm and chose decay rates $\beta_1 = 0.9$ and $\beta_2 = 0.999$ [78]. The loss adopted is the cross-entropy loss (CE), introduced in Section 2.2.3. To account for the random initializations, we averaged the results over 20 iterations.

In this experiment, we compare the following models: (*i*) our proposed GTCNN; (*ii*) a standard LSTM model; and (*iii*) the GGRNN model [3]. We will additionally consider the random classifier as a baseline. The LSTM model [cf. Section 2.4.1] sees the waveform data as a multi-dimensional time series and does not use any information regard-

ing topology. The GGRNN model [cf. Section 2.4.2] consists of an RNN model where the internal matrix multiplications are replaced with graph convolutions and an attention mechanism is added to weight information differently along edges[4]. For the GTCNN and LSTM models, we grid searched the respective hyperparameters, while for the GGRNN we kept the hyperparameters from [3]: 20 features for the hidden state and 4 filter taps for each graph convolutional filter.

The grid search yielded the following configurations: the GTCNN consists of three layers with features $F_1 = 4$, $F_2 = 8$, $F_3 = 12$, filter taps $K_1 = K_2 = K_3 = 2$, slicing ratios $R_1 = R_2 = R_3 = 2$. Moreover, at the first layer 100% of the nodes per timestep are kept, while at the second and third layer the GTCNN retains 90% and 70% of the nodes per timestep, respectively; the LSTM has 20 hidden units.

RESULTS

In Figure 5.15, we show the radius-based accuracy for the above mentioned models. We see that the GTCNN has the highest accuracy, followed by the GGRNN model and the LSTM model. This result shows that exploiting the graph structure aids learning by reducing the number of parameters in the LSTM efficiently. Furthermore, all three models perform better than the random classifier. Although the models seem to maintain the same gap between the accuracies as we increase the radius, it is insightful to analyze their behaviour for a small radius.

In Figure 5.15b, we show a zoom of the first 25 kilometres for the radius-based accuracy. We can observe the GTCNN, LSTM, and GGRNN show a steeper increase in accuracy compared to the random classifier, meaning that several predictions which were considered wrong with a standard accuracy metric (when the radius is zero) are correct if we consider a radius of three kilometres around the target station. This finding confirms the usefulness of radius-based metrics. Interestingly, when observing this increase in accuracy, we can see how the GTCNN is the model exhibiting the steepest increase over the first kilometres. Nevertheless, we shall recall these accuracies are still far from satisfactory in practice. This is in part attributed to the complexity of the problem, and to the limited amount of data we are dealing with.

For the remaining radius-based metrics, we show them in Figure 5.16 for $R \in \{0, 20, 90\}$. We see that the GTCNN has the highest recall for the reported radiuses, while the precision and f1 is lower compared to the other models. However, this effect is mitigated for $R = 90$, where the precision and f1 of the GTCNN reach the level of the other models while maintaining a higher recall. We refer the reader to Appendix D for the full plots of radius-based precision, recall, and f1-score.

VALIDATION OF THE DOWNSAMPLING STEP

Notice in the above results we downsampled the waveforms from 100 Hz down to 2 Hz following [3]. This harsh down sampling represents information loss if we approach the problem from the signal reconstruction perspective. Likewise, one may also argue this downsampling is also responsible for the low performance reported in the previous section. We here analyze the classification accuracies when using the full waveform

[4] In [3], this model is referred to as e-GGRNN. It is the model achieving the best performance with earthquake data, and thus more suitable for our comparisons.

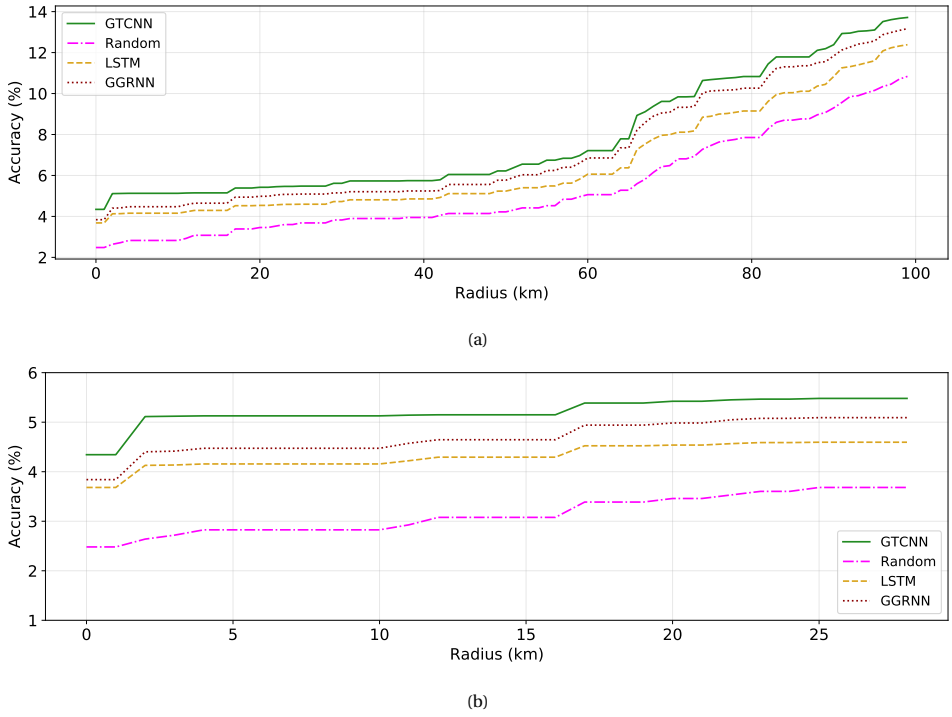


Figure 5.15: Radius-based accuracy for the models. We also compute this metrics for a random classifier as baseline. **a)** Radius-based accuracy for radius ranging from 0 km to 100 km. **b)** Zoom of the initial part of the plot in a), showing the different behaviour of the models for a slight increase in the radius.

of 100Hz. To investigate this, we trained the LSTM model on the raw data, and then compare its performance with the LSTM model trained on the downsampled data. We choose to compare the two models using the radius-based metrics of Section 5.4.1.

We report the radius-based accuracy of the models in Figure 5.17, while recall, precision, and f1 can be found in Appendix D (Figures D.2a, D.2b, and D.2c). We see that there is no significant difference in accuracy between the model trained on the downsampled data (2 Hz) compared to the model trained on the original data (100 Hz). Moreover, for larger radiuses the raw data led to a poorer performance when compared to the downsampled data. We believe this is because the downsampling step may filter out high-frequency noise contained in the raw data, which would make the classification harder. Therefore, it is safe to assume the downsampling step is not the main factor leading to low accuracies saw above, but that is somewhat due to the complexity of the problem.

5.4.3. TOWARDS A MORE SIMPLIFIED SETTING

We argue that the multi-class problem is a difficult one, arising from the fact that predicting (or classifying) earthquakes is inherently a hard task [74, 71]. On top of this, a multi-class experimental setting with 45 classes and only 4633 data points adds to this

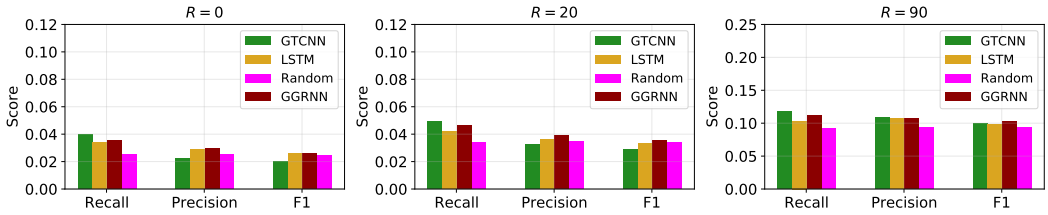


Figure 5.16: Radius-based metrics (precision, recall, f1) for the models. We compute such metrics with radius $R \in \{0, 20, 90\}$. Note the y-axis scale is different for the plot with $R = 90$.

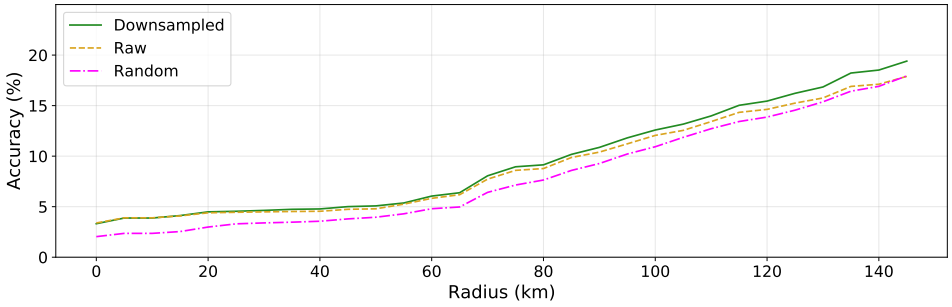


Figure 5.17: LSTM radius-based accuracy with and without downsampling (2 Hz data versus 100 Hz data). As a baseline, we also add the radius-based accuracy of a random classifier.

complexity. Although superior to a random classifier, all models are still far from a desired accuracy. Thus, we investigated the possibility of simplifying the problem without loss of generality, in order to gain more insights about the performance of the models.

We converted the multi-class classification problem (with C classes) into C binary classification problems, one per labelled station. This means a model is trained to predict whether the earthquake will occur at a particular station or somewhere else. Let us consider the binary classification problem for the i-th station, with label c_i. Suppose there are N_i data samples whose label is station c_i. We refer to these samples as samples assigned to the *positive* class. Then, to obtain a balanced dataset, we uniformly sample the same number of data points, i.e., N_i data points, from the remaining $C-1$ classes. We refer to these samples as samples assigned to the *negative* class. Then, the model is trained on this smaller dataset for the binary classification, i.e., a new dataset where we assign label 1 to the positive class and label 0 to the negative class.

EXPERIMENTAL SETTING AND RESULTS

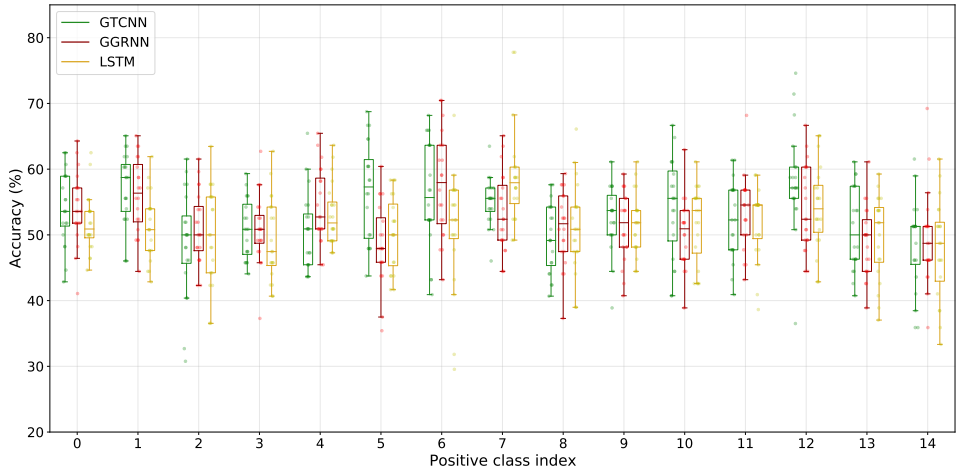
For these experiments, we decided not to perform grid search a second time and instead used the same hyperparameters found for the multi-class setting [cf. Section 5.4.2]. We compared the GTCNN, the LSTM, and the GGRNN models.

Figure 5.18 shows the boxplots of the accuracy for each binary classification problem, computed running 20 iterations in each case. Although the task is inherently hard,

we can see the proposed models can reach a median accuracy of about 60% in about 11 binary problems, with an accuracy hitting up to 70%, such as when the index of the positive class is 20, 34 or 43.

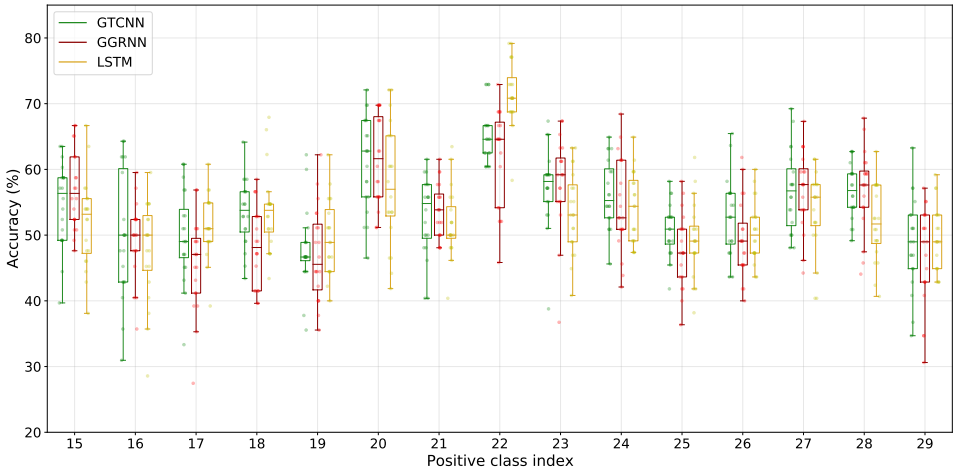
It is also clear that certain classes lead, on average, to higher performance than others. For example, when the positive class is class 22 or 43, all models obtain higher performance for all models. This result is consistent with the fact that these two stations are situated in less dense areas, and thus it should be intuitively easier to discriminate them from the other stations. The same consistent behaviour can be observed in the boxplots computed for the precision, recall, and f1 (Figure D.3, Figure D.4, and Figure D.5 in Appendix D).

To evaluate the different models on a global context, we also show in Figure 5.19 the histograms for the adopted metrics obtained across the 20 iterations of the 45 binary problems. That is, each model has 900 values in the histogram. We can see that although the performances of the GTCNN and the LSTM model are similar, the GTCNN histogram exhibits higher values of accuracy. We observe a similar behaviour when comparing the GTCNN and the GGRNN, although the performance of the two models is more similar. This result again indicates that the knowledge of the topology proved beneficial for the learning process.

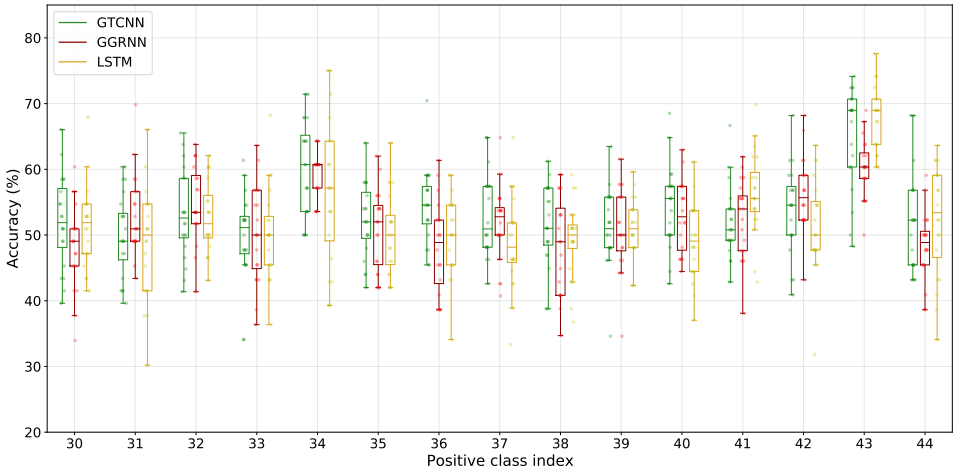


(a)

Figure 5.18: Boxplots of the accuracy for each binary classification setting. The number of iterations (per boxplot) is 20.

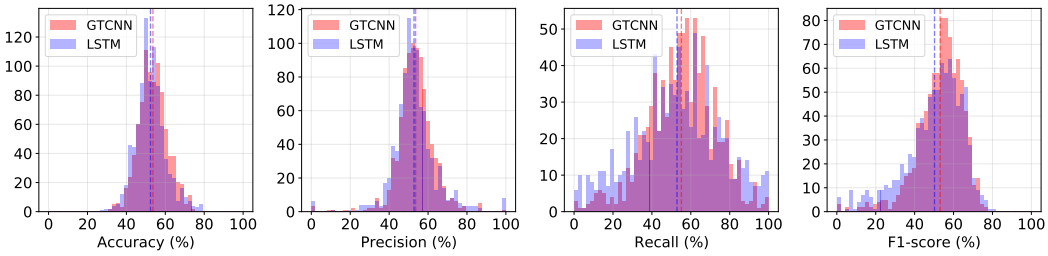


(b)

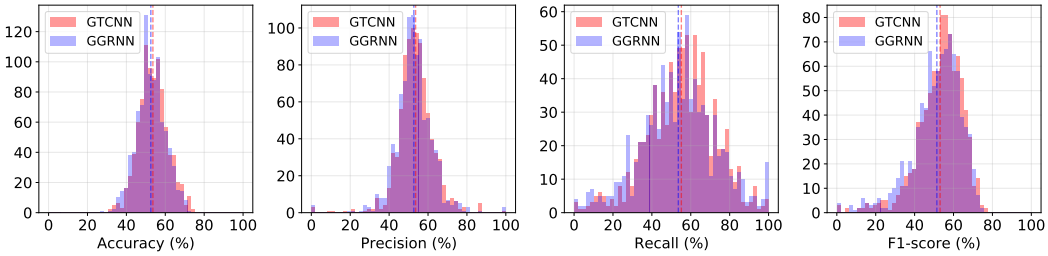


(c)

Figure 5.18: Boxplots of the accuracy for each binary classification setting (cont.). The number of iterations (per boxplot) is 20.



(a) Histogram of accuracy, precision, recall, and f1-score for the GTCNN and the LSTM models.



(b) Histogram of accuracy, precision, recall, and f1-score for the GTCNN and the GGRNN models.

Figure 5.19: Histogram of accuracy, precision, recall, and f1-score for the GTCNN, the LSTM, and the GGRNN models. For each of the 45 binary classification problems, a different positive class was chosen, and 20 iterations were performed. The vertical lines represent the average score for each model.

5.5. CONCLUSION

In this chapter, we tested the capability of the GTCNN for earthquake classification problems. For this, we used the FDSN service of New Zealand to gather a balanced dataset consisting of 4633 earthquakes across New Zealand [cf. Section 5.2]. Then, we experimented on this dataset with two different experimental settings. Firstly, we formulated the earthquake classification problem as a multi-class classification where the model predicts the station closest to the epicentre of the upcoming earthquake, out of the 45 available stations [cf. Section 5.4.2]. Secondly, we simplified the problem by transforming the multi-class classification into 45 binary classifications (one per station). In this second setting, the model is trained to predict whether the earthquake will happen at a particular station or somewhere else [cf. Section 5.4.3]. To better measure the performance of the models in the multi-class setting, we devised an approach to transform standard metrics to radius-based metrics, able to take into account the topology of the sensor network. This approach allows the metrics to discriminate between predictions that are entirely wrong (from a geographic perspective) and predictions that are still close to the correct station [cf. Section 5.4.2].

We compared the proposed GTCNN model with a conventional LSTM and a state-of-the-art graph-based RNN model called GGRNN. Our results on the multi-class experimental setting suggest that models that know the graph topology can achieve higher accuracy. In fact, both the GTCNN and the GGRNN both yield higher accuracy than

the LSTM for all the considered radius values. Moreover, when comparing the GTCNN and the GGRNN, it appears that our model can obtain slightly higher accuracy for all radius values. When looking at the binary classification setting, we noticed the models exhibit a significantly different performance based on which class is considered as *positive* class in the binary classification setting. This finding suggests that different stations have different properties, and further work is required to understand whether this stems from our process of data gathering or, instead, is a natural property of the earthquakes data. However, when observing the models' behaviour across all the binary problems, our proposed GTCNN appears to take on higher values for all the considered metrics from a global perspective.

A concluding remark for this chapter is that all the experiments have been performed with a monodimensional time-varying graph signal as input. That is, the input consists of weak motion (velocity) measurements across the sensor network over time. However, including different sources of data, such as temperature, pressure, etcetera, may be beneficial for the problems we dealt with in this chapter.

6

FORECASTING TIME-VARYING GRAPH SIGNALS

In this chapter, we evaluate the GTCNN capability of predicting time series with two real-world datasets: the Molene temperature dataset and the NOAA temperature dataset. The chapter is structured as follows. Section 6.1 formulates the problem of forecasting time-varying graph signals. Section 6.2 provides details about the two datasets. Section 6.3 describes the experimental setup. Section 6.4 presents the results and Section 6.5 concludes the chapter.

6.1. PROBLEM FORMULATION

In this forecasting task, the dataset consists of a graph representation matrix $\boldsymbol{A}$ (such as the Adjacency matrix) of $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and a collection of T graph signals $\boldsymbol{x}^{(t)} \in \mathbb{R}^N$, with $|\mathcal{V}| = N$. The GTCNN receives as input T_0 consecutive graph signals $\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \dots, \boldsymbol{x}^{(t-T_0+1)}$ and provides an estimate for h number of steps ahead $\hat{\boldsymbol{y}}^{(t+h)}$ of the true value $\boldsymbol{y}^{(t+h)} = \boldsymbol{x}^{(t+h)}$. More formally, the forecasting task is expressed as

$$[\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \dots, \boldsymbol{x}^{(t-T_0+1)}] \xrightarrow[\mathcal{G}]{f(\cdot)} \boldsymbol{x}^{(t+h)} \in \mathbb{R}^N .$$

Our aim is to learn function $f(\cdot)$, given a number of labeled examples.

Our models are optimized to make the prediction only at $h = 1$ step ahead. If $h > 1$, the predicted output is fed back into the GTCNN in an iterative fashion to reach a longer prediction horizon. As shown in [81], this method (called *iterated prediction*) is often reliable only for short prediction horizons, i.e., for small h values, since the model can quickly diverge due to accumulated errors. In these experiments, the maximum horizon considered is $h = 5$ (five hours ahead) likewise [7].

6.2. DATASETS

We consider the Molene [19] and the NOAA [20] datasets, which contain hourly temperature measurements across several locations. To make a fair comparison with other models, we follow the graph construction and preprocessing techniques adopted in [7] for both datasets.

For all the experiments, we adopt 35% of the data for training, 15% for validation, and 50% for testing. This choice is made to enable a fair comparison with the models in [7]. The split is performed sequentially. For instance, assume we have 100 successive hourly measurements: the first 35 measurements are used for training, the next 15 measurements are used for validation, and the last 50 measurements for testing. We refer to the union of the training and validation data as the "in-sample" data, while we refer to the test data as "out-of-sample" data. To provide a fair comparison between our approach and the ones reported in [7], we subtract the in-sample mean from the raw data for both datasets: we compute the in-sample mean for each station, and we subtract it from the raw data of such station[1].

MOLENE TEMPERATURES

The first dataset will be referred to throughout this section as "Molene". It provides $T = 744$ hourly measurements across $N = 32$ weather stations in the region of Brest, France. The data was recorded in January 2014. The graph is constructed starting from the node coordinates using a 10-NN approach and a Gaussian kernel weighting of the edges, as in [7]. The 10-NN graph is constructed based on the Euclidean distance between the stations, considering the Lambert II coordinates (x, y coordinates) and the altitude (in meters)[2]. We indicate as $\boldsymbol{A} \in \mathbb{R}^{N \times N}$ the weighted adjacency matrix of the

[1] Notice mean-centering time series is a standard procedure for forecasting [82].

[2] https://pygsp.readthedocs.io/en/stable/reference/graphs.html#pygsp.graphs.NNGraph

obtained graph. The edge weights are defined as

$$A_{ij} = e^{-d(i,j)/\bar{d}}, \tag{6.1}$$

where $d(i,j)$ is the Euclidean distance between stations i and j, and $\bar{d}$ is the average Euclidean distance in the dataset. Figure 6.1 shows the hourly measurements available at a specific station.

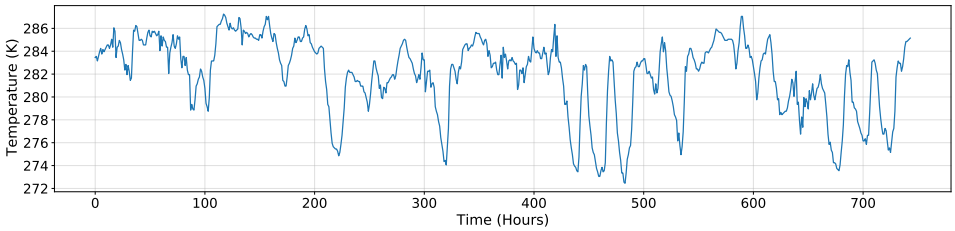


Figure 6.1: Example of time series on a specific station for the Molene dataset. Station id: 56007001. Station name: AURAY.

The Molene dataset contains little time series and, therefore, a strong prior is needed to aid learning. Our rationale is that full end-to-end solutions may fail in these circumstances, and we hypothesize our graph-based prior will help further.

NOAA TEMPERATURES

The NOAA dataset offers hourly temperature measurements across $N = 109$ weather stations in the United States, recorded over the year 2010. In total, this dataset provides $T = 8579$ hourly measurements, each consisting of 109 values. Therefore, the NOAA dataset is larger than the Molene dataset. To construct the graph, we follow the method adopted in [1], [26], and [7], which relies on the 7-NN geographical distances. Figure 6.2 shows the hourly measurements available at a specific station in the dataset for the first month.

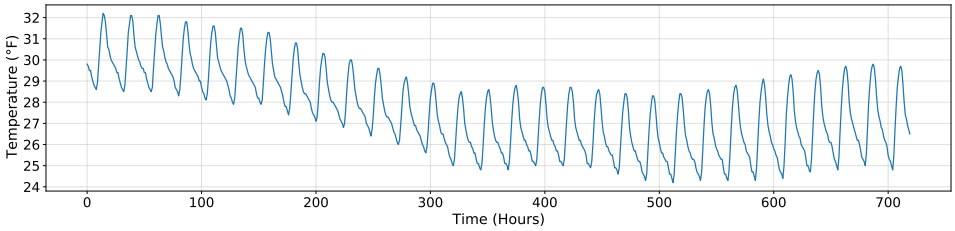
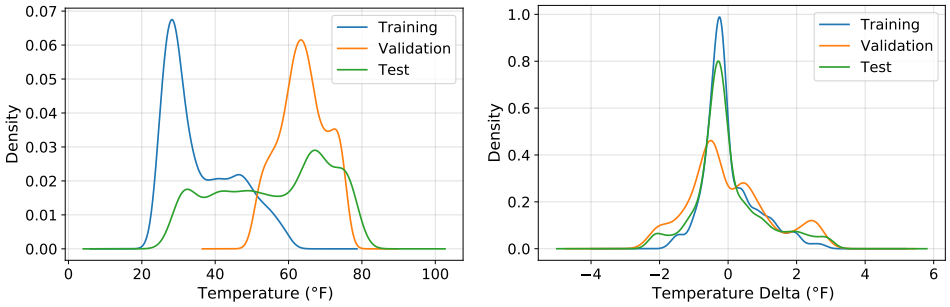


Figure 6.2: Example of time series (first month of the year) on a specific station for the NOAA dataset. Node 78.

Since the NOAA dataset contains data for one year only, it exhibits different distributions between training, validation and test sets. To be precise, data belonging to the beginning and the end of the year will be lower temperatures than the middle months.

This distribution problem is illustrated in Figure 6.3a. To address this issue, we adopt the technique of computing the first-order difference of the time series data [83]. That is, we subtract from the graph signal $\boldsymbol{x}^{(t)}$ the graph signal $\boldsymbol{x}^{(t-1)}$, for all the T timesteps available in the dataset. By doing this, we obtain the distributions shown in Figure 6.3b. We provide additional details in Appendix C.



(a) Original distributions.

(b) Distributions on differenced data.

Figure 6.3: NOAA distributions (training, validation and test) before and after differencing the data.

6.3. EXPERIMENTAL SETUP

TRAINING LOSS

When training neural networks for regression, there are some popular choices with regards to the loss function. The most common include Mean Squared Error [cf. (2.18)], the Mean Absolute Error (MAE[3]), or a combination of the two (Huber loss[4]). Since both these datasets do not contain outliers, we adopt the MSE as the training loss. Following (4.18), we can add an ℓ_1-based regularization term to encourage the GTCNN to learn a sparser product graph. For a set of data $\mathcal{T}$, our complete training loss is expressed as

$$\mathscr{L}_{\mathcal{T}} = \mathscr{L}_{\mathcal{T}}(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \beta \mathscr{L}(\mathbf{s}) = \frac{1}{R} \sum_{i \in \mathcal{T}} (\hat{\boldsymbol{y}}_i - \boldsymbol{y}_i)^2 + \beta \sum_{i,j,l} |s_{ij,l}|\,, \tag{6.2}$$

where R represents the number of datapoints, and β is a hyperparameter that controls the importance of the sparsity regularizer $\mathscr{L}(\mathbf{s})$ [cf. Section 4.5].

To compare the performance of the GTCNN to the graph-based forecasting methods in [7], we adopt the *root Normalized Mean Squared Error* (rNMSE) as our evaluation metric. We compute the rNMSE by comparing the true labels $\boldsymbol{y}$ and the corresponding predictions $\hat{\boldsymbol{y}}$:

$$\text{rNMSE}_{\mathcal{T}} = \sqrt{\frac{\sum_{i \in \mathcal{T}} \left\| \hat{\boldsymbol{y}}_i - \boldsymbol{y}_i \right\|_2^2}{\sum_{i \in \mathcal{T}} \left\| \boldsymbol{y}_i \right\|_2^2}}\,. \tag{6.3}$$

[3] https://pytorch.org/docs/stable/nn.html#torch.nn.L1Loss
[4] https://pytorch.org/docs/stable/nn.html#torch.nn.SmoothL1Loss

MODEL SELECTION

We selected the model yielding the lowest one step-ahead rNMSE on the validation set. Grid Search was used to choose the hyperparameters of the models. We adopted the ADAM optimizer with decay rates $\beta_1 = 0.9$ and $\beta_2 = 0.999$ and optimized the learning rate (from 0.01 to 0.0005). We also looked at the metrics on the validation set to decide when to stop training and avoid overfitting. For the GTCNN and the GGRNN models, we used the weighted adjacency matrix of the graph as GSO.

For the GTCNN, we experimented with networks consisting of two and three layers, followed by a fully connected layer. The features F in each layer are $F \in \{2,\dots,12\}$, the number of taps K in each layer are $K \in \{2,3,4\}$, the window of observation T_0, i.e., the number of graph signals in input, is $T_0 \in \{3,4,5\}$, the ℓ_1-norm sparsity regularizer has a weight β varying from 0 to 0.05. For the LSTM, we varied the number of hidden units from 8 to 64, and for the GGRNN we varied the number of state features $F \in \{2,\dots,20\}$ and the number of filter taps $K \in \{2,3,4,5\}$.

6.4. RESULTS

MOLENE

We compare the GTCNN model with the conventional LSTM network [cf. Section 2.4.1] and with the GGRNN [cf. Section 2.4.2]. We also compare our model with the two graph-based models in [7], named G-VARMA and GP-VAR [cf. Section 2.4.2].

Figure 6.4 shows the rNMSE for each step-ahead prediction. In general, we see that

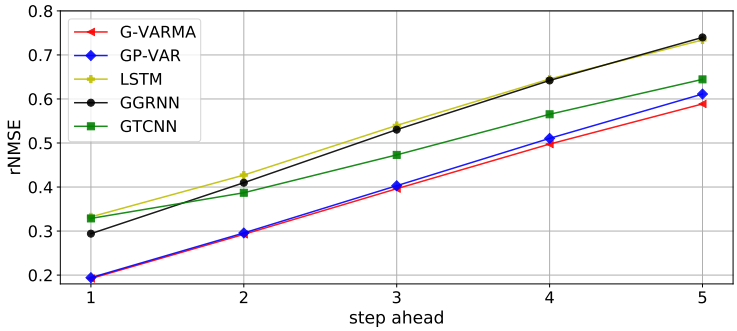


Figure 6.4: rNMSE versus step-ahead for the Molene dataset: comparison of the GTCNN model and other approaches proposed in the literature. G-VARMA and GP-VAR were introduced in [7]. The GGRNN was introduced in [3].

the linear model (G-VARMA and GP-VAR) outperform the neural network-based models. This may be due to the size of the dataset, which only contains a total of 744 data points. Furthermore, we see our GTCNN performs worse for shorter horizons, but better for longer horizons because it accumulates less error when predicting over longer horizons. This can be seen by looking at the slope of the lines in Figure 6.4. To better show this behaviour, we show in Figure 6.5 the rNMSE relative increment w.r.t. the one step-ahead prediction. We see that the neural network-based models show smaller increases

compared to the linear models, particularly in the case of the GTCNN. This result may be due to the GTCNN being more conservative and predicting values closer to the mean of the training data, thus diverging less when iterating to reach longer prediction horizons. A second motivation for this behaviour may be that the sparse regularizer on the product graphs reduces the overfitting on the one step-ahead, thus performing better for larger steps-ahead.

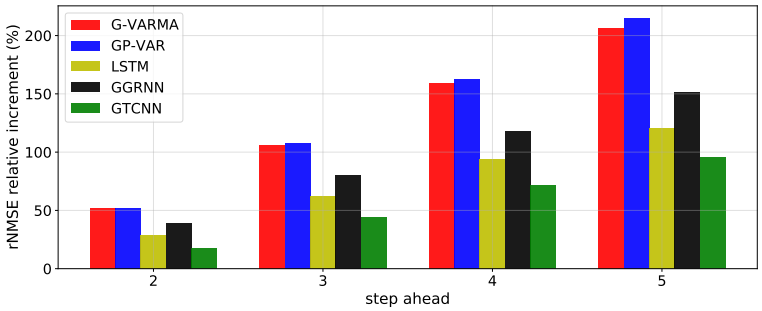


Figure 6.5: rNMSE relative increment (%) versus step-ahead for the Molene dataset. The relative increment is computed with respect to the rNMSE obtained for the one step-ahead prediction. The values used for this plot are the values shown in Figure 6.4.

The parameters of the GTCNN shown in Figure 6.4 are as follows: window of observation $T_0 = 4$; features at first layer $F_1 = 4$; features at second layer $F_2 = 12$; filter taps at first and second layer $K_1 = K_2 = 2$; slicing ratio at first layer $R_1 = 2$; slicing ratio at second layer $R_2 = 2$; number of nodes per timestep at first layer $N_1 = 25$; number of nodes per timestep at second layer $N_2 = 16$; ℓ_1 regularization weight $\beta = 0.00025$. For the LSTM, the model has 32 hidden units. For the GGRNN, the model has $F = 4$ state features and $K = 2$ filter taps. We refer the reader to Appendix C for additional visualizations regarding these experiments.

NOAA

We compare the GTCNN model against the same models chosen for the Molene dataset. We show the performance in Figure 6.6. We see that the nonlinear learning approaches (GTCNN, GGRNN, and LSTM) achieve better performance than the linear models. This result is not surprising since this dataset is much bigger than the Molene dataset, and this boosts the performance of the deep learning models over the linear ones. However, when comparing the LSTM and the GTCNN, we see the performance of the two models is very similar for shorter horizons, and we observe a lower rNMSE for the LSTM only for the four and five step-ahead predictions. This result suggests that the prior of the graph is not necessary when the size of the training data is large enough. An interesting result, however, is that the GTCNN model outperforms the other graph-aware models, confirming again that the proposed approach can learn from time-varying graph signals effectively. Finally, we show in Figure 6.7 the rNMSE relative increment w.r.t. the one step-ahead prediction. Differently from the Molene dataset, the model with the smallest increases is the LSTM, followed by the GTCNN and the other

graph-aware methods. This finding again suggests how the prior of the graph may not bring as many benefits when the amount of data is significantly larger.

The parameters of the GTCNN shown in Figure 6.2 are as follows: window of observation $T_0 = 4$; features at first layer $F_1 = 4$; features at second layer $F_2 = 12$; filter taps at first and second layer $K_1 = K_2 = 2$; slicing ratio at first layer $R_1 = 2$; slicing ratio at second layer $R_2 = 2$; number of nodes per timestep at first layer $N_1 = N = 109$; number of nodes per timestep at second layer $N_2 = 76$; ℓ_1 regularization weight $\beta = 0.00025$. For the LSTM, the model has 64 hidden units. For the GGRNN, the model has $F = 3$ state features and $K = 2$ filter taps.

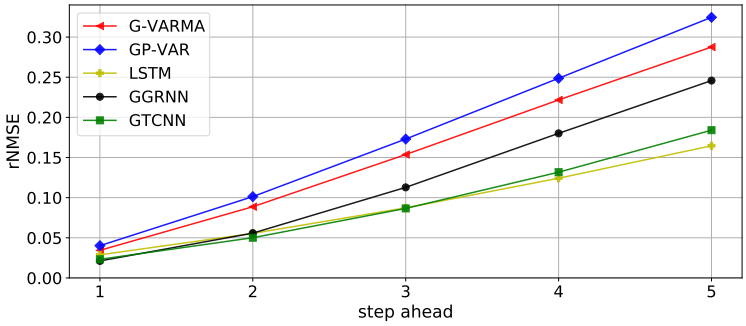


Figure 6.6: rNMSE versus step-ahead for the NOAA dataset: comparison of the GTCNN model and other approaches proposed in the literature. G-VARMA and GP-VAR were introduced in [7]. The GGRNN was introduced in [3].

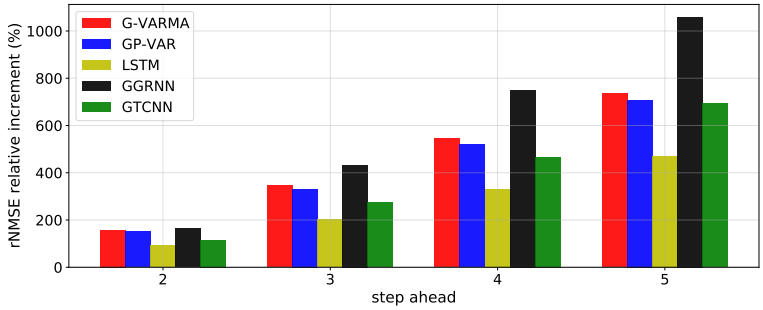


Figure 6.7: rNMSE relative increment (%) versus step-ahead for the NOAA dataset. The relative increment is computed with respect to the rNMSE obtained for the one step-ahead prediction. The values used for this plot are the values shown in Figure 6.6.

PARAMETRIC PRODUCT GRAPH VALIDATION

Without performing additional hyperparameter tuning, we investigated the performance of the GTCNN when adopting either the parametric formulation of the product graph [cf. (2.24)] of a fixed product graph. We show in Figure 6.8 the rNMSE for the

five different horizons. We see that the strong product graph leads to a lower rNMSE for longer horizons, while the Cartesian does not. We also notice that the parametric product graph follows the performance of the strong product graph for shorter horizons, and the performance of the Cartesian product graph for longer horizons. This result suggests that the parametric product graph formulation lets the GTCNN learn how to combine these product graphs and adapt them to the task at hand.

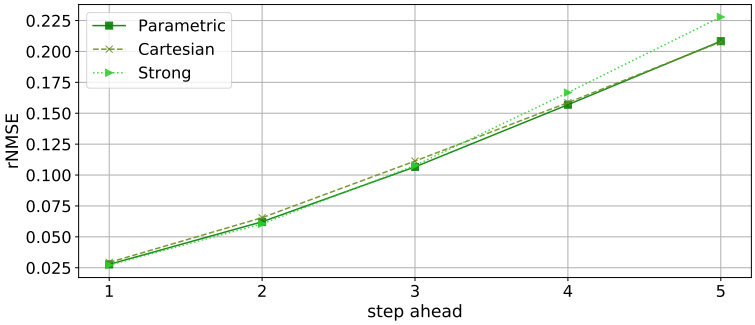


Figure 6.8: rNMSE versus step-ahead for the NOAA dataset: comparison of the parametric product graph, the Cartesian product graph, and the Strong product graph.

6.5. CONCLUSION

In this chapter, we investigated the applicability of the GTCNN to forecasting problems. We considered two datasets of different sizes, providing hourly temperature measurements across several locations. We see the deep-learning methods (GTCNN, GGRNN, and LSTM) perform better than the standard graph-based methods G-VARMA and GP-VAR on the bigger dataset (NOAA). This suggests the GTCNN is flexible enough to outperform standard graph-based approaches, provided enough data. This is an expected behaviour, since both the neural network-based models have a higher number of parameters and can learn more complex functions [79].

Furthermore, when looking at the LSTM and the GTCNN models, we notice that the gap between their performance is wider in the Molene setting (Figure 6.4) than in the case of NOAA (Figure 6.6). This suggests the prior knowledge provided by the graph is useful and can improve the performance of the GTCNN in case of shortage of data. Interestingly, we found for the Molene dataset that the GTCNN accumulates less error when iterating over the predictions to reach longer prediction horizons (higher step ahead). In other words, the rNMSE of the GTCNN grows with a smaller slope compared to the other models when predicting over longer horizons. On the NOAA dataset, this effect is less pronounced, and the slope of the rNMSE is similar for the models.

Finally, we evaluated the usefulness of employing a parametric product graph rather than a fixed product graph in the GTCNN architecture. We found that the performance of the parametric product graph seems to be a combination of the performances obtained by the fixed product graphs, suggesting that the GTCNN was able to learn the correct product graph for the task at hand.

7

CONCLUSION

In this chapter, we conclude the thesis and discuss possible future work. The chapter is structured as follows. Section 7.1 provides a summary of the work carried out in this thesis, outlining the results obtained. Section 7.2 answers the research questions posed in Chapter 1. Finally, Section 7.3 lays down possible future research directions that stem from this work.

7.1. THESIS SUMMARY

In Chapter 1, we motivated this research and set the context for the remaining chapters. Chapter 2 provided the background information, including Graph Convolutional Neural Networks and graph convolutional filters. Chapter 3 reviewed current literature w.r.t. learning of spatial and temporal dependencies, which play a fundamental role in learning from time-varying graph signals. In particular, we highlighted methods to capture either spatial or temporal dependency and the differences between hybrid and fused approaches when learning spatio-temporal dependency from the data.

We presented in Chapter 4 our main contribution, consisting of a novel convolutional-inspired neural network architecture that can learn from time-varying graph signals. Instead of combining different models to capture the spatial and temporal dependencies, the GTCNN constructs a parametric product graph to represent the time-varying graph signal as a standard graph signal on this larger graph. Then, it employs graph convolutions over the larger graph to generate features [cf. Section 4.2], and applies graph-time pooling to reduce the dimensionality and allow for a deeper structure [cf. Section 4.3]. We evaluated the different building blocks on the GTCNN in Section 4.6, showing the benefits of learning sparse parametric product graphs and providing an interpretation of the learned product graphs at each layer.

We then evaluated the proposed GTCNN for classification and regression using real-world datasets. In Chapter 5, we evaluated the GTCNN for earthquake classification and cured a real-world dataset for the experiments. Finally, in Chapter 6, we evaluated the GTCNN for temperature prediction on two different real-world datasets. In earthquake classification, we found that models using a prior about the data topology obtained a higher performance compared to graph-unaware models. In temperature forecasting, we saw that the models behaved differently on the two datasets, with neural network models (including the proposed GTCNN) performing best on the largest dataset.

7.2. ANSWERS TO RESEARCH QUESTIONS

In this section, we address the research questions posed in Section 1 based on the work carried out in this thesis.

(RQ1) *"How to learn the correct product graph for modelling signal variations over graph and time domains?"*

We answered this question in Chapter 4, where we included the parametric formulation of product graphs within the layer structure of the GTCNN architecture. The parametric product graph addresses the fact that different types of product graphs [cf. Section 2.3.1] are possible and lets the model learn which product graph best fits the task at hand [cf. Section 4.2]. The parametric formulation [cf. (2.24)] performs a weighted sum of the Kronecker, the Cartesian, and the Strong product graph (also considering self-loops) [cf. (2.25)], where the weights are learned during training. Moreover, we introduced an ℓ_1-norm regularization that can be added to the loss function during training [cf. Section 4.5] to enforce sparsity in these learned product graphs.

(RQ2) *"How to develop a graph convolutional architecture to jointly process time-varying graph signals over parametric product graphs?"*

We answered this question in Chapter 4, where we proposed the GTCNN model. The GTCNN includes two novel components: (i) the parametric graph-time convolutional layer that learns the parametric product graph and generates features through graph convolutions [cf. Section 4.2]; and (ii) the graph-time pooling layer, which can reduce the dimensionality along the graph domain (through downsampling) and the time domain (through slicing) [cf. Section 4.3]. During training, the neural network learns the graph convolutional filters as well as the coefficients to construct the parametric product graphs. The GTCNN is, therefore, able to perform convolutions over the learned graph-time product graph, thus learning how to aggregate and combine values both over the graph and over time.

(RQ3) *"How to use the developed architecture for classification and regression tasks?"*

We answered this question in Chapter 5 and Chapter 6, respectively. In Chapter 5, we evaluated the GTCNN's capability for predicting the station closest to the earthquake's epicentre, given a time-varying graph signal consisting of seismic measurements before the strike. We concluded that graph-aware models obtained a higher performance, indicating the structural knowledge is useful to aid learning. The proposed GTCNN achieved higher accuracy than other models, suggesting the approach of learning graph-time correlations through product graphs and graph convolutions is effective. In Chapter 6, we evaluated the GTCNN's capability for forecasting temperatures on two real-world datasets. We observed different behaviours for the two datasets, which differ in size. On the smaller dataset, linear models outperformed the neural network approaches, including the proposed GTCNN. However, the GTCNN accumulated less error when iterating over the predictions to reach longer horizons, showing higher robustness. On the bigger dataset, we found the opposite behaviour, with the neural network approaches outperforming the linear models, suggesting these more complex models are a reasonable choice only when there is enough training data. The latter result also suggested that the use of prior knowledge regarding the graph structure seems to be less beneficial in case of abundance of data.

Based on the answer to these research questions, we can now answer the thesis main question.

(RQ) *"How to learn meaningful representations from time-varying network data by means of graph convolutions and product graphs?"*

The proposed GTCNN effectively captured and exploited the graph-time correlations in time-varying graph signals by performing graph convolutions over graph-time product graphs learned directly from the data. The GTCNN uses these learnable product graphs as a prior about the graph-time interactions. The graph-time convolutional features generated by the GTCNN proved to be effective representations in both classification and regression tasks.

7.3. FUTURE WORK

To conclude this thesis, we discuss possible future research directions that stem from this work. In general, these suggestions pursue two goals. The first two extensions aim

to improve the GTCNN performance by increasing its flexibility and adaptability to the task at hand. The remaining research direction aims to create new experimental settings to better evaluate the capabilities of the proposed GTCNN architecture.

DILATED TIME MODELLING

One limitation of the proposed GTCNN architecture is that the parametric product graph assumes time interactions happen between values at adjacent timesteps. While this prior seems a reasonable assumption, it has one main drawback. Suppose the graph-time convolutions [cf. Section 4.2] need to aggregate temporal information from K timesteps ahead. This necessarily implies that a graph filter of order K that captures this temporal information will also capture spatial information from the K-hop neighbourhood. However, it would be useful for the GTCNN to learn graph filters that capture graph-time interactions at different temporal gaps, without necessarily increasing the neighbourhood along the spatial domain.

Building upon the idea of dilated temporal convolutions [52], it would be possible to model time according to different dilation factors. At each layer, instead of constructing a single graph-time parametric product graph $\boldsymbol{S}_l$ [cf. Section 4.2], the GTCNN could learn an arbitrary number of graph-time product graphs $\boldsymbol{S}_{l,d}$, each modelling time with a different time dilation factor d. With this new notation, it is straightforward to see that the proposed GTCNN adopts a time dilation factor $d = 1$. We show in Figure 7.1 two product graphs with different time dilation factors.

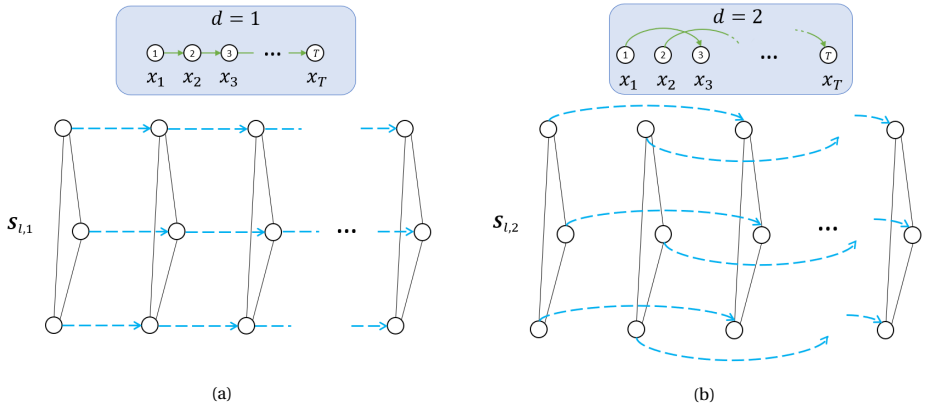


Figure 7.1: Two graph-time product graphs obtained with a different dilation factor d. **a)** A graph-time product graph obtained by modelling time with a dilation factor $d = 1$, equivalent to the formulation of the proposed GTCNN [cf. Chapter 4]. **b)** A graph-time product graph obtained by using a dilation factor $d = 2$. We note this graph connects over time each node with its second next version.

Following the construction of these product graphs $\boldsymbol{S}_{l,d}$, the convolutional features generated by performing graph convolution over each graph can be aggregated according to a certain criterion. For example, suppose the GTCNN constructs D product graphs at a given layer, each with a different time dilation factor d. Then, the aggregated convo-

lutional features could be defined as

$$\boldsymbol{u}^{f}_{l,\diamond} = \sum_{d=1}^{D} w_d \, \boldsymbol{u}^{f}_{l,\diamond,d} \,, \tag{7.1}$$

where $\boldsymbol{u}^{f}_{l,\diamond,d}$ denotes the convolutional features generated by performing graph convolutions [cf. (4.4)] over the product graph having time dilation factor d and w_d are learnable coefficients. This extension would extend the reach of the graph convolutions over time, without necessarily extending it also over the nominal graph. In fact, by considering different dilation factors, the order of the graph filters can remain low while capturing interactions between values that are distant in time.

HIGHER ORDER PRODUCT GRAPH

Throughout this thesis, we considered the parametric product graph formulation of (2.26), which we recall here:

$$\boldsymbol{S}_{\diamond} = \sum_{i=0}^{1} \sum_{j=0}^{1} s_{ij} (\boldsymbol{C}_T^i \otimes \boldsymbol{S}^j) \,.$$

A possible extension of the GTCNN would be to consider higher powers for the GSOs of the two factor graphs, yielding:

$$\boldsymbol{S}_{\diamond} = \sum_{i=0}^{K_c} \sum_{j=0}^{K_n} s_{ij} (\boldsymbol{C}_T^i \otimes \boldsymbol{S}^j) \,, \tag{7.2}$$

where K_c defines the maximum power considered w.r.t. the GSO $\boldsymbol{C}_T$ of the time graph, and K_n defines the maximum power considered w.r.t. the GSO $\boldsymbol{S}$ of the nominal graph.

Let us briefly discuss how this revisited formulation of the product graph would affect how the GTCNN models graph-time interactions. For the sake of this discussion, let us consider the adjacency matrix as GSO of a given graph. The k-th power of the adjacency matrix carries information about the number of k-hop walks between each pair of nodes in the graph [84]. Therefore, when considering higher powers of the GSO $\boldsymbol{C}_T$, the resulting product graph will also include time interactions between nodes that are non-adjacent along the temporal evolution. Analogously, when considering higher powers of the GSO $\boldsymbol{S}$, the resulting product graph will also include spatial interactions between nodes that are not direct neighbours in the nominal graph $\mathscr{G}$. However, detailed research is needed to reduce the computational cost of computing powers of the adjacency matrices as well as to understand the benefits of it.

MULTIDIMENSIONAL FEATURES FOR EARTHQUAKES CLASSIFICATION

In Chapter 5, we dealt with earthquakes classification. As detailed in Section 5.2.3, the input data consists of weak motion (velocity) measured along the vertical axis and recorded at 100 Hz. However, since earthquakes are complex phenomena, there may be a different set of measurements that represents a better input for the task we dealt with[1],

[1] A list of the available measurements can be found at https://www.geonet.org.nz/data/supplementary/channels in the "Channel Code" section.

such as temperature and pressure, to name just a few. Therefore, a future research direction on earthquakes classification may be pursued by providing to the GTCNN multidimensional measurements at each seismic station, and fully exploit the end-to-end learning capability of the GTCNN.

BIBLIOGRAPHY

[1] P. Di Lorenzo et al. "Adaptive graph signal processing: Algorithms and optimal sampling strategies". In: *IEEE Transactions on Signal Processing* 66.13 (2018), pp. 3584–3598.

[2] L. Zhao et al. "T-GCN: A Temporal Graph Convolutional Network for Traffic Prediction". In: *IEEE Transactions on Intelligent Transportation Systems* (2019), pp. 1–11. DOI: 10.1109/TITS.2019.2935152.

[3] L. Ruiz, F. Gama, and A. Ribeiro. "Gated Graph Recurrent Neural Networks". In: *arXiv* (2020), arXiv–2002.

[4] S. Wang, J. Cao, and P. S. Yu. "Deep learning for spatio-temporal data mining: A survey". In: *arXiv preprint arXiv:1906.04928* (2019).

[5] J. Ye et al. "How to Build a Graph-Based Deep Learning Architecture in Traffic Domain: A Survey". In: *arXiv preprint arXiv:2005.11691* (2020).

[6] Z. Wu et al. "A comprehensive survey on graph neural networks". In: *IEEE Transactions on Neural Networks and Learning Systems* (2020).

[7] E. Isufi et al. "Forecasting time series with varma recursions on graphs". In: *IEEE Transactions on Signal Processing* 67.18 (2019), pp. 4870–4885.

[8] E. Isufi, G. Leus, and P. Banelli. "2-dimensional finite impulse response graph-temporal filters". In: *2016 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*. IEEE. 2016, pp. 405–409.

[9] F. Manessi, A. Rozza, and M. Manzo. "Dynamic graph convolutional networks". In: *Pattern Recognition* 97 (Jan. 2020), p. 107000. ISSN: 0031-3203. DOI: 10.1016/j.patcog.2019.107000. URL: http://dx.doi.org/10.1016/j.patcog.2019.107000.

[10] T. Bogaerts et al. "A graph CNN-LSTM neural network for short and long-term traffic forecasting based on trajectory data". In: *Transportation Research Part C: Emerging Technologies* 112 (2020), pp. 62–77.

[11] Y. Seo et al. "Structured sequence modeling with graph convolutional recurrent networks". In: *International Conference on Neural Information Processing*. Springer. 2018, pp. 362–373.

[12] J. Zhang et al. "Gaan: Gated attention networks for learning on large and spatiotemporal graphs". In: *arXiv preprint arXiv:1803.07294* (2018).

[13] Z. Lu et al. "Leveraging Graph Neural Network with LSTM For Traffic Speed Prediction". In: *2019 IEEE SmartWorld, Ubiquitous Intelligence Computing, Advanced Trusted Computing, Scalable Computing Communications, Cloud Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP* 2019, pp. 74–81.

[14] A. Sandryhaila and J. M. Moura. “Big data analysis with signal processing on graphs”. In: *IEEE Signal Processing Magazine* 31.5 (2014), pp. 80–90.

[15] A. Natali, E. Isufi, and G. Leus. “Forecasting Multi-Dimensional Processes Over Graphs”. In: *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2020, pp. 5575–5579.

[16] A. Sandryhaila and J. M. Moura. “Discrete signal processing on graphs”. In: *IEEE transactions on signal processing* 61.7 (2013), pp. 1644–1656.

[17] F. Gama et al. “Convolutional neural network architectures for signals supported on graphs”. In: *IEEE Transactions on Signal Processing* 67.4 (2018), pp. 1034–1049.

[18] *FDSN webservice for New Zealand*. https://www.geonet.org.nz/data/tools/FDSN. Accessed: 2020-04-05.

[19] *Molene dataset*. https://www.data.gouv.fr/fr/datasets/donnees-horaires-des-55-stations-terrestres-de-la-zone-large-molene-sur-un-mois/. Accessed: 2020-02-12.

[20] A. Arguez et al. “NOAA’s 1981–2010 US climate normals: An overview”. In: *Bulletin of the American Meteorological Society* 93.11 (2012), pp. 1687–1697.

[21] A. Ortega et al. *Graph Signal Processing: Overview, Challenges and Applications*. 2017. arXiv: 1712.00468 [eess.SP].

[22] E. Isufi. “Graph-time signal processing: Filtering and sampling strategies”. In: (2019).

[23] D. I. Shuman et al. “The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains”. In: *IEEE signal processing magazine* 30.3 (2013), pp. 83–98.

[24] Y. Chauvin and D. E. Rumelhart. *Backpropagation: theory, architectures, and applications*. Psychology Press, 2013.

[25] R. Hammack, W. Imrich, and S. Klavžar. *Handbook of product graphs*. CRC press, 2011.

[26] D. Romero, V. N. Ioannidis, and G. B. Giannakis. “Kernel-based reconstruction of space-time functions on dynamic graphs”. In: *IEEE Journal of Selected Topics in Signal Processing* 11.6 (2017), pp. 856–869.

[27] F. Grassi et al. “A time-vertex signal processing framework: Scalable processing and meaningful representations for time-series on graphs”. In: *IEEE Transactions on Signal Processing* 66.3 (2017), pp. 817–829.

[28] H. Lütkepohl. “Forecasting with VARMA models”. In: *Handbook of economic forecasting* 1 (2006), pp. 287–325.

[29] H. Lütkepohl. *New introduction to multiple time series analysis*. Springer Science & Business Media, 2005.

[30] Y. Bengio, P. Simard, and P. Frasconi. “Learning long-term dependencies with gradient descent is difficult”. In: *IEEE transactions on neural networks* 5.2 (1994), pp. 157–166.

[31] Y. Yu et al. “A review of recurrent neural networks: LSTM cells and network architectures”. In: *Neural computation* 31.7 (2019), pp. 1235–1270.

[32] S. Hochreiter and J. Schmidhuber. “Long short-term memory”. In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[33] A. Krizhevsky, I. Sutskever, and G. E. Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[34] F. Gama et al. “Graphs, convolutions, and neural networks”. In: *arXiv preprint arXiv:2003.03777* (2020).

[35] Z. Wu et al. “Graph wavenet for deep spatial-temporal graph modeling”. In: *arXiv preprint arXiv:1906.00121* (2019).

[36] M. Defferrard, X. Bresson, and P. Vandergheynst. “Convolutional neural networks on graphs with fast localized spectral filtering”. In: *Advances in neural information processing systems*. 2016, pp. 3844–3852.

[37] E. Isufi, F. Gama, and A. Ribeiro. “Generalizing Graph Convolutional Neural Networks with Edge-Variant Recursions on Graphs”. In: *arXiv preprint arXiv:1903.01298* (2019).

[38] T. N. Kipf and M. Welling. “Semi-supervised classification with graph convolutional networks”. In: *arXiv preprint arXiv:1609.02907* (2016).

[39] P. Veličković et al. “Graph attention networks”. In: *arXiv preprint arXiv:1710.10903* (2017).

[40] J. Bruna et al. “Spectral networks and locally connected networks on graphs”. In: *arXiv preprint arXiv:1312.6203* (2013).

[41] M. Henaff, J. Bruna, and Y. LeCun. “Deep convolutional networks on graph-structured data”. In: *arXiv preprint arXiv:1506.05163* (2015).

[42] P. J. Brockwell and R. A. Davis. *Introduction to time series and forecasting*. springer, 2016, pp. 203–210.

[43] R. Fu, Z. Zhang, and L. Li. “Using LSTM and GRU neural network methods for traffic flow prediction”. In: *2016 31st Youth Academic Annual Conference of Chinese Association of Automation (YAC)*. 2016, pp. 324–328.

[44] V. Gómez and A. Maravall. “Estimation, prediction, and interpolation for nonstationary series with the Kalman filter”. In: *Journal of the American Statistical Association* 89.426 (1994), pp. 611–624.

[45] A. J. Smola and B. Schölkopf. “A tutorial on support vector regression”. In: *Statistics and computing* 14.3 (2004), pp. 199–222.

[46] M. T. Asif et al. “Spatiotemporal patterns in large-scale traffic speed prediction”. In: *IEEE Transactions on Intelligent Transportation Systems* 15.2 (2013), pp. 794–804.

[47] S. Sonoda and N. Murata. “Neural network with unbounded activation functions is universal approximator”. In: *Applied and Computational Harmonic Analysis* 43.2 (2017), pp. 233–268.

[48] Y. Tian and L. Pan. “Predicting short-term traffic flow by long short-term memory recurrent neural network”. In: *2015 IEEE international conference on smart city/SocialCom/SustainCom (SmartCity)*. IEEE. 2015, pp. 153–158.

[49] X. Ma et al. “Long short-term memory neural network for traffic speed prediction using remote microwave sensor data”. In: *Transportation Research Part C: Emerging Technologies* 54 (2015), pp. 187–197.

[50] R. Pascanu, T. Mikolov, and Y. Bengio. “Understanding the exploding gradient problem”. In: *CoRR, abs/1211.5063* 2 (2012), p. 417.

[51] K. Cho et al. “On the properties of neural machine translation: Encoder-decoder approaches”. In: *arXiv preprint arXiv:1409.1259* (2014).

[52] S. Bai, J. Z. Kolter, and V. Koltun. “An empirical evaluation of generic convolutional and recurrent networks for sequence modeling”. In: *arXiv preprint arXiv:1803.01271* (2018).

[53] X. Cao et al. “Interactive temporal recurrent convolution network for traffic prediction in data centers”. In: *IEEE Access* 6 (2017), pp. 5276–5289.

[54] J. Ke et al. “Short-term forecasting of passenger demand under on-demand ride services: A spatio-temporal deep learning approach”. In: *Transportation Research Part C: Emerging Technologies* 85 (2017), pp. 591–608.

[55] H. Yu et al. *Spatiotemporal Recurrent Convolutional Networks for Traffic Prediction in Transportation Networks*. 2017. arXiv: 1705.02699 [cs.LG].

[56] Y. Sun et al. “Constructing Geographic and Long-term Temporal Graph for Traffic Forecasting”. In: *arXiv preprint arXiv:2004.10958* (2020).

[57] D. Chai, L. Wang, and Q. Yang. “Bike flow prediction with multi-graph convolutional networks”. In: *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. 2018, pp. 397–400.

[58] M. Khodayar and J. Wang. “Spatio-temporal graph deep neural network for short-term wind speed forecasting”. In: *IEEE Transactions on Sustainable Energy* 10.2 (2018), pp. 670–681.

[59] S. Guo et al. “Attention based spatial-temporal graph convolutional networks for traffic flow forecasting”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 2019, pp. 922–929.

[60] L. Ge et al. “Temporal Graph Convolutional Networks for Traffic Speed Prediction Considering External Factors”. In: *2019 20th IEEE International Conference on Mobile Data Management (MDM)*. IEEE. 2019, pp. 234–242.

[61] K. Lee and W. Rhee. *DDP-GCN: Multi-Graph Convolutional Network for Spatiotemporal Traffic Forecasting*. 2019. arXiv: 1905.12256 [cs.LG].

[62] B. Yu, H. Yin, and Z. Zhu. “Spatio-temporal graph convolutional networks: A deep learning framework for traffic forecasting”. In: *arXiv preprint arXiv:1709.04875* (2017).

[63] F. Yu and V. Koltun. *Multi-Scale Context Aggregation by Dilated Convolutions*. 2015. arXiv: 1511.07122 [cs.CV].

[64] S. Fang et al. “Gstnet: Global spatial-temporal network for traffic flow prediction”. In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI*. 2019, pp. 10–16.

[65] Z. Wu, Y. Wang, and L. Zhang. “MSSTN: Multi-Scale Spatial Temporal Network for Air Pollution Prediction”. In: *2019 IEEE International Conference on Big Data (Big Data)*. 2019, pp. 1547–1556.

[66] Z. Diao et al. “Dynamic spatial-temporal graph convolutional neural networks for traffic forecasting”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 2019, pp. 890–897.

[67] Y. N. Dauphin et al. “Language modeling with gated convolutional networks”. In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 933–941.

[68] C. Si et al. “An attention enhanced graph convolutional lstm network for skeleton-based action recognition”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019, pp. 1227–1236.

[69] Y. Li et al. “Diffusion convolutional recurrent neural network: Data-driven traffic forecasting”. In: *arXiv preprint arXiv:1707.01926* (2017).

[70] S. Yan, Y. Xiong, and D. Lin. “Spatial temporal graph convolutional networks for skeleton-based action recognition”. In: *Thirty-second AAAI conference on artificial intelligence*. 2018.

[71] T. Bhandarkar et al. “Earthquake trend prediction using long short-term memory RNN”. In: *International Journal of Electrical and Computer Engineering* 9.2 (2019), p. 1304.

[72] T. Perol, M. Gharbi, and M. Denolle. “Convolutional neural network for earthquake detection and location”. In: *Science Advances* 4.2 (2018), e1700578.

[73] N. R. Council et al. *Predicting earthquakes: A scientific and technical evaluation, with implications for society*. National Academies, 1976, pp. 7–14.

[74] Q. Wang et al. “Earthquake prediction based on spatio-temporal data mining: an LSTM network approach”. In: *IEEE Transactions on Emerging Topics in Computing* (2017).

[75] A. Bhatia, S. Pasari, and A. Mehta. “EARTHQUAKE FORECASTING USING ARTIFICIAL NEURAL NETWORKS.” In: *International Archives of the Photogrammetry, Remote Sensing & Spatial Information Sciences* (2018).

[76] J. Friedman, T. Hastie, and R. Tibshirani. *The elements of statistical learning*. Vol. 1. 10. pg. 611. Springer series in statistics New York, 2001.

[77] A.-L. Barabási et al. *Network science*. Cambridge university press, 2016.

[78] D. P. Kingma and J. Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[79] W. Zheng and M. Jin. “The Effects of Class Imbalance and Training Data Size on Classifier Learning: An Empirical Study”. In: *SN Computer Science* 1.2 (2020), pp. 1–13.

[80] R. Bullock. “Great circle distances and bearings between two locations”. In: *MDT, June* 5 (2007).

[81] R. Boné and M. Crucianu. “Multi-step-ahead prediction with neural networks: a review”. In: *9emes rencontres internationales: Approches Connexionnistes en Sciences* 2 (2002), pp. 97–106.

[82] R. A. Yaffee and M. McGee. *An introduction to time series analysis and forecasting: with applications of SAS® and SPSS®*. Elsevier, 2000.

[83] R. Adhikari and R. K. Agrawal. “An introductory study on time series modeling and forecasting”. In: *arXiv preprint arXiv:1302.6613* (2013).

[84] A. Duncan. “Powers of the adjacency matrix and the walk matrix”. In: (2004).

[85] K. Schacke. “On the kronecker product”. In: *Master’s thesis, University of Waterloo* (2004).

A

KRONECKER PRODUCT

A

Given two matrices $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ and $\boldsymbol{B} \in \mathbb{R}^{p \times q}$, their *Kronecker product* $\boldsymbol{C} = \boldsymbol{A} \otimes \boldsymbol{B} \in \mathbb{R}^{mp \times nq}$ is obtained as [85]:

$$\boldsymbol{A} \otimes \boldsymbol{B} = \begin{bmatrix} a_{11}\boldsymbol{B} & \cdots & a_{1n}\boldsymbol{B} \\ \vdots & \ddots & \vdots \\ a_{m1}\boldsymbol{B} & \cdots & a_{mn}\boldsymbol{B} \end{bmatrix}.$$

Among all the properties of the Kronecker product, which are listed in [85], we report here an important one which we exploit in this work. Given four matrices $\boldsymbol{A} \in \mathbb{R}^{p \times q}$, $\boldsymbol{B} \in \mathbb{R}^{r \times s}$, $\boldsymbol{C} \in \mathbb{R}^{q \times k}$, $\boldsymbol{D} \in \mathbb{R}^{s \times l}$, it follows that:

$$(\boldsymbol{A} \otimes \boldsymbol{B})(\boldsymbol{C} \otimes \boldsymbol{D}) = \boldsymbol{AC} \otimes \boldsymbol{BD}. \tag{A.1}$$

Given two square matrices $\boldsymbol{A} \in \mathbb{R}^{n \times n}$, $\boldsymbol{B} \in \mathbb{R}^{m \times m}$, it follows from (A.1) that:

$$(\boldsymbol{A} \otimes \boldsymbol{B})^k = \boldsymbol{A}^k \otimes \boldsymbol{B}^k. \tag{A.2}$$

B

Source Localization Supplementary Material

B

B.1. NON-LEARNING RUNS

Most likely due to the initialization of the network parameters, it may happen that the models do not learn and behave similarly to a random classifier (they exhibit an accuracy close to $\frac{1}{C}$, where C is the number of classes). For the sake of completeness, we report in this Appendix some additional information about these runs. Table B.1 shows information about the baseline models adopted for the Source Localization task. Moreover, we show in Table B.2 detailed information about the runs involving product graph-based models (without graph-time pooling).

B.2. SPARSE REGULARIZATION

In Section 4.6, we showed the boxplots for different values on the parameter β controlling the importance of the sparse regularizer [cf. Figure 4.10]. We report here in Table B.3 the average accuracy and its standard deviation for these experiments.

Table B.1: Statistics about the baselines for the Source Localization task

Model	Window	Accuracy	# Failed
GCNN	1	0.642 (0.16)	4
GCNN	2	0.424 (0.217)	17
GCNN	3	0.446 (0.227)	17

Table B.2: Statistics about all the product graph-based models for the Source Localization task.

Parametric	Product type	Time	Init	Window	Accuracy	# Failed
No	Cartesian	Directed	-	2	0.645 (0.186)	5
No	Cartesian	Directed	-	3	0.662 (0.185)	5
No	Cartesian	Undirected	-	2	0.623 (0.194)	6
No	Cartesian	Undirected	-	3	0.646 (0.201)	6
No	Strong	Directed	-	2	0.634 (0.201)	6
No	Strong	Directed	-	3	0.67 (0.174)	4
No	Strong	Undirected	-	2	0.505 (0.262)	14
No	Strong	Undirected	-	3	0.608 (0.227)	8
Yes	-	Directed	1	2	0.677 (0.18)	4
Yes	-	Directed	1	3	0.655 (0.203)	6
Yes	-	Directed	Random	2	0.606 (0.244)	9
Yes	-	Directed	Random	3	0.693 (0.182)	4
Yes	-	Undirected	1	2	0.46 (0.26)	17
Yes	-	Undirected	1	3	0.53 (0.257)	13
Yes	-	Undirected	Random	2	0.589 (0.246)	10
Yes	-	Undirected	Random	3	0.539 (0.262)	13

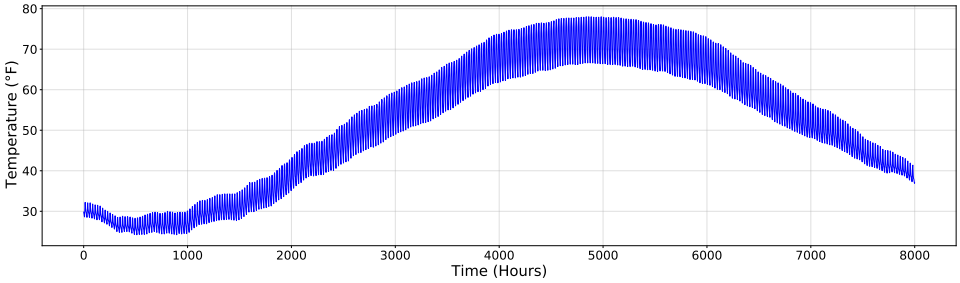
Table B.3: Average accuracy and standard deviation of the experiments on the ℓ_1-norm regularization.

β	0	2.5×10^{-4}	5×10^{-4}	0.001	**0.0025**	0.005	0.0075	0.01	0.05	0.1	0.5	1
Avg. Acc.	0.616	0.681	0.69	0.702	**0.718**	0.658	0.704	0.694	0.684	0.688	0.616	0.626
Std.	0.22	0.158	0.148	0.129	**0.086**	0.188	0.106	0.126	0.122	0.11	0.139	0.073

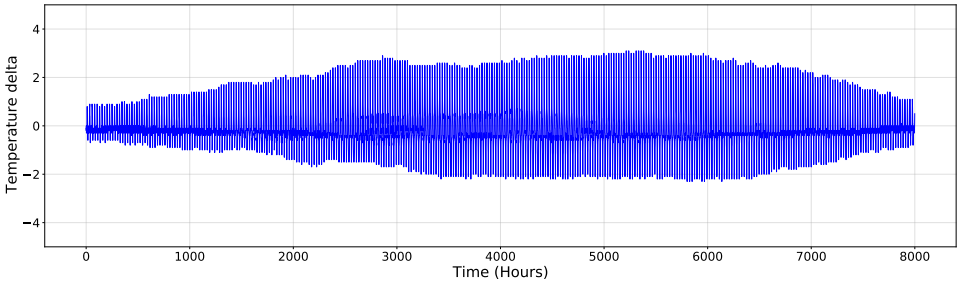
C

TIME SERIES FORECASTING SUPPLEMENTARY MATERIAL

We have shown in Section 6.2 the problem with the training, validation, and test set distribution with the NOAA dataset. We show in Figure C.1 the NOAA time series data before and after the differencing step. The resulting distributions were shown in Figure 6.3.



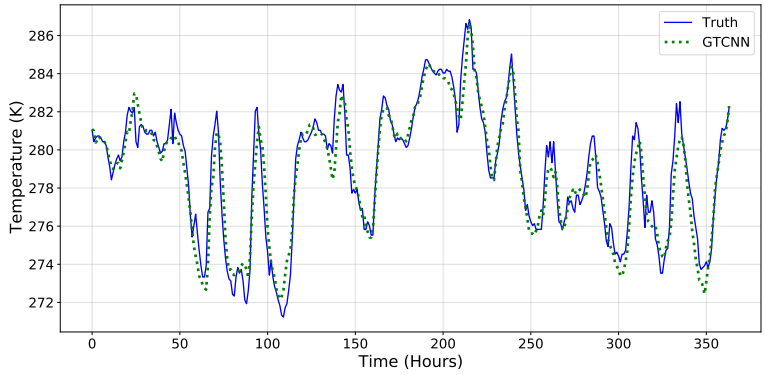
(a) Original data.



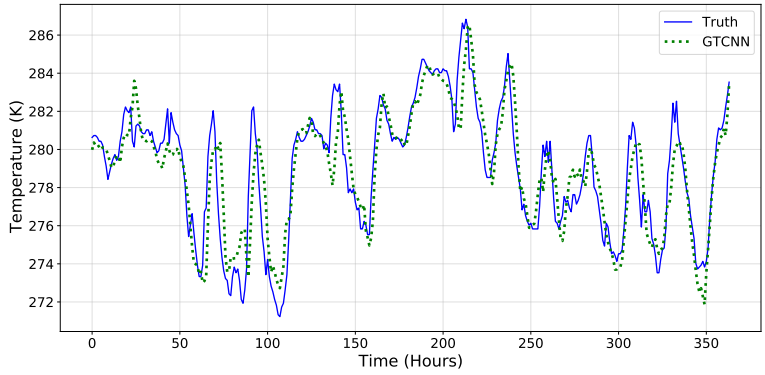
(b) Differenced data (first order difference).

Figure C.1: NOAA full data at node 78 before and after first order difference.

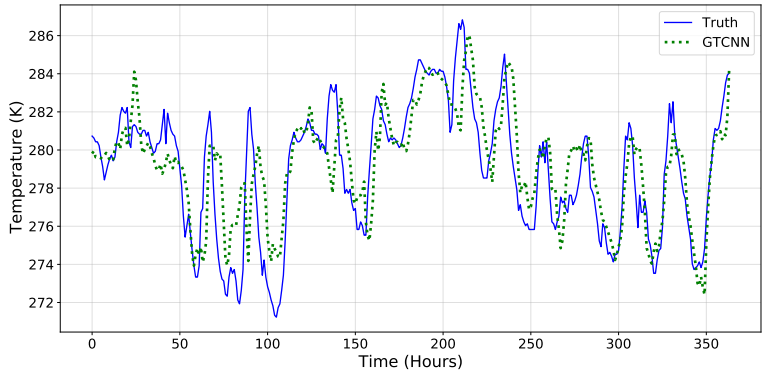
Moreover, to demonstrate the prediction capability of the GTCNN, we show in Figures C.2 and C.3 the predictions for the Molene and NOAA datasets. We show predictions for one, three, and five steps-ahead to illustrate how the prediction worsens as we increase the length of the horizon h.



(a) One step-ahead prediction.

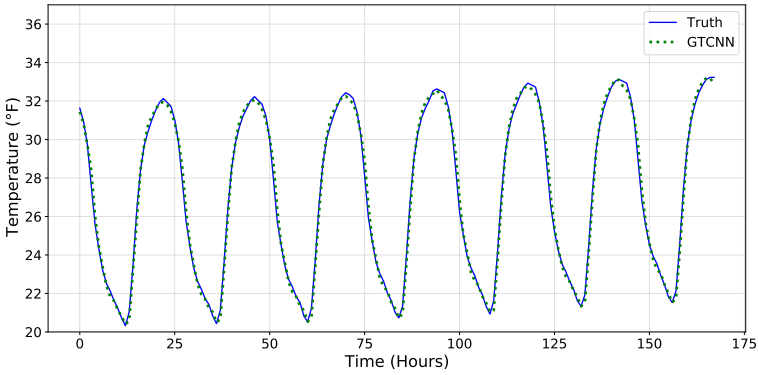


(b) Three steps-ahead prediction.

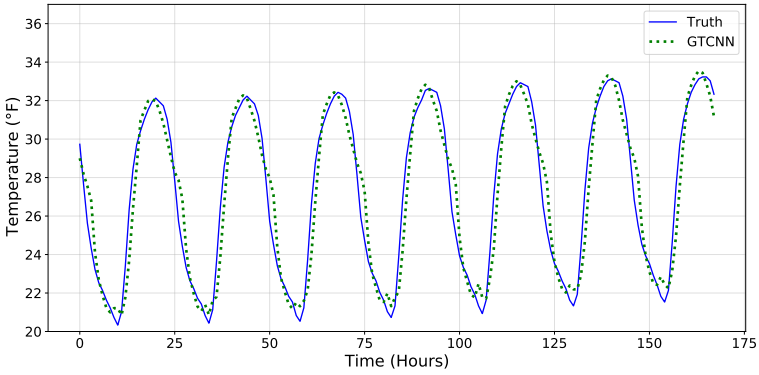


(c) Five steps-ahead prediction.

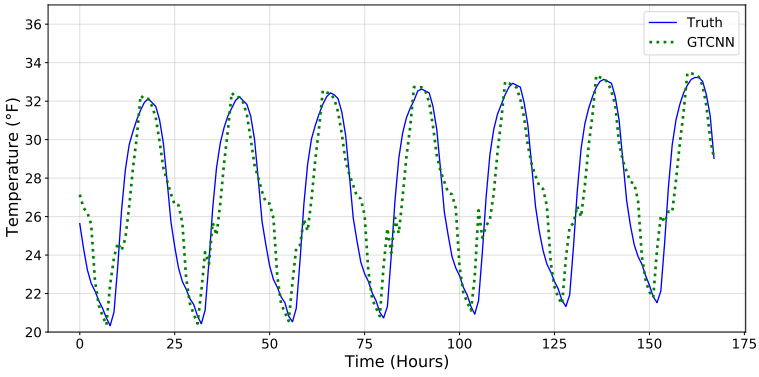
Figure C.2: GTCNN prediction on the Molene dataset (test set) at a specific node, randomly chosen. Station id: 22135001. Station name: LOUARGAT. The model never saw this data before.



(a) one step-ahead prediction.



(b) three steps-ahead prediction.



(c) five steps-ahead prediction.

Figure C.3: GTCNN prediction for the NOAA dataset (test set) at node 78, randomly chosen. For illustration purposes, only a week of data is shown. The model never saw this data before.

D

EARTHQUAKES CLASSIFICATION SUPPLEMENTARY MATERIAL

D.1. SEISMIC STATIONS

Section 5.2.1 introduced the earthquake dataset, discussing the 58 seismic stations providing the wave measurements. We show in Table D.1 the details regarding these stations. Each station can be uniquely identified on the FDSN service using a 'network' code and 'station' code. The most important details for us are the 'index' column, which relates each station with a node in the graph, and the 'latitude' and 'longitude' columns, locating the stations on the geographic map.

D.2. MULTI-CLASS CLASSIFICATION METRICS

In Section 5.4.2, we performed the multi-class classification experiment and showed the plot for the radius-based accuracy only. We report in Figures D.1a, D.1b, and D.1c the other radius-based metrics (precision, recall, and f1) for the multi-class experiment. While the recall of the GTCNN is higher than the other models, the GTCNN exhibits a lower precision and f1 score. However, we also see that for a value of radius $R > 65$, the precision of the GTCNN catches up with the GGRNN model, also improving the resulting f1 score.

D.3. DOWNSAMPLING STEP VALIDATION

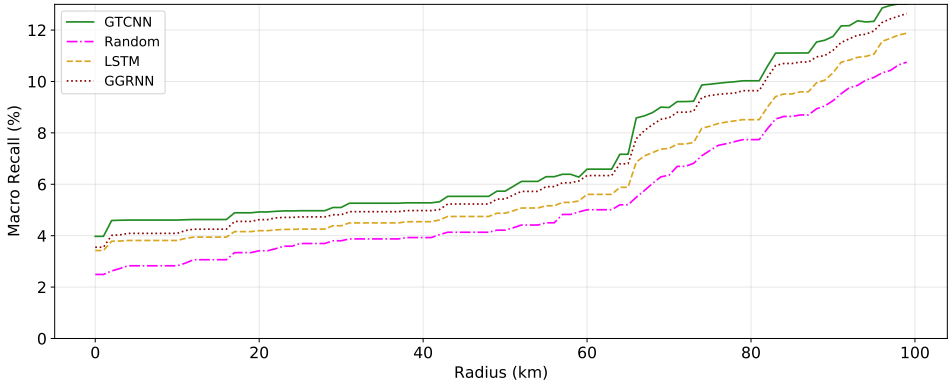
In Section 5.4.2, we performed an experiment to make sure our downsampling step does not represent information loss and showed the plot for the radius-based accuracy only. We report in Figures D.2a, D.2b, and D.2c the other radius-based metrics (precision, recall, and f1) for this experiment. The behaviour for these metrics is similar to the one observed in the accuracy [cf. Figure 5.17], i.e., the model trained on the raw data performs similarly to the model trained on the downsampled data for small values of R, and worse for larger radiuses.

D.4. BINARY CLASSIFICATION

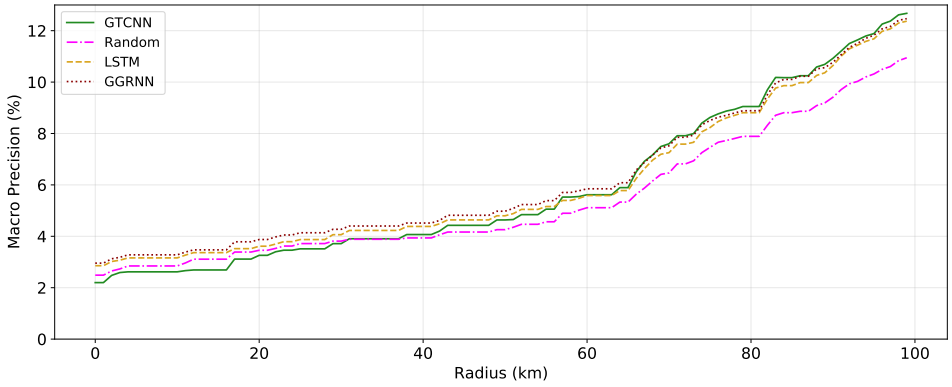
In Section 5.4.3, we performed the binary classification experiment and showed the boxplots for the accuracy only. We report in Figures D.3, D.4, and D.5 the boxplots for the precision, recall and f1. We can still observe that certain classes lead, on average, to higher performance on these metrics. This result is consistent with the findings shown for the accuracy [cf. Figure 5.18]. Moreover, we see that the spread of the boxplots is significantly higher for the recall than for any other chosen metric. For example, when the positive class is class 8, the GGRNN scored both recall 0% and 100%.

Table D.1: Stations for the New Zealand earthquake dataset presented in section 5.2.

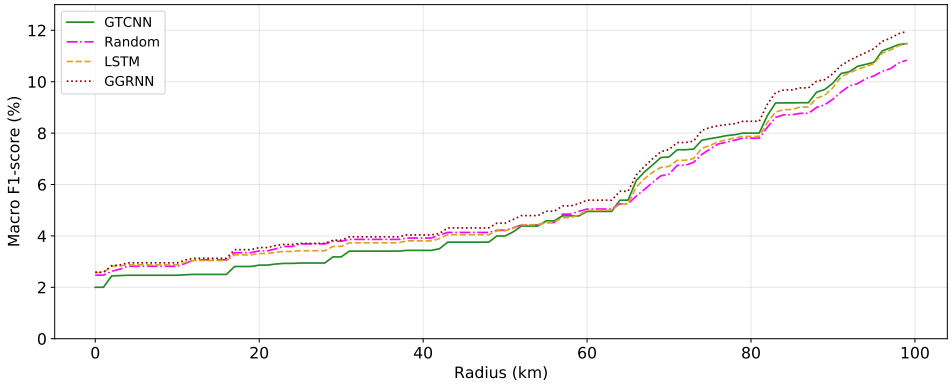
index	network	station	latitude	longitude	sitename
0	NZ	BFZ	-40.679647	176.246245	Birch Farm
1	NZ	BHW	-41.408231	174.871115	Baring Head
2	NZ	BKZ	-39.165666	176.492544	Black Stump Farm
3	NZ	COVZ	-39.199914	175.542402	Chateau Observatory
4	NZ	CVZ	-44.383180	171.006160	Cave
5	NZ	DCZ	-45.464713	167.153533	Deep Cove
6	NZ	DSZ	-41.744961	171.804614	Denniston North
7	NZ	EAZ	-45.231053	169.308253	Earnscleugh
8	NZ	ETVZ	-39.135700	175.710600	East Tongariro
9	NZ	FWVZ	-39.254945	175.552952	Far West
10	NZ	GRZ	-36.250200	175.457800	Great Barrier Island
11	NZ	GVZ	-42.967365	173.034750	Greta Valley
12	NZ	HAZ	-37.756100	177.782600	Te Kaha
13	NZ	HIZ	-38.512929	174.855686	Hauiti
14	NZ	INZ	-42.724500	171.444100	Inchbonnie
15	NZ	JCZ	-44.073210	168.785473	Jackson Bay
16	NZ	KHEZ	-39.294200	174.014500	Kahui Hut
17	NZ	KHZ	-42.415980	173.538970	Kahutara
18	NZ	KNZ	-39.021755	177.673669	Kokohu
19	NZ	KUZ	-36.745229	175.720873	Kuaotunu
20	NZ	LBZ	-44.385553	170.184420	Lake Benmore
21	NZ	LTZ	-42.781667	172.271035	Lake Taylor Station
22	NZ	MLZ	-45.366544	168.118407	Mavora Lakes
23	NZ	MQZ	-43.706082	172.653766	McQueen's Valley
24	NZ	MRZ	-40.660545	175.578527	Mangatainoka River
25	NZ	MSZ	-44.673334	167.926399	Milford Sound
26	NZ	MWZ	-38.334001	177.527779	Matawai
27	NZ	MXZ	-37.562259	178.306631	Matakaoa Point
28	NZ	NNZ	-41.217103	173.379477	Nelson
29	NZ	ODZ	-45.043982	170.644622	Otahua Downs
30	NZ	OPRZ	-37.844300	176.554929	Ohinepanea
31	NZ	OPZ	-45.884356	170.597767	Otago Peninsula
32	NZ	OUZ	-35.219689	173.596133	Omahuta
33	NZ	OXZ	-43.325900	172.038300	Oxford
34	NZ	PUZ	-38.071548	178.257209	Puketiti
35	NZ	PXZ	-40.030644	176.862145	Pawanui
36	NZ	QRZ	-40.825522	172.529148	Quartz Range
37	NZ	RATZ	-38.866498	175.772176	Rangitukua
38	NZ	RPZ	-43.714608	171.053865	Rata Peaks
39	NZ	RTZ	-38.615440	176.980518	Ruatahuna
40	NZ	SYZ	-46.536890	169.138823	Scrubby Hill
41	NZ	THZ	-41.762474	172.905218	Top House
42	NZ	TLZ	-38.329400	175.538000	Tolley Road
43	NZ	TMVZ	-39.115610	175.704064	Te Maari
44	NZ	TOZ	-37.730956	175.501847	Tahuroa Road
45	NZ	TRVZ	-39.298816	175.547822	Turoa
46	NZ	TSZ	-40.058553	175.961124	Takapari Road
47	NZ	TUZ	-45.953975	169.631143	Tuapeka
48	NZ	URZ	-38.259249	177.110894	Urewera
49	NZ	VRZ	-39.124341	174.758453	Vera Road
50	NZ	WCZ	-35.938642	174.345043	Waipu Caves
51	NZ	WEL	-41.284048	174.768184	Wellington
52	NZ	WHVZ	-39.282500	175.588600	Whangaehu Hut
53	NZ	WHZ	-45.892428	167.947031	Wether Hill Road
54	NZ	WIZ	-37.524511	177.189302	White Island
55	NZ	WKZ	-44.827021	169.017562	Wanaka
56	NZ	WSRZ	-37.518110	177.177805	White Island Summit
57	NZ	WVZ	-43.074350	170.736754	Waitaha Valley



(a)

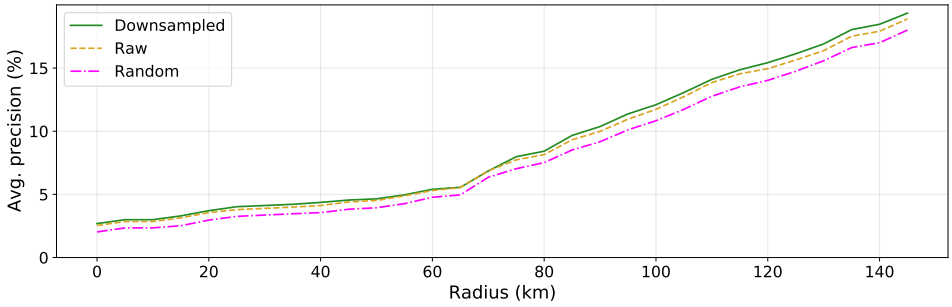


(b)

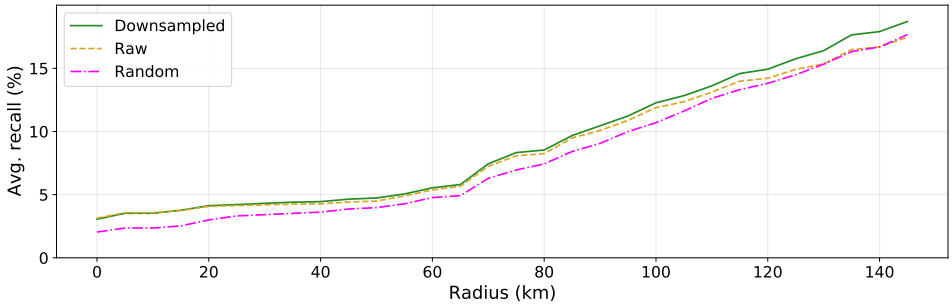


(c)

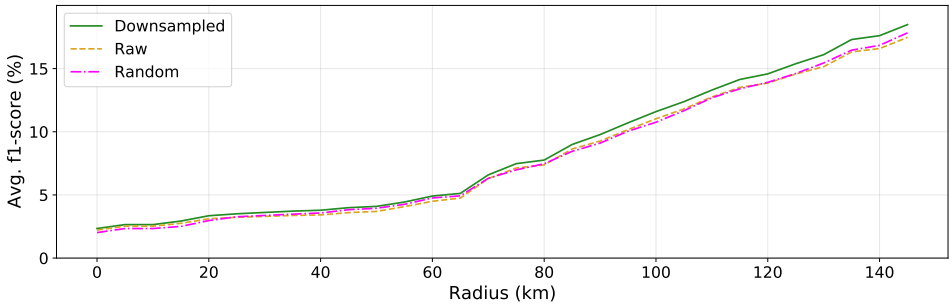
Figure D.1: Radius-based macro-averaged metrics for the multi-class classification problem presented in section 5.4.2. **a)** Macro-averaged recall, **b)** Macro-averaged precision, **c)** Macro-averaged f1-score. The radius ranges from 0km (standard macro-averaged metrics) to 100km.



(a)

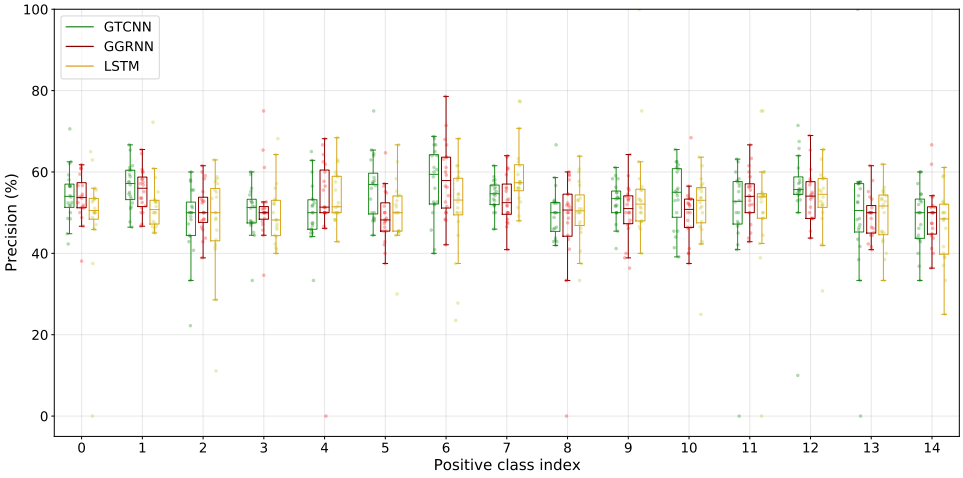


(b)

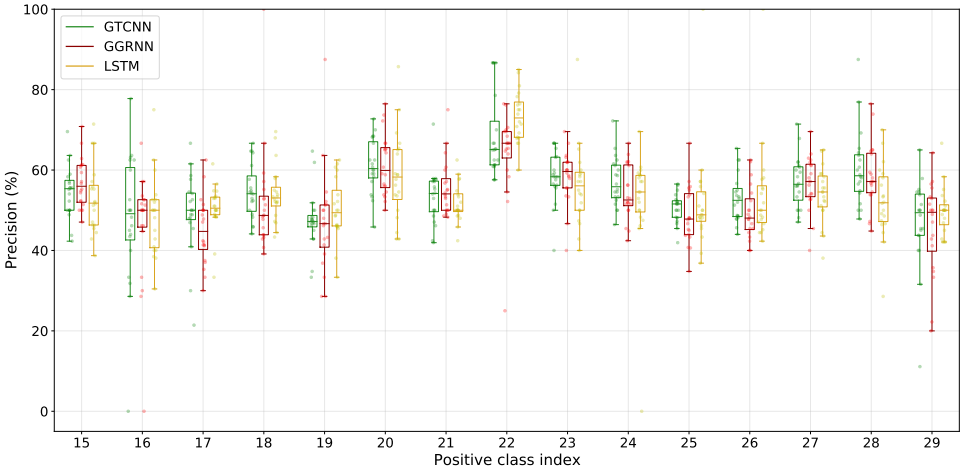


(c)

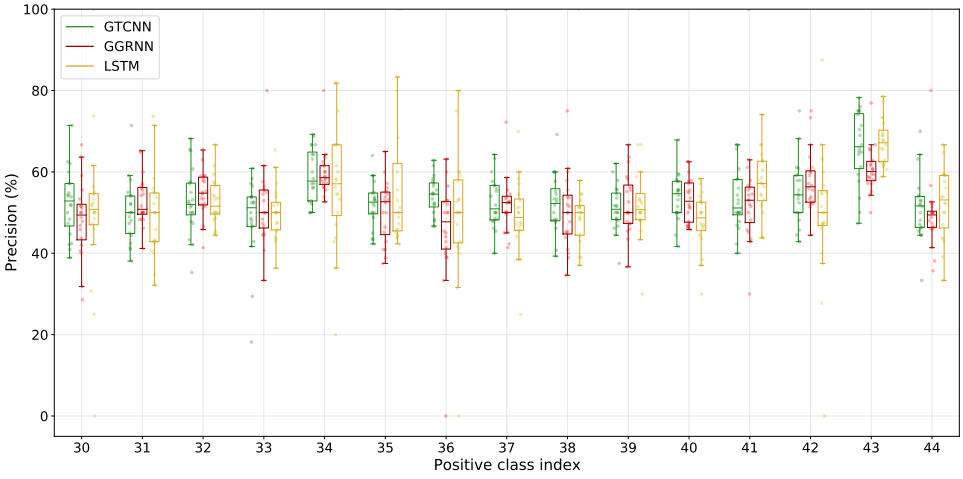
Figure D.2: LSTM radius-based metrics with and without downsampling (2Hz data versus 100Hz data). The experiment is detailed in section 5.4.2. **a)** Macro-averaged precision, **b)** Macro-averaged recall, **c)** Macro-averaged f1-score. The radius ranges from 0km (standard macro-averaged metrics) to 150km.



(a)

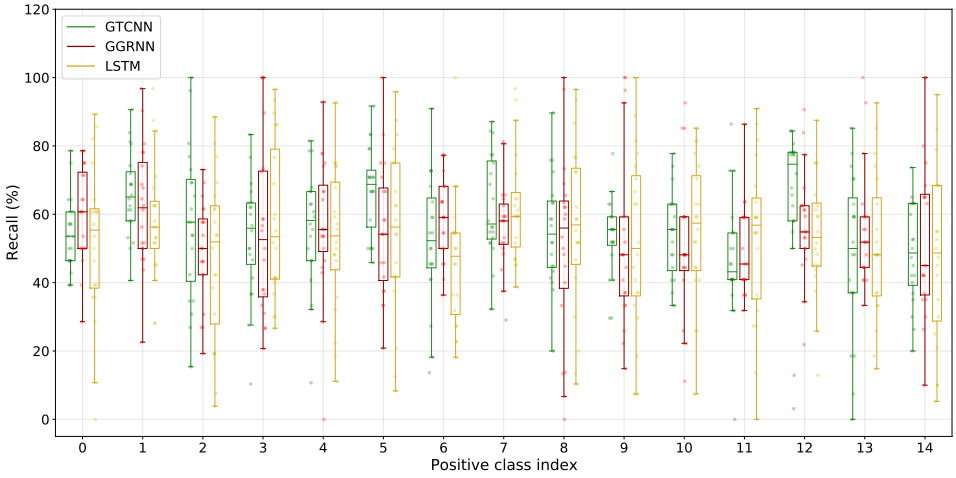


(b)

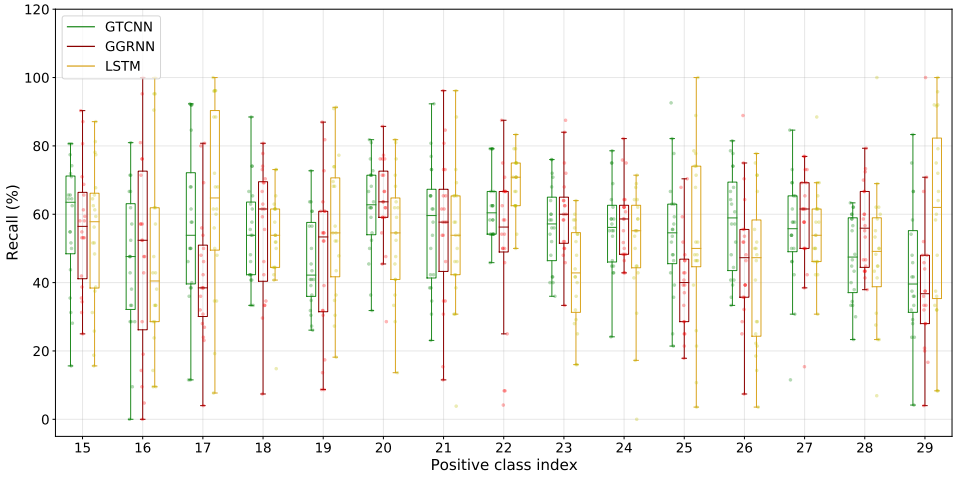


(c)

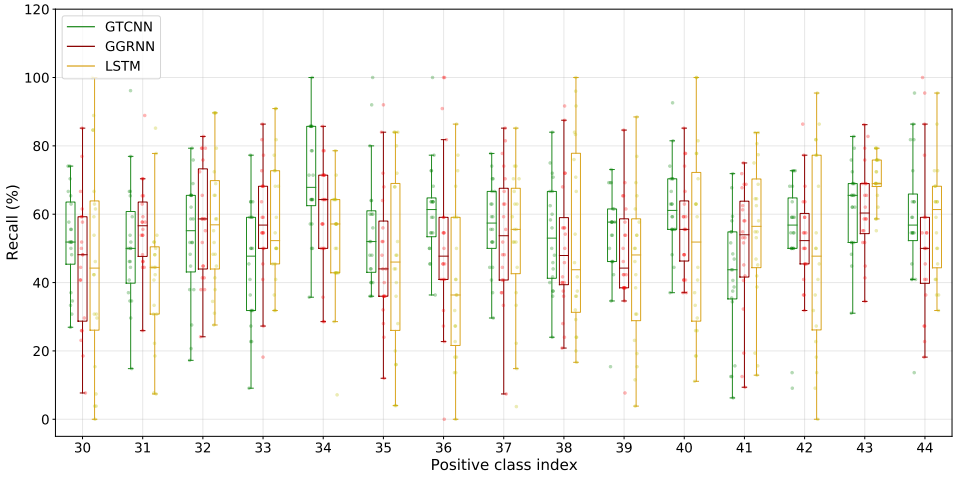
Figure D.3: Boxplots of the precision for each binary classification setting. The number of iterations (per boxplot) is 20.



(a)

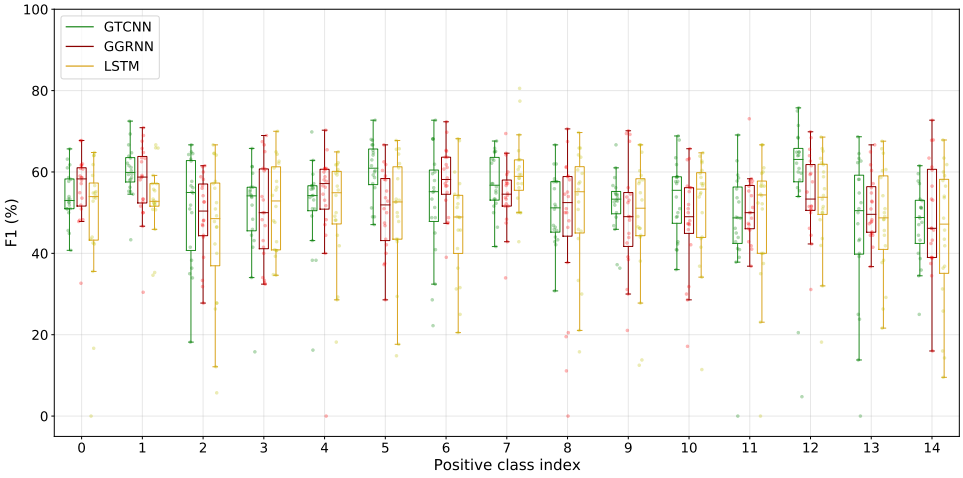


(b)

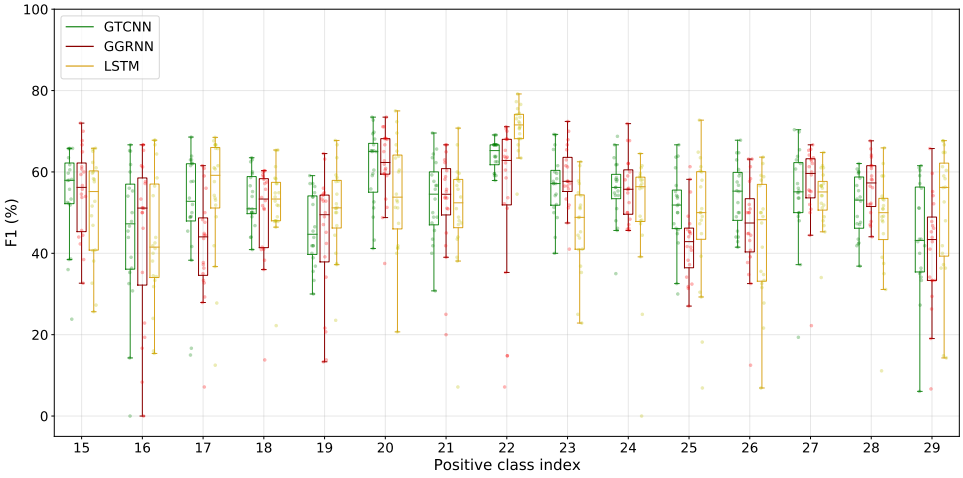


(c)

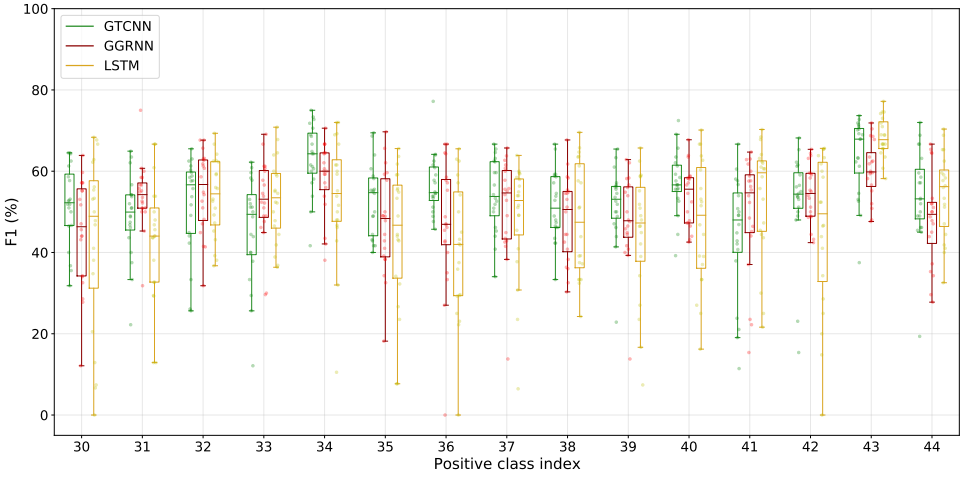
Figure D.4: Boxplots of the recall for each binary classification setting. The number of iterations (per boxplot) is 20.



(a)



(b)



(c)

Figure D.5: Boxplots of the f1-score for each binary classification setting. The number of iterations (per boxplot) is 20.