Online Adaptive Graph Neural Networks

Msc Thesis Computer Science & Engineering

Alex Jeleniewski

TU Delft

Online Adaptive Graph Neural Networks

Msc Thesis Computer Science & Engineering

Thesis report

by

Alex Jeleniewski

to obtain the degree of Master of Science
at the Delft University of Technology
to be defended publicly on 04-07-2024

Thesis committee:

Chair:	Elvin Isufi
Core Member 2:	Stjepan Picek
Core Member 3:	Seyran Khademi
Supervisor:	Elvin Isufi
Daily Co-Supervisor:	Mohammad Sabbaqi
Place:	Faculty of Electrical Engineering, Mathematics, Computer Science, Delft
Project Duration:	June 2023 - July 2024
Student number:	4701925

Faculty of EEMCS · Delft University of Technology

Abstract

Analyzing and forecasting multivariate time series using networks is interesting in traffic, energy consumption, or financial forecasting applications. The main challenge is to capture both spatial and temporal dependencies in the data alongside the dynamics of the network itself. Graph Neural Networks (GNNs) have shown reliable performance in network-based data by modeling the connections as a graph. GNNs, using temporal and recurrent structures, are further developed as a prominent tool for processing multivariate time series. However, the dynamics of the network, the evolution of the network over time, and its effect on the data should be studied more. Recent works augment temporal GNNs with graph learning modules to account for the changes in the network; however, often, these methods do not consider the dynamics directly, and all of them stop the graph changing after the training phase. This thesis focuses on an online graph adapting approach to deal with the dynamic data over networks. The proposed method is a hybrid model consisting of a temporal GNN inspired by the physics of dynamic networks, trains its parameters during the training phase, and adapts the graph with the stream of temporal data online. The experiments study the graph adaptation module's role on various benchmark datasets in forecasting tasks, such as traffic, windmill energy, and financial forecasting.

Preface

Contents

List of Figures vi

List of Tables vii

I Preliminaries 1

1 Introduction 2

2 Preliminaries 4
- 2.1 Graphs . . . 4
- 2.2 Neural Networks . . . 6
- 2.3 Training a Neural Network . . . 7
- 2.4 Neural Networks for Graphs . . . 7
- 2.5 Neural Network on Dynamic graphs . . . 9
- 2.6 Multivariate Time Series Forecasting . . . 9
- 2.7 Graph learning for multivariate timeseries forecasting . . . 9
- 2.8 Discussion . . . 10

3 Related work 11
- 3.1 Learning from Dynamic graphs . . . 11
- 3.2 Learning/Adapting dynamic graphs . . . 12
- 3.3 Online learning on graphs . . . 13
- 3.4 Position of this thesis . . . 13

II Implementation 15

4 Proposed method 16
- 4.1 Evolving Attention Weights . . . 17
- 4.2 Graph Adaptation . . . 17
- 4.3 Predicting with the adapted graph . . . 17
- 4.4 Implementation . . . 18
- 4.5 Discussion . . . 19

5 Experiments and Results 20
- 5.1 Baselines . . . 20
- 5.2 Models . . . 20
- 5.3 Datasets . . . 21
- 5.4 Hyperparameters . . . 21
- 5.5 Learning and Evaluation metrics . . . 21
- 5.6 Results . . . 22
- 5.7 Ablation studies . . . 27

III Closure 29

6 Conclusion 30

References 33

A Hyperparameter search 34

B Baseline experiments **37**
B.1 GCRN . . . 37
B.2 EvolveGCN . . . 37
B.3 MTGNN . . . 38
B.4 DGCRN . . . 38
B.5 ADLNN . . . 39
B.6 ASTGAT . . . 40

C Changing the graph structure **41**
C.1 Results . . . 42

D Ablation study on adaptiveness **43**

List of Figures

2.1 A graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$, where node 1 (green) is the target node, nodes 2 and 3 (blue) are the 1-hop-neighbours of node 1, and node 4 and 5 (red) are the 2-hop-neighbours of node 1. . 4
2.2 Illustration of how a dynamic graph can differ in topology. Left is $\mathcal{G}_t$ and right is $\mathcal{G}_{t+1}$. 5
2.3 An example of the graph adaptation process using node features over time. $\mathcal{G}_0$ is an initial given graph. 6

4.1 High-level overview of the OAGNN model. The EvolvingGA and Adaptation of W_t define the adaptation process. 16
4.2 Overview of the prediction module. It consists of L stacked message passing module that uses the adapted graph and a single skip connection MLP. 18

5.1 METR-LA. The average speed of the window at each timestep plotted (top), and the normalized norm of difference of the adjacency matrix at each timestep (bottom). Adapt rate $\gamma = 0.01$. 23
5.2 PEMS-BAY. The average power of the window at each timestep plotted (top) and the normalized norm of the difference of the adjacency matrix at each timestep (bottom). Adapt rate $\gamma = 0.01$. 23
5.3 SDWPF. The average power of the window at each timestep plotted (top) against the norm difference of the adjacency matrix at each timestep (middle) and the normalized norm of difference at each timestep (bottom). Adapt rate $\gamma = 0.1$. 25
5.4 FINANCE. The average close price of the window at each timestep plotted (top) against the norm difference of the adjacency matrix at each timestep (middle) and the normalized norm of difference at each timestep (bottom). Adapt rate $\gamma = 0.001$. 26
5.5 Convergence of the OAGNN model. Log-scale. 27

C.1 Overview of the adaptation module with graph construct (GC). The previous prediction, $\mathbf{B}_{t-1}$ and $\mathbf{A}_{t-1}$ are stored from the previous iteration. 42

List of Tables

3.1 Table showing the structure of works in the field related to adaptive dynamic networks. . . . 14

5.1 Prediction performance with standard deviation on METR-LA dataset for horizons 3 and 12. Averages of 5 runs. Lower is better. 22
5.2 Prediction performance with standard deviation on PEMS-BAY dataset for horizons 3 and 12. Averages of 5 runs. Lower is better. 23
5.3 Prediction performance and standard deviations for horizons 3 and 12 on the SDWPF dataset. Averages of 5 runs. All results are our own, and the MAE+RMSE loss refers to the loss function introduced in [21] and defined in **MW**. We define the MAE and *sd*MAE losses in **kW**. Lower is better. *The model did not always converge. 24
5.4 Class prediction performance with the standard deviation for horizon 1 on the **FINANCE** dataset. Averages of 5 runs. Higher is better. *Results from [22]. 24
5.5 Results of ablation studies on FINANCE, METR-LA, and SDWPF. METR-LA and SDWPF use $h = 12$. Average over five runs, with the standard deviation denoted. Lower is better. . 28

A.1 Chosen hyperparameters for all datasets. Selected hyperparameters are based on averages of 3 runs and the stability of the constructed graph. 35
A.2 Table illustrating the relation between the number of evolving attention heads and the evolving attention hidden dimension with the number of learnable parameters in the model. . . . 35
A.3 Runtime in seconds per epoch of the model per datasets. Averages over five runs with the standard deviation denoted. 36

B.1 Parameters for MTGNN, and how they were derived. Note that several parameters are left out, as these are dataset-dependent. 38
B.2 Parameters for DGCRN, and how they were derived. Note that several parameters are left out, as these are dataset-dependent. 39
B.3 Parameters for ADLNN, and how they were derived. Note that several parameters are left out, as these are dataset-dependent. 40
B.4 Parameters for ASTGAT, and how they were derived. Note that several parameters are left out, as these are dataset-dependent. 40

C.1 Result of runs using GC and the normal OAGNN model. The METR-LA, PEMS-BAY, and SDWPF datasets use $h = 12$. Averages over five runs with the standard deviation denoted. Lower is better for METR-LA, PEMS-BAY, and SDWPF; for FINANCE, higher is better. . . . 42

D.1 Result of additional ablation study using 50, 40, and 10 train-test-validation split. The METR-LA, PEMS-BAY, and SDWPF datasets use $h = 12$. Averages over five runs with the standard deviation denoted. Lower is better. 43

Part I

Preliminaries

1 Introduction

In multivariate timeseries environments such as financial, traffic, and weather, the data and their interactions depend on dynamic factors such as macroeconomics for finance, news, and the time of day [1, 2]. We define such time series as *dynamic*. Solutions model the spatial dependencies between the time series using a graph structure. A graph can be constructed from physical distances or metrics, such as cosine similarity, which is then efficiently processed using a Graph Neural Network (GNN) [3]. A GNN also has the benefits of capturing arbitrary inductive biases and allowing for new tasks.

However, due to the dynamic behavior of the timeseries, a static graph cannot represent the changing interactions of the data. For example, a financial graph based on correlations displays different behavior depending on the macroeconomic environment, which a single static graph cannot represent. To capture the changes in the dynamic data, a *dynamic graph* models the spatial dependencies between time series at each timestep [4, 5, 6]. By leveraging a GNN to that of a dynamic GNN, the model learns the spatial and temporal dependencies, improving on the static graph solutions [7, 8, 9, 10, 11].

A dynamic graph represents the network at a timestep; it does not represent the evolution of the network. A network depends on the current timestep and all previous timesteps and is a snapshot of the evolution of the network. To improve on the dynamic graph, we see a graph learning module used [12, 13, 14, 15], with several works incorporating metrics over the data itself [16, 17, 18]. In [19], the authors argue from a physical point of view about a two-way relationship in which the *dynamics* (i.e., node features) influence the structure and how the structure influences dynamics. To use this, the authors introduce an *adaptive* dynamical network that captures this physical property.

However, a dynamic environment is never static, and the models mentioned above do not capture changing dynamics and network structure outside of a training environment. This results in eventual network mismatch with the actual network and performance degradation. Online graph learning approaches exist to capture the network changes continuously, but these do not capture the two-way relationship between the structure and the dynamics. In addition, such methods require assumptions over the data.

Therefore, in this thesis, we follow the physical point of view, and we investigate how to construct an adaptive graph neural network that can model the two-way relationship and continuously capture changes in the dynamic environment:

Research Objective

”How to construct an online adaptive graph model for multivariate time series analysis?”

We first build upon the theoretical framework proposed in [19] to a machine learning model. The model contains an adaptation operator to adapt the graph, for which we first ask:

Research Question 1

”How to capture the network dynamics in the adaptation operator?”

We propose a similarity-based score that can capture the network dynamics with a scalable and data-model-independent approach. We construct the adaptation operator by modifying the graph attention [20] operator to have a temporal component and introduce a temporal connection in the graph adaptation process. Finally, we ask:

Research Question 2

”How does the online adaptive graph model perform on real-world applications?”

We compare the performance and effect of the adaptation operator on datasets that display different dynamic behaviors: the often-used traffic datasets METR-LA and PEMS-BAY and introduce more applicable and more dynamic networks by using SDWPF [21] and a FINANCE dataset [22].

Together, these research questions yielded the contribution of the Online Adaptive Graph Neural Network (OAGNN). The hybrid model contains learnable weights and an online graph adaptation mechanism. By building upon the adaptive dynamic network in [19] we construct an adaptive graph neural network using a simple message passing structure. To capture the continuously changing network, we use an online mechanism in the graph adaptation process to always capture the evolution of the network. We use Graph Attention to compute the graph attention coefficients and evolve the weights in graph attention to create a temporal dependency.

This work is structured as follows: we first provide the necessary background knowledge to understand this work in Chapter 2; then, we discuss prior works in Chapter 3; in Chapter 4 we introduce the OAGNN model; Chapter 5 covers the experiments and the performance of the model; in Section 5.7 we discuss several ablation studies and in Chapter 6 we conclude this work and discuss future directions.

2 Preliminaries

This section provides background knowledge to facilitate understanding of this work. We introduce graphs, including dynamic and adaptive graphs. Next, we discuss neural networks and learning them, neural networks for graphs, and dynamic graphs. Then, we introduce multivariate timeseries forecasting on static graphs and with graph learning approaches. Finally, we present adaptive dynamic graph neural networks for multivariate timeseries forecasting.

2.1. Graphs

A (static) graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$ defines a network using the set of N nodes/vertices as $\mathcal{V}$, and the set of M edges as $\mathcal{E} \in \mathbb{R}^{N \times N}$. If a connection exists between nodes i and j, an entry $(i, j) \in \mathcal{E}$ exists. A connection $(i, i) \in \mathcal{E}$ indicates a *self-loop*, i.e. the node is connected with itself. Graphs can either be *directed* or *undirected*. In an undirected graph, there is no distinction between an edge's direction, meaning that for each $(i, j) \in \mathcal{E}$, there exists a $(j, i) \in \mathcal{E}$. In a directed graph, the edges have a direction, such that $(i, j) \in \mathcal{E}$ does not imply the existence of (j, i). For each node, we can define the k-hop-neighbourhood $\mathcal{N}^k(i)$ that defines the nodes connected with node i within k edges. The 1-hop-neighbourhood or the direct neighborhood of a node i is the set of nodes $\mathcal{N}^1(i)$, or $\mathcal{N}(i)$ for short, that have a direct edge with node i. We illustrate this in Figure 2.1.

The adjacency matrix **A** $\in \mathbb{R}^{N \times N}$, is defined such that $\mathbf{A}_{ij} = 1$ if there exists an edge between node i and node j, i.e. $(i, j) \in \mathcal{E}$ exists. If there exists no edge between node i and node j, $\mathbf{A}_{ij} = 0$. A graph with 0 or 1 edges is an *unweighted* graph. To construct a *weighted* graph, we define the graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}, \mathcal{W}\}$, where $\mathcal{W}$ is the set of edge weights. We can apply this definition to the adjacency matrix **A** as $\mathbf{A}_{ij} > 0$ if there exists an edge (i, j), with the value indicating the edge weight, and $\mathbf{A}_{ij} = 0$ if there exists no edge from i to j. The adjacency matrix of an undirected graph is symmetric.

When **A** is symmetric, we can define the *degree* matrix **D** $\in \mathbb{R}^{N \times N}$. The degree of a node i defines the sum of all edge weights of the direct neighborhood of i. The matrix **D** is a *diagonal* matrix, i.e., all values except the main diagonal entries are 0. We can define this as

$$\mathbf{D}_{ij} = \begin{cases} \mathbf{D}_{ij} = \text{deg}(i) = \sum_{j=1}^{N} \mathbf{A}_{ij} & \text{if } i = j, \\ 0 & \text{otherwise,} \end{cases}$$

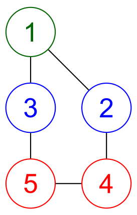

Figure 2.1: A graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$, where node 1 (green) is the target node, nodes 2 and 3 (blue) are the 1-hop-neighbours of node 1, and node 4 and 5 (red) are the 2-hop-neighbours of node 1.

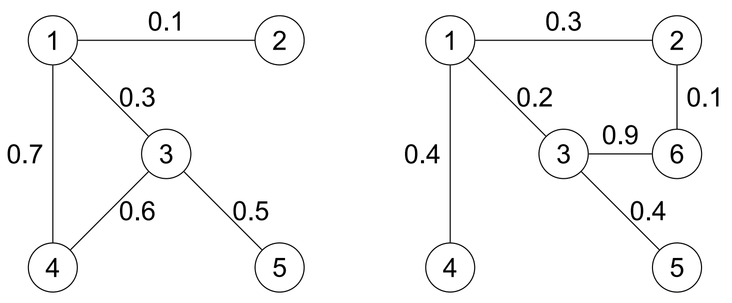

Figure 2.2: Illustration of how a dynamic graph can differ in topology. Left is $\mathcal{G}_t$ and right is $\mathcal{G}_{t+1}$.

where deg$(\cdot)$ indicates the *degree* function. In many cases, we prefer a normalized version of the adjacency matrix. We can normalize the matrix using row-wise normalization such as

$$\mathbf{A}_{RW} = \mathbf{D}^{-1}\mathbf{A}.$$

The result of this normalization method is also called the random-walk adjacency matrix. However, it does not preserve symmetry. The symmetric normalization is

$$\hat{\mathbf{A}} = \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}.$$

Another representation of the graph is the *Laplacian* matrix

$$\mathbf{L} = \mathbf{D} - \mathbf{A}.$$

The Laplacian's main diagonal is each node's degree, and negative real numbers elsewhere depending on **A**. The Laplacian is useful for many graph properties, such as spanning trees or spectral decomposition. Again, we can define a normalized version of the Laplacian as

$$\mathbf{L} = \mathbf{I} - \hat{\mathbf{A}} = \mathbf{I} - \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}},$$

where **I** is the identity matrix, containing ones on its main diagonal and zeros elsewhere.

2.1.1. Dynamic Graph

A dynamic graph describes a network that changes in topology over time. We now have a time-indexed graph $\mathcal{G}_t$. The changes in topology can consist of differences in the node set $\mathcal{V}$, the edge set $\mathcal{E}$, or the edge weights $\mathcal{W}$. We can use a discrete or continuous representation to represent such a network. In a discrete setting, we define a set of graph snapshots $\mathcal{G}$, where each snapshot $\mathcal{G}_t$ contains a full representation of the graph at timestep t

$$\mathcal{G} = \{\mathcal{G}_1, ..., \mathcal{G}_T\}$$
$$\mathcal{G}_t = \{\mathcal{V}_t, \mathcal{E}_t, \mathcal{W}_t\}.$$

where T is the max timestep. Figure 2.2 shows two graphs at consecutive timesteps. Similar to graphs in 2.1, we can define the adjacency matrix $\mathbf{A}_t$, the degree matrix $\mathbf{D}_t$, and the Laplacian $\mathbf{L}_t$. We use the discrete setting when the entire graph is captured at periodic intervals, such as minutes, hours, or days. An important distinction with a static graph is that $\mathbf{A}_t$ is not necessarily fixed in size, as the number of nodes can differ between timesteps.

In a continuous setting, only some of the network is available. Instead, we use an event-based temporal graph [4], in which $\mathcal{G}_t$ is defined by edge insertions $\epsilon^+_{\mathcal{E}}$, deletions $\epsilon^-_{\mathcal{E}}$, node insertions $\epsilon^+_{\mathcal{V}}$, and deletions $\epsilon^-_{\mathcal{V}}$. Formally we define

$$\mathcal{G}_t = \{\epsilon : \epsilon \in \{\epsilon^+_{\mathcal{E}}, \epsilon^-_{\mathcal{E}}, \epsilon^+_{\mathcal{V}}, \epsilon^-_{\mathcal{V}}\}\}.$$

Using the original graph, the node and edge insertions, and deletions, it is possible to construct a graph at timestep t.

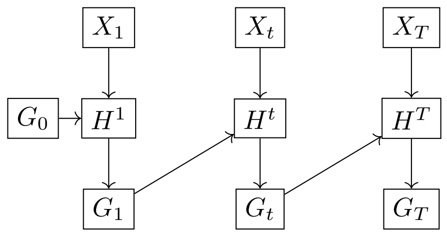

Figure 2.3: An example of the graph adaptation process using node features over time. $\mathcal{G}_0$ is an initial given graph.

2.1.2. Features on graphs

Each node can contain information about itself; we call these the *node features*. For a static graph, node i has its feature vector represented by $\mathbf{x}_i \in \mathbb{R}^F$, where F indicates the number of features. We also define the feature matrix for all nodes as $\mathbf{X} \in \mathbb{R}^{N\times F}$. In a dynamic setting, node i has its own feature vector at timestep t defined by $\mathbf{x}_{ti} \in \mathbb{R}^F$, or $\mathbf{X}_t \in \mathbb{R}^{N\times F}$ for all nodes.

Each edge can contain information as edge features. We previously introduced the edge weight set $\mathcal{W}$ or $\mathcal{W}_t$, where each edge has a value. We can view this as each edge having a feature vector with a single dimension. In this work, we do not focus on multidimensional edge features and only use a single feature for each edge.

2.1.3. Adaptive Dynamic Graph

In an adaptive dynamic graph, we adapt the graph at the current timestep using the *network dynamics*, i.e., the node features. Authors in [19] construct an adaptive operator for each node pair (i, j) as

$$a_{ij}(t) = H^t_{ij}[x(\cdot), A(\cdot), t], \tag{2.1}$$

where H^t_{ij} indicates a general adaptation operator, $x(\cdot)$ the states and history of the nodes, and $A(\cdot)$ the states and history of the adjacency matrices. The adaptation operator depends on the node features and the previous adjacency matrices. A graph can be adapted from a static or given graph at timestep t. This work focuses on the case where only the initial graph is given. With this, we descritize (2.1) with the adaptation operator

$$\mathbf{A}_{tij} = H^t_{ij}(\mathbf{x}_{ti}, \mathbf{x}_{tj}, \mathbf{A}_{t-1}), \tag{2.2}$$

where the time dependent adaptation operator is defined by H^t_{ij}, that depends on the node embeddings $\mathbf{x}_{ti}$ and $\mathbf{x}_{tj}$ at timestep t, and the previous adjacency matrix $\mathbf{A}_{t-1}$. We illustrate the process in Figure 2.3.

2.2. Neural Networks

Neural networks are part of deep learning and comprise an input layer, hidden layers, and an output layer. For any layer, we have some input $\mathbf{x} \in \mathbb{R}^F$, and the output $\mathbf{x}' \in \mathbb{R}^{F'}$. Input features are transformed to output features using parameters; for example, for a layer l, a learnable linear transformation transforms input features $\mathbf{x}_{l-1}$ to $\mathbf{x}_l$ using the weight matrix $\mathbf{W}_l$ and bias $\mathbf{b}_l$

$$\mathbf{x}_l = \sigma(\mathbf{W}_l \mathbf{x}_{l-1} + \mathbf{b}_l).$$

By optimizing the weights and bias, the features are transformed into the correct output. Layers of a neural network can define multidimensional input, but there is always a dimension F for the features.

2.2.1. Recurrent Neural Network

A recurrent neural network (RNN) is a type of network that contains memory and can process data sequences. Gated recurrent units (GRU) or long short-term memory (LSTM) networks exist for memory. We highlight the structure of a GRU as it is most relevant to this work. However, an LSTM would be a possible substitute anywhere we use GRU in this work. The inputs of an RNN are different from that of a CNN, as the input data is now in a sequence. We use the input as $\mathbf{x} \in \mathbb{R}^{s\times F}$, with s defining the sequence length, as input to our GRU as

$$\begin{aligned}\mathbf{r}_t &= \sigma(\mathbf{W}_r\mathbf{x}_t + \mathbf{W}_{rh}\mathbf{h}_{t-1} + \mathbf{b}_r),\\ \mathbf{z}_t &= \sigma(\mathbf{W}_z\mathbf{x}_t + \mathbf{W}_{zh}\mathbf{h}_{t-1} + \mathbf{b}_z),\\ \hat{\boldsymbol{h}}_t &= \tanh(\mathbf{W}_h\mathbf{z}_t + \mathbf{W}_{rh}(\mathbf{r}_t \odot \mathbf{h}_{t-1}) + \mathbf{b}_h),\\ \mathbf{h}_t &= (1-\mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \hat{\boldsymbol{h}}_t,\end{aligned}$$

where $\mathbf{x}_t$ is a value within the sequence at index t, **W** and **b** indicate a learnable weight matrix and bias respectively, σ a non-linear function, $\tanh(\cdot)$ the tanh function, $\odot$ the Hadamard product, and h_t the *hidden* state, used as memory for the unit.

2.3. Training a Neural Network

Optimizing parameters happens by splitting the given dataset into three sets: the *training* set, the *validation* set, and the *test* set. With the training set, we train the model, and after each execution, we update the models' parameters to generate a better output. To determine how to adjust the parameters, we use a loss function.

2.3.1. Loss functions

For a model to update its weights, it must have some method of computing how wrong or correct the output is. Loss functions define the objective the neural network has to minimize. A neural network will adjust its weights during the training phase according to the output of the loss function. A loss function is selected based on the task and the network. We use the Mean Absolute Error (MAE) loss and the root mean squared error (RMSE) for the forecasting task, defined as

$$\mathcal{L}_{MAE} = \frac{1}{N}\sum_{i=1}^{N}|\hat{y}_i - y_i|, \tag{2.3}$$

$$\mathcal{L}_{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}(\hat{y}_i - y_i)^2}, \tag{2.4}$$

where in all loss functions, the prediction $\hat{y}$ is compared against the correct value y, and N indicating the number of predicted values. For classification, we use the cross entropy loss for binary classification (two classes), defined as

$$\mathcal{L}_{CE} = \frac{1}{N}\sum_{i=1}^{N} y_i \log(\hat{y}) + (1-y_i)\log(1-\hat{y}). \tag{2.5}$$

2.3.2. Evaluation

We train for multiple *epochs*; that is, training the model on the training dataset is performed multiple times, as a single run using the training dataset does not allow the model to become descriptive enough. At the end of each epoch, we get a reference for the models' performance using the validation set, but the parameters are not updated. Lastly, after the training epochs finish, we test the models' performance on new data with the test set. The test loss is the primary score with which to compare models.

2.3.3. Online learning

After training, the model parameters are never adjusted. For models deployed in a dynamic setting, where changes in the network can even cause concept drift, it might not be desired to train the model using (traditional) batch learning. The training data might not represent the input correctly, the training data cannot capture the dynamics of the input correctly, prior data might not be (widely) available, or there needs to be more training data to train on. To this end, we use online learning. The model does not use training, validation, and test sets but learns on every input. After each input, the model adjusts its parameters, which allows it to change over time. The main difference with batch learning is that the samples are processed once in an online setting.

2.4. Neural Networks for Graphs

In this section, we discuss graph neural networks in general and introduce attention on graphs.

2.4.1. Graph Neural Networks

A graph neural network (GNN) [3] is a type of neural network aimed at processing data based on graphs. We define a GNN layer in its highest-level form as

$$\mathbf{X}' = f_{GNN}(\mathbf{X}, \mathbf{A}), \tag{2.6}$$

where **A** the adjacency matrix, and $\mathbf{X} \in \mathbb{R}^{N\times F}$ indicates a feature matrix, and $\mathbf{X}' \in \mathbb{R}^{N\times F'}$ the output node embeddings. A GNN can take any form, but at its core lies *message passing* to update representations with neighboring information. In GNNs, we aggregate information from the neighbors of a node to the target node. We define a message passing function for a single node i as

$$\mathbf{x}'_i = g(\mathbf{x}_i, \mathsf{AGG}_{j\in\mathcal{N}(i)}(f(\mathbf{x}_i, \mathbf{x}_j)) \tag{2.7}$$

where $\mathbf{x}_i \in \mathbb{R}^F$ is the feature vector for i, $\mathbf{x}'_i \in \mathbb{R}^{F'}$ the output feature vector of the message passing function with $f(.)$ and $g(.)$ some MLPs, and an aggregation scheme AGG over the neighborhood $\mathcal{N}$ of i. Aggregation schemes often are addition or multiplication. The work in [23] introduced a GNN as a graph convolutional (GC) layer defined as

$$\mathbf{x}'_i = \sum_{j\in N(i)\cup\{i\}} \frac{1}{\mathsf{deg}(i)\cdot\mathsf{deg}(j)} \cdot (\mathbf{W}\mathbf{x}_j) + \mathbf{b}, \tag{2.8}$$

where the aggregation scheme is defined using additions through $\sum$, the $\mathsf{deg}(\cdot)$ function as the degree of the node, and a learnable weight matrix **W**, and bias **b**. GNN layers can generally be represented for the entire graph instead of by a message passing function. We can rewrite the previous layer to the general GNN

$$\mathbf{X}' = \sigma(\hat{\mathbf{D}}^{-\frac{1}{2}}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-\frac{1}{2}}\mathbf{X}\mathbf{W}), \tag{2.9}$$

where, $\hat{\mathbf{D}}$ is the degree matrix for $\hat{\mathbf{A}}$, σ a non-linear function, and $\hat{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ with **I** as the identity matrix.

2.4.2. Attention on graphs

In attention on graphs, the input vectors are first transformed using a learnable weight matrix $\mathbf{W} \in \mathbb{R}^{F'\times F}$. A self-attention formula is then defined as

$$e_{ij} = a(\mathbf{W}\mathbf{x}_i, \mathbf{W}\mathbf{x}_j), \tag{2.10}$$

where a is any attention mechanism performing self-attention between two nodes, **W** is a learnable weight matrix, and $\mathbf{x}_i$ is the feature vector for node i [20]. To compute the attention coefficients, a softmax function is used

$$\alpha_{ij} = \mathsf{softmax}_j(e_{ij}). \tag{2.11}$$

We can formulate a function to update a single node's features using message passing. To this end, we sum over the neighborhood of node i using self-attention

$$\mathbf{x}'_i = \sigma(\sum_{j\in\mathcal{N}(i)} \alpha_{ij}\mathbf{W}\mathbf{x}_j), \tag{2.12}$$

where σ is a non-linear function.

To improve the expressiveness and the stability of self-attention, we can extend attention to multi-head attention. For this we use K attention mechanism and K weight matrices **W**

$$\mathbf{x}'_i = \sigma(\sum_{k=1}^{K}\sum_{j\in\mathcal{N}(i)} \alpha_{ij}^k\mathbf{W}^k\mathbf{x}_j). \tag{2.13}$$

We sum the heads to get one output vector for the K heads. In [20], the authors propose the full graph attention operator for a single node (one head) as

$$\alpha_{ij} = \frac{\exp\big(\mathsf{LeakyReLU}(\mathbf{a}^T[\mathbf{W}\mathbf{x}_i \parallel \mathbf{W}\mathbf{x}_j])\big)}{\sum_{k\in\mathcal{N}_i}\exp(\mathsf{LeakyReLU}(\mathbf{a}^T[\mathbf{W}\mathbf{x}_i \parallel \mathbf{W}\mathbf{x}_j]))}, \tag{2.14}$$

where $\cdot^T$ represents the transpose, $\parallel$ the concatenation operation and $\mathbf{a} \in \mathbb{R}^{2F'}$ a learnable vector.

2.5. Neural Network on Dynamic graphs

A neural network on dynamic graphs is an extension of a GNN, in which a temporal component is introduced to handle a dynamic graph as input. A neural network on dynamic graphs derives temporal and spatial dependencies of the current graph and input and their history. We define such a network in its highest form as

$$\mathbf{X}'_t = f_{DNN}(\mathbf{X}_t, \mathbf{A}_t). \tag{2.15}$$

where $\mathbf{A}_t$ is the adjacency matrix and $\mathbf{X}_t$ the node features at timestep t, and $\mathbf{X}'_t$ the new node features. Such a network has a structure similar to that of graph neural networks, with the difference in input with (2.6) being the time-indexed input and including some temporal component to connect with the previous timestep. We do not discuss such a network's implementation as it is irrelevant to this work.

2.6. Multivariate Time Series Forecasting

This work focuses on timeseries forecasting, consisting of value-based prediction and classification. For both, the goal is to predict based on some input timeseries, e.g., an asset's price. However, timeseries often do not exist in a vacuum. For example, in weather forecasting, many weather stations report the temperature. Instead of a single input timeseries, we have N timeseries that we are predicting simultaneously. By processing multiple timeseries, we can use structural information of other timeseries, such as physical distances, to improve prediction performance. We call this problem *multivariate time series forecasting*. In (multivariate) timeseries forecasting, we forecast for some *horizon* h using a *window* w of input. If we have N timeseries with F features to predict, the sliding window is defined as

$$\mathcal{X}_{t-w:t} = [\mathbf{X}_{t-w}, ..., \mathbf{X}_{t-1}, \mathbf{X}_t],$$

where $\mathbf{X}_t \in \mathbb{R}^{N\times F}$, and $\mathcal{X}_{t-w:t} \in \mathbb{R}^{N\times w\times F}$ defining the entire sequence. We predict the next sequence (the *horizon*) of h timesteps

$$\hat{\mathcal{Y}}_{t+1:t+h} = [\hat{\mathbf{Y}}_{t+1}, ..., \hat{\mathbf{Y}}_{t+h-1}, \hat{\mathbf{Y}}_{t+h}],$$

where $\hat{\mathbf{Y}}_{t+1} \in \mathbb{R}^{N\times d_0}$ is the prediction for the timestep $t+1$, $\hat{\mathcal{Y}}_{t+1:t+h} \in \mathbb{R}^{h\times N\times d_0}$ indicating the full sequence of predictions, where d_0 is the output dimension. If $d_0 = 1$, without the loss of generality we define $\hat{\mathbf{Y}}_{t+1:t+h} \in \mathbb{R}^{n\times N}$.

2.6.1. Fixed-Graph Multivariate Time Series Forecasting

A graph-based approach is very suitable since a multivariate time series forecasting problem consists of predicting N variables simultaneously. In many such methods, the input graph is generated beforehand and is not adapted/learned. The input graph could be generated based on the distances between sensors in the physical network or correlations of the timeseries.

2.7. Graph learning for multivariate timeseries forecasting

Several works have added a graph learning component to improve the fixed graph in multivariate time series forecasting. In this section, we discuss approaches that we use in this paper and are relevant to capturing the dynamics of a network and the limitations of current methods.

Node embeddings

In traffic forecasting, several works use node embeddings with a difference layer [24, 12, 17], where two embedding matrices $\mathbf{E}_1, \mathbf{E}_2$ are used in a learnable layer. The embedding matrices are defined as $\mathbf{E} \in \mathbb{R}^{N\times d_E}$ with d_E the embedding dimensions. In [24], the authors define the layer as

$$\mathbf{A} = \text{ReLU}(tanh(\alpha(\mathbf{M}_1\mathbf{M}_2^T - \mathbf{M}_2\mathbf{M}_1^T))), \tag{2.16}$$

where α is a hyperparameter, $\cdot^T$ the transpose, ReLU and *tanh* activation functions and $\mathbf{M}_1$ and $\mathbf{M}_2$ defined as

$$\mathbf{M}_1 = tanh(\alpha(\mathbf{E}_1\Theta_1)), \tag{2.17}$$

$$\mathbf{M}_2 = tanh(\alpha(\mathbf{E}_2\Theta_2)), \tag{2.18}$$

with again α the hyperparameter and Θ_1, Θ_2 learnable parameters. To prevent too much complexity, top-k values from $\mathbf{A}$ are used by the model. The layer is thus the result of the difference between the two node embeddings and does not take a physical approach to adapting the graph. Instead, the approach is more from a functional/deep learning perspective.

Using metrics

Multiple works have used metrics to compute a graph at a given timestep. Metrics give a similarity score between nodes and can describe the network's structure. For example, the cosine similarity defines the similarity between two vectors using the cosine operator. As nodes have a feature vector, computing the cosine similarity between two nodes is as straightforward as

$$s_{ij} = cos(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{x}_i^T \mathbf{x}_j}{\|\mathbf{x}_i\| \|\mathbf{x}_j\|} = \frac{\sum_{n=1}^{F} \mathbf{x}_{in}^T \mathbf{x}_{jn}}{\sqrt{\sum_{n=1}^{F} \mathbf{x}_i^T \mathbf{x}_i} \sqrt{\sum_{n=1}^{F} \mathbf{x}_j^T \mathbf{x}_j}}, \tag{2.19}$$

where s_{ij} defines the cosine similarity between nodes i and j, and F the number of features. We can define any metric that applies to two vectors to compute the similarity.

Learning with metrics

However, no learning is applied when using metrics to construct the graph. We must adjust the metric functions to contain learnable parameters to learn with metrics. For cosine similarity, this is possible using a learnable weight matrix **W**

$$s_{ij} = cos(\mathbf{W}\mathbf{x}_i, \mathbf{W}\mathbf{x}_j). \tag{2.20}$$

The structure of a learnable cosine similarity function is similar to that of Graph Attention, which we defined in 2.4.2. Specifically, we can define the resulting graph attention coefficients as a metric of similarity between nodes i and j as

$$\alpha_{ij} = \text{softmax}_j(a(\mathbf{W}\mathbf{x}_i, \mathbf{W}\mathbf{x}_j)), \tag{2.21}$$

where $a(\cdot)$ is some self attention mechanism, and **W** is a learnable weight matrix.

2.8. Discussion

This chapter introduced three different graphs: static, dynamic, and adaptive. Dynamic graphs introduce a temporal component to improve the non-dynamic setting of static graphs. By leveraging temporal dependencies, a dynamic GNN is better at handling time-indexed data. However, a dynamic graph is a snapshot of the network at a timestep and does not encode the network's evolution. We use an adaptive dynamic graph to leverage the information of the network's prior states. In addition, we introduced neural networks on dynamic graphs. These networks can efficiently process the spatial and temporal dependencies to induce new information. We mention several approaches to improve dynamic GNNs through graph learning. However, these methods do not capture the two-way relationship between the network structure and its dynamics. Lastly, we highlighted how a neural network is usually trained offline, where the models' parameters no longer change after the training phase. Due to the highly dynamic data of problems, an online method can be preferred.

We use an adaptive dynamic graph to capture the network's evolution. We expand on dynamic GNNs to use adaptive dynamic graphs and capture the two-way relationship between the structure and the dynamics. To ensure constant graph adaptation, we use an online mechanism.

3 Related work

This chapter discusses the relevant literature on online adaptive dynamic graphs and learning from the data residing over them. We divide this section into dynamic graph approaches, dynamic graph learning/adapting approaches, and online learning on graphs. We illustrate the classification of the works in Table 3.1.

3.1. Learning from Dynamic graphs

A dynamic graph is a model for a network with time-varying topology, i.e., edges and nodes can exist at time t but not at time $t+1$. We can represent a dynamic graph in two ways, either discrete or continuous. In a discrete setting, we know the entire graph at discrete intervals, so-called graph ”snapshots” [5, 6] or snapshot-based temporal graphs (STG) [4]. In a continuous setting, a common representation is an event-based model that encapsulates changes in a sequence of events, each accounting for a minimal change, i.e., the addition or deletion of an edge or a node.

Several surveys have collected dynamic graph tasks, which include link prediction, edge classification, node clustering and classification, anomaly detection, event time prediction, and graph classification [25, 6, 4]. We distinguish between classification tasks and regression tasks. Classification tasks involve classifying the graph, such as classifying the entire graph, just one node or edge, or a group of nodes and edges. Regression tasks like link or event-time predictions aim to predict the graph structure. In addition, tasks combining these two categories include node clustering, anomaly detection, or recommender systems.

3.1.1. Methods

We group work on dynamic graphs into two categories: statistical and machine learning models. Statistical models use mathematical assumptions to perform the desired tasks. Authors in [26, 27, 28] extend matrix factorization models for static graphs into a dynamic setting. The works in [29] and [30] apply random walks on dynamic graphs while [31] introduces random causal walks to account for temporal causality of the data. These models achieve an acceptable performance even with a small training set; however, these methods usually rely on assumptions, do not scale well, and are not adaptable to the task. An often-explored strategy is to use machine learning methods. Such methods allow for task-specific networks that perform well on larger scales.

Recurrence

Since dynamic graphs contain a temporal component, in many machine learning-based approach approaches, it is most common to use a recurrent component. A Graph Convolutional Network (GCN) is often used at a timestep to compute the spatial dependencies due to their proven usefulness on graphs. Gated recurrent units (GRU) and long-term-short-term (LSTM) networks are used to compute the temporal dependencies between timesteps [11, 7, 32]. In [9], the authors propose a new Dice metric to measure the similarity between a node and its neighbors and use these to improve the computation of the spatial dependencies in the GCN layer. In [8], the authors use a recurrent component to ”evolve” the GCN weights, not to learn actual node embeddings from the GCN. Work in [33] uses message passing for spatial dependencies instead of convolutions, where an update and propagation component makes the model suitable for an event-based setting.

Due to the limited capabilities of GCNs to learn a structure, several works have proposed to use graph attention [20]. Such approaches are common in recommender systems [34, 35].

Temporal attention
RNNs suffer from fixed temporal connections and a sequential nature preventing parallel computations. Temporal attention models can address both of these issues. By using attention to learn the temporal dependencies, an RNN is no longer required [36, 37]. The works in [10, 38, 39] use self-attention in so-called Temporal Graph Attention (TGAT) to aggregate temporal and spatial dependencies into a single graph.

Autoencoder
The advantage of autoencoders lies in their ability to use unlabeled data and their applicability to various downstream tasks. Work [40] uses a supra-adjacency matrix, combining graphs in a time window into one large graph and using an autoencoder to learn a representation for downstream tasks. The work in [41] uses a Variational Graph Recurrent Neural Network (VGRNN) to learn representations, improving previous work by adding latent random variables.

However, several factors limit dynamic graphs. For example, a dynamic graph based on correlations represents the network correlations at the specific timestep. It does not represent the network's evolution over time. In addition, in many real-life scenarios, the graph might be unavailable, be constructed from static prior data, or incorrectly represent the current state of the network.

3.2. Learning/Adapting dynamic graphs

The literature surrounding dynamic graph learning and adapting dynamic graphs are closely related, often interchanging definitions between these fields. However, these two fields are different; in graph learning, the goal is to learn some network representation to improve the task performance, but not necessarily based on any physical property. When adapting a dynamic graph, we adjust the graph from a physical perspective, modifying the network topology based on the data evolution. An adaptive dynamic graph is thus more constrained and similar to a physical network. This section discusses approaches based on graph learning or a mix of adaptive and graph learning.

3.2.1. Methods

Graph learning/adapting comes from works in traffic flow prediction. They argue against pre-defined graphs, as they cannot capture the relations between nodes well enough over time [42, 17, 13]. Several works use graph learning from a deep learning perspective, using learnable hidden node embedding matrices in combination with a learnable difference layer. The layer subtracts the two node embedding matrices to construct a graph at a timestep [24, 12]. The work in [43] improves the difference layer by incorporating attention scores. In addition, they construct the initial adjacency matrix using several different metrics to improve initial graph construction. In [17], the authors extend the difference layer above by using the results of a graph convolution layer. The graph convolutions are computed on the node features, thus incorporating the network dynamics in the graph learning.

Link predictor
A link predictor is used to compute the edge probabilities to construct a distribution for the network. The predictor is a pre-trained model, not necessarily for the applied domain. Works using these, sample a graph using the edge probabilities and use it for downstream tasks [16, 44].

Learning the distribution
The core of the link predictor strategy lies in having a well-working predictor, which is task-specific and does not necessarily translate between domains. For this, works improve the predictor by learning a distribution for the graph and sample from it, allowing the network to learn a task-specific distribution [16, 18, 13]. The work in [45] uses a denoising module that denoises the input graph by creating a subgraph filter to remove task-irrelevant edges. In [14], the authors propose an iterative model. By using layers of GCNs, during which the node embeddings are saved as observation graphs, the model trains for the

task. The observation graphs, the initial adjacency matrix, and the labels are combined to create a graph estimator.

Approaches using a link predictor or learning the distributions are focused on learning the graph but do not capture the data evolution.

Applying metrics
The limitation of learning a distribution is that we focus on the graph's structure without considering the data. Similarity-based metrics or attention are similar to dynamic graphs in that they compute a score based on the current state of the data. The difference is that these compute the graph at each in these methods, whereas a dynamic graph is an input [46, 42].

Learning based on metrics
By applying metrics, methods can capture the dynamics of a network at the timestep. However, they do not capture the evolution of the network. In [19], the authors argue theoretically that a two-way relationship exists between a network's structure and its dynamics. The structure of a network influences its dynamics and vice versa. They define an adaption function for an adjacency matrix that depends on the states, the history of nodes, and the previous states of the adjacency matrices. [47] argues similarly that a better graph means better node embeddings and vice versa. Using an iterative approach, they optimize the node embeddings and the graph simultaneously for the downstream task. In [15], the authors construct an adaptive graph at each timestep by computing a self-defined metric between nodes to define the Laplacian.

Methods involving graph learning/adapting approaches aim to construct a graph best suited for the downstream task. Depending on the method used, the network dynamics are either directly or indirectly taken into account. However, all the solutions have the problem of the graph becoming static when such a model has finished training, while the underlying network is always dynamic and constantly evolving during testing. The network in the real world could differ significantly from the training environment.

3.3. Online learning on graphs

In online graph learning, approaches focus on learning the graph regardless of whether the model is training or testing. For this, a differentiable function must be constructed and optimized. Works in online graph learning make assumptions about the data and the network, so-called data models. Authors in [48, 49] assume a smooth signal and compute prior smoothness over the data. They learn the graph by leveraging the priors and using smoothness properties such as sparsity and connectivity [50, 51]. In other works [52, 53], instead of a smooth signal, a heat diffusion process is assumed. However, requiring priors about the data restricts the possibilities of the algorithm and does not allow for real-time decision-making. Therefore, several works use model-independent or prior-free approaches. Authors in [54] assume a static data model in which neighboring nodes behave similarly to the data. Work in [55] proposes graph-learning approaches for different signal assumptions, including a corrective mechanism for forecasting tasks.

3.4. Position of this thesis

We position this thesis as a blend of an adaptive dynamic graph approach using metric learning and online graph learning. We build upon the model proposed in [19] into that of a GNN, resulting in an adaptive GNN. To prevent the need for data assumptions, we learn how to adapt the graph for the downstream task in the training phase, but the actual adaptation of the graph is an online operation. An online operation ensures that the model is always adapted.

Type	Category	Works
Dynamic graph	Statistical	[28, 30, 31, 27, 29]
	Recurrence	[8, 32, 34, 9, 7, 35, 33, 11]
	Temporal attention	[37, 10, 36, 39]
	Autoencoder	[40, 26, 41, 38]
Graph learning	Node embeddings	[43, 24, 12, 17]
	Link predictor	[44, 16]
	Learning distribution	[13, 18, 14, 45]
	Apply metrics	[42, 46]
	Learn based on metrics	[47, 15]
Online graph learning	Smoothness	[51, 48, 49, 55, 50]
	Diffusion	[52, 53]

Table 3.1: Table showing the structure of works in the field related to adaptive dynamic networks.

Part II

Implementation

4 Proposed method

In this chapter, we propose the Online Adaptive Graph Neural Network (OAGNN) model that adapts the graph using the graph attention coefficients in an evolving manner. As shown in Figure 4.1, the model consists of three modules: the EvolvingGA and Adaptation of W_t modules define the adaptation process, and the prediction module generates the output. The EvolvingGA module is the first step in the adaptation process, in which we compute node similarity scores. Then, we update the edge weights using an online step, resulting in the adapted adjacency matrix $\mathbf{A}_t$. The prediction module uses the adapted adjacency matrix to predict the labels. Due to its online nature, we continuously adapt the graph, even when deployed in the real world. In addition, no weights or embeddings exist that rely on the number of nodes, allowing for domain changes when deployed in the real world. In a non-evolving or traditional approach, introducing a temporal component to graph attention scales its complexity linearly with the number of timesteps processed. However, an evolving approach to induce temporal dependencies between timesteps avoids linear growth in time. In addition, in [8], the authors argue that this has the advantage of better handling dynamic data as a node does not always have to be present.

We construct the full OAGNN model by combining the characteristics of the adaptation operator with that of a GNN for dynamic graphs. We define a message passing function

$$\mathbf{x}'_{ti} = f(\mathbf{x}_{ti}) + \sum_{j=1}^{N} \mathbf{A}_{tij} \cdot g(\mathbf{x}_{ti}, \mathbf{x}_{tj}), \tag{4.1}$$

where $\mathbf{x}_{ti}$ and $\mathbf{x}_{tj}$ are node features at timestep t for nodes i and j respectively, $\mathbf{x}'_{ti}$ the new node features for node i, the MLPs $g(\cdot)$ and $f(\cdot)$ to transform the features, and $\mathbf{A}_{tij}$ the result from the graph adapting operator (2.2).

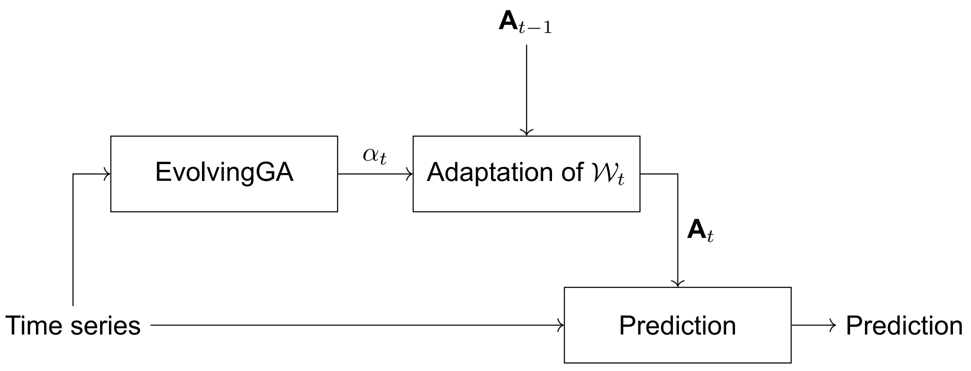

Figure 4.1: High-level overview of the OAGNN model. The EvolvingGA and Adaptation of W_t define the adaptation process.

4.1. Evolving Attention Weights

To construct the first module, the evolving graph attention module (EvolveGA), we translate the general self-attention function (2.10), with the learnable weight matrix **W**, to a time-dependent variant as

$$\alpha_{tij} = \text{softmax}(a(\mathbf{W}_t\mathbf{x}_{ti}, \mathbf{W}_t\mathbf{x}_{tj})), \tag{4.2}$$

where $\mathbf{x}_{ti}$ and $\mathbf{x}_{tj}$ are node features, $a(\cdot)$ any self-attention mechanism, t indicates the timestep and $\mathbf{W}_t \in \mathbb{R}^{F' \times F}$. However, due to the complexity of learning $\mathbf{W}_t$ for each timestep, we instead *evolve* it. We construct the recurrence function $f_{RW}(\cdot)$ that models the transition of the weight matrix as

$$\mathbf{W}_t = f_{RW}(\mathbf{W}_{t-1}). \tag{4.3}$$

Attention mechanisms might use additional weights next to $\mathbf{W}_t$, requiring a separate recurrence function such as $f_{RW}(\cdot)$ for each weight. In the graph adaption, we use the resulting graph attention coefficients α_{tij} to express the similarity between two nodes. Because of this, we repurpose the weight matrix $\mathbf{W}_t$ to transform the initial feature dimension F to a hidden dimension F', such that $F' > F$, to increase the expressiveness of the self-attention module $a(\cdot)$.

4.2. Graph Adaptation

Adapting a network to its dynamics allows the graph to evolve as the network changes. In many networks, such as traffic or financial networks, where new conditions can influence the entire network structure, more than a (pre-trained) static graph might be needed. We use an online mechanism to adapt the graph, keeping the model coherent with the network dynamics even in the real world. We update the edge weights using the output of the evolving graph attention module α_{tij} (4.2).

To adapt the edge weights for the adjacency matrix $\mathbf{A}_t$, we define an update step with the previous adjacency matrix as

$$\mathbf{A}_{tij} = \gamma\alpha_{tij} + (1-\gamma)\mathbf{A}_{(t-1)ij}, \tag{4.4}$$

where α_{tij} (4.2) is the graph attention score, and $\mathbf{A}_{(t-1)ij}$ the edge weight between node pair (i, j) at the previous timestep. The hyperparameter γ indicates the rate of graph adaptation. In the adaptation process, due to the complexity of graph adaptation, we compute the graph attention scores over the neighborhoods defined in $\mathbf{A}_0$ and solely adapt these edges.

4.3. Predicting with the adapted graph

We illustrate the prediction module in Figure 4.2. To utilize the adapted adjacency matrix $\mathbf{A}_{tij}$ (4.4), we construct a multi-layer message passing neural network, where a single layer l is defined as

$$\mathbf{x}_{ti}^{(l+1)} = MP_l(\mathbf{x}_{ti}^{(l)}, \mathbf{A}_t) = \mathbf{W}_{l2}^T(\sum_{j=1}^{N} \mathbf{A}_{tij} \cdot g_l(\mathbf{x}_{ti}^{(l)}, \mathbf{x}_{tj}^{(l)}) + \mathbf{b}_{l1}), \tag{4.5}$$

with message passing function,

$$g_l(\mathbf{x}_{ti}^{(l)}, \mathbf{x}_{tj}^{(l)}) = g_l(\mathbf{x}_{tj}^{(l)}) = \sigma(\mathbf{W}_{l1}^T\mathbf{x}_{tj}^{(l)}), \tag{4.6}$$

where $\mathbf{A}_{tij}$ is the adapted edge and the MLP $g(\cdot)$ which is solely dependent on the value of $\mathbf{x}_{tj}$. The learnable inner weight $\mathbf{W}_{l1}$, with l indicating the layer, is used to scale the input features to a hidden dimension with shape $\mathbb{R}^{d_l}$, and a bias $\mathbf{b}_{l1} \in \mathbb{R}^{d_l}$. We use the outer learnable weight $\mathbf{W}_{l2}$ to scale the hidden dimension to the desired output dimension $\mathbb{R}^{d_{out}}$. The message passing layer repeats for several layers to include the k-hop neighbors, such as

$$MP_L(\mathbf{x}_{ti}) = \sum_{l=1}^{L} MP_l(\mathbf{x}_{ti}^{(l)}), \tag{4.7}$$

where L indicates the number of layers. We define an additional skip connection consisting of a single linear transformation that does not take into account the graph structure

$$f(\mathbf{x}_{ti}) = \mathbf{W}_3^T\mathbf{x}_{ti} + \mathbf{b}_3. \tag{4.8}$$

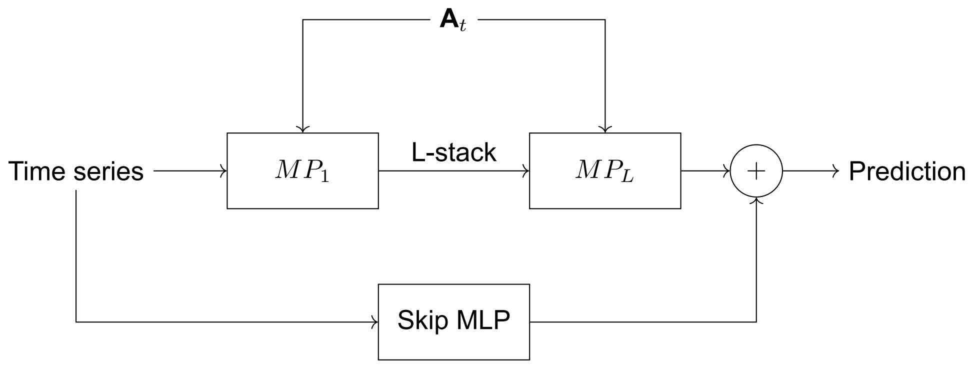

Figure 4.2: Overview of the prediction module. It consists of L stacked message passing module that uses the adapted graph and a single skip connection MLP.

We define the full prediction module as

$$\hat{\mathbf{y}}_{t+1,i} = f(\mathbf{x}_{ti}) + MP_L(\mathbf{x}_{ti}, \mathbf{A}_t), \tag{4.9}$$

where $\hat{\mathbf{y}}_{t+1,i} \in \mathbb{R}^{d_{out}}$, and d_{out} is the output dimension.

4.4. Implementation

In implementing multivariate timeseries analysis, we define a window of inputs instead of a single timestep. The input consists of $\mathcal{X}_{t-w:t} \in \mathbb{R}^{N \times w \times F}$ and the previous adjacency matrix $\mathbf{A}_{t-1}$ to predict the label as

$$\hat{\mathbf{Y}}_{t+1}, \mathbf{A}_t = f(\mathcal{X}_{t-w:t}, \mathbf{A}_{t-1}), \tag{4.10}$$

where $\hat{\mathbf{Y}}_{t+1} \in \mathbb{R}^{N \times d_o}$ is the predicted label with output dimension d_o. For a single node i, we define

$$\hat{\mathbf{y}}_{t+1,i} = f(\mathbf{X}_{t-w:t,i}) + \sum_{j=1}^{N} \mathbf{A}_{tij} \cdot g(\mathbf{X}_{t-w:t,i}, \mathbf{X}_{t-w:t,j}), \tag{4.11}$$

where $\mathbf{A}_{tij}$ is the output of the adaptation operator H_{ij}^t (2.2), the window of features for node i as $\mathbf{X}_{t-w:t,i} \in \mathbb{R}^{w \times F}$, similarly defined for node j, and $g(\cdot)$ and $f(\cdot)$ some MLP. For the implementation we flatten all inputs such that $\mathbf{X}_{t-w:t} \in \mathbb{R}^{N \times wF}$ and for a single node $\mathbf{x}_{t-w:t,i} \in \mathbb{R}^{wF}$.

We implement the model: we define a GRU layer for the recurrence function of evolving the attention weights. We use the attention mechanism introduced by Graph Attention [20], defined as (2.14). Graph Attention has the additional weight vector $\mathbf{a}_t \in \mathbb{R}^{F'}$ to evolve, for which we define another GRU layer. As a GRU layer defines two inputs, the input features and the optional hidden state, we set both inputs to the weight of the previous timestep. We define the evolving attention operator as

$$\alpha_{tij} = \frac{\exp\big(\text{LeakyReLU}(\mathbf{a}_t^T[\mathbf{W}_t\mathbf{x}_{t-w:t,i} \parallel \mathbf{W}_t\mathbf{x}_{t-w:t,j}])\big)}{\sum_{k \in \mathcal{N}_i} \exp\big(\text{LeakyReLU}(\mathbf{a}_t^T[\mathbf{W}_t\mathbf{x}_{t-w:t,i} \parallel \mathbf{W}_t\mathbf{x}_{t-w:t,j}])\big)}, \tag{4.12}$$

where $\mathbf{x}_{t-w:t,i} \in \mathbb{R}^{wF}$ is the flattened input and $\mathbf{W}_t \in \mathbb{R}^{wF \times F'}$ to account for the windowed input. The pseudo-code for evolving attention is visible in Algorithm 1. The pseudo-code for the full Online Adaptive Graph Neural Network model can be seen in Algorithm 2. Note that in the experiments we focus on multivariate timeseries *forecasting* for which we predict the next sequence of timesteps as $\hat{\mathcal{Y}}_{t+1:t+h} \in \mathbb{R}^{N \times h \times d_0}$, where h is the forecasting horizon as

$$\hat{\mathcal{Y}}_{t+1:t+h}, \mathbf{A}_t = f(\mathcal{X}_{t-w:t}, \mathbf{A}_{t-1}).$$

Algorithm 1: Evolving Graph Attention

```
$\mathbf{W}_0 \leftarrow (F, F')$
$\mathbf{a}_0 \leftarrow (2F')$
Function EvolvingGA($\mathbf{X}_t$, $\mathbf{A}_t$):
    $\mathbf{W}_t = GRU(\mathbf{W}_{t-1}, \mathbf{W}_{t-1})$
    $\mathbf{a}_t = GRU(\mathbf{a}_{t-1}, \mathbf{a}_{t-1})$
    for $(i, j) \in \{\mathbf{A}_t | \mathbf{A}_{tij} > 0\}$ do
        $\alpha_{tij} \leftarrow$ result from equation (4.12)
    end
    return
```

Algorithm 2: Online Adapting Graph Neural Network

```
Function OAGNN($\mathcal{X}_t$):
    $\mathbf{X}_t \leftarrow$ flatten($\mathcal{X}_t$)
    $\mathbf{A}_t \leftarrow$ using $\alpha_{tij}$ from Evolving Graph Attention algorithm (1)
    $\hat{\mathbf{Y}}_{t+1} \leftarrow$ result from equation (4.9)
    return $\hat{\mathbf{Y}}_{t+1}, \mathbf{A}_t$
```

4.5. Discussion

The OAGNN model has the advantage of adapting the graph from a physical point of view, modeling the two-way relations of the structure and dynamics in the network. It is always adaptive, even when deployed in the real world, and the model does not rely on data assumptions or hidden node embeddings. However, we acknowledge its limitations; first, the model does not allow for edges to be changed. Only the edges defined in the initial adjacency matrix are adapted. This limits the model's "freedom" in adapting the graph and the possibility of changes in the node set. We mitigate most of this limitation by integrating the *k*-hop neighbors through the prediction module. However, the limitation still exists. The model uses a fixed hyperparameter γ for the adapt rate. The adapt rate would change in an ideal scenario depending on the network dynamics. Lastly, we use the implementation of graph attention for the similarity scores. We have not evaluated this against other methods.

The OAGNN model's complexity comes from the three modules. The EvolvingGA module uses the graph attention operator without the feature transformation. With K indicating the number of heads, we have $\mathcal{O}(KNFF'_2)$. We omit the complexity of evolving the weights as this is not dependent on the N or M. The adaptation through $\mathcal{W}_t$ solely sums over the edges with complexity $\mathcal{O}(M)$. Lastly, the prediction module uses message passing over all nodes for L layers. One message has complexity $\mathcal{O}(MF'_1)$, where we omit the input feature dimension and M is the number of edges. We define the model's full complexity as

$$\mathcal{O}(KNFF'_2 + M + LMF'_1).$$

where $L \geq 1$. This results in linear scaling with the number of nodes and edges.

The work in [17] comes most similar in task and approach by incorporating the network dynamics. However, it uses hidden node embeddings, does not capture the two-way relationship between the network structure and its dynamics, and is not online. In [47], the authors use a similar approach to the two-way relationship but do not design the model for the tasks we focus on in this work. The focus is on optimizing the graph for time-independent data. Additionally, the work does not capture graph changes online.

5 Experiments and Results

In this section, we discuss the experiments and their results. First, we describe all the experiments performed and the datasets used. Then, we discuss the performance using several metrics and the graph adaptation process. We conduct experiments on several prediction horizons and tasks depending on the dataset.

5.1. Baselines

We perform several baseline experiments to compare the performance of the proposed model. The baseline approaches include methods that rely on static graphs, graph learning based on node embeddings, and graph distribution learning methods. We do not compare with fully online approaches as they cannot keep up with the performance. We train the baselines as closely as possible for the performance comparisons following the hyperparameters defined in the respective works (if available). Additional information on each baseline and their configurations is in Appendix B.

- **GCRN** [56]: baseline model which uses a single layer of the Chebysev Graph Convolutional Gated Recurrent Unit Cell. Then, we use a readout layer. The graph is not adapted.
- **EvolveGCN** [8]: baseline evolving model that implements an evolving GCN. The model is designed for link prediction, edge classification, or node classification, but we translate it to timeseries forecasting. The graph is not adapted.
- **MTGNN** [24]: baseline graph learning model. The graph is learned through a difference layer using node embeddings.
- **DGCRN** [17]: baseline graph learning model. Improves on MTGNN [24] by incorporating GCN outputs that use the node features in their graph learning layer. The graph is learned through a difference layer using node embeddings.
- **ADLNN** [43]: baseline graph learning model. It incorporates results from self-attention in a similar difference layer as MTGNN [24]. The graph is learned through a difference layer using node embeddings.
- **ASTGAT** [13]: baseline graph learning model. The graph is learned by learning the distribution of its structure.
- **MGDPR** [22]: baseline financial prediction model. Only compared with the FINANCE dataset. The model generates graphs using precomputed graphs based on entropy and signal energy. We do not rerun this model.

5.2. Models

We propose a single strategy for the OAGNN model. We perform a hyperparameter search for the most impactful hyperparameter for the OAGNN model in Appendix A.

- **OAGNN**: the full Online Adaptive Graph Neural Network as proposed in section 4.

5.3. Datasets

We test the performance of our models on four datasets, including two traffic datasets, a windmill energy forecasting dataset, and a financial dataset. Both traffic datasets have been used prior in works using graph learning for forecasting problems, giving us a direct performance comparison. We use windmill energy forecasting and finance to introduce a more applicable setting for adaptive networks.

- **PEMS-BAY**: a traffic speed dataset consisting of 325 sensors and measurements from January 2017 to May 2017 in 5-minute intervals. The dataset contains two features at each timestep: the measured speed and the relative time. We construct the initial adjacency matrix using a metadata file containing the longitude and latitude of each sensor. We apply kNN with $k = 20$ and create an undirected graph.
- **METR-LA**: a traffic speed dataset of 207 loop detectors and measurements between March 2012 and June 2012 in 5-minute intervals. The initial adjacency matrix is a static pre-defined adjacency matrix. The dataset comprises two features at each timestep: the measured speed and the relative time. This dataset is regarded as more complex than the PEMS-BAY dataset.
- **SDWPF**: a wind power forecasting dataset introduced in [21] consisting of 134 wind turbines with measurements over half a year in 10-minute intervals. We apply kNN with $k = 20$ using the locations of the windmills. We use the SDWPF dataset to introduce a more applicable setting for adaptive timeseries forecasting. The SDWPF dataset uses ten input features: the wind speed in m/s, the wind direction in degrees, the temperature of the surrounding environment and the internal temperature of the turbine nacelle, the nacelle direction in degrees, the direction of all three blades in degrees (as three separate features), and the reactive and active power in kilowatt (kW).
- **FINANCE**: dataset from [22] containing financial timeseries from NASDAQ, from 2013-01-01 to 2017-12-31 in daily intervals. We use the same 1026 timeseries as in [22]. The features include the open and close price, the high and low for the day, the volume, and a trend class (binary, 0 or 1) indicating if today's close is less than the previous. We z-score normalize the full dataset and construct the initial adjacency matrix by computing the correlation matrix and taking the top $k = 20$ neighbors for each node. We fill in missing data with zeros.

5.4. Hyperparameters

We train the OAGNN model using the Adam Optimizer; 100 epochs, average the results over five runs, and use early stopping with patience 20. For the traffic and SDWPF datasets, we use batch size 64 and a train, validation, and test split of 75, 10, and 15, respectively. For the FINANCE dataset, we use batch size six and a split of 40, 8, and 52 to reproduce [22] as closely as possible. We define all hyperparameters and results from the hyperparameter search for METR-LA, SDWPF, and FINANCE in Appendix A.

We use the DelftBlue [57] supercomputer to train and evaluate our models. We have a single Intel XEON E5-6248R 24C 3.0GHz, with 128G of RAM. As our GPU, we used an NVIDIA Tesla V100s 32GB. We implement all models using PyTorch, PyG, and PyTorch Geometric Temporal [58].

5.5. Learning and Evaluation metrics

We forecast future measured speed for the traffic datasets, and for the SDWPF dataset, the goal is to forecast future active power usage. We forecast a single value such that $d_0 = 1$. We use an input window $w = 12$, predicting $h = 3$ and $h = 12$. To train the traffic models, we use the MAE loss as defined in (2.3), and we evaluate using the MAE loss and the RMSE loss (2.4). Since we predict a horizon for all nodes, we redefine the MAE at timestep *t* as follows

$$MAE_t = \frac{1}{NH} \sum_{i=1}^{N} \sum_{h=1}^{H} |\hat{y}_{hi} - y_{hi}|, \tag{5.1}$$

where N is the number of nodes and H is the horizon. We define the RMSE loss similarly. For both losses, we average the loss over all time steps. Models on the SDWPF dataset are trained and evaluated based on a combination of the MAE and the RMSE loss as defined in [21]. The SDWPF dataset contains several special rules in pre-processing and evaluation, which we all apply to the dataset, and we include the MAE loss and the *sd*MAE as evaluation metrics. The *sd*MAE loss is the *scaled disjoint* MAE, which only evaluates non-overlapping horizons and is scaled by the average value of the target, defined as

$$\mathcal{L}_{sdMAE} = \frac{\mathcal{L}_{dMAE}}{\frac{1}{N}\sum_{i=1}^{N} y_i}, \tag{5.2}$$

with,

$$\mathcal{L}_{dMAE} = \frac{1}{K}\sum_{t=1}^{K} MAE_{Ht}, \tag{5.3}$$

K is the number of sliding windows fitting into the test set, and H is the horizon. On the FINANCE dataset, the goal is to correctly classify the trend class of the next timestep ($h = 1$). We use an input window $w = 21$ and train and evaluate the model using the cross-entropy loss (2.5). Additionally, we assess performance using accuracy and the F1 score.

5.6. Results

For each dataset, we display the performance results and the adaptation process on the test set. We plot the average values for selected features for each timestep to highlight the dynamics of the dataset and, from a single run, the Frobenius norm of the difference of the adjacency matrix at the current timestep normalized by the average value of the edge weights to highlight the amount of adaptation. Note that for all plots, we omit the first timestep for clarity because the model has adapted to the training and validation data, and when moving to the test set, the data is new, and much adaptation happens.

In Table 5.1, we present the performance results for METR-LA, and we see that the model cannot outperform some of the graph-learning-based approaches. We attribute this to the data in the METR-LA dataset and its effect on the graph adaptation. Figure 5.1 provides a visual representation of this, displaying the adaptation process and the frequent sudden zeroes in the data (top) to which the OAGNN model adapts (bottom). With drastic changes in the data, the network dynamics are significantly different from the previous instances. This difference in dynamics negatively impacts the model's performance in this case.

Table 5.1: Prediction performance with standard deviation on METR-LA dataset for horizons 3 and 12. Averages of 5 runs. Lower is better.

Model	$h = 3$		$h = 12$	
	MAE	RMSE	MAE	RMSE
GCRN	3.40±0.14	5.90±0.07	5.05±0.04	9.04±0.04
EvolveGCN	9.45±0.61	12.65±0.44	10.26±0.23	14.09±0.32
MTGNN	4.81±0.54	9.00±1.07	6.53±0.10	12.15±0.14
DGCRN	**2.89**±0.06	**5.12**±0.02	**2.93**±0.23	**5.62**±0.32
ASTGAT	5.09±1.33	7.97±1.47	6.18±0.46	10.21±0.34
ADLNN	2.94±0.02	5.58±0.03	4.51±0.03	8.55±0.08
OAGNN	3.22±0.01	5.84±0.01	4.96±0.02	9.09±0.02

Table 5.2 displays the performance results on the PEMS-BAY dataset, showing that the model outperforms the baselines. Looking at the adaptation process (Fig. 5.2), we see the dataset contains a single zero instance drastic change, thus not negatively impacting the performance as in METR-LA. The PEMS-BAY dataset displays more harmonic and consistent data than the METR-LA dataset; however, in the adaptation, we see that the model does not care for patterns in the data. The difference in average speed between each timestep is more significant for PEMS-BAY than METR-LA, resulting in more adaptation.

On the SDWPF dataset, we see the OAGNN model significantly outperform all baselines (Table 5.3), with the adaptation process in Figure 5.3. Compared to the traffic datasets, the plotted features (top), the

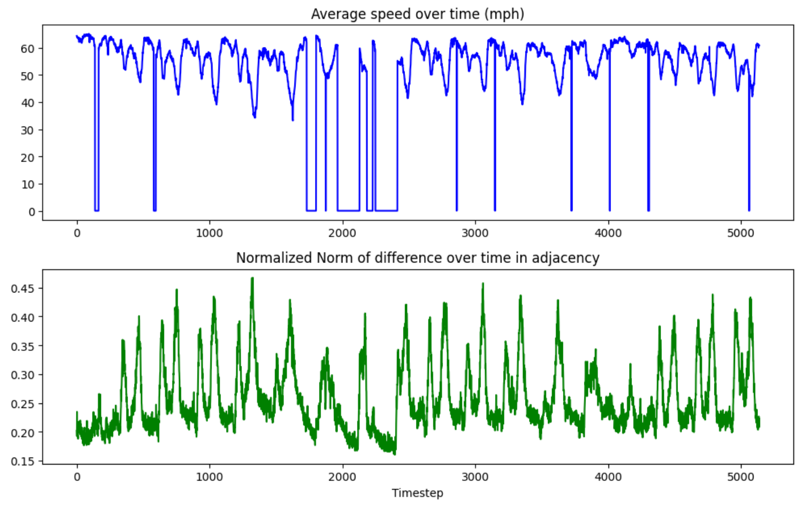

Figure 5.1: METR-LA. The average speed of the window at each timestep plotted (top), and the normalized norm of difference of the adjacency matrix at each timestep (bottom). Adapt rate $\gamma = 0.01$.

Table 5.2: Prediction performance with standard deviation on PEMS-BAY dataset for horizons 3 and 12. Averages of 5 runs. Lower is better.

Model	$h = 3$		$h = 12$	
	MAE	RMSE	MAE	RMSE
GCRN	2.52±0.13	3.56±0.17	3.55±0.24	5.34±0.29
EvolveGCN	6.34±0.38	8.40±0.35	6.27±0.31	8.34±0.31
MTGNN	2.38±0.17	4.14±0.32	2.82±0.04	4.81±0.04
DGCRN	2.38±1.58	3.78±2.33	2.42±0.19	**4.19**±0.28
ADLNN	2.47±0.04	4.46±0.05	2.54±0.03	4.57±0.02
OAGNN	**1.35**±0.03	**2.50**±0.03	**2.20**±0.01	4.21±0.02

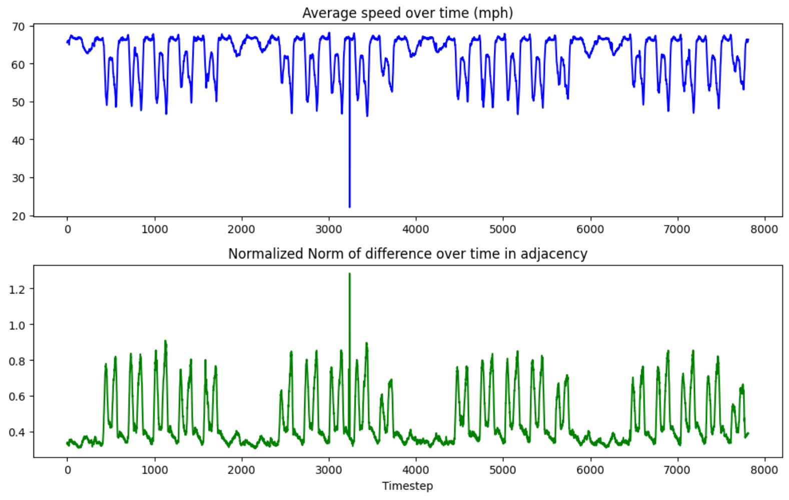

Figure 5.2: PEMS-BAY. The average power of the window at each timestep plotted (top) and the normalized norm of the difference of the adjacency matrix at each timestep (bottom). Adapt rate $\gamma = 0.01$.

Table 5.3: Prediction performance and standard deviations for horizons 3 and 12 on the SDWPF dataset. Averages of 5 runs. All results are our own, and the MAE+RMSE loss refers to the loss function introduced in [21] and defined in **MW**. We define the MAE and *sd*MAE losses in **kW**. Lower is better. *The model did not always converge.

	$h = 3$			$h = 12$		
Model	MAE+RMSE	MAE	*sd*MAE	MAE+RMSE	MAE	*sd*MAE
GCRN*	126.08±1.42	94.09±1.06	0.36±0.00	180.72±26.2	134.87±19.6	0.47±0.08
EvolveGCN*	116.02±13.9	86.58±10.4	0.29±0.01	167.20±6.21	124.78±4.63	0.42±0.03
MTGNN	81.88±3.10	61.11±2.98	0.24±0.01	119.23±5.12	88.98±3.82	0.32±0.01
ADLNN	80.26±1.45	59.90±1.08	0.23±0.01	126.31±4.68	94.26±3.49	0.33±0.01
OAGNN	**61.93**±0.86	**46.22**±0.64	**0.18**±0.00	**107.17**±0.34	**79.98**±0.00	**0.29**±0.05

Table 5.4: Class prediction performance with the standard deviation for horizon 1 on the **FINANCE** dataset. Averages of 5 runs. Higher is better. *Results from [22].

Model	Acc(%)	F1
GCRN	66.09±0.01	0.66±0.00
EvolveGCN	65.43±0.04	0.65±0.00
MTGNN	50.01±0.04	0.50±0.00
ADLNN	50.08±0.03	0.51±0.01
MGDPR*	62.77±0.65	0.62±0.01
OAGNN	**66.35**±0.25	**0.67**±0.00

reactive and active power, are not harmonic and seem more sporadic, and the dynamics of the features change over time. For example, seasons influence the dynamics of many features over a more extended period, and the weather is much more impactful in wind power forecasting. We see that the static graph approaches cannot always converge on the SDWPF dataset, and the graph learning approaches cannot capture the complex network dynamics in the graph, highlighting the difficulty of the dataset, whereas the OAGNN model performs better. We see the difficulty in capturing the network dynamics by inspecting the adaptation process (bottom), in which the model requires significant change between timesteps to adapt the graph to the dynamics. We hypothesize that this results from the highly dynamic environment of SDWPF. However, it can result from the model's inability to capture the complex dynamics.

The model outperforms the baselines in Table 5.4 for FINANCE, but the static graph approaches perform similarly well. We can conclude from this that the graph matters less in the FINANCE dataset than in the other datasets. We plot the average close and volume for the FINANCE dataset (Fig. 5.4), and we see the evolving network dynamics through the close prices, where the value starts at ± 0.02 and ends at ± 0.2 without frequent sporadic movements. The difference graph plot shows us that the model can consistently capture these changes in the dynamics through minimal changes compared to the SDWPF dataset, where the model requires more adaptation. We conclude from this that the model can accurately capture the dynamics of an evolving network. However, the standard deviation of most baselines performs in a lower order compared to the OAGNN model. We argue that due to the larger test set on the FINANCE dataset and the online nature of the model, we see better but less stable behavior over a more extended period. We conclude that the OAGNN model is generally more consistent over the datasets due to its adaptive nature and better at capturing the evolution of a dynamic network; however, with smaller training and larger test sets comes more difficulty in ”correctly” capturing the dynamics of an evolving network.

5.6.1. Convergence analysis

We analyze the convergence of the graph adaptation during the training phase of the model by studying the Frobenius norm of the difference between the current adapted adjacency matrix and the previous one. The adaptation module must converge during training to adapt the graph ”correctly” in the real world.

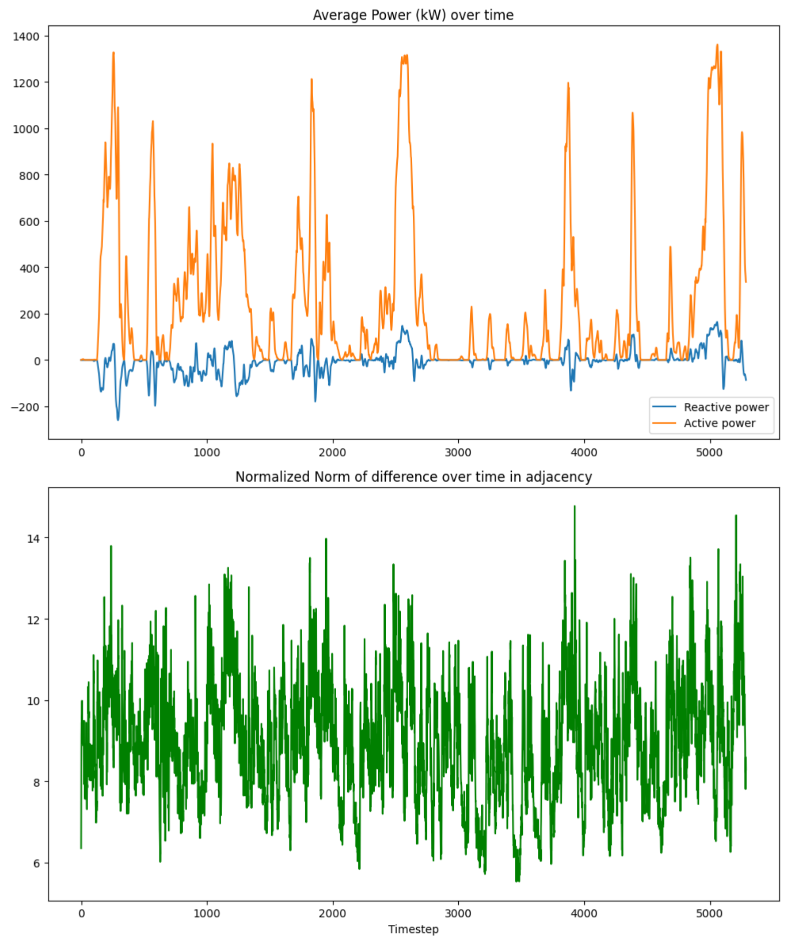

Figure 5.3: SDWPF. The average power of the window at each timestep plotted (top) against the norm difference of the adjacency matrix at each timestep (middle) and the normalized norm of difference at each timestep (bottom). Adapt rate $\gamma = 0.1$.

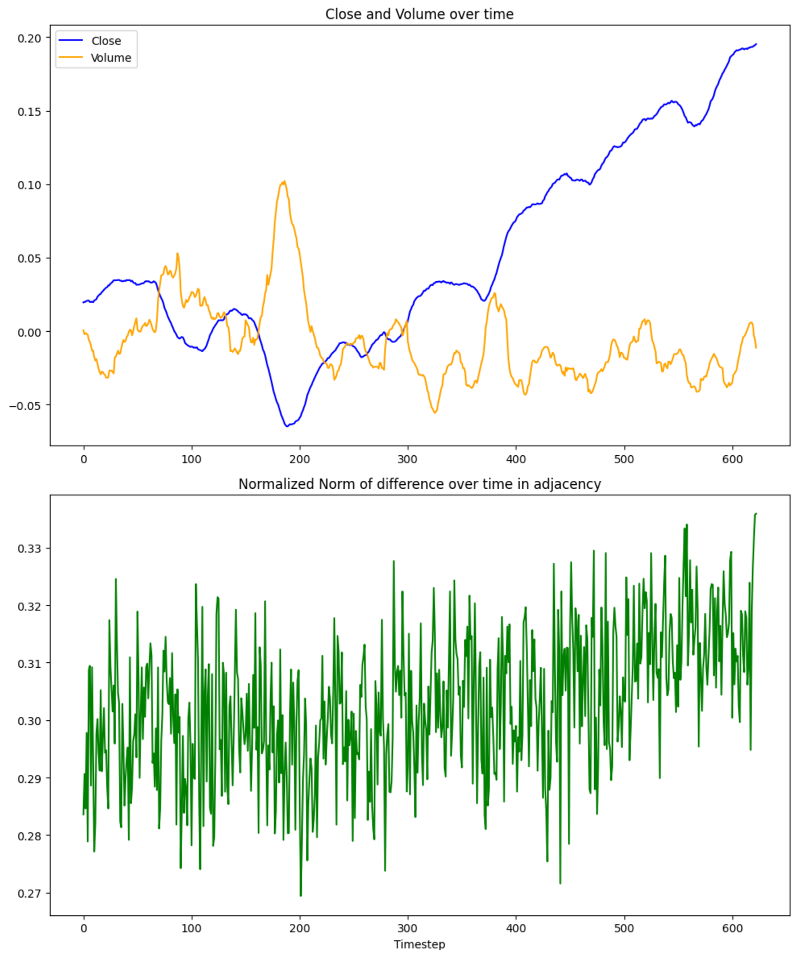

Figure 5.4: FINANCE. The average close price of the window at each timestep plotted (top) against the norm difference of the adjacency matrix at each timestep (middle) and the normalized norm of difference at each timestep (bottom). Adapt rate $\gamma = 0.001$.

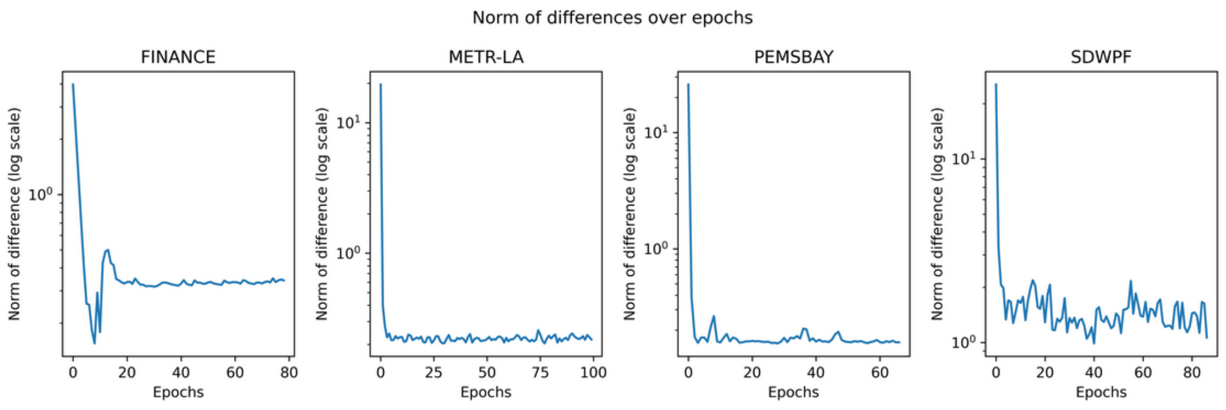

Figure 5.5: Convergence of the OAGNN model. Log-scale.

Looking at Figure 5.5, we see the adaptation converge clearly on the traffic and FINANCE datasets. As theorized, the SDWPF dataset shows us how the model cannot fully converge to a single consistent graph. We hypothesize this is because of the highly dynamic nature of SDWPF, resulting in no single ”correct” graph structure for the dataset.

5.7. Ablation studies

In this section, we discuss ablations studies to test the effect of several components of OAGNN. We study the following ablations:

- **No graph-adapt**: the OAGNN model without any graph adapting, such that $\mathbf{A}_t = \mathbf{A}_0$. Denoted as **OAGNN-NO-GA**. We use this ablation to study the effectiveness of the graph adaptation.
- **Only attention**: the OAGNN model without the online edge weight update connection or graph construction, using the resulting attention coefficients directly, such that $\mathbf{A}_{tij} = \alpha_{tij}$. Denoted as **OAGNN-O-Att**. We use this ablation to study the need for the previous states of the adjacency matrix.
- **No evolving-attention**: the OAGNN model without evolving attention. The graph attention module weights are learned without time-dependence, such that $\mathbf{W}_t = \mathbf{W}$. Denoted as **OAGNN-NO-EA**. We use this ablation to study the need to evolve the attention weights.
- **No online adapt**: the OAGNN model without graph adapting when the model is no longer in the training phase. Denoted as **OAGNN-NO-OA**. We use this ablation to study the need for an online adapting process when adapting the graph.

From the results of the ablations studies in Table 5.5, we conclude that each of the tested components of the OAGNN models is required, but for some, it depends on the dataset. Using the **OAGNN-NO-GA** ablation, we conclude that the graph adapting module is a crucial component. Each dataset shows a drop-off in performance when the graph adapting module is absent. Although the zeroes in the METR-LA dataset negatively impact the performance, adapting is still a key component. The results for the **OAGNN-O-Att** show us the need for the adaptation operator and why capturing the evolution of a network is required for the models' performance. We further see the OAGNN-O-Att diverge much earlier. The **OAGNN-NO-EA** ablation shows us that the evolving attention mechanism is not required for performance, whereas on the PEMS-BAY dataset, we see a performance improvement. We hypothesize that the model encodes most of the temporal dependencies through the adaptation operator and that evolving the graph attention weight does not impact performance much. However, the evolving module does introduce more stability in the trained model, and similar to the **OAGNN-O-Att** ablation, the model diverges earlier more often. With the **OAGNN-NO-OA** ablation, we see minimal to no impact on the traffic and SDWPF datasets. We conclude that the datasets do not contain enough testing data in which the network dynamics evolve to highlight the impact of online adapting. However, on the FINANCE dataset, we do see a difference in accuracy due to the larger test set. We study this effect additionally in Appendix D by experimenting with a different train-test-validation split.

Table 5.5: Results of ablation studies on FINANCE, METR-LA, and SDWPF. METR-LA and SDWPF use $h = 12$. Average over five runs, with the standard deviation denoted. Lower is better.

	METR-LA	PEMS-BAY	SDWPF	FINANCE
Model	MAE	MAE	*sd*MAE	Acc(%)
OAGNN-NO-GA	5.03±0.02	2.21±0.00	0.30±0.00	65.912±0.90
OAGNN-O-Att	4.98±0.01	2.22±0.02	0.30±0.00	64.255±0.08
OAGNN-NO-EA	4.97±0.01	**2.19**±0.02	**0.29**±0.00	66.300±0.34
OAGNN-NO-OA	4.97±0.01	2.20±0.01	**0.29**±0.00	66.13±0.19
OAGNN	**4.96**±0.02	2.20±0.01	**0.29**±0.00	**66.35**±0.25

Part III

Closure

6 Conclusion

In this thesis, we answered the main research objective *”How to construct an online adaptive graph model for multivariate time series analysis?”*. We introduced the Online Adaptive Graph Neural Network (OAGNN). Using a physical point of view when viewing a graph network, we model the relationship between the network dynamics and structure using a two-way relationship. With this, we capture the evolution of the network over time. As the evolution of a network is never static, we construct an online adaptation mechanism to prevent performance degradation and network mismatch. We have shown by using four datasets with different dynamic behaviors that the model can perform competitively in different environments due to its adaptive nature. We highlight the differences in the dynamic nature of the datasets and hypothesize on the model’s ability to adapt to a dynamic environment and its limitations regarding bad data. We conclude the models’ ability to capture a dynamic network’s evolution and handle complex dynamic environments where other methods cannot. We show the need for the most critical components in OAGNN by studying several ablation studies. We highlight the impact of the graph adaptation regardless of bad data and the significance of adapting online when there is enough test data. With this work, we have shown a direction for approaches on dynamic multivariate timeseries, in which the model’s ability to adapt to the data is no longer limited to any degree. We allow the model complete freedom to adjust itself accordingly. We acknowledge the limitations of OAGNN, primarily that we do not change what edges are adapted. We solely adapt the edges defined in the initial adjacency matrix. In future works, the impact of changing the edges must be studied, and for this, the adaptation module should be expanded to adjust what edges are adapted. To improve the freedom of the model to adapt to the data, the adapt rate itself should be a dynamic component of this process. We focused on using graph attention for the similarity score. Other methods for similarity scores should be studied.

References

[1] Thanh Trung Huynh et al. “Efficient integration of multi-order dynamics and internal dynamics in stock movement prediction”. In: *Proceedings of the Sixteenth ACM International Conference on Web Search and Data Mining*. 2023, pp. 850–858.

[2] Junbo Zhang et al. “Predicting citywide crowd flows using deep spatio-temporal residual networks”. In: *Artificial Intelligence* 259 (2018), pp. 147–166.

[3] Franco Scarselli et al. “The graph neural network model”. In: *IEEE transactions on neural networks* 20.1 (2008), pp. 61–80.

[4] Antonio Longa et al. “Graph Neural Networks for temporal graphs: State of the art, open challenges, and opportunities”. In: *arXiv preprint arXiv:2302.01018* (2023).

[5] Joakim Skarding et al. “Foundations and modeling of dynamic networks using dynamic graph neural networks: A survey”. In: *IEEE Access* 9 (2021), pp. 79143–79168.

[6] Shubham Gupta et al. “A Survey on Temporal Graph Representation Learning and Generative Modeling”. In: *arXiv preprint arXiv:2208.12126* (2022).

[7] Jinyin Chen et al. “GC-LSTM: Graph convolution embedded LSTM for dynamic network link prediction”. In: *Applied Intelligence* (2022), pp. 1–16.

[8] Aldo Pareja et al. “Evolvegcn: Evolving graph convolutional networks for dynamic graphs”. In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 34. 04. 2020, pp. 5363–5370.

[9] Chao Gao et al. “A novel representation learning for dynamic graphs based on graph convolutional networks”. In: *IEEE Transactions on Cybernetics* (2022).

[10] Da Xu et al. “Inductive representation learning on temporal graphs”. In: *arXiv preprint arXiv:2002.07962* (2020).

[11] Ling Zhao et al. “T-gcn: A temporal graph convolutional network for traffic prediction”. In: *IEEE transactions on intelligent transportation systems* 21.9 (2019), pp. 3848–3858.

[12] Ming Jin et al. “Multivariate time series forecasting with dynamic graph neural ODEs”. In: *IEEE Transactions on Knowledge and Data Engineering* (2022).

[13] Xiangyuan Kong et al. “Adaptive spatial-temporal graph attention networks for traffic flow forecasting”. In: *Applied Intelligence* (2022), pp. 1–17.

[14] Ruijia Wang et al. “Graph structure estimation neural networks”. In: *Proceedings of the Web Conference 2021*. 2021, pp. 342–353.

[15] Lei Bai et al. “Adaptive graph convolutional recurrent network for traffic forecasting”. In: *Advances in neural information processing systems* 33 (2020), pp. 17804–17815.

[16] Tong Zhao et al. “Data augmentation for graph neural networks”. In: *Proceedings of the aaai conference on artificial intelligence*. Vol. 35. 12. 2021, pp. 11015–11023.

[17] Fuxian Li et al. “Dynamic graph convolutional recurrent network for traffic prediction: Benchmark and solution”. In: *ACM Transactions on Knowledge Discovery from Data* 17.1 (2023), pp. 1–21.

[18] Luca Franceschi et al. “Learning discrete structures for graph neural networks”. In: *International conference on machine learning*. PMLR. 2019, pp. 1972–1982.

[19] Rico Berner et al. “Adaptive Dynamical Networks”. In: *arXiv preprint arXiv:2304.05652* (2023).

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

[21] Jingbo Zhou et al. “Sdwpf: A dataset for spatial dynamic wind power forecasting challenge at kdd cup 2022”. In: *arXiv preprint arXiv:2208.04360* (2022).

[22] Zinuo You et al. “Multi-relational Graph Diffusion Neural Network with Parallel Retention for Stock Trends Classification”. In: *arXiv preprint arXiv:2401.05430* (2024).

[23] Thomas N Kipf et al. “Semi-supervised classification with graph convolutional networks”. In: *arXiv preprint arXiv:1609.02907* (2016).

[24] Zonghan Wu et al. “Connecting the dots: Multivariate time series forecasting with graph neural networks”. In: *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*. 2020, pp. 753–763.

[25] Claudio DT Barros et al. “A survey on embedding dynamic graphs”. In: *ACM Computing Surveys (CSUR)* 55.1 (2021), pp. 1–37.

[26] Nahla Mohamed Ahmed et al. “DeepEye: link prediction in dynamic networks based on non-negative matrix factorization”. In: *Big Data Mining and Analytics* 1.1 (2018), pp. 19–33.

[27] Jundong Li et al. “Attributed network embedding for learning in a dynamic environment”. In: *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. 2017, pp. 387–396.

[28] Yu-Ru Lin et al. “Facetnet: a framework for analyzing communities and their evolutions in dynamic networks”. In: *Proceedings of the 17th international conference on World Wide Web*. 2008, pp. 685–694.

[29] Aditya Grover et al. “node2vec: Scalable feature learning for networks”. In: *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. 2016, pp. 855–864.

[30] Yanbang Wang et al. “Inductive representation learning in temporal networks via causal anonymous walks”. In: *arXiv preprint arXiv:2101.05974* (2021).

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

[32] Franco Manessi et al. “Dynamic graph convolutional networks”. In: *Pattern Recognition* 97 (2020), p. 107000.

[33] Yao Ma et al. “Streaming graph neural networks”. In: *Proceedings of the 43rd international ACM SIGIR conference on research and development in information retrieval*. 2020, pp. 719–728.

[34] Weiping Song et al. “Session-based social recommendation via dynamic graph attention networks”. In: *Proceedings of the Twelfth ACM international conference on web search and data mining*. 2019, pp. 555–563.

[35] Srijan Kumar et al. “Predicting dynamic embedding trajectory in temporal interaction networks”. In: *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 2019, pp. 1269–1278.

[36] Rakshit Trivedi et al. “Dyrep: Learning representations over dynamic graphs”. In: *International conference on learning representations*. 2019.

[37] Aravind Sankar et al. “Dysat: Deep neural representation learning on dynamic graphs via self-attention networks”. In: *Proceedings of the 13th international conference on web search and data mining*. 2020, pp. 519–527.

[38] Chenguang Song et al. “Temporally evolving graph neural network for fake news detection”. In: *Information Processing & Management* 58.6 (2021), p. 102712.

[39] Emanuele Rossi et al. “Temporal graph networks for deep learning on dynamic graphs”. In: *arXiv preprint arXiv:2006.10637* (2020).

[40] Mounir Haddad et al. “Temporalizing static graph autoencoders to handle temporal networks”. In: *Proceedings of the 2021 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*. 2021, pp. 201–208.

[41] Ehsan Hajiramezanali et al. “Variational graph recurrent neural networks”. In: *Advances in neural information processing systems* 32 (2019).

[42] Hao Peng et al. “Spatial temporal incidence dynamic graph neural networks for traffic flow forecasting”. In: *Information Sciences* 521 (2020), pp. 277–290.

[43] Abishek Sriramulu et al. “Adaptive Dependency Learning Graph Neural Networks”. In: *Information Sciences* (2023).

[44] Chao Shang et al. “Discrete graph structure learning for forecasting multiple time series”. In: *arXiv preprint arXiv:2101.06861* (2021).

[45] Dongsheng Luo et al. “Learning to drop: Robust graph neural network via topological denoising”. In: *Proceedings of the 14th ACM international conference on web search and data mining*. 2021, pp. 779–787.

[46] Min Shi et al. “GAEN: graph attention evolving networks”. In: *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence (IJCAI)*. 2021.

[47] Yu Chen et al. “Iterative deep graph learning for graph neural networks: Better and robust node embeddings”. In: *Advances in neural information processing systems* 33 (2020), pp. 19314–19326.

[48] Seyed Saman Saboksayr et al. “Online graph learning under smoothness priors”. In: *2021 29th European Signal Processing Conference (EUSIPCO)*. IEEE. 2021, pp. 1820–1824.

[49] Seyed Saman Saboksayr et al. “Online discriminative graph learning from multi-class smooth signals”. In: *Signal Processing* 186 (2021), p. 108101.

[50] Vassilis Kalofolias. “How to learn a graph from smooth signals”. In: *Artificial intelligence and statistics*. PMLR. 2016, pp. 920–929.

[51] Xiang Zhang et al. “Online Graph Learning In Dynamic Environments”. In: *2022 30th European Signal Processing Conference (EUSIPCO)*. IEEE. 2022, pp. 2151–2155.

[52] Stefan Vlaski et al. “Online graph learning from sequential data”. In: *2018 IEEE Data Science Workshop (DSW)*. IEEE. 2018, pp. 190–194.

[53] Dorina Thanou et al. “Learning heat diffusion graphs”. In: *IEEE Transactions on Signal and Information Processing over Networks* 3.3 (2017), pp. 484–499.

[54] Rasoul Shafipour et al. “Online topology inference from streaming stationary graph signals with partial connectivity information”. In: *Algorithms* 13.9 (2020), p. 228.

[55] Alberto Natali et al. “Learning time-varying graphs from online data”. In: *IEEE Open Journal of Signal Processing* 3 (2022), pp. 212–228.

[56] Youngjoo Seo et al. “Structured sequence modeling with graph convolutional recurrent networks”. In: *Neural Information Processing: 25th International Conference, ICONIP 2018, Siem Reap, Cambodia, December 13-16, 2018, Proceedings, Part I 25*. Springer. 2018, pp. 362–373.

[57] Delft High Performance Computing Centre (DHPC). *DelftBlue Supercomputer (Phase 1)*. `https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase1`. 2022.

[58] Benedek Rozemberczki et al. “Pytorch geometric temporal: Spatiotemporal signal processing with neural machine learning models”. In: *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 2021, pp. 4564–4573.

[59] Xingjian Shi et al. “Convolutional LSTM network: A machine learning approach for precipitation nowcasting”. In: *Advances in neural information processing systems* 28 (2015).

A
Hyperparameter search

In this section, we discuss the selected hyperparameters using the results of hyperparameter tuning. For all hyperparameters searches, we use the full specified dataset, 100 epochs with early stopping, three runs, and to select the hyperparameters; for the traffic datasets, we use the METR-LA dataset with the L1 loss; for the SDWPF dataset, the MAE+RMSE loss; and for the FINANCE dataset, we use the F1 loss. We optimize the traffic datasets and SDWPF for $h = 12$. We display all results and the selected hyperparameters in Table A.1. We do not run a hyperparameter search for the PEMS-BAY dataset and not for the graph attention dropout and leaky ReLU, which we set to 0.5 and 0.2, respectively. The number of evolving attention heads and the hidden dimension of evolving attention have the most impact on the number of parameters in the model, which we display in A.2. We see a near-exponential growth in the number of parameters with an increase in the hidden dimension and the number of heads. However, the computation time is not affected by this. Compared to the baselines, we see consistent timing performance on all datasets, recorded per epoch basis in A.3. We see a longer runtime than the baselines on FINANCE, possibly because we have a more expensive operator due to the online component.

Dataset	Hyperparameter	Searches	Selected
Traffic METR-LA/ PEMS-BAY	Learning rate	{0.001,0.0005,0.00005}	0.0005
	Weight decay	$\{1e^{-5}, 1e^{-9}\}$	$1e^{-5}$
	Number of layers	{1,...,4}	2
	Adapt rate	{0.1, 0.05, 0.01, 0.005, 0.001}	0.01
	MP hidden dimension	{32, 64, 128}	32
	Perm prob train	{0, 0.005, 0.001}	0.001
	Perm prob test	{0, 0.0005, 0.0001}	0.0001
	GC rate	{0.005, 0.004, 0.003, 0.002, 0.001, 0.0005}	0.003
	Evolving att heads	{2, 4, 8}	8
	Evolving at hidden dim	{8, 16}	8
SDWPF	Learning rate	{0.001,0.0005,0.00005}	0.00005
	Weight decay	$\{1e^{-2}, 1e^{-3}, 1e^{-5}, 1e^{-9}\}$	$1e^{-5}$
	Number of layers	{1,...,4}	2
	Adapt rate	{0.1, 0.05, 0.01, 0.005, 0.001}	0.1
	MP hidden dimension	{32, 64, 128}	32
	Perm prob train	{0, 0.005, 0.001}	0
	Perm prob test	{0, 0.0005, 0.0001}	0
	GC rate	{0.005, 0.004, 0.003, 0.002, 0.001, 0.0005}	0.004
	Evolving att heads	{2, 4, 8}	8
	Evolving att hidden dim	{8, 16}	8
FINANCE	Learning rate	{0.001,0.0005,0.00005}	0.001
	Weight decay	$\{1e^{-3}, 1e^{-4}, 1e^{-5}, 1e^{-7}\}$	$1e^{-7}$
	Number of layers	{1,...,4}	4
	Adapt rate	{0.1, 0.05, 0.01, 0.005, 0.001}	0.001
	MP hidden dimension	{32, 64, 128}	128
	Perm prob train	{0, 0.005, 0.001}	0
	Perm prob test	{0, 0.0005, 0.0001}	0
	GC rate	{0.005, 0.004, 0.003, 0.002, 0.001, 0.0005}	0.5
	Evolving att heads	{2, 4, 8}	4
	Evolving att hidden dim	{8, 16}	16

Table A.1: Chosen hyperparameters for all datasets. Selected hyperparameters are based on averages of 3 runs and the stability of the constructed graph.

Heads \ Dim	8	16	32	64
2	6248	16088	57292	207244
4	16088	54200	207244	802060
8	54200	204152	802060	3171340

Table A.2: Table illustrating the relation between the number of evolving attention heads and the evolving attention hidden dimension with the number of learnable parameters in the model.

Table A.3: Runtime in seconds per epoch of the model per datasets. Averages over five runs with the standard deviation denoted.

Model	METR-LA	PEMS-BAY	SDWPF	FINANCE
MTGNN	15.51±0.06	35.86±0.17	11.87±0.03	2.92±0.02
ADLNN	15.88±0.14	36.61±0.98	12.98±0.04	3.23±0.04
GCRN	15.00±0.07	47.98±0.13	97.69±0.24	10.49±0.01
EvolveGCN	113.58±3.45	165.25±3.25	131.40±0.151	3.76±0.06
DGCRN	99.83±0.54	184.97±0.48	-	-
ASTGAT	177.81±0.45	-	-	-
OAGNN	10.20±0.22	25.04±2.25	9.85±0.05	3.46±0.20

B
Baseline experiments

In this appendix, we go more in-depth into the baseline experiments. Namely, we explain the selected hyperparameters for each baseline model and why. We do this to create complete transparency between the comparisons and to motivate other works to reproduce them.

For several baselines, we only use a single feature from the METR-LA and PEMS-BAY datasets as the input instead of the two input features for the datasets. This change has minimal impact as the second feature of these datasets is not crucial, representing the time delta, but is needed as either the model did not support multidimensional input or performed significantly worse with it.

We do not perform any baseline experiments using fully online graph learning methods, as these do not translate well to multivariate timeseries forecasting. We theorize that this is because the prediction is separate from graph learning.

B.1. GCRN

GCRN [56] is a static-graph approach for forecasting and was initially trained and tested on the moving-MNIST dataset [59] and the Penn Treebank dataset [56]. However, since the network combines recurrence and graph convolutions, the network can also be applied to multivariate timeseries forecasting. The implementation in PyTorch Geometric Temporal [58] has no implementation for a sequence length, so we flatten the input $\mathcal{X}_t \in \mathbb{R}^{N \times w \times F}$ to a two-dimensional input sequence $\mathbf{X}_t \in \mathbb{R}^{N \times wF}$. The complete baseline network first uses the GCRN model to transform $\mathbf{X}_t$ into the output dimension $\mathbb{R}^{N \times 16wF}$, then a readout layer is used to reduce the output to the horizon h sequence length. We set the learning rate to 0.001 and weight decay to $1e^{-4}$ for the traffic datasets; for SDWPF, we set the learning rate to 0.0005 and the weight decay to $1e^{-9}$; for the FINANCE dataset we set the learning rate to 0.00005 and the learning rate to $1e^{-4}$, and for all we set the hyperparameter $K = 2$.

B.2. EvolveGCN

EvolveGCN [8] is a static graph approach that introduces an evolving GCN. The GCN weight is evolved using a GRU in the same manner as this work. The authors test EvolveGCN on multiple datasets for link prediction, edge classification, and node classification. However, due to its approach of using evolving weights, we use this work as a baseline model. We use the implementation of the *-H* version from PyTorch Geometric Temporal [58]. As EvolveGCN does not allow for windowed input, therefor, we flatten the input $\mathcal{X}_t \in \mathbb{R}^{N \times w \times F}$ to a two-dimensional input sequence $\mathbf{X}_t \in \mathbb{R}^{N \times wF}$. The complete baseline network uses the EvolveGCN model to transform $\mathbf{X}_t$ into new features with the same dimensions as the input and then a readout layer to reduce the output to the horizon h sequence length. We set the learning rate to 0.001 and weight decay to $1e^{-4}$ for METR-LA and PEMS-BAY. For SDWPF, we set the learning rate to 0.0005 and the weight decay to $1e^{-9}$. We set the learning rate for FINANCE to 0.00005 and the weight decay to $1e^{-4}$.

Hyperparameter	Value	Reason
gcn_true	True	derived from the paper
build_adj	True	derived from the paper
gcn_depth	2	derived from the mix-hop propagation depth
kernel_set	[6,7]	derived from the paper
kernel_size	7	derived from **kernel_set**
dropout	0.3	derived from the paper
subgraph_size	-	derived from the paper
node_dim	40	derived from the paper
dilation_exponential	1	derived from the paper
conv_channels	32	derived from the paper
residual_channels	32	derived from **conv_channels**
skip_channels	64	derived from the paper
end_channels	128	derived from output channels of the first layer
seq_length	12	derived from w
out_dim	12	derived from h
layers	3	derived from the number of temporal convolution modules
propalpha	0.05	derived from the paper
tanhalpha	3	derived from the paper
layer_norm_affline	True	derived from the paper

Table B.1: Parameters for MTGNN, and how they were derived. Note that several parameters are left out, as these are dataset-dependent.

B.3. MTGNN

MTGNN [24] uses node embeddings in a so-called difference layer to learn the graph. The layer is defined as follows

$$\mathbf{M}_1 = tanh(\alpha(\text{MLP}_1(\mathbf{E}_1))),$$
$$\mathbf{M}_2 = tanh(\alpha(\text{MLP}_2(\mathbf{E}_2))),$$
$$\mathbf{A} = ReLU(tanh(\alpha(\mathbf{M}_1\mathbf{M}_2^T - \mathbf{M}_2\mathbf{M}_1^T)),$$

where $\mathbf{E}_1$ and $\mathbf{E}_2 \in \mathbb{R}^{N\times d_e}$ are learnable embedding matrices, with d_e the embedding dimension, $\text{MLP}_1(\cdot)$ and $\text{MLP}_2(\cdot)$ learnable MLPs, $\cdot^T$ the transpose, α a hyperparameter and $tanh(\cdot)$ and $ReLU(\cdot)$ non-linear functions. Finally, the top values for each row in **A** are selected to use in the prediction module. This difference layer does not use the node features and uses the learned node embeddings.

We mostly derive the hyperparameters from the paper and several from the provided code to learn the model. We set the learning rate to 0.001 and the weight decay to $1e^{-4}$ for METR-LA and PEMS-BAY; for SDWPF, we set the learning rate to 0.0005 and the weight decay to $1e^{-9}$. We set the learning rate for FINANCE to 0.00005 and the weight decay to $1e^{-4}$. We use the implementation of the model in PyTorch Geometric Temporal [58], and for each hyperparameter, we define how they are derived with their name in code in Table B.1.

B.4. DGCRN

DGCRN [17] is a follow-up of MTGNN [24], improving the graph learning by incorporating the results of two GCN layers that process the node features. The authors define the matrix $\mathbf{I}_t$ as

$$\mathbf{I}_t = \mathbf{V}_t||\mathbf{T}_t||\mathbf{H}_{t-1},$$

Param	Value	Reason
gcn_depth	1	reduced for complexity
dropout	0.3	derived from the code
subgraph_size	20	derived from the code
node_dim	40	derived from the paper
middle_dim	1	derived from the paper
rnn_size	64	derived from the paper
hyperGNN_dim	16	derived from the code
list_weight	[0.05, 0.95, 0.95]	derived from the paper
cl_decay_steps	4000	derived from the paper
layers	3	derived from the paper
tanhalpha	3	derived from the paper

Table B.2: Parameters for DGCRN, and how they were derived. Note that several parameters are left out, as these are dataset-dependent.

where $\mathbf{V}_t$ is the speed, $\mathbf{T}_t$ the time of day, which are both part of the features, and $\mathbf{H}_{t-1}$, a previous hidden state. Then, the adjacency matrix is defined with

$$\mathbf{DE}_1 = tanh(\alpha(\text{CONV}_1(\mathbf{I}_t) \odot \mathbf{E}_1)),$$

$$\mathbf{DE}_2 = tanh(\alpha(\text{CONV}_2(\mathbf{I}_t) \odot \mathbf{E}_2)),$$

$$\mathbf{A} = ReLU(tanh(\alpha(\mathbf{DE}_1\mathbf{DE}_2^T - \mathbf{DE}_2\mathbf{DE}_1^T)),$$

where $\mathbf{E}_1$ and $\mathbf{E}_2 \in \mathbb{R}^{N \times d_e}$ are learnable embedding matrices, with d_e the embedding dimension, CONV$_1(\cdot)$ and CONV$_2(\cdot)$ learnable convolution operators, $\cdot^T$ the transpose, $\odot$ the Hadammard product, α a hyper-parameter and $tanh(\cdot)$ and $ReLU(\cdot)$ non-linear functions. Finally, the top values for each row in A are selected to use in the prediction module. The difference layer does include the node features. However, there is no graph adaptation from a physical point of view, and the node embeddings are still used.

We use code from github[1] and derive most of the hyperparameters from the paper or the code directly. The code for DGCRN does not support multidimensional input, so we are restricted to only using a single feature. Because of this, we cannot run this model on SDWPF or FINANCE, as it requires all the features used. We modify the code to support different horizons other than $h = 12$. We set the learning rate to 0.0005 and the weight decay to $1e^{-9}$ for METR-LA and PEMS-BAY. For each hyperparameter, we define how they are derived with their name in code in Table B.2.

B.5. ADLNN

In ADLNN [43], the authors propose a similar structure for graph learning as MTGNN [17] and use attention scores in the difference layer. They define the full layer as follows

$$\mathbf{Z}_1 = tanh(\alpha(\text{MHA}(\mathbf{E}_1, \mathbf{E}_1, \mathbf{E}_1))),$$

$$\mathbf{M}_1 = tanh(\alpha(\text{MLP}_1(\mathbf{Z}_1))),$$

$$\mathbf{M}_2 = tanh(\alpha(\text{MLP}_2(\mathbf{E}_2))),$$

$$\mathbf{A} = ReLU(tanh(\alpha(\mathbf{M}_1\mathbf{M}_2^T - \mathbf{M}_2\mathbf{M}_1^T)),$$

where $\mathbf{E}_1$ and $\mathbf{E}_2 \in \mathbb{R}^{N \times d_e}$ are learnable embedding matrices, with d_e the embedding dimension, MLP$_1(\cdot)$ and MLP$_2(\cdot)$, MHA($\mathbf{Q}, \mathbf{K}, \mathbf{V}$) a multi head attention layer with **Q** the queries, **K** the keys, and **V** the values, $\cdot^T$ the transpose, α a hyperparameter and $tanh(\cdot)$ and $ReLU(\cdot)$ non-linear functions. As a final step, the top values for each row in A are selected to use in the prediction module. This difference layer does not

[1]https://github.com/tsinghua-fib-lab/Traffic-Benchmark

Param	Value	Reason
gcn_depth	2	-
dropout	0.3	derived from the code
subgraph_size	20	derived from the code
node_dim	40	derived from the code
skip_channels	64	derived from the code
residual_channels	32	derived from the code
end_channels	128	derived from the code
conv_channels	32	derived from the code
dilation_exponential	1	derived from the code
proalpha	0.05	derived from the code
layers	3	derived from the code
tanhalpha	3	derived from the code

Table B.3: Parameters for ADLNN, and how they were derived. Note that several parameters are left out, as these are dataset-dependent.

Param	Value	Reason
hidden_dim	48	-
dropout	0.6	derived from the code
alpha	0.2	derived from the code
layers	2	reduced for complexity
num_heads	4	derived from the code

Table B.4: Parameters for ASTGAT, and how they were derived. Note that several parameters are left out, as these are dataset-dependent.

use the node features and uses the learned node embeddings.

For the training of ADLNN, we use the code from github[2]. We set the learning rate to 0.0005 and weight decay to $1e^{-9}$ for METR-LA, PEMS-BAY and SDWPF. We set the learning rate for FINANCE to 0.00005 and the weight decay to $1e^{-5}$, and we adjust the code to support multidimensional input. In METR-LA, we chose only the measured speed as the input feature, as the performance with both features was significantly worse(+40%). We do not use their initial graph construction method and solely rely on the difference layer. For each hyperparameter, we define how they are derived with their name in code in Table B.3.

B.6. ASTGAT

ASTGAT [13] uses a sequential model to learn the graph distribution. From the learned distribution, a graph is sampled at each timestep and used in a subsequent spatiotemporal model. A graph regularizer based on smoothness and sparsity learns the distribution. The distribution is not learned using the node features and requires spatial embeddings for the dataset. We only have spatial embeddings from METR-LA, so we solely use that dataset.

We train ASTGAT using the code from github[3], and for METR-LA, we use the L1 loss function defined in the paper to train the model. We set the learning rate to 0.0005 and weight decay to $1e^{-9}$ for METR-LA. For each hyperparameter, we define how they are derived with their name in code in Table B.4.

[2]https://github.com/AbishekSriramulu/ADLGNN

[3]https://github.com/xyk0058/ASTGAT

C Changing the graph structure

In this section, we discuss an approach to changing the graph structure. The most significant limitation of the current method is that only existing edges are adapted. To change the graph structure in step 1, we consider a different adaptation module as illustrated in Figure C.1. For this, we consider several requirements. The method must be online to keep the entire adaptation process online; it cannot depend on the edge weights, as the weight of an edge does not indicate the preference for the edge itself, and it must be a data-model-independent approach. In a forecasting problem where we sequentially process the data, when we predict for a timestep, we have the real value for our predicted value at the next timestep. We can compute the node-wise loss for the previous timestep and change the graph structure through $\mathbf{B}_t$. The node-wise loss does not give us a performance metric for each edge. However, due to the requirements above and the need for the method to solely compute what edges to adapt, not the actual values, we interpret the loss as the loss of the node *using* the neighborhood.

Namely, before we adapt the edge weights at each timestep, we compute the prediction performance of the model using a (normalized) loss metric $\bar{\mathcal{L}}_{pred,t}(\hat{\mathbf{y}}_t, \mathbf{y}_t) \in \mathbb{R}^{N\times 1}$. Then, we translate this loss to a matrix $\bar{\mathbf{L}}_t \in \mathbb{R}^{N\times N}$ by repeating it column-wise, such that

$$\bar{\mathbf{L}}_t = \bar{\mathcal{L}}_{pred,t} \cdot \mathbf{1}^T, \tag{C.1}$$

where $\mathbf{1}^T$ is a $1 \times N$ vector of ones with $\cdot^T$ indicating the transpose. To create a sparse matrix of only the edges used for the previous prediction, we compute $\mathbf{L}_t$

$$\mathbf{L}_t = \bar{\mathbf{L}}_t \odot \mathbf{B}_{t-1}, \tag{C.2}$$

Since the loss can fluctuate significantly between timesteps, we use a similar connecting step to (C.6)

$$\mathbf{H}_t = \omega \mathbf{L}_t + (1-\omega)\mathbf{H}_{t-1} \odot \mathbf{B}_{t-1} + \mathbf{H}_{t-1} \odot (1 - \mathbf{B}_{t-1}), \tag{C.3}$$

where ω is the graph construct rate, $\odot$ the Hadamard product. Finally, we select the row-wise **lowest**-k node pairs (i,j) from $\mathbf{H}_t$ for each node and save this selection through the binary matrix $\mathbf{B}_t$ and adapt these edges solely.

$$\mathbf{B}_{tij} = \begin{cases} 1, & \text{if } (i,j) \in \text{lowest-k}(\mathbf{H}_{ti}), \\ 0, & \text{otherwise.} \end{cases} \tag{C.4}$$

For the initial adjacency matrix, the graph structure $\mathbf{B}_0$ is defined as the support of $\mathbf{A}_0$, $supp(\mathbf{A}_0)$, such that

$$\mathbf{B}_0 = \begin{cases} 1, & \text{if } \mathbf{A}_{0ij} \neq 0, \\ 0, & \text{otherwise.} \end{cases} \tag{C.5}$$

The matrix $\mathbf{B}_t$ provides more control over the graph adaptation process using the GC module, which we introduce later in this section. To adapt the edge weights for the sparse matrix $\mathbf{A}_t$, we define a connecting step with the previous adjacency matrix

$$\bar{\mathbf{A}}_{tij} = \gamma \alpha_{tij} + (1-\gamma)\bar{\mathbf{A}}_{(t-1)ij}\mathbf{B}_{tij} + \beta \bar{\mathbf{A}}_{(t-1)ij}(1 - \mathbf{B}_{tij}), \tag{C.6}$$

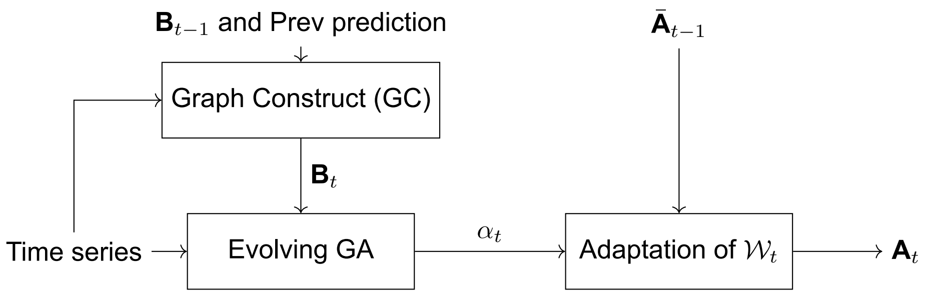

Figure C.1: Overview of the adaptation module with graph construct (GC). The previous prediction, $\mathbf{B}_{t-1}$ and $\mathbf{A}_{t-1}$ are stored from the previous iteration.

Table C.1: Result of runs using GC and the normal OAGNN model. The METR-LA, PEMS-BAY, and SDWPF datasets use $h = 12$. Averages over five runs with the standard deviation denoted. Lower is better for METR-LA, PEMS-BAY, and SDWPF; for FINANCE, higher is better.

	METR-LA	PEMS-BAY	SDWPF		FINANCE	
Model	MAE	MAE	MAE+RMSE	*sd*MAE	Acc(%)	F1
OAGNN-GC	4.99±0.01	2.21±0.02	107.83±0.97	0.30±0.00	**66.46**±0.00	0.66±0.01
OAGNN	**4.96**±0.01	**2.20**±0.01	**107.17**±0.45	**0.29**±0.00	66.35±0.00	**0.67**±0.00

C.1. Results

We run the same test setup as in Chapter 5 and test the model's performance with GC, displayed in Table C.1. We see from the results that the impact of the GC module is often negative. We conjugate several possible reasons for this behavior; the GC module is not efficient enough. It does converge during the training phase, but when deployed in the real world, the model can not adjust the graph correctly. Secondly, what edges the model adapts does not matter. When we adapt the edge weights, we adapt them to their optimal value, and by integrating the k-hop neighbors with the prediction module, the exact edges that exist do not matter.

D

Ablation study on adaptiveness

This section discusses an additional study of the **OAGNN-NO-OA** ablation using a different train-test-validation split. The experiment setup is the same as the ablation study from section 5.7. We study the effect of the online adaptation mechanism by using a split of 50, 40, and 10 for training, testing, and validating, respectively. We see from the results in Table D.1 how the traffic datasets do not benefit from the online adapting due to their non-dynamic nature. However, the SDWPF shows a clear advantage when more test data is present.

Table D.1: Result of additional ablation study using 50, 40, and 10 train-test-validation split. The METR-LA, PEMS-BAY, and SDWPF datasets use $h = 12$. Averages over five runs with the standard deviation denoted. Lower is better.

	METR-LA		PEMS-BAY		SDWPF	
Model	MAE	RMSE	MAE	RMSE	MAE+RMSE	*sd*MAE
OAGNN-NO-OA	**4.90**±0.01	**8.95**±0.01	2.24±0.02	**4.27**±0.04	141.15±0.85	0.31±0.00
OAGNN	**4.90**±0.01	**8.95**±0.02	**2.23**±0.01	**4.27**±0.05	**137.67**±0.57	**0.30**±0.00