
# Comparative study of Graph Convolutional Recurrent Network models in traffic forecasting using the Pedal Me's bike transport demand example of London

Student name: Leon Reiß
Student ID: I6337206
Supervisor: Prof. Dr. Roselinde Kessels
Second reader: Yicheng Mao (M.Sc.)
Date of defense: 26.06.2024

**Acknowledgements**

I would like to thank Prof. Dr. Roselinde Kessels for supervising my work. Thanks to her feedback, I was able to keep my research on the right track.

I would also like to thank my Family Ralf, Sabine and Timo Reiß for their mental support during this intensive and interesting time.

## Abstract

This research evaluates Graph Convolutional Recurrent Network (GCRN) models—specifically GConvGRU and GConvLSTM—against a baseline ARIMA model for predicting bike transport demand for Pedal Me in different regions of London. Using the PyTorch Geometric Temporal Library, the GCRN models were tested with different epochs and dropout levels, and their performance was evaluated against the ARIMA model using MSE, RMSE and MAE in a holdout validation. The results show that while GCRN models achieve a balance between training and testing for short training periods, they overfit for longer training periods, resulting in poor generalization. Despite the usefulness of dropout layers, they could not completely prevent overfitting. The GCRN models did not perform better than the ARIMA baseline model due to the small size of the dataset and the lack of features such as additional contextual information, which prevented them from fully utilizing their capabilities in spatio-temporal modeling. Future work should therefore investigate other regularization techniques and transfer learning to improve performance. Furthermore, additional metrics such as connectivity should be used to gain a better understanding of memory processing in GCRN models. In addition, the extension of ARIMA to ARIMAX could improve the prediction of volatile fluctuations by including further contextual information despite a smaller data set. Given the challenges faced by the GCRN models, ARIMA is currently the better choice despite similar performance and could serve as a decision support system for resource optimization in the respective regions of Pedal Me in London.

# Table of Contents

## List of Tables and Figures

### Tables



### Figures

## 1. Introduction

Intelligent Transport Systems (ITS) use modern telecommunication and information technologies, such as artificial intelligence and machine learning, to improve traffic management, as outlined by the European Commission (2010) in the EU Directive 2010/40. Studies show that ITS can reduce traffic congestion, increase road utilization, and reduce emissions and fuel consumption (Chen & Chen, 2019; Lozić, 2016; Qureshi & Abdullah, 2013; Shaheen & Finson, 2013). It also contributes to passenger satisfaction, reduces travel and waiting times, and promotes a safer, better coordinated and more sustainable transport infrastructure (European Commission, 2010; Jiber, Lamouik, Yahyaouy, & Sabri, 2018). Economically, ITS can provide support by providing decision support systems (DSS), for example by generating demand forecasts for transport requests, as in the case of Uber, in order to adjust prices in real time and use resources effectively (Chen, Thakuriah, & Ampountolas, 2021; Shaheen & Finson, 2013; Wang et al., 2020).

With the development of ITS, traffic forecasting is becoming increasingly important. According to Jiang and Luo (2022) as well as Zhao et al. (2020),traffic forecasting refers to the analysis of traffic patterns to predict future traffic trends, which include different mobility solutions such as car, train, bike, and others. It is an umbrella term that encompasses different forecasting problems such as Forecasting traffic flow and speed in road traffic, passenger flow in urban rail systems, and demand forecasting for transit and ride-sharing services are just some of the many applications (Wang et al., 2020; Jiang & Luo, 2022).

Common to all traffic forecasting problems is the need to model both temporal and spatial dynamics (Jiang & Luo, 2022). This modeling may also include external factors such as holidays, major events or weather, and is a challenging time series problem with significant spatial dependencies (Li, Zhou, & Pan, 2022; Ye, Zhao, Ye, & Xu, 2022). The spatial dependency is affected by the topological structure of the road network, as traffic conditions in one area (such as congestion on major roads) may also affect neighboring areas (Zhao et al., 2020). This is particularly relevant for demand forecasting, as demand for transport services in one region is co-determined by conditions in neighboring regions (Lan, 2020). Furthermore, temporal patterns show dynamic changes through daily and weekly periodicities, as well as trends throughout the day, with current traffic conditions influenced by previous ones (Zhao et al., 2020).

In the past, well-known time series models such as ARIMA (Auto Regressive Integrated Moving Average) and classical machine learning methods such as Support Vector Machine and k-nearest neighbour have been used for traffic forecasting (Zhao, Chen, Wu, Chen, & Liu, 2017). These approaches are usually considered ineffective because they only capture the temporal dimension and are unable to model the complex nonlinear relationships of traffic (Zhao et al., 2017; Zhao et al., 2020). Research by Jiang, Han, Jiang, Zhao and Wang (2023) points to significant progress in the field of traffic forecasting using deep learning, which can model complex non-linear patterns and achieve convincing results in spatio-temporal traffic forecasting. These networks consist of multiple layers with numerous nodes that process input data and adjust weights through backpropagation based on prediction errors by iteratively trying to minimize the error function (Manibardo, Laña, & Del Ser, 2020).

Different deep learning architectures are used synergistically to combine spatial and temporal dimensions in a single model. This combination of models is also known as Graph Convolutional Recurrent Networks (GCRNs). These are currently a common practice for modeling complex spatiotemporal dynamics for traffic forecasting (Jiang et al., 2023). Convolutional Neural Networks (CNNs) and Graph Convolutional Networks (GCNs) are particularly suitable for modeling the spatial dimension (Jiang et al., 2023). On the other hand, Recurrent Neural Networks (RNNs), such as Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) cells, are often used for the temporal dimension (Jiang et al., 2023).

However, various studies have shown that the effectiveness and behavior of LSTM and GRU models depend on the specific type and amount of data, as well as the respective application context, to achieve good performance (Chung, Gülçehre, Cho, & Bengio, 2014; Yang, Yu, & Zhou, 2020). In particular, for deep learning models such as LSTM and GRU, performance can be highly dependent on the amount of data available, as they often require large amounts of data to effectively learn and generalize complex patterns (Alzubaidi et al., 2023). This poses a challenge when limited datasets are available, making models more prone to overfitting (Alzubaidi et al., 2023). Overfitting occurs when a model learns the training data too well, including noise and too specific patterns, causing it to perform poorly on new, unseen data (Xue, 2019).

This study deals with a specific use case of traffic forecasting in the context of a decision support system for a private company. Pedal Me, founded in 2017, is a bike transport service that provides passenger and goods transport in London (https://pedalme.co.uk/). We compare two GCRN models to predict the demand for transport requests based on the publicly available dataset of Pedal Me. These models include either LSTM or GRU cells. The dataset includes 15 geographical regions, their distances to each other and the number of requests per week recorded over time for each region. This is a relatively small data set, which was chosen due to a lack of computational resources. However, due to the limited size of the dataset and the complexity of the models, there is a risk of overfitting (Alzubaidi et al., 2023).

In this research, the Python library Pytorch Geometric Temporal by Rozemberzcki et al. (2021) is used. This library contains the model and the dataset, but the models were not specifically tuned to the Pedal Me dataset. Furthermore, no baseline model was created as a critical reference point. At this point, we continue the work of Rozemberzcki et al. (2021) and provide a more detailed analysis of model performance on this dataset. The epochs and dropout layers are varied to compare the nuances of their modeling performance in terms of error measures (MSE, MAE, RMSE). The models are also compared with an ARIMA as a baseline model at regional and global level to assess their suitability for use in a decision support system and to investigate whether they can maintain their superior functionality over traditional time series models despite the risk of overfitting on a smaller dataset.

This research has four main objectives: First, the literature review presents advanced deep learning models for traffic prediction, with a focus on Graph Convolutional Neural Networks and Recurrent Neural Networks. Second, the GCRN models were developed and tuned using the Pytorch Geometric Temporal Library by Rozemberczki et al. (2021) and then compared with an ARIMA model. Third, it will be investigated whether the ARIMA model can be outperformed by the developed models in holdout validation. Finally, this investigation aims to show how effective GCRN models are in this particular context of traffic forecasting, especially when applied to a smaller data set, and what measures can be taken to improve the performance of the models and research in this area.

## 2. Literature Review

This literature review introduces traffic forecasting theories, focusing on spatio-temporal models that utilize graph (convolutional) networks and recurrent neural networks. It finishes with an introduction of the GCRN models used in this study.

### *2.1 Capturing the spatial dimension with GNNs*

Graphs represent entities such as social networks, molecules and road systems and provide rich relational information (Zhou et al., 2021). Graph neural networks (GNNs) were developed to process graph-based data by updating node representations through aggregating the data of their own and neighboring nodes (Wu, Cui, Pei, Zhao, & Guo, 2022). These networks excel in areas such as urban intelligence, bioinformatics and recommender systems by effectively processing non-Euclidean data, which CNNs are not capable of (Rossi et al., 2020; Ye et al., 2022; Zhou et al., 2021). Although GNNs address varied tasks, they primarily engage in two critical steps: graph construction and node representation learning. Graph construction identifies dependencies among the input data, while learning node representations refines node features by integrating those of adjacent nodes (Wu et al., 2022). The first chapter on graph construction will provide a basis of matrix representations for traffic forecasting. This is followed by a chapter about graph convolutions, which presents a method for node representation learning on graph-based data.

#### *2.1.1 Introduction to Graph Construction*

This section deals with fundamental matrix representation methods for graphs in traffic forecasting, which was derived from spectral graph theory. Chung (1996) has characterized this area of research through his work on the eigenvalues of Laplacian matrices to derive the properties of graphs. However, to stay in the context of traffic forecasting, we also draw on other sources.

A traffic Graph (with node features) can be defined as $G = (V, E, A)$, where $V$ is the node set, $E$ is the edge set, and $A$ is the adjacency matrix (Jiang & Luo, 2022). Each node can, for example, represent a sensor, a section of road or a specific region. The edge set represents the connections between the nodes, which can be used to construct the adjacency matrix that contains the topology of the network (Ye et al., 2022). A traffic graph can be directed, undirected, weighted or unweighted, depending on its purpose (Jiang & Luo, 2022). This flexibility allows the graph to accurately represent different aspects of traffic systems, e.g. the direction of traffic flow with directed edges or the distance of different regions indicated by the weighting of the edges (Ye et al. 2022). For a single time step $t$, the node feature matrix $X_t \in R^{nxd}$ for $G$ could contain specific feature values for each node, where $n$ is the number of nodes and $d$ is the number of feature variables (Jiang & Luo, 2022). These features could be, for example, the traffic flow or the demand for a specific means of transportation at a specific node (Ye et al. 2022). Figure 1 shows a simple, undirected and static traffic network in which the nodes represent sensor stations at different locations and the edges represent the connections between these locations through road sections (Wu et al., 2022). Please note that this figure is only intended to illustrate the concept of neighboring nodes in graphs. In our case, the nodes represent regions, and the weighted edges represent the distances between them.

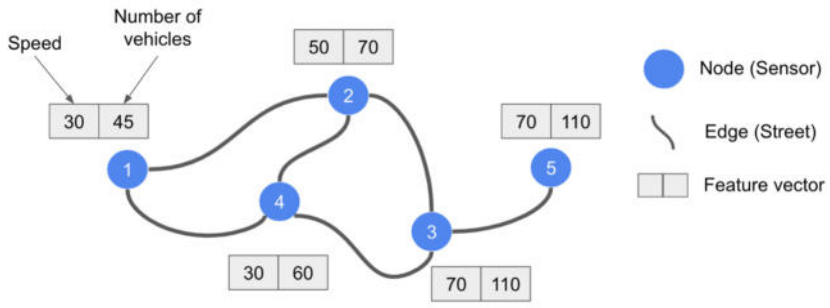



*Figure 1.* Simple representation of a graph for a road network

The adjacency matrix of $G$, denoted as $A(G)$, is an $n \times n$ matrix where each entry $a_{ij}$ indicates the presence of an edge between nodes $v_i$ to $v_j$ (Chung, 1996). Adjacency matrices are seen as the key to capturing spatial dependency in traffic forecasting, because they precisely reflect the connectivity between different locations within a graph (Ye et al., 2022). For binary encoding, $a_{ij} = 1$ an edge exists, and $a_{ij} = 0$ indicates no edge (Jiang & Luo, 2022). In weighted graphs, $a_{ij}$ can be a float value representing the edge weight, such as distance or correlation of the

connection between nodes (Chai, Wang, & Yang, 2018). The construction of the adjacency matrix depends on whether the graph is static or dynamic. In static graphs, the connections and weights do not change over time, making $A$ constant. In dynamic graphs, the connections and weights evolve, resulting in time-varying $A_t$ (Ye et al., 2022). This study focuses on static graphs where the adjacency matrix remains unchanged.

The Degree matrix $D(G)$ of a graph $G$ is where each diagonal entry $D_{ii}$ is determined by $D_{ii} = deg(v_i)$ and all off-diagonal entries $D_{ij}$ are $D_{ij} = 0$ (Chung, 1996). Here, $deg(v_i)$ is the number of edges incident to node $v_i$, which could also be the sum of the weights of the edges attached to $v_i$ in a weighted graph (Jiang & Luo, 2022).

The Laplacian matrix $L(G)$ of a graph $G$ can be constructed by subtracting the adjacency matrix A from the degree matrix $D$ (Jiang & Luo, 2022).

$$L(G) = D(G) - A(G)$$

Figure 2 shows a simple example of how the corresponding Laplacian matrix can be computed from the adjacency matrix and the degree matrix of a simple traffic network.

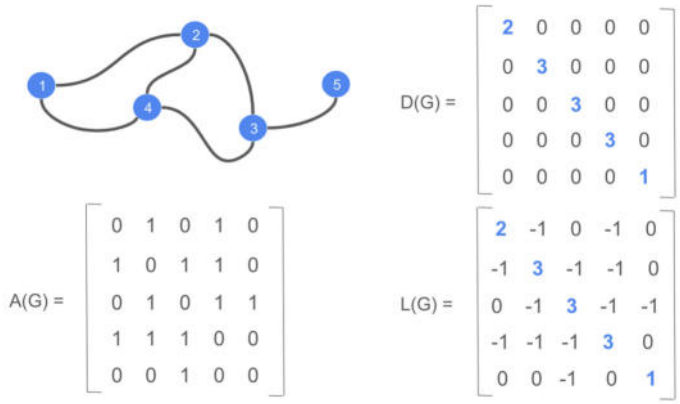



*Figure 2.* Computation of a Laplacian Matrix based on a static unweighted graph

Finally, another important matrix to understand convolutions on graphs is the identity matrix. The identity matrix $I(G)$ of a graph $G$ is a matrix where each diagonal entry $I_{ii}$ is determined

by 1 and all other entries $I_{ij}$ are 0. The tools presented in this chapter can now be used to derive convolutions on graphs.

### *2.1.2 Convolutions on Graphs*

Node representation learning updates and improves node representations by incorporating feature information from other nodes through message passing, a process that GNNs use to refine these representations and improve traffic prediction accuracy (Sánchez-Lengeling, Reif, Pearce, & Wiltschko, 2021). Graph Convolutional Networks (GCNs), a subclass of GNN construct a filter in the spectral domain to identify spatial relationships among the nodes using the eigenvalues of the Laplacian Matrix (Jiang & Luo, 2022; Zhao et al., 2020). Different GCN variants differ in how these filters are constructed (Daigavane, Ravindran, & Aggarwal, 2021).

The Chebyshev spectral CNN (ChebNet) by Defferrard, Bresson, and Vandergheynst (2016) uses a truncated expansion of Chebyshev polynomials up to $K^{th}$ order to approximate the eigenvalues of the Laplacian matrix, which enables efficient spectral filtering within the graph (Jiang & Luo, 2022). This enables an aggregation of neighbors that are any number of hops (K-hops) away from the original node (Daigavane et al., 2021).

Kipf and Welling (2016) presented a widely used approach for graph convolution that is known for its simplicity and computational efficiency (Jiang & Luo, 2022). Their method simplifies the process of ChebNet by using the adjacency matrix directly in the spatial domain as a first-order approximation of Chebyshev polynomials, thus avoiding the need to transform the data into the spectral domain (Jiang & Luo, 2022). This strategy not only avoids the complex computations typical of spectral methods, but also focuses on immediate neighborhood interactions (K=1), improving both computational efficiency and simplicity of implementation. In this literature review, Kipf and Welling's approach is examined in detail as to not go beyond the scope of this research. The goal of this model is to learn an update function of neighbor features on a graph, which takes as input (Kipf & Welling, 2016):

- A feature description $x_i$ for every node i at time step t, summarized in a $N \; x \; D$ matrix, denoted as $H^{NxD}$ ($N$: number of nodes, $D$: number of input features)
- An adjacency matrix *A(G)*

An initial layer-wise propagation rule can be formulated as a nonlinear function (Kipf & Welling, 2016):

$$f(H^{(l)}, A) = \sigma(A\, H^{l}\, W^{l})$$

where $W^{l}$ is a learnable weight matrix for the $l^{th}$ neural network layer which is trained through backpropagation. $\sigma(\cdot)$ is a non-linear activation function like the ReLU (Kipf & Welling, 2016):

$$y(x) = \max(x, 0)$$

However, this initial propagation rule has two limitations: It does not take into account the information of its own nodes, and it is not normalized (Kipf & Welling, 2016; Jiang & Luo, 2022). Firstly, it is important that the node retains its own properties and is not completely dependent on the values of its neighbors (Kipf & Welling, 2016). Furthermore, non-normalization could lead to an over scaling of the feature vectors (Kipf & Welling, 2016). For example, if a graph contains strong nodes with many neighbors, these can have a disproportionate influence on the aggregated features and overlay the signals of nodes with less strong connections.

To include self-information in graphs without self-loops, the identity matrix *I* with its entries on the diagonal is added to the adjacency matrix A (Kipf & Welling):

$$\hat{A} = A + I.$$

To counteract the problem of an unnormalized A, which can distort the node features during multiplication, normalization is achieved by using the inverse of the degree matrix D by applying $D^{-1}A$ (Kipf & Welling, 2016). However, a symmetrical normalization is often used in practice instead of averaging through the degree of each node.

These adjustments ultimately lead to the propagation rule by Kipf and Welling (2016) using a symmetric normalized adjacency matrix:

$$f(H^l, A) = \sigma(D^{-\frac{1}{2}}\hat{A}D^{-\frac{1}{2}}H^l W^l)$$

This weighted matrix multiplication with the neighboring values enables a local message passing procedure, by aggregating the information of all nodes with their direct neighbors (K=1). Graph convolutional networks are trained using backpropagation and gradient descent to adjust the weights and biases, methods originally introduced by Rumelhart, Hinton, and Williams (1986). This requires a differentiable loss function, such as the Root Mean Square Error (RMSE), which represents the difference between predictions and actual results (Werbos, 1990). Backpropagation then calculates partial derivatives starting from the loss function via the non-linear activation functions to the parameters of each node using the chain rule. The gradient descent minimizes the loss function by iteratively updating the parameters towards the negative gradient which leads to the best approximation of the target function (Nielsen, 2019).

### *2.2 Capturing the temporal dimension with RNNs*

In the following chapter, several recurrent neural network architectures are introduced as well as their mechanisms for processing sequential data.

#### *2.2.1 Recurrent Neural Networks and the Vanishing Gradient Problem*

Recurrent neural networks (RNNs) are supervised machine learning models characterized by their unique memory component (Haykin, 1998). Unlike conventional neural networks, which treat each input as if it were the first, RNNs retain information from previous inputs. For this reason, RNNs are often used for pattern recognition in various data sequences such as text- and time series data (Chung et al., 2014). This also comes into play in traffic forecasting, as they are able to capture the temporal dependencies in traffic data, which usually consist of chronological observations such as hourly vehicle traffic flow at an intersection (Chung et al., 2014; Ye et al., 2022).

This ability to remember information over time, is due to the hidden state, denoted as $H_t$, integrated in each cell of a recurrent neural network, which is passed on from one calculation step

to the next and contains important information about the past states (Salehinejad et al., 2017). Basically, a cell of a recurrent neural network consists of 3 different components: Input, hidden state and output (Schmidt, 2019). All components have weight matrices ($W_{hx}$, $W_{hh}$ and $W_{ho}$) and output as well as the hidden state have additional bias parameters ($b_h$, $b_o$) that are adjusted during training by Backpropagation Through Time (Schmidt, 2019).

Figure 3 shows an unfolded cell of a simple vanilla RNN. This network processes a sequence of input data ($X_1, X_2,..., X_p$) over several time steps. At each time step $t$, the RNN combines the current input vector ($X_t$) with the previous hidden state ($H_{t-1}$) to calculate the new hidden state ($H_t$) and the output ($O_t$) in a cell using the associated weight matrix and bias (Chung et al., 2014; Salehinejad et al., 2017). This architecture allows the RNN to store information from previous inputs which is crucial for recognizing patterns in sequential data such as speech or time series (Schmidt, 2019). Please note that the sources cited in this section used vectors to illustrate the functionality of RNNs. However, as this research focuses on graph-based matrices, notations were adapted accordingly in this and subsequent chapters on LSTM, GRU, and GCRN models.

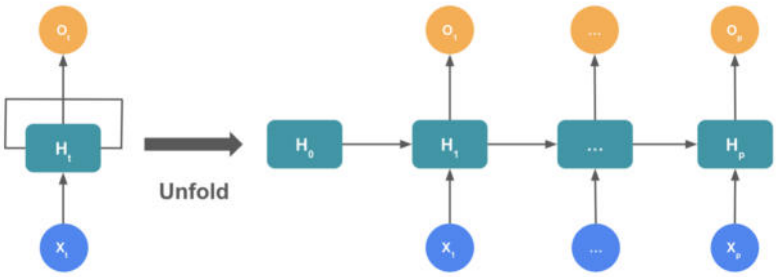



*Figure 3.* Architecture of a Vanilla Recurrent Neural Network. Adapted from "How to build a Graph-Based Deep Learning Architecture in traffic Domain: a survey" by J. Ye, J. Zhao, K. Ye, & C. Xu, 2022, *IEEE Transactions on Intelligent Transportation Systems*, 23(5), p 11 (Figure 7). Copyright 2022 by IEEE.

Backpropagation Through Time (BPTT) extends conventional backpropagation in RNNs by propagating errors backwards through all time steps to update parameters (weights, biases) (Yamak, Li, & Gadosey, 2019). This method aims to minimize the loss function by calculating partial derivatives using the chain rule to achieve a local/ global minimum along the negative

gradient (Werbos, 1990; Schmidt, 2019). However, RNN architectures using BPTT often encounter problems such as vanishing and exploding gradients caused by the extensive backpropagation over many time steps of the network (Bengio, Simard, & Frasconi, 1994). Since the hidden state $H_t$ in each time step is derived from the previous time steps using the chain rule, a sequential chain of derivation dependencies is created across all time steps (Schmidt, 2019). As an exact mathematical derivation of this problem is beyond the scope of this literature review, interested readers can find further details in Appendix A, which were derived by Schmidt (2019). Disappearing gradients occur when the products of these derivatives in the backpropagation process are too small, resulting in minimal updates to the weights, which in turn limits the ability of the network to learn long-term dependencies (Hochreiter, 1998). Salehinejad et al. (2017) point out that these problems are particularly prevalent for standard activation functions such as the sigmoid $\sigma$, which tend to produce gradients close to zero, exacerbating the vanishing gradient problem. Conversely, exploding gradients occur when these derivatives are too large, leading to drastic updates that destabilize the learning process (Salehinejad et al., 2017). To mitigate these problems, advanced architectures such as Long Short-Term Memory (LSTM) units and Gated Recurrent Units (GRUs) based on RNN have been developed (Madsen, 2019). These units contain information flow regulation mechanisms that allow them to maintain a stable gradient over time and improve learning from data with long-range dependencies, leading to higher prediction accuracy than the standard RNN (Cho, Van Merrienboer, Bahdanau, & Bengio, 2014; Hochreiter & Schmidhuber, 1997; Noh, 2021).

### *2.2.2 Long-Short-Term-Memory*

The Long-Short-Term-Memory (LSTM) architecture is considered as one of the most efficient methods to mitigate the vanishing/ exploding gradient problem (Madsen, 2019). This is mainly because LSTM are able to utilize a more constant error during training, which allows them to learn over more than 1000-time steps (Nicholson, 2019). Long-short-term-memory cells were first introduced by Hochreiter and Schmidhuber (1997). Since then, several small adjustments have been made to the original LSTM unit, as outlined by Chung et al. (2014). In addition, modifications such as the Fully Connected LSTM (FC-LSTM) have also been introduced, which will be explained in detail later.

In the context of traffic forecasting, LSTM units are used in various applications. For example, Chai et al. (2018) uses stacked LSTM cells in a multi-graph approach to model bike demand

predictions in NewYork (USA) based on usage data from the company CitiBike in combination with external influencing factors such as weather information. Furthermore, Song et al. (2021) used a deep convolutional LSTM network to improve the prediction of pedestrian movements and Poonia and Jain (2020) utilized LSTM cells for short-term traffic forecasting.

The architecture of a basic LSTM cell, shown below in Figure 4, consists of two key components: the hidden state and the cell state. The cell state ($C_t$) passes through the entire cell chain with minimal linear interactions and is updated at each time step via gated operations that contribute to the maintenance of long-term information, and thus serves as a component of the LSTM's long-term memory (Colah, 2015; Hochreiter & Schmidhuber, 1997). According to Noh (2021), it acts as a dynamic memory with a self-referential connection that allows it to adapt its real-valued state at each time step and integrate new inputs. The hidden state ($H_t$), on the other hand, is recalculated at each time step, reflects short-term dependencies, and acts as both a short-term memory and an output of the LSTM cell (Hochreiter & Schmidhuber, 1997). This distinction emphasizes the dual ability of the LSTM to manage immediate and long-term temporal dependencies by treating the hidden and cell states separately.

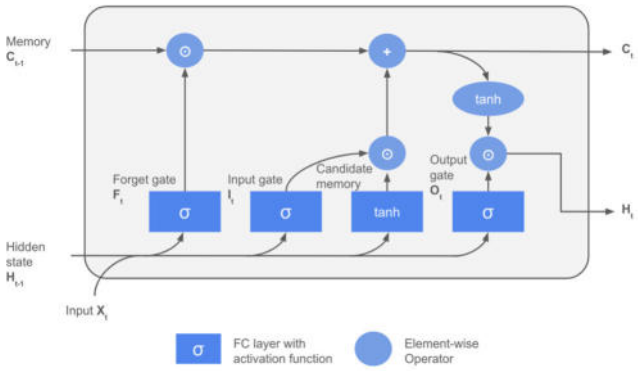



*Figure 4.* Basic architecture of a Long-Short-Term-Memory (LSTM) cell. Adapted from "Recurrent Neural Networks (RNNs): A gentle Introduction and Overview" by R. M. Schmidt, 2019, p 12 (Figure 10). Copyright 2019 by Cornell University.

The LSTM cell uses gates to regulate the information flow, i.e. decide what should be removed or added to the cell- and hidden states (Yang et al., 2020). A LSTM cell has 3 gates: an input, a forget and an output gate.

The gates are defined as follows (Schmidt, 2019):

$$I_t = \sigma\,(X_t W_{xi} + H_{t-1} W_{hi} + b_i)$$
$$F_t = \sigma\,(X_t W_{xf} + H_{t-1} W_{hf} + b_f)$$
$$O_t = \sigma\,(X_t W_{xo} + H_{t-1} W_{ho} + b_o)$$

where $W_{xf}$, $W_{xi}$, $W_{xo} \in R^{dxh}$ and $W_{hf}$, $W_{hi}$, $W_{ho} \in R^{hxh}$ are weight matrices and $b_f$, $b_i$, $b_o \in R^{1xh}$ are their respective biases. It is important to note that all 3 gates receive the same input (the features $X$ at time step $t$, as well as the last hidden state $H_{t-1}$). The weights and biases of an LSTM cell are learned by backpropagation through time, which is performed after processing the entire sequence in a forward pass (Gruslys, Munos, & Danihelka, 2019). The sigmoid activation function σ, which introduces non linearity for the different gates, plays a crucial role in the LSTM gates by deciding which information is allowed through or blocked (Colah, 2015).

The forget gate of an LSTM cell uses the input in order to determine how much of the last cell state $C_{t-1}$ is retained for the next computation step (Schmidt, 2019). This gate ensures that the LSTM can discard irrelevant information, e.g. when a text analysis reaches the end of a document and the subsequent document is unrelated, so that it is not necessary to remember the previous content (Nicholson, 2019). By applying a sigmoid activation function σ in the forget gate, resulting in values between 0 (completely forgotten) and 1 (completely retained), the cell state $C_{t-1}$ is adjusted by an element wise multiplication (Hadamard product) with the output of the forget gate (Yu, Si, Hu, & Zhang, 2019).

The input gate works together with the candidate memory to decide which current information should be included in the cell state based on the inputs $X_t$ and $H_{t-1}$ (Nichsolson, 2019). The input gate, just like the forget gate, uses a sigmoid activation function σ to decide which values of the inputs should be included in the cell state by values between 0 and 1. The candidate memory generates a matrix of potential new values using a tanh function, which outputs values $\in (-1,1)$. The tanh function can be defined as:

$$tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

The candidate memory is defined as follows (Schmidt, 2019):

$$\widehat{C_t} = tanh(X_t W_{xc} + H_{t-1} W_{hc} + b_c)$$

where $W_{xc} \in R^{dxh}$ and $W_{hc} \in R^{hxh}$, as well as biases $b_c \in R^{1xh}$. The two matrices from the input gate and the candidate memory are now merged again with an element wise multiplication. This results in a new matrix that contains the relevant (input gate) and transformed values (candidate memory) of the current input, which is then added to the new cell state.

The new cell state is calculated by (Schmidt, 2019):

$$C_t = F_t \odot C_{t-1} + I_t \odot \widehat{C_t}$$

This equation combines the already explained components of the forget gate and the input gate. In the first part of the equation, it is decided what is to be forgotten from the old state by performing an element wise multiplication of the forget gate and the old cell state, while the second part adds new memory content through the merging of input gate and candidate memory.

The central plus sign in LSTMs is crucial for mitigating the vanishing gradient problem, as it maintains a stronger gradient flow over multiple time steps (Nicholson, 2019; Ye et al., 2022). This additive component allows the LSTM cell to retain existing content and add new information, thus enabling long-term storage and preventing important features from being overwritten (Chung et al., 2014). In contrast, the traditional RNN unit always replaces the activation value or content of a unit with a new value calculated from the current input and the previous hidden state (Chung et al., 2014).

Finally, the output or hidden state of the cell is calculated based on a filtering of the current cell state as well as the current input and the last hidden state (Schmidt, 2019):

$$H_t = O_t \odot tanh(C_t)$$

*2.2.3 Gated-Recurrent-Unit*

The Gated Recurrent Unit (GRU) was proposed by Cho et al. (2014). The GRU receives/gives the same input/output as an ordinary RNN, but its internal structure is a simplified version of the LSTM, which also includes the additive component when updating from *t* to *t+1* (Chung et al., 2014; Yang et al., 2020). The simplification is achieved by reducing the number of gates to just an update- and a reset gate (Dey & Salemt, 2017), which can be seen in Figure 5. Another difference is that there is no separate memory cell (Chung et al., 2014). This streamlined architecture uses the hidden state directly, which is regulated by the update and reset gates, making GRU a leaner variant with fewer parameters compared to LSTM.

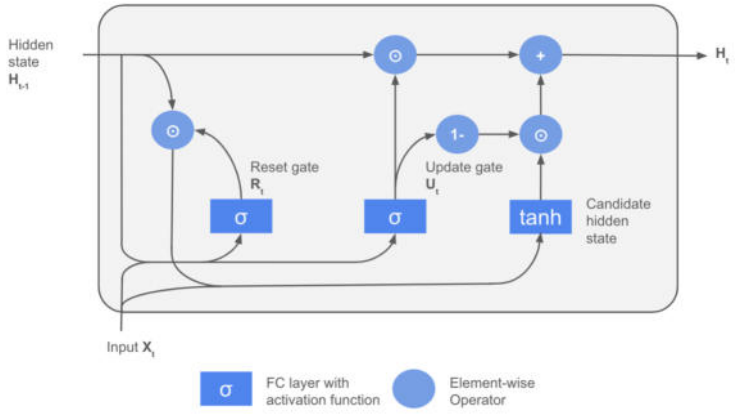



*Figure 5.* Basic architecture of a Gated Recurrent Unit (GRU) cell. Adapted from "LSTM and GRU Neural Network Performance Comparison Study: Taking Yelp Review Dataset as an Example" by S. Yang, X. Yu, & Y. Zhou, 2020, *IEEE International Workshop on Electronic Communication and Artificial Intelligence (IWECAI), p 99 (Figure 2)*. Copyright 2020 by IEEE.

GRU cells are used in similar applications as LSTM cells for traffic forecasting. For example, Zhao et al. (2020) uses the graph convolution of Kipf & Welling (2016) in their T-GCN model and combines it with GRU cells to evaluate the traffic prediction for the SZ-taxi (Shenzhen, China) and Los-loop (Los Angeles County, USA) datasets, which both record real-time traffic speed but in different time intervals.

The gates of a GRU cell are defined as follows (Chung et al., 2014):

$$R_t = \sigma\left(X_t W_{xr} + H_{t-1} W_{hr} + b_r\right)$$
$$U_t = \sigma\left(X_t W_{xu} + H_{t-1} W_{hu} + b_u\right)$$

where $W_{xr}$, $W_{xu} \in R^{dxh}$ and $W_{hr}$, $W_{hu} \in R^{hxh}$ are weight matrices and $b_r$, $b_u \in R^{1xh}$ are their respective biases. Please note that the formulas originate from Chung et al. (2014) and not from Cho et al. (2014), as they provide a different syntactic formulation. The indices have been adapted to those used by Schmidt (2019) for better readability.

The reset gate decides how much of the hidden state $H_{t-1}$ should be retained and is therefore similar to the forget gate of the LSTM (Fu, Zuo, & Li, 2016). The update gate determines the extent to which the unit refreshes its current information (Chung et al., 2014). It should be noted that the sigmoid activation function σ is also utilized here to output values between 0 and 1, which is used to regulate how much and what information should be passed through.

The candidate for the new state is calculated by (Chung et al., 2014):

$$\widehat{H_t} = tanh(R_t \odot (H_{t-1} W_{hr}) + X_t W_{xh} + b_h)$$

It uses the reset gate to store the relevant information from the past. If the Reset Gate $R_t$ is off ($R_t$ is close to 0), then the previously calculated state is forgotten (Chung et al., 2014). Finally, the new hidden state is calculated by a linear interpolation of the previous state and the candidate for the new state using the update gate (Chung et al., 2014):

$$H_t = (1 - U_t) \odot H_{t-1} + U_t \odot \widehat{H_t}$$

*2.3 Reproduction of the GCRN models*

The spatio-temporal models used in this research originate from Seo, Defferrard, Vandergheynst, and Bresson (2016), who present Graph Convolutional Recurrent Networks (GCRN). Two variants of this model were presented in their paper: The first variant operates on grid-based data using GNNs as the spatial component, which was validated by an experiment to predict moving handwritten digits (Moving MNIST Dataset). The second variant, which will be used in this research, is based on graph convolutions and was used by Seo et al. (2016) to improve language modeling by capturing spatial and semantic relationships in the Penn Treebank dataset. These deep learning networks integrate the spatial capabilities of Graph Convolutional Networks (GCNs) with the dynamic pattern recognition of Recurrent Neural Networks (RNNs) to capture both spatial and temporal relationships within the data (Seo et al., 2016). This reproduction being the last chapter of this literature review, aims to provide the reader with a clear understanding of the mechanisms that link these different neural network models.

The input $X_t$ and $H_t$ are first processed by a graph convolutional network with Chebyshev polynomials, denoted through G (Seo et al., 2016):

$$X_t^{GCN} = W_t^{GCN} * G * X_t$$
$$H_{t-1}^{GCN} = W_t^{GCN} * G * H_{t-1}$$

The matrix outputs of these terms are then used as inputs to the RNN cell computations, which include various gates and memory components as explained earlier in this literature review. In particular, Seo et al. (2016) use Fully-Connected Long-Short-Term-Memory (FC-LSTM), and Gated Recurrent Unit cells as RNN variants. Fully-Connected Long-Short-Term-Memory (FC-LSTM) introduce additional connections Peephole connections, which link the cell state directly to the LSTM gates, allowing them to more precisely decide what information to retain or discard by examining the current or previous memory cells (Seo et al., 2016; Yu et al., 2019; Gers & Schmidhuber, 2000). These peephole connections are denoted as $W_{ci}$, $W_{cf}$ and $W_{co}$. The gates and cell states of these architectures therefore receive sequential data whose underlying spatial information for each node has already been aggregated with its neighbors using a convolutional filter. This integration allows the model to use both the spatial dependencies captured by GCNs and the sequential dependencies modeled by RNNs.

The following equations, derived from Seo et al. (2016) present the final equations used in this research. The GConvLSTM is defined as:

$$I_t = \sigma(X_t^{GCN} W_{xi} + H_{t-1}^{GCN} W_{hi} + W_{ci} \odot C_{t-1} + b_i)$$

$$F_t = \sigma(X_t^{GCN} W_{xf} + H_{t-1}^{GCN} W_{hf} + W_{cf} \odot C_{t-1} + b_f)$$

$$O_t = \sigma(X_t^{GCN} W_{xo} + H_{t-1}^{GCN} W_{ho} + W_{co} \odot C_t + b_o)$$

$$C_t = F_t \odot C_{t-1} + I_t \odot tanh(X_t^{GCN} W_{Xc} + H_{t-1}^{GCN} W_{hc} + b_c)$$

$$H_t = O_t \odot tanh(C_t)$$

and the GConvGRU is defined as:

$$R_t = \sigma(X_t^{GCN} W_{xr} + H_{t-1}^{GCN} W_{hr} + b_r)$$

$$U_t = \sigma(X_t^{GCN} W_{xu} + H_{t-1}^{GCN} W_{hu} + b_u$$

$$H_t = (1 - U_t) \odot H_{t-1}^{GCN} + U_t \odot \widehat{H}_t$$

$$\widehat{H_t} = tanh(R_t \odot (H_{t-1}^{GCN} W_{hr}) + X_t^{GCN} W_{x\mathrm{h}} + b_{\mathrm{h}})$$

It should be noted that the work of Seo et al. (2016) also uses an encoder-decoder architecture (seq2seq) for sequence modeling, which is well suited for multi-step traffic forecasting (Ye et al., 2022). This approach allows the model to generate multiple future steps in a sequence, which improves its utility for predicting traffic patterns over longer time periods. However, as this has not been implemented in the PyTorch library, this architecture will not be discussed further. Instead, this research focuses on single-step forecasting, where only the traffic state in the next time step is predicted (Jiang & Luo, 2022).

# 3. Methodology

Currently, there are only few research papers that comprehensively compare LSTM and GRU models with spatial dimensions using graphical neural networks for traffic forecasting, especially for bike transport demand prediction. The effectiveness of those models is highly dependent on the data and application context, which requires further research (Chung et al., 2014; Yang et al., 2020). Furthermore, the 'black box' nature of their training processes limits understanding and raises reliability concerns, particularly in critical areas such as autonomous driving and intelligent transport (Gilpin et al., 2018; Liang et al., 2021). Rozemberczki et al. (2021) only tested these models in their PyTorch Geometric Temporal Framework generically. They did not optimize any key hyperparameters and did not use a baseline model as a critical reference point. This systematic optimization and evaluation is carried out in this study.

## *3.1 Research Design*

This research aims to investigate the effectiveness of two Graph Convolutional Recurrent Networks (GCRN) from the PyTorch Geometric Temporal library of Rozemberczki et al. (2021), where one contains LSTM cells (GConvLSTM) and the other GRU cells (GConvGRU). Different hyperparameters are systematically varied in order to find the best performance per model and to check its generalisation capability. These models and a baseline model are compared on the publicly available data set from Pedal Me in London. This dataset contains historical data of weekly counts for bike transportation requests, which is structured in graphs for 15 regions (see table 1). The bike transport requests here reflect the demand for bike transport. A regression task is to be solved on the data set, i.e. in one step the numerical demand for the next point in time is forecasted on the basis of the 4 given lags as features for each region. It is important to note that other features such as weather data and information on major events, including sporting events or royal holidays, have been taken into account. However, the dataset lacks detailed descriptions of the specific weeks or geographical regions covered between 2020 and 2021, which is why the given lagged features are used, in line with the approach of Rozemberczki et al. (2021). Furthermore, the Pedal Me dataset is smaller than other datasets used for transportation demand- or traffic-forecasting in general, which could let to overfitting for such complex models. It was selected because it is more manageable in terms of available computational resources. Despite the limitations of the dataset and the risk of overfitting, this study aims to investigate whether the GCRN models are more suitable than the baseline model

for use in a decision support system at Pedal Me, as they can maintain their superior functionality on the smaller dataset.

Table 1

*Overview of Pedal Me dataset*

| **Category** | **Description** |
|---|---|
| Name | Pedal Me |
| Location | London |
| Application context | Weekly counts of bike transport requests. Nodes represent geographical units and edges are mutual adjacency relationships based on proximity. |
| Source | PyTorch Geometric Temporal |
| Number of nodes (regions) | 15 |
| Number of time steps | 31 weeks |
| Number of features per node | 4 Lags (numerical) |
| Target | Weekly counts of bike transport requests (numerical) |
| Attributes | edges, edge_weights, targets_and_features, targets |

*Note.* Adapted from: "PyTorch Geometric Temporal: Spatiotemporal Signal Processing with Neural Machine Learning Models," by B. Rozemberczki, P. Scherer, Y. He, G. Panagopoulos, A. Riedel, M. Astefanoaei, O. Kiss, F. Beres, G. López, N. Collignon, & R. Sarkar, 2021, p 7 (Table 4). Copyright 2021 by the authors.

The ARIMA (Auto Regressive Integrated Moving Average) model is used as the baseline model. It has been used in several comparative studies in the transport sector as well as in similar studies examined for this research design (Fu et al., 2016; Wang et al., 2020; Chai et al., 2018). More precisely, the Auto-ARIMA from the 'pmdarima' package in Python is used to determine the best parameters per region. This and the restructuring of graph data into time series data is explained in more detail in section '3.2 Modeling'.

Mean Squared Error (MSE), Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE) are used as metrics for performance measurement:

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y_i})$$

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

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

This decision is based on the fact that this research deals with a regression task, where the deviation of the predictions from the actual values must be measured, which can be done by using mean error measures. The smaller these values, the better the predictive performance. These measures have also been used in similar comparative studies in the traffic domain (Fu et al., 2016; Wang et al., 2020; Rozemberczki et al., 2021). To get a comprehensive overview of the model's performance, the analysis and collection of performance metrics is divided into two parts. Firstly, the training and test performance of the models in 3 selected regions with different volatility is compared using the MSE (region-level analysis). Here, epochs and drop-out layers are varied for an in-depth analysis. In addition, time series plots for the predictions of selected regions will be presented. At the global level, the epochs for the GCRN models with dropout are varied first, while collecting test and training MSEs for all regions together, with regard to the learning curves and generalization capability of the two GCRN models. The overall model performances are then analyzed with their best configuration of epochs and drop-out layer across all regions to find the best model for this application context, using the MSE, MAE and RMSE for training and testing.

Holdout validation using the ‘temporal_signal_split’ function integrated in the library is used for the GCRN models, as well as a time-based train/test split for the ARIMA models. Holdout validation was chosen for its simplicity and ability to provide a straightforward assessment of model performance. An 85/15 train/test split was chosen to single-step forecast the last 5 timesteps of the Pedal Me data set. As the hidden states (and cell states in the LSTM) are

initialized with random numbers close to 0, this could lead to fluctuations in model performance as the backpropagation algorithm finds different local/global minima each time. For this reason, each model configuration is run 5 times and the standard deviation is calculated. Significant overfitting was identified at the GCRN models during the hold-out validation. Due to the limited size of the data set and the complexity of the models, no time series cross-validation was performed, as it could lead to even more intensive training on individual folds and thus exacerbate overfitting.

The following hyperparameters are included in this comparison to see if performance can be improved: Number of epochs and variation with an additional dropout layer. Here, dropout is a regularization technique that helps to prevent overfitting by randomly dropping units during training (Srivastava et al., 2014). The number of epochs represent the training time. 4, 10 and 30 epochs are considered after initial tests. Other parameter variations are not considered to avoid going beyond the scope of this research. The best learning rate of 0.01 was determined by initial tests in which the learning rates 0.0001, 0.001 and 0.1 were also tested. The comparison was then carried out by changing the epochs and adding an additional dropout layer before the linear output layer. An attempt was made to find a dropout value that works equally well for each model without flattening the predictions too much. A value of 0.3 was therefore considered appropriate as it could effectively balance regularization and model performance.

### *3.2 Data Inspection and cleaning*

An initial data inspection is carried out before modeling begins. To do this, the PyTorch Geometric and PyTorch Geometric Temporal libraries had to be installed in order to access the Pedal Me dataset and models (see appendix B for the library installation code). First, it was tested tested if there are missing values in the dataset and it turned out that all values are complete. After that, time series with target and features (see appendix C for uncleaned time series plot per region), boxplots (see appendix D) as well as descriptive statistics including mean, standard deviation, min- & max-values and quartiles (see appendix E) from the various regions were plotted. The adjacency matrix of the Pedal Me dataset can be found in appendix F and includes the mutual adjacency relationships based on proximity between the regions.

The different volatility of the demand for Pedal Me bike transport in the different regions of London became clear during data inspection. The mean values of the target variable (number

of requests) are close to zero in all regions, e.g. -0.027 in region 1 and -0.020 in region 3, but the standard deviations vary more, e.g. 0.233 in region 1, 0.785 in region 3 and 0.408 in region 14, reflecting the different volatility of demand in the regions. In region 5, for example, the quartiles range from -0.474 to 0.246, while in region 12 they vary from -0.665 to 0.425. In general, the distribution of values is similar, with a median close to 0 in each region. This distribution suggests a relatively symmetrical pattern of demand, with some outliers in certain regions, such as region 3 or 12. This analysis suggests that the data has been standardized, resulting in mean values close to zero and comparable scales across regions. As the authors of the library PyTorch Geometric Temporal do not provide any information on the data transformation process, the dataset is not modified in this respect. Repeating, albeit non-linear, patterns were observed in the data. These patterns show significant fluctuations and outliers, which are likely to be influenced by external factors affecting the demand for bike transport at Pedal Me. Nevertheless, these fluctuations show some consistency between different regions, suggesting potentially predictable behavior despite the apparent volatility.

However, it is noticeable that in each region in snapshot 0 there is an unrealistically high value for lag 0 and for the target (demand) in snapshot 30 an unrealistically low value compared to the other data points in each time series. These outliers could significantly influence the model performance and the reliability of the analysis. For this reason, the 'trim_to_quartiles' function is used per region to trim the outliers to the lower or upper quartiles of the respective region (see appendix G for data cleaning/ inspection code and appendix H for the cleaned time series per region). Other fluctuations and outliers are representative of the real demand behavior of customers from Pedal Me in London and are therefore not adjusted in order to minimize distortion of reality.

### *3.3 Modeling*

This research utilizes the PyTorch Geometric Temporal Library by Rozemberczki et al. (2021), an open-source Python library built on the PyTorch Geometric Ecosystem. It provides modular neural network layers for spatio-temporal machine learning with accessible hyperparameters and integration into existing frameworks. This library supports tasks such as epidemiological forecasting and ride-hail demand prediction (Rozemberczki et al., 2021).

*3.3.1 Auto-ARIMA as Baseline*

First, the dataset from the 'torch_geometric_temporal.dataset' package is loaded using the 'PedalMeDatasetLoader'. After data cleaning, the target (number of requests) of each region is extracted from the graph format and organized into data frames. Each data frame contains the time series for one region. The Augmented Dickey-Fuller test is then used in a for loop to check the stationarity of the time series data for each region and to correct it if necessary. This step is crucial as non-stationary data can lead to misleading model estimates and forecasts, and is also a common practice for determining stationarity (Alghamdi, Elgazzar, Bayoumi, Sharaf, & Shah, 2019). In time series analysis, stationarity is a fundamental assumption for many models, including ARIMA, as it ensures that the statistical properties of the series do not change over time. ARIMA models are powerful forecasting tools as they can capture the dynamics of the time series through their parameters (Alghamdi et al., 2019). 'Auto_arima' from the ‘pmdarima’ package is then used to determine the best-fitting ARIMA model parameters (p, d, q) for each region's time series data. According to Alghamdi et al. (2019), the three parameters that have statistical significance for model accuracy can be defined as follows:

- p: The number of lagged observations included in the model
- d: The degree of differentiation for the time series
- q: The size of the moving average window

Since the previous stationarity analysis function already differentiates the time series, when necessary, Auto-ARIMA only specifies the minimum and maximum parameters for p and q. These have been set to a minimum of 0 and a maximum of 5 after initial experiments. The code is shown in appendix I.

*3.3.2 GCRN models*

The development of the GCRN models was based on the documentation by Rozemberczki et al. (2021). The architecture of each model is initialized by the ’_init_’ method, which incorporates GConvLSTM or GConvGRU cells. These cells utilize symmetric normalization for the graph convolution, perform 2-hop neighbor aggregation, and use MSE as the loss function for backpropagation. The GConvLSTM or GConvGRU layer are followed by a ReLU activation function to increase the non-linearity in the network by setting negative values to zero and leaving positive values unchanged. In addition, one dropout layer is implemented which can

be toggled to check differences in performance. The `forward` method controls the forward pass of the model, starting with the processing of node features, edge indices and edge weights. After the GConvLSTM layer, the result is transformed by a linear layer that generates the final output of the model. This linear activation function is crucial for scaling the output to specific prediction targets. During training, the model iterates through the training data in epochs to optimize the parameters. In each epoch, the losses are calculated using the mean squared error and adjusted by back propagation using the Adam optimizer. The Adam optimizer is a modification of the backpropagation algorithm by varying the learning rates with gradient estimates, which helps the model to converge faster (Kingma & Ba, 2014). During evaluation, predictions are made for the test dataset and the losses and actual and predicted values are stored in lists for later evaluation. Once training is complete, performance metrics are determined through predicting on the test dataset. Performance metrics (regional- and global-level), learning curves as well as time series for the selected regions are plotted afterwards. The code for the GCRN models and the plotting can be seen in appendix J.

## 4. Evaluation

In this section, the performance results of the GCRN and ARIMA models for bike transport demand prediction are presented and analyzed by predicting the number of weekly requests in the Pedal Me dataset. A training/test split of 85/15 is used for single-step forecasting of the last 5 timesteps (weeks) of the data set, while varying dropout and training time (in epochs) for the GCRN models. The analysis is split into two parts:

1. Detailed analysis of the selected regions: Detailed analysis of three specific regions (regions 1, 3 and 15), each with a different degree of demand volatility,
2. Model performance across all regions: Comparison of the model performance metrics on average across all regions.

The approach allows for a detailed assessment of model performance at both a granular level (specific regions) and a broader level (all regions). Due to the limited scope of this study, the

detailed analysis was restricted to selected regions. While this approach provides valuable insights into specific regions, it also means that the results cannot be directly generalized to all regions. However, analysis at the regions-level highlights unique patterns and differences in performance that might be overlooked in a purely global analysis, where the averaging of performance metrics across all regions might blur important insights. For the performance on the test dataset, "test" is used, followed by the respective metric. For the performance on the training dataset, "train" is used, followed by the respective metric.

### *4.1 Region-Level Analysis*

Region 1 shows low volatility with a target standard deviation of demand of ± 0.232, which indicates relatively stable demand. Region 3 shows high volatility with a target standard deviation of ± 0.784. This reflects regions with strong fluctuations and variable capacity utilization, which may be due to external factors. Region 15 represents moderate volatility with a target standard deviation of ± 0.495, reflecting a balanced mix of stable and volatile demand. This selection ensures a comprehensive analysis reflecting different demand dynamics for Pedal Me in London. All detailed results for the performance comparison at region level can be seen in appendix K.

#### *4.1.1 Best model performance for selected regions*

In regions 1 and 3, the ARIMA shows the best performance of all models on the test dataset with a MSE of 0.0242 for region 1 and a MSE of 0.3168 for region 3. The Auto-ARIMA achieved this performance with parameters p=2, d=0 and q=0 in both regions. In the low volatile region 1, GConvLSTM performs best of the GCRN models with a test MSE of 0.0362 ± 0.0062, being close to the ARIMA. This is followed by GConvGRU with a test MSE of 0.05066 ± 0.00847. In region 3, the average performance over the 5 repetitions for both GCRN models is slightly worse compared to ARIMA with test MSEs of 0.3096 ± 0.0315 for the GConvGRU and 0.3196 ± 0.0615. However, the ARIMA performance is within the standard deviation of these values. In region 15, both GCRN models are better than the baseline ARIMA and achieve similar performances: The GConvLSTM has a test MSE of 0.1098 ± 0.0176 and the GConvGRU a test MSE of 0.1117 ± 0.0271. In this region, the ARIMA achieves a test MSE of 0.3992, also with the parameters p=2 and 0 for d and q. The best test performance of the GCRN models was achieved in all regions with a training time of 4 epochs and the use of

an additional dropout layer, except for the GConvGRU in region 3 (4 epochs but without an additional dropout layer).

### *4.1.2 Trend Analysis for Training and Testing Performance*

In all three regions, there is a tendency for the test performance of the GCRN models to deteriorate with increasing training time, accompanied by a decrease in the train MSE in the more volatile regions 3 and 15. For region 3, which has a high volatility of bike transportation demand, we observe significant changes with different training times. With 4 epochs and no dropout layer, the GConvGRU model achieves a train MSE of 0.4553 ± 0.0043 and a test MSE of 0.3096 ± 0.0315, while the GConvLSTM model has a train MSE of 0.4694 ± 0.0098 and a test MSE of 0.3196 ± 0.0616. This balance at 4 epochs suggests that the GCRN models are able to moderately generalize on unseen data under strongly fluctuating demand over the weeks at a low training time for this region. When the training time is increased to 10 epochs without adding a dropout layer, the train MSE for GConvGRU improves to 0.3644 ± 0.0116, but the test MSE deteriorates to 0.5774 ± 0.1184. Similarly, the train MSE of the GConvLSTM model improves to 0.3701 ± 0.0077, while the test MSE increases to 0.8481 ± 0.0866. At 30 epochs, the increasing difference between training and testing performance becomes more evident: The train MSE of GConvGRU decreases further to 0.1912 ± 0.0135, but the test MSE increases significantly to 1,4751 ± 0.2689. Similarly, GConvLSTM has a train MSE of 0.1948 ± 0.0128 and a test MSE of 1.3646 ± 0.3315.

In region 15, which represents moderate volatility, the GCRN models show a similar trend. With 4 epochs and without an additional dropout layer, the GConvGRU model achieves a train MSE of 0.1808 ± 0.0050 and a test MSE of 0.1252 ± 0.0357, while the GConvLSTM model has a train MSE of 0.1891 ± 0.0031 and a test MSE of 0.1159 ± 0.0265. This balance shows that the models with 4 epochs can maintain a relatively stable relationship between training and test performance for this region with moderate volatility. With an increase to 10 epochs and no dropout layer, the train MSE for GConvGRU improves to 0.1465 ± 0.0047, while the corresponding test MSE deteriorates to 0.1900 ± 0.0548. The training MSE of GConvLSTM also improves to 0.1589 ± 0.0071 and the test MSE increases to 0.1798 ± 0.0273. At 30 epochs with no dropout layer, GConvGRU shows a further reduced train MSE of 0.0937 ± 0.0149, but the test MSE increases significantly to 0.4878 ± 0.1246. Similarly, the GConvLSTM model shows a train MSE of 0.0945 ± 0.0160 and a higher test MSE of 0.4165 ± 0.0935.

Region 1, which is characterized by lower volatility, shows a slightly different pattern to regions 3 and 15. For 4 epochs with no dropout layer, the GConvGRU model achieves a train MSE of 0.0325 ± 0.0013 and a test MSE of 0.0572 ± 0.0078, while the GConvLSTM model has a train MSE of 0.0346 ± 0.0015 and a test MSE of 0.0519 ± 0.0097. This demonstrates a relatively balanced relation between training and testing. When training is increased to 10 epochs without a dropout layer, the train MSE for GConvGRU changes only slightly to 0.0389 ± 0.0029 and the test MSE increases to 0.0862 ± 0.0313. Similarly, the train MSE of the GConvLSTM model adjusts to 0.0354 ± 0.0018, while the test MSE increases to 0.0711 ± 0.0127. After 30 epochs, both GCRN models show a deterioration in training performance. The training MSE of the GConvGRU model increases to 0.0525 ± 0.0036, followed by a significantly higher test MSE of 0.2715 ± 0.1003. The GConvLSTM model shows a similar trend: the training MSE increases to 0.0551 ± 0.0051 and the test MSE deteriorates to 0.3029 ± 0.0536.

The observed trend of increasing test MSE combined with decreasing training MSE across epochs suggests that models in regions 3 and 15 begin to overfit after more than four training epochs, with some evidence also visible in region 1. Overfitting occurs when a model learns the training data too well, capturing noise and details that are specific to the training set, resulting in poor generalization to new, unseen data (Xue, 2019). The higher standard deviations of the test MSEs in the different regions at higher training time also indicate that the performance of the models varies significantly when applied to new data, suggesting unstable and unreliable predictions. This effect is more pronounced in the volatile regions 3 and 15 at higher epochs, which may indicate a greater sensitivity to fluctuations in these regions. In region 1, although test performance deteriorates, there is also an increase in MSE during training. This could indicate that the model is losing its ability to learn the training data correctly, indicating overfitting to noise or too specific patterns in the training data.

In contrast to the GCRN models with training times of more than 4 epochs, the ARIMA model shows a moderate deviation between training and test performance across all regions. The train MSEs for ARIMA (0.0449 for region 1, 0.1932 for region 15, and 0.4720 for region 3) are comparable to those of the GCRN models. However, the test MSEs (0.0242 for region 1, 0.3992 for region 15 and 0.3168 for region 3) show that ARIMA has relatively stable generalization capabilities for these regions.

At the end of the Region-Level analysis, visualizations of the predictions for all three models in the selected regions are presented. Figure 6 shows the predictions with 4 epochs and additional dropout layer, while Figure 7 presents the predictions with 30 epochs without an additional dropout layer.

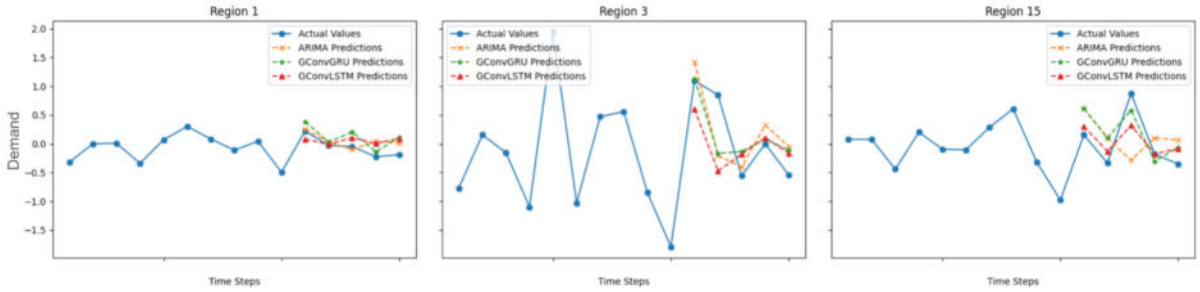



*Figure 6.* Visualization of predictions for selected regions on 4 epochs with dropout

Figure 6 shows that all three models can adapt to the actual values of the target variable to a certain extent in these regions. In region 1, characterized by lower volatility, the predictions of all models agree well with the actual values. It can also be seen that both GCRN models (GConvGRU and GConvLSTM) have similar prediction curves. However, in region 3, all the models correctly predict the first time step, but they are not able to correctly predict the sustained high demand in the following week.

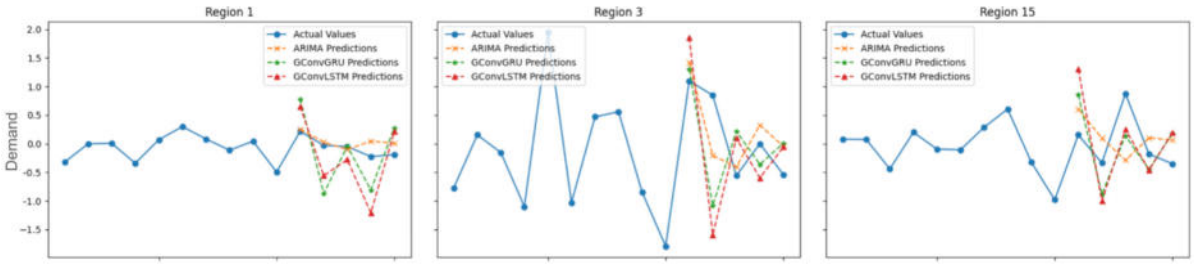



*Figure 7*. Visualization of predictions for selected regions on 30 epochs without dropout

Figure 7 shows the predictions for selected regions after 30 epochs without dropout. The GCRN models tend to be overdriven, which is recognizable by stronger fluctuations in the predictions. It is also evident that similar patterns are produced by the GCRN models in all regions, when the training time is increased.

### *4.1.3 The influence of Dropout Layers*

There is a consistent trend across all regions, suggesting that the dropout counteracts the increasing deviation between test and train performance in both GCRN models at a certain extent. For this reason, only the key performance indicators of the GConvGRU are presented below. Looking at region 15 (moderately volatile), this effect is clearly visible with increasing training time. For 4 epochs, the Train MSE without dropout is 0.1808 ± 0.0050 and the Test MSE is 0.1252 ± 0.0357, while with dropout the Train MSE is 0.1912 ± 0.0079 and the Test MSE is 0.1117 ± 0.0271. At 10 epochs the Train MSE without dropout is 0.1465 ± 0.0047 and the Test MSE is 0.1900 ± 0.0548, whereas with dropout the Test MSE is 0.17212 ± 0.05636 and the Train MSE is 0.16906 ± 0.00529. After 30 epochs, the Train MSE without dropout reaches 0.0937 ± 0.0149 and the Test MSE increases significantly to 0.4878 ± 0.1246, while with dropout the Train MSE is 0.1582 ± 0.0093 and the Test MSE is 0.4108 ± 0.0717. These results indicate that dropout helps the model to improve generalization and avoid overfitting on this dataset at a certain extent. Nevertheless, the deviation of both values still increases as the number of epochs increases, which could mean that overfitting could not be completely prevented.

## *4.2 Global Performance Analysis*

For the global analysis, the learning curves are presented first in order to assess whether the trend of increasing deviation between training and test MSE is transferable to all regions. A detailed table of the train/test MSE values per epoch with dropout can be found in Appendix L, which is consistent with the observations of the learning curves. In this section, only the learning curves of the GConvGRU are shown in Figures 8 (4 epochs), 9 (10 epochs) and 10 (30 epochs). These are similar to the learning curve of the GConvLSTM, which indicates a similar learning and prediction behavior of both models on the Pedal Me data set. The learning curves of the GConvLSTM can be seen in Appendix M.

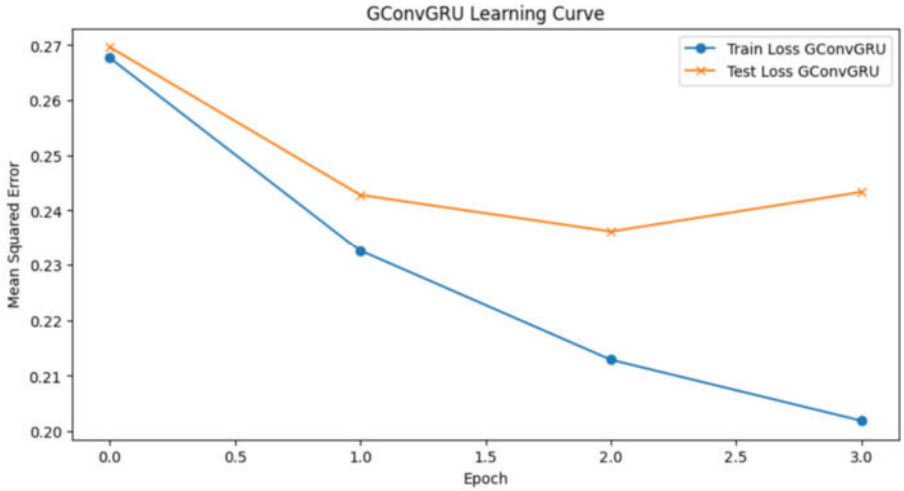



*Figure 8.* Learning Curve GConvGRU on 4 epochs and dropout

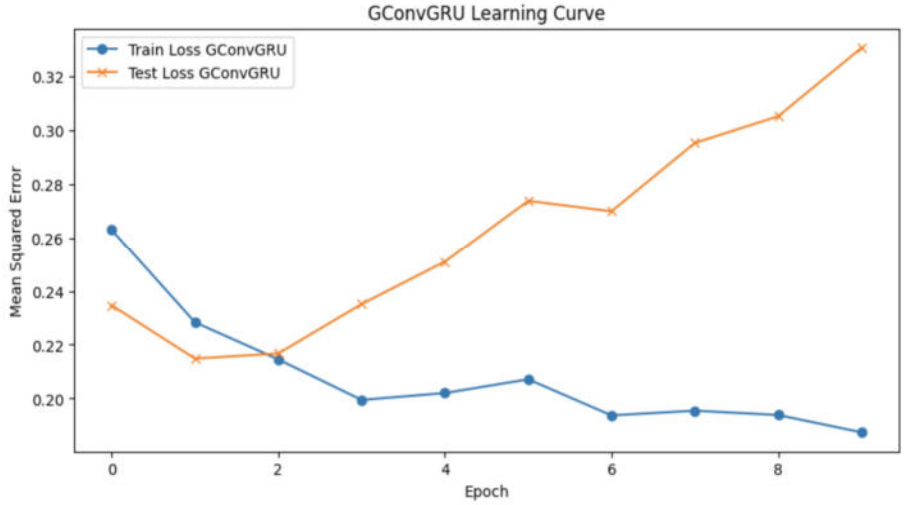



*Figure 9.* Learning Curve GConvGRU on 10 epochs and dropout

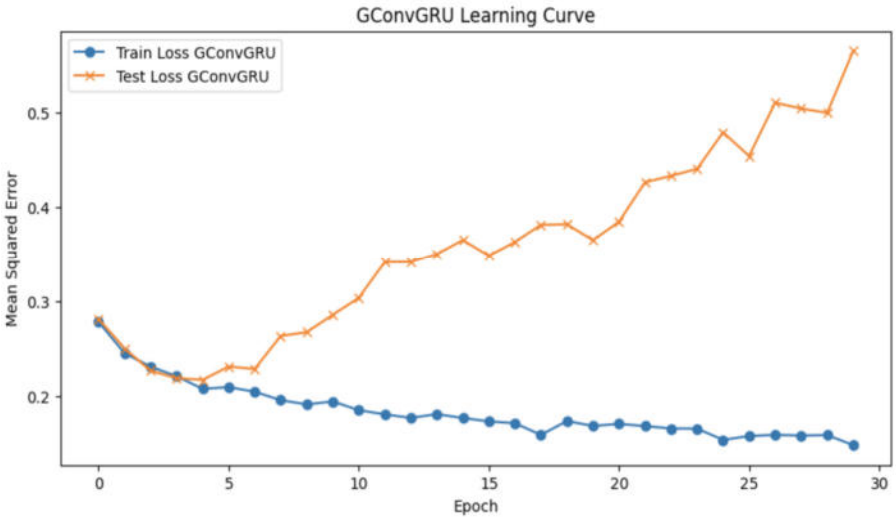



*Figure 10.* Learning Curve GConvGRU on 30 epochs and dropout

The learning curves of the GConvGRU model on 4 epochs show that the train MSE starts low early on and stabilizes quickly, which indicates rapid pattern recognition. Although the test MSE remains relatively stable up to 4 epochs, the widening divergence between training and test errors at 10 and 30 epochs indicates overfitting in all regions. The continuous increase in test errors with longer training, despite the use of a dropout layer with a rate of 0.3, suggests the need to adapt regularization strategies or to closely monitor both training and test performance over longer training durations to identify overfitting early and improve model performance. Furthermore, the deviating test MSEs from the train MSEs at 30 epochs indicate Rozemberzcki et al. (2021) chose a too high number of epochs (100) for this data set on their initial tests.

Now the overall model performance metrics in the form of MSE, RMSE and MAE for both training and testing will be presented, which can be seen in Table 2. For the GCRN models, the averages and their standard deviations were again calculated using 5 experimental repetitions. It is important to note that only the performance with 4 epochs and an additional dropout layer (dropout value: 0.3) are used for the GCRN models, as these have proven to be the best parameter settings for most of the regions. To calculate the overall performance metrics, the actuals and predictions per model during training and testing (after the last epoch for GCRN models), were collected together in all regions, and then the error metrics were calculated.

Table 2

*Performance metrics across all regions*

| | GConvGRU | GConvLSTM | ARIMA |
|---|---|---|---|
| Train MSE | 0.2106 ± 0.0042 | 0.2185 ± 0.0088 | 0.2152 |
| Test MSE | 0.2338 ± 0.0074 | 0.2407 ± 0.0096 | 0.2481 |
| Train RMSE | 0.4586 ± 0.0045 | 0.4653 ± 0.0079 | 0.4639 |
| Test RMSE | 0.4839 ± 0.0075 | 0.4905 ± 0.0100 | 0.4981 |
| Train MAE | 0.3311 ± 0.0025 | 0.3357 ± 0.0048 | 0.3325 |
| Test MAE | 0.3672 ± 0.0089 | 0.3706 ± 0.0101 | 0.3609 |

The GConvGRU, GConvLSTM and ARIMA models show similar values in all performance metrics. GConvGRU achieves the best test MSE, followed by the GConvLSTM (GConvGRU 0.2338 ± 0.0074, GConvLSTM 0.2407 ± 0.0096 and ARIMA 0.2481). In addition to the MSE, the RMSE values show that GConvGRU with a test RMSE of 0.4839 ± 0.0075 performs slightly better than GConvLSTM (0.4905 ± 0.0100) and ARIMA (0.4981). This indicates that the GConvGRU model has, on average, smaller quadratic deviations between the predicted and actual values, indicating a more robust and accurate predictive ability. In addition, ARIMA has the lowest mean absolute error with a test MAE of 0.3609 compared to GConvLSTM (0.3706 ± 0.0101) and GConvGRU (0.3672 ± 0.0089), indicating that it is the most accurate in prediction on average. These results mirror the comparison at the region-level, where the models (with 4 epochs and dropout for the GCRN models) also showed similar performance values. All models show a relatively small deviation between training and testing, which suggest a moderate generalization capability. However, it should be noted that aggregated performance indicators across all regions blur the individual performances of the individual models, i.e. a model could perform very poorly in one region but not appear clearly in this indicator because it performs very well in another region. The results are now interpreted in more depth in the following discussion and placed in the context of similar studies.

# 5. Discussion

## *5.1 GCRN models on the Pedal Me dataset*

The results confirm the initial assumption that the GCRN models, in particular GConvGRU and GConvLSTM, tend to overfit the data set used. This can be seen at the region level, where the training MSE decreases continuously, indicating an increasing fit of the models to the training data, while at the same time the test MSE increases, indicating poorer generalization to unseen data (Xue, 2019). The learning curves for all regions also show an increasing discrepancy between training and test errors with longer training times. According to Xue (2019) and Alzubaidi et al. (2023), overfitting is a common problem in machine learning models (especially in deep learning) and is usually caused by a combination of several factors. These include the complexity of the model, the limited size of the dataset, and the presence of noise in the training data that cannot be explained by the features (Xue, 2019).

These factors are important in the context of this study. The relatively small size of the Pedal Me dataset contributes to the fact that the models can only learn a limited number of patterns due to the small number of training examples. This limits their ability to generalize to unseen data. The noise that disrupts the limited patterns due to unpredictable fluctuations is most likely due to external factors such as the weather, holidays or major events that could influence the behavior of customers using Pedal Me's transport services (Chen et al., 2021). However, these cannot be adequately taken into account due to the lack of corresponding characteristics in the dataset, as only the historical data is available in the form of lagged characteristics. The increased complexity of the GCRN models can exacerbate the problem, as the numerous parameters are optimized for the limited and noisy sample data.

As similar patterns and increasing standard deviation of the average test MSE across the selected regions occur with increasing number of epochs, there may be a correlation between these phenomena and overfitting. When overfitting, GCRN models may tend to replicate dominant global trends to individual regions in training, ignoring the unique patterns per region, which could lead to instability in predictions on unseen data in the regions. This could be due to inefficient message passing between regions during overfitting and should be further investigated in future research.

The addition of a dropout layer has been shown to improve generalization and test performance compared to configurations with the same training time without a dropout layer. However, a dropout layer with a value of 0.3 cannot completely prevent overfitting with increasing training time, as the discrepancy between training and test still drifts apart with increasing training time, as the learning curves show. Future research could therefore investigate increasing the dropout value and other regularization strategies on this dataset to see if generalizability can be further improved. For example, L2 regularization could help by assigning larger weights only to those features that significantly improve the loss function, while assigning smaller weights to less informative features with noise (Xue, 2019). In addition, reducing complexity can also improve the ability to generalize to unseen data (Xue, 2019). In this case, for example, the peephole connections of the FC-LSTM could be removed to reduce the number of parameters. The results also highlight the need to continuously monitor the overfitting of the GCRN models for predicting bike transport demand on the Pedal Me dataset throughout training, and to implement mechanisms for early-stopping when test performance gradually deteriorates (Xue, 2019). Future research could also look more closely at transfer learning. Transfer learning allows one to use pre-trained models for similar prediction problems and refine them for the specific Pedal Me dataset (Safonova et al., 2023). For example, future research could use a demand prediction model of the ride-sharing service Uber and refine it to the specific dataset of Pedal Me to test whether the demand for bike transport services in London can be better predicted for Pedal Me.

Nevertheless, the results show that, apart from overfitting with increasing training time, the GCRN models are generally able to adapt to the demand values in selected regions. They also show some generalization on this small dataset when the epochs are kept small. This adaptability is evidenced by the fact that the models maintain a relatively moderate deviation between training and test performance over 4 epochs. Furthermore, the visualizations show that the models fit the patterns of the target variables in the three selected regions of different volatility, in some cases achieving close hits in the less volatile region. However, although they approximate the reality of demand, they also show higher deviations in the volatile regions 3 and 15. It should be noted that although the regions were selected with different standard deviations to cover a wide range of volatility in demand for Pedal Me's services across regions, they may not fully represent all other regional characteristics, which limits the generalizability of the results.

Nevertheless, it can be stated that the GCRN models are not able to significantly outperform the baseline ARIMA, as illustrated by the performance metrics at both regional and global levels. In other studies, dealing with similar spatio-temporal forecasting problems, the results cannot be directly compared with this study due to modified GCRN architectures, data sets and methods. However, they show a clear trend that GCRN models predominantly outperform conventional time series models such as ARIMA in traffic forecasting. For example, Chai et al. (2018) used a multi-graph convolutional approach combining GCN and LSTM to forecast the demand for shared bikes in New York City. Their model outperformed ARIMA by an average of 51% in terms of RMSE per sharing station. The dataset included 49.7 million trip records from July 2013 to September 2017 and included characteristics such as temperature, day of the week and wind speed, as well as arrival and departure data from 827 stations. In addition, Zhao et al. (2020) developed the T-GCN model, which combines GCNs to capture spatial dependencies and GRUs to capture temporal dependencies to predict traffic speed. The model was applied to the SZ-Taxi dataset, which contains taxi trajectories in Shenzhen (China) in January 2015 at 15-minute intervals for 156 major roads, and the Los Loop dataset from Los Angeles (United States of America), which contains traffic data from 207 sensors from 1 to 7 March 2012 at 5-minute intervals. Compared to ARIMA, T-GCN performed 47.5% better on the SZ-Taxi dataset and 51% better on the Los Loop dataset.

Comparisons with these studies suggest that the limitations of the Pedal Me dataset, with its limited size and lack of features, do not only contribute to overfitting. However, it is also clear that the limitations of the dataset prevent the GCRN models from fully exploiting their ability to effectively integrate spatial and temporal dimensions. This is because the dataset restricts model training to too little and insufficient information, thereby compromising their ability to fully capture complex, dynamic spatio-temporal patterns of bike transport demand. Although efforts were made to extend the dataset to include additional contextual features, this was not possible due to a lack of information on the geographical coordinates of the respective regions as well as specific date information, further emphasizing the limitations of the dataset. These limitations probably meant that the GCRN models in their current configuration could not outperform the baseline ARIMA model. This highlights the need for larger and more comprehensive datasets, in line with the literature on the data size requirements of deep learning models, in order to realize the full potential of GCRN models for spatio-temporal prediction tasks (Alzubaidi et al., 2023).

*5.2 ARIMA on the Pedal Me dataset*

However, the regional analysis for selected regions also shows that less complex ARIMA, as a baseline model, is in principle also able to detect patterns in the underlying data by iteratively testing different parameter combinations (p, d, q) to achieve the best test performance. However, Alghamdi et al. (2019) point out in their study that the size of the dataset is also crucial for the performance of ARIMA, as in smaller datasets it is difficult to distinguish between normal and abnormal traffic patterns due to the lack of distinguishable features, further highlighting the limitation of the Pedal Me dataset. Future research could therefore investigate whether ARIMAX, as an extension of traditional ARIMA, offers additional benefits by including explanatory variables such as weather and socio economic data to better predict strong variations, even when the size of the dataset is limited (Chen et al., 2021). From the current point of view, ARIMA remains the less problematic choice over the GCRN models due to its low complexity, even if the performance is similar. A decision support system for Pedal should therefore currently use the ARIMA model, for example to allocate resources such as drivers and bikes appropriately to the regions.

*5.3 Comparison of GCRN models*

Looking more closely at the architecture of the two GCRN models, GConvLSTM differs by using a cell state as a long-term memory element, which is controlled by an additional output gate to regulate the forwarding of memory contents (Chung et al., 2014). In contrast, GConvGRU does not have a separate cell state and releases its entire memory content without additional control by controlling the information flow of the previous activation when computing the new activation (Chung et al., 2014). However, these architectural differences appear to have a limited impact on model performance in the context of bike transport demand prediction and the small Pedal Me dataset. A direct comparison of the two GCRN models does not show a clear advantage of one model over the other, as evidenced by the similar regional and global performance metrics, as well as the comparable learning curves across different epochs. However, the error metrics and tools used in this study, such as learning curves, do not allow a deeper interpretation of the internal mechanisms of the backpropagation algorithm. The mechanisms of pattern recognition in sequential data remain opaque, and therefore a deeper understanding of these algorithms is difficult to achieve with current evaluation methods. It is therefore useful to consider additional metrics to the traditional performance metrics in further studies. Connectivity, for example, measures how these models remember their inputs (Madsen,

2019). Connectivity is already used in language processing and assesses how much changes in inputs affect the model's outputs by calculating the strength of the connection between inputs at one point in time and outputs at another point in time. Assessing the gradients between inputs and desired outputs helps to understand how well the model is able to use relevant information from the past for future predictions, and could provide insights into the model's memory and context processing. Such insights could improve the selection and adaptation of GCRN architectures for different spatio-temporal forecasting problems by providing a clearer picture of how GConvLSTM and GConvGRU process and remember relevant patterns and what differences in their functioning are important for different scenarios.

## 6. Conclusion

In this study, different Graph Convolutional Recurrent Networks (GConvGRU and GConvLSTM) were compared with the baseline model ARIMA for predicting the bike transport demand of the company Pedal Me. The results showed that GCRN models with very short training times are generally able to maintain a balance between training and testing on the limited Pedal Me dataset and to detect patterns in selected regions. However, they also show larger deviations in the selected volatile regions. Furthermore, the models overfit after a short time due to their own complexity, as well as the size and lack of features of the dataset, leading to poor generalization to unseen data. Overfitting could be related to phenomena such as variability in test performance, as well as increasingly similar prediction patterns across multiple regions, which need to be further investigated in future work. Dropout was found to be helpful in limiting overfitting to some extent. Therefore, further research could consider the use of transfer learning, as well as other regularization techniques such as L2 regularization, early stopping and complexity reduction of the models, to test whether the generalization ability can be further improved on this dataset. Future studies should also consider additional metrics such as connectivity alongside traditional performance metrics to gain a better understanding of the memory and context processing of these models for spatio-temporal prediction tasks.

A comparison with similar studies shows that the Pedal Me data set not only favors overfitting in the GCRN models. It also shows that these models could not outperform the ARIMA baseline model due to the limited size and lack of features of the dataset. This illustrates that the

GCRN models in their current form require larger and more comprehensive data sets to realize their full potential. The use of cross-validation would generally have provided more robust results, but was not practical due to the small amount of data, as this would have increased overfitting by categorizing into smaller folds. Although this methodological choice improves the comparability of the different models by limiting overfitting in a controlled manner, no model could be clearly identified as the best model for the Pedal Me decision support system based on the performance metrics. Nevertheless, due to its low complexity, ARIMA remains the less problematic choice, even if the performance is similar. Pedal Me should therefore currently use the ARIMA model for a decision support system to optimize resource allocation.

**References**

Alghamdi, T., Elgazzar, K., Bayoumi, M., Sharaf, T., & Shah, S. (2019). Forecasting Traffic Congestion Using ARIMA Modeling. *International Wireless Communications and Mobile Computing Conference, IWCMC*. https://doi.org/10.1109/iwcmc.2019.8766698

Alzubaidi, L., Bai, J., Al-Sabaawi, A., Santamaría, J., Albahri, A. S., Al-Dabbagh, B. S. N., Fadhel, M. A., Manoufali, M., Zhang, J., Al-Timemy, A. H., Duan, Y., Abdullah, A., Farhan, L., Lu, Y., Gupta, A., Albu, F., Abbosh, A., & Gu, Y. (2023). A survey on deep learning tools dealing with data scarcity: definitions, challenges, solutions, tips, and applications. *Journal of Big Data*, *10*(1). https://doi.org/10.1186/s40537-023-00727-2

Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, *5*(2), 157–166. https://doi.org/10.1109/72.279181

Chai, D., Wang, L., & Yang, Q. (2018). Bike Flow Prediction with Multi-Graph Convolutional Networks. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.1807.10934

Chen, L., Thakuriah, P., & Ampountolas, K. (2021). Short-Term Prediction of demand for Ride-Hailing Services: A deep learning approach. *Journal of Big Data Analytics in Transportation*, *3*(2), 175–195. https://doi.org/10.1007/s42421-021-00041-4

Chen, X., & Chen, R. (2019). A Review on Traffic Prediction Methods for Intelligent Transportation System in Smart Cities. *12th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI)*. https://doi.org/10.1109/cisp-bmei48845.2019.8965742

Cho, K., Van Merrienboer, B., Bahdanau, D., & Bengio, Y. (2014). On the Properties of Neural Machine Translation: Encoder-Decoder Approaches. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.1409.1259

Chung, F. (1996). *Spectral graph Theory*. https://mathweb.ucsd.edu/~fan/research/revised.html

Chung, J., Gülçehre, Ç., Cho, K., & Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv (Cornell University)*. https://arxiv.org/pdf/1412.3555

Colah, C. (2015, August). *Understanding LSTM Networks*. Colah's Blog. https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Daigavane, A., Ravindran, B., & Aggarwal, G. (2021). Understanding convolutions on graphs. *Distill*, *6*(8). https://doi.org/10.23915/distill.00032

Defferrard, M., Bresson, X., & Vandergheynst, P. (2016). Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.1606.09375

Dey, R., & Salemt, F. M. (2017). Gate-variants of Gated Recurrent Unit (GRU) neural networks. *IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*. https://doi.org/10.1109/mwscas.2017.8053243

European Commission. (2010, July). *Directive 2010/40/EU of the European Parliament and of the Council on the Framework for the deployment of Intelligent Transport Systems in the field of Road Transport and for interfaces with other modes of transport*. https://transport.ec.europa.eu/. Retrieved March 28, 2024, from https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32010L0040

Fu, R., Zuo, Z., & Li, L. (2016). Using LSTM and GRU neural network methods for traffic flow prediction. *2016 31st Youth Academic Annual Conference of Chinese Association of Automation (YAC)*. https://doi.org/10.1109/yac.2016.7804912

Gers, F., & Schmidhuber, J. (2000). Recurrent nets that time and count. *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks*. https://doi.org/10.1109/ijcnn.2000.861302

Gilpin, L. H., Bau, D., Yuan, B. Z., Bajwa, A., Specter, M., & Kagal, L. (2018). Explaining Explanations: An Overview of Interpretability of Machine Learning. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.1806.00069

Gruslys, A., Munos, R., & Danihelka, I. (2019). Memory-Efficient Backpropagation Through Time. *Advances in Neural Information Processing Systems*, *29*. https://proceedings.neurips.cc/paper/2016/hash/a501bebf79d570651ff601788ea9d16d-Abstract.html

Haykin, S. (1998). *Neural Networks: a comprehensive foundation*. http://www.cis.hut.fi/Opinnot/T-61.3030/luennot2007/lect1.pdf

Hochreiter, S. (1998). The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, *06*(02), 107–116. https://doi.org/10.1142/s0218488598000094

Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term memory. *Neural Computation*, *9*(8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

Jiang, J., Han, C., Jiang, W., Zhao, W. X., & Wang, J. (2023). LibCitY: a unified library towards Efficient and Comprehensive Urban Spatial-Temporal Prediction. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.2304.14343

Jiang, W., & Luo, J. (2022). Graph neural network for traffic forecasting: A survey. *Expert Systems With Applications*, *207*, 117921. https://doi.org/10.1016/j.eswa.2022.117921

Jiber, M., Lamouik, I., Yahyaouy, A., & Sabri, M. A. (2018). Traffic flow prediction using neural network. *2018 International Conference on Intelligent Systems and Computer Vision (ISCV)*. https://doi.org/10.1109/isacv.2018.8354066

Kingma, D. P., & Ba, J. L. (2014). Adam: A method for stochastic optimization. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.1412.6980

Kipf, T. N., & Welling, M. (2016, September 9). *Semi-Supervised Classification with Graph Convolutional Networks*. arXiv.org. https://arxiv.org/abs/1609.02907

Lan, R. (2020). Region-level Ride-hailing Demand Prediction with Deep Learning. *Journal of Physics. Conference Series*, *1678*(1), 012111. https://doi.org/10.1088/1742-6596/1678/1/012111

Li, Y., Zhou, X., & Pan, M. (2022). Graph Neural Networks in Urban Intelligence. In *Graph Neural Networks: Foundations, Frontiers, and Applications* (pp. 579–593). https://doi.org/10.1007/978-981-16-6054-2_27

Liang, Y., Li, S., Yan, C., Li, M., & Jiang, C. (2021). Explaining the black-box model: A survey of local interpretation methods for deep neural networks. *Neurocomputing*, *419*, 168–182. https://doi.org/10.1016/j.neucom.2020.08.011

Lozić, J. (2016). Resource-efficient intelligent transportation systems as a basis for sustainable development. Overview of initiatives and strategies. *Journal of Sustainable Development of Transport and Logistics*, *1*(1), 6–10. https://doi.org/10.14254/jsdtl.2016.1-1.1

Madsen, A. (2019). Visualizing memorization in RNNs. *Distill*, *4*(3). https://doi.org/10.23915/distill.00016

Manibardo, E. L., Laña, I., & Del Ser, J. (2020). Transfer Learning and Online Learning for Traffic Forecasting under Different Data Availability Conditions: Alternatives and Pitfalls. *23rd International Conference on Intelligent Transportation Systems (ITSC)*. https://doi.org/10.1109/itsc45102.2020.9294557

Nicholson, C. V. (2019). *A Beginner's Guide to LSTMs and recurrent neural networks*. Pathmind. Retrieved March 3, 2024, from https://wiki.pathmind.com/lstm

Nielsen, M. (2019). *Neural Networks and Deep Learning* (1st ed.). http://neuralnetworksanddeeplearning.com/

Noh, S. (2021). Analysis of gradient vanishing of RNNs and performance comparison. *Information*, *12*(11), 442. https://doi.org/10.3390/info12110442

Poonia, P., & Jain, V. K. (2020). Short-Term Traffic Flow Prediction: Using LSTM. *2020 International Conference on Emerging Trends in Communication, Control and Computing (ICONC3)*. https://doi.org/10.1109/iconc345789.2020.9117329

Qureshi, K. N., & Abdullah, A. H. (2013). A survey on intelligent transportation systems. *Middle-East Journal of Scientific Research*, *15*(5), 629–642. https://www.researchgate.net/publication/257367335_A_Survey_on_Intelligent_Transportation_Systems

Rossi, E., Chamberlain, B., Frasca, F., Eynard, D., Monti, F., & Bronstein, M. M. (2020). Temporal Graph networks for deep learning on dynamic graphs. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.2006.10637

Rozemberczki, B., Scherer, P., He, Y., Panagopoulos, G., Riedel, A., Astefanoaei, M., Kiss, O., Beres, F., López, G., Collignon, N., & Sarkar, R. (2021). PyTorch Geometric Temporal: Spatiotemporal Signal Processing with Neural Machine Learning Models. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.2104.07788

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533–536. https://doi.org/10.1038/323533a0

Safonova, A., Ghazaryan, G., Stiller, S., Main-Knorn, M., Nendel, C., & Ryo, M. (2023). Ten deep learning techniques to address small data problems with remote sensing. *International Journal of Applied Earth Observation and Geoinformation*, *125*, 103569. https://doi.org/10.1016/j.jag.2023.103569

Salehinejad, H., Baarbé, J., Sankar, S., Barfett, J., Colak, E., & Valaee, S. (2017). Recent advances in recurrent neural networks. *arXiv (Cornell University)*. http://export.arxiv.org/pdf/1801.01078

Sánchez-Lengeling, B., Reif, E., Pearce, A., & Wiltschko, A. B. (2021). A gentle introduction to graph neural networks. *Distill*, *6*(8). https://doi.org/10.23915/distill.00033

Schmidt, R. M. (2019). Recurrent Neural Networks (RNNs): A gentle Introduction and Overview. *arXiv (Cornell University)*. https://arxiv.org/pdf/1912.05911.pdf

Seo, Y., Defferrard, M., Vandergheynst, P., & Bresson, X. (2016). Structured Sequence Modeling with Graph Convolutional Recurrent Networks. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.1612.07659

Shaheen, S., & Finson, R. (2013). *Intelligent Transportation Systems*. Open Access Publications From the University of California. https://escholarship.org/uc/item/3hh2t4f9

Song, X., Chen, K., Li, X., Sun, J., Hou, B., Cui, Y., Zhang, B., Xiong, G., & Wang, Z. (2021). Pedestrian trajectory prediction based on Deep Convolutional LSTM network. *IEEE Transactions on Intelligent Transportation Systems*, *22*(6), 3285–3302. https://doi.org/10.1109/tits.2020.2981118

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research 15*, *15*(1), 1929–1958. https://jmlr.csail.mit.edu/papers/volume15/srivastava14a/srivastava14a.pdf

Wang, X., Ma, Y., Wang, Y., Jin, W., Wang, X., Tang, J., Jia, C., & Yu, J. (2020). Traffic Flow Prediction via Spatial Temporal Graph Neural Network. *International World Wide Web Conference Committee*. https://doi.org/10.1145/3366423.3380186

Werbos, P. J. (1990). Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, *78*(10), 1550–1560. https://doi.org/10.1109/5.58337

Wu, L., Cui, P., Pei, J., Zhao, L., & Guo, X. (2022). Graph Neural Networks: foundation, frontiers and applications. *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. https://doi.org/10.1145/3534678.3542609

Xie, B., Wang, M., & Tao, D. (2011). Toward the optimization of normalized graph LaPlacian. *IEEE Transactions on Neural Networks*, *22*(4), 660–666. https://doi.org/10.1109/tnn.2011.2107919

Xue, Y. (2019). An Overview of Overfitting and its Solutions. *Journal of Physics. Conference Series*, *1168*, 022022. https://doi.org/10.1088/1742-6596/1168/2/022022

Yamak, P. T., Li, Y., & Gadosey, P. K. (2019). A Comparison between ARIMA, LSTM, and GRU for Time Series Forecasting. *ACAI '19: Proceedings of the 2019 2nd International Conference on Algorithms, Computing and Artificial Intelligence*. https://doi.org/10.1145/3377713.3377722

Yang, S., Yu, X., & Zhou, Y. (2020). LSTM and GRU Neural Network Performance Comparison Study: Taking Yelp Review Dataset as an Example. *IEEE International*

*Workshop on Electronic Communication and Artificial Intelligence (IWECAI)*. https://doi.org/10.1109/iwecai50956.2020.00027

Ye, J., Zhao, J., Ye, K., & Xu, C. (2022). How to build a Graph-Based Deep Learning Architecture in traffic Domain: a survey. *IEEE Transactions on Intelligent Transportation Systems*, *23*(5), 3904–3924. https://doi.org/10.1109/tits.2020.3043250

Yu, Y., Si, X., Hu, C., & Zhang, J. (2019). A review of Recurrent Neural networks: LSTM cells and network architectures. *Neural Computation*, *31*(7), 1235–1270. https://doi.org/10.1162/neco_a_01199

Zhao, L., Song, Y., Zhang, C., Liu, Y., Wang, P., Lin, T., Deng, M., & Li, H. (2020). T-GCN: A Temporal graph Convolutional Network for traffic Prediction. *IEEE Transactions on Intelligent Transportation Systems*, *21*(9), 3848–3858. https://doi.org/10.1109/tits.2019.2935152

Zhao, Z., Chen, W., Wu, X., Chen, P. C. Y., & Liu, J. (2017). LSTM network: a deep learning approach for short-term traffic forecast. *IET Intelligent Transport Systems*, *11*(2), 68–75. https://doi.org/10.1049/iet-its.2016.0208

Zhou, J., Cui, G., Hu, S., Zhang, Z., Yang, C., Liu, Z., Wang, L., Li, C., & Sun, M. (2021). Graph Neural Networks: A Review of Methods and applications. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.1812.08434

# Appendix

**Appendix A:** Backpropagation through time and the vanishing gradient problem

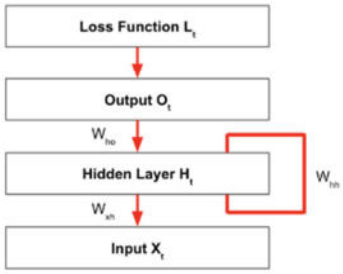
$$\frac{\partial L}{\partial W_{ho}} = \sum_{t=1}^{T} \frac{\partial l_t}{\partial O_t} \cdot \frac{\partial O_t}{\partial \phi_o} \cdot \frac{\partial \phi_o}{\partial W_{ho}} = \sum_{t=1}^{T} \frac{\partial l_t}{\partial O_t} \cdot \frac{\partial O_t}{\partial \phi_o} \cdot H_t$$

$$\frac{\partial L}{\partial W_{hh}} = \sum_{t=1}^{T} \frac{\partial l_t}{\partial O_t} \cdot \frac{\partial O_t}{\partial \phi_o} \cdot \frac{\partial \phi_o}{\partial H_t} \cdot \frac{\partial H_t}{\partial \phi_h} \cdot \frac{\partial \phi_h}{\partial W_{hh}} = \sum_{t=1}^{T} \frac{\partial l_t}{\partial O_t} \cdot \frac{\partial O_t}{\partial \phi_o} \cdot W_{ho} \cdot \frac{\partial H_t}{\partial \phi_h} \cdot \frac{\partial \phi_h}{\partial W_{hh}}$$

$$\frac{\partial L}{\partial W_{xh}} = \sum_{t=1}^{T} \frac{\partial l_t}{\partial O_t} \cdot \frac{\partial O_t}{\partial \phi_o} \cdot \frac{\partial \phi_o}{\partial H_t} \cdot \frac{\partial H_t}{\partial \phi_h} \cdot \frac{\partial \phi_h}{\partial W_{xh}} = \sum_{t=1}^{T} \frac{\partial l_t}{\partial O_t} \cdot \frac{\partial O_t}{\partial \phi_o} \cdot W_{ho} \cdot \frac{\partial H_t}{\partial \phi_h} \cdot \frac{\partial \phi_h}{\partial W_{xh}}$$

$$\frac{\partial L}{\partial W_{hh}} = \sum_{t=1}^{T} \frac{\partial l_t}{\partial O_t} \cdot \frac{\partial O_t}{\partial \phi_o} \cdot W_{ho} \cdot \sum_{k=1}^{t} \frac{\partial H_t}{\partial H_k} \cdot \frac{\partial H_k}{\partial W_{hh}}$$

$$\frac{\partial L}{\partial W_{xh}} = \sum_{t=1}^{T} \frac{\partial l_t}{\partial O_t} \cdot \frac{\partial O_t}{\partial \phi_o} \cdot W_{ho} \cdot \sum_{k=1}^{t} \frac{\partial H_t}{\partial H_k} \cdot \frac{\partial H_k}{\partial W_{xh}}$$



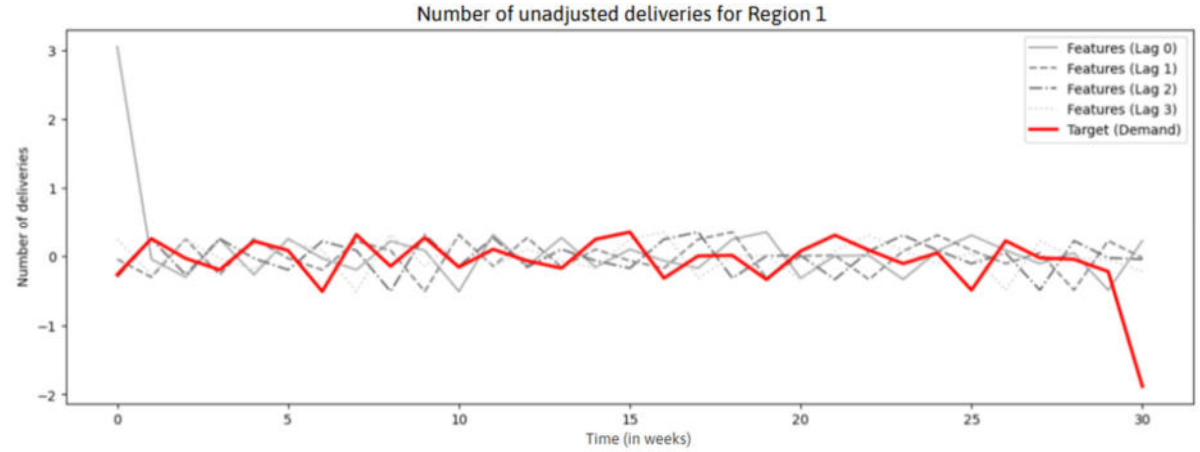

**Appendix B:** Installation code for PyTorch Geometric and PyTorch Geometric Temporal as well as other libraries

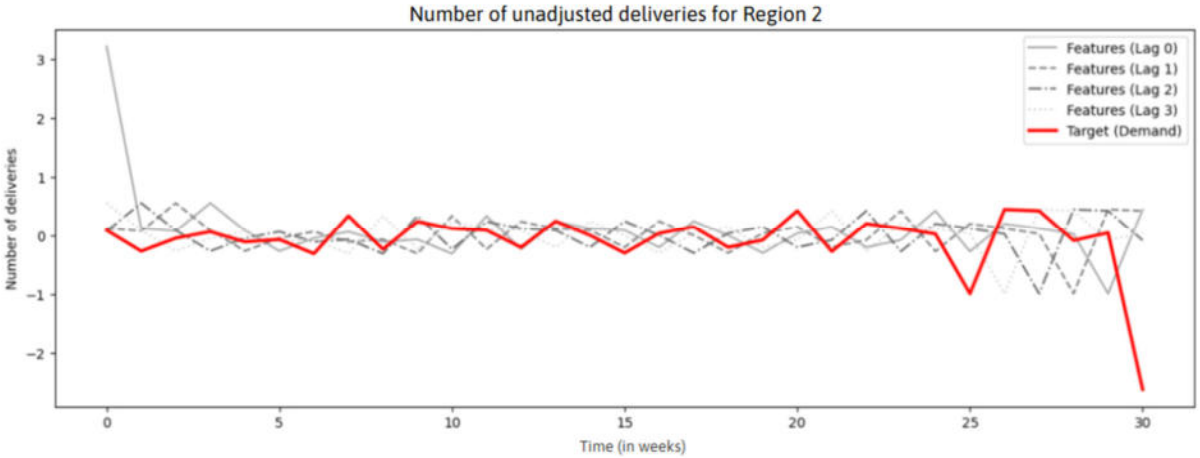
```
############### Installing Pytorch Geometric ###############

!pip install torch_geometric
import torch
import torch_geometric

print("PyTorch Version:", torch.__version__)
print("PyTorch Geometric Version:", torch_geometric.__version__)
#PyTorch Version: 2.2.1+cu121
#PyTorch Geometric Version: 2.5.2

############ Installing Pytorch Geometric Temporal ############

! pip install torch-geometric-temporal
! pip freeze | grep torch-geometric-temporal
#Result: torch-geometric-temporal==0.54.0


############ Updating Import Path in tsagcn.py ###############
```

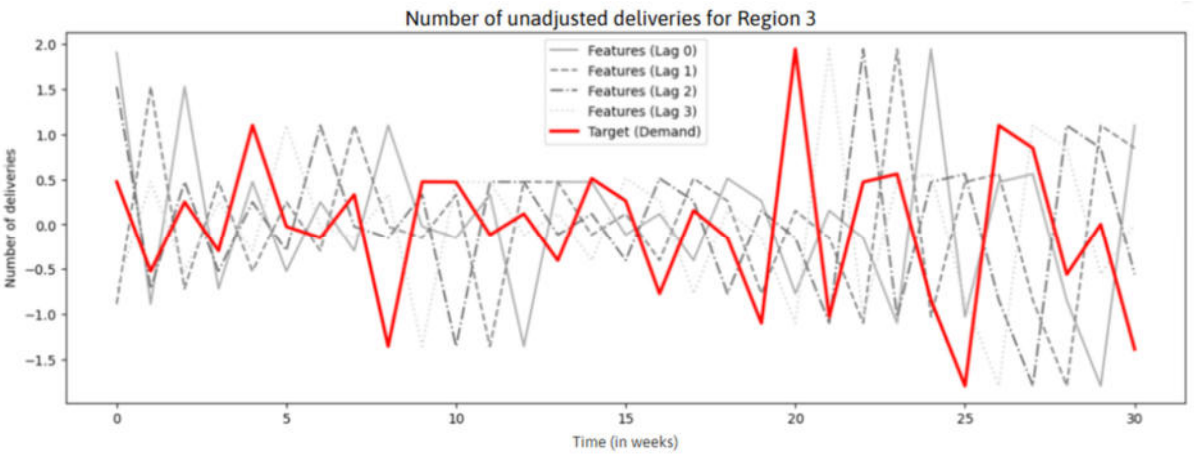
```
# Note: This script updates the import path in 'tsagcn.py' for compati-
bility with the latest 'torch_geometric' library.
# It creates a backup of the original file before replacing the old im-
port statement with the new one.

# The path to the file
file_path = '/usr/local/lib/python3.10/dist-packages/torch_geomet-
ric_temporal/nn/attention/tsagcn.py'

# Create a backup file
!cp {file_path} {file_path}.bak

# Reads the contents of the file
with open(file_path, 'r') as file:
    content = file.readlines()

# Change the specific row
content = [line.replace("from torch_geometric.utils.to_dense_adj import
to_dense_adj",
                        "from torch_geometric.utils import
to_dense_adj") for line in content]

# Write the changed content back into the file
with open(file_path, 'w') as file:
    file.writelines(content)

######################Installing all other libraries ###############

#!pip install pmdarima
#!pip install matplotlib
import numpy as np
import pandas as pd
from pmdarima import auto_arima
from sklearn.metrics import mean_squared_error, mean_absolute_error
from statsmodels.tsa.stattools import adfuller
from tqdm import tqdm
from torch_geometric_temporal.dataset import PedalMeDatasetLoader
from torch_geometric_temporal.signal import temporal_signal_split
import json
import urllib.request
from torch_geometric_temporal.signal import StaticGraphTemporalSignal
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
import time
import torch
import torch.nn as nn
import torch.nn.functional as F
```

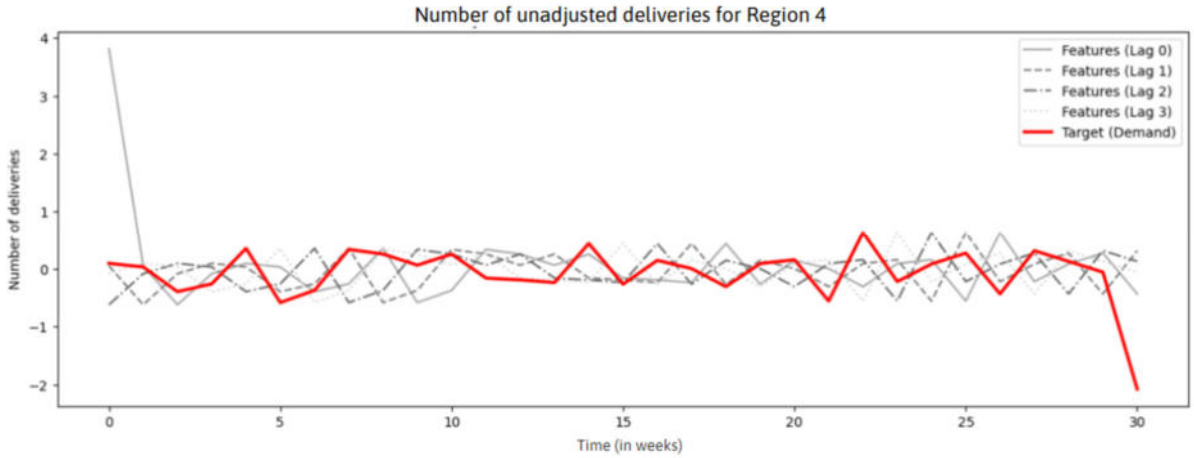
```
from torch_geometric_temporal.nn.recurrent import GConvGRU
from torch_geometric_temporal.nn.recurrent import GConvLSTM
import matplotlib
import numpy as np
import pandas as pd
import seaborn as sns
```

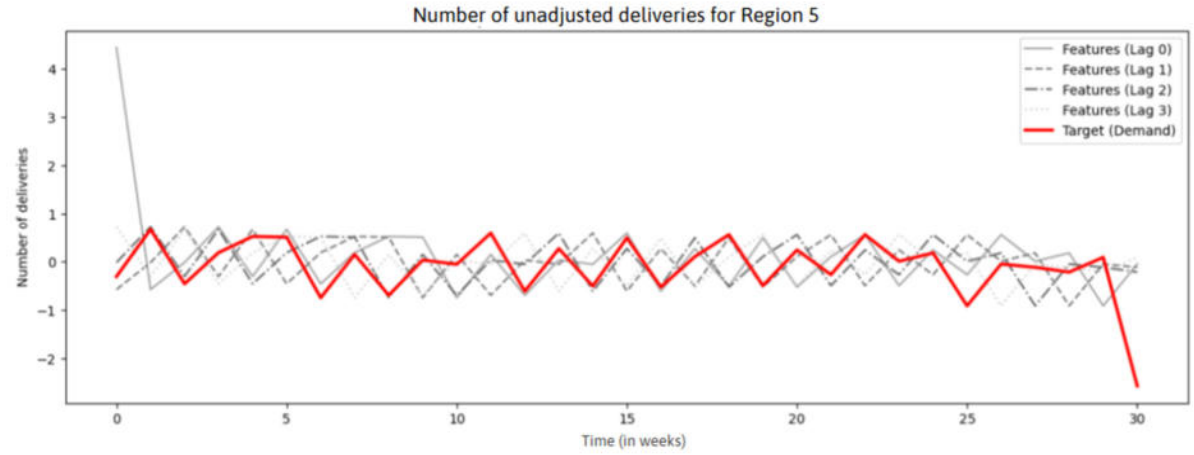
**Appendix C:** Unadjusted time series with target (demand) and features (lagged demand) for each region

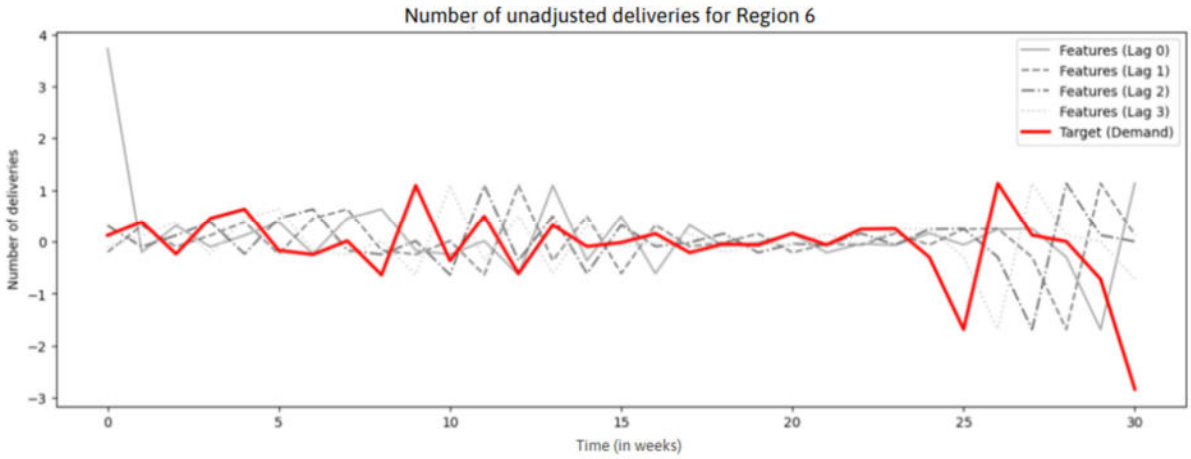


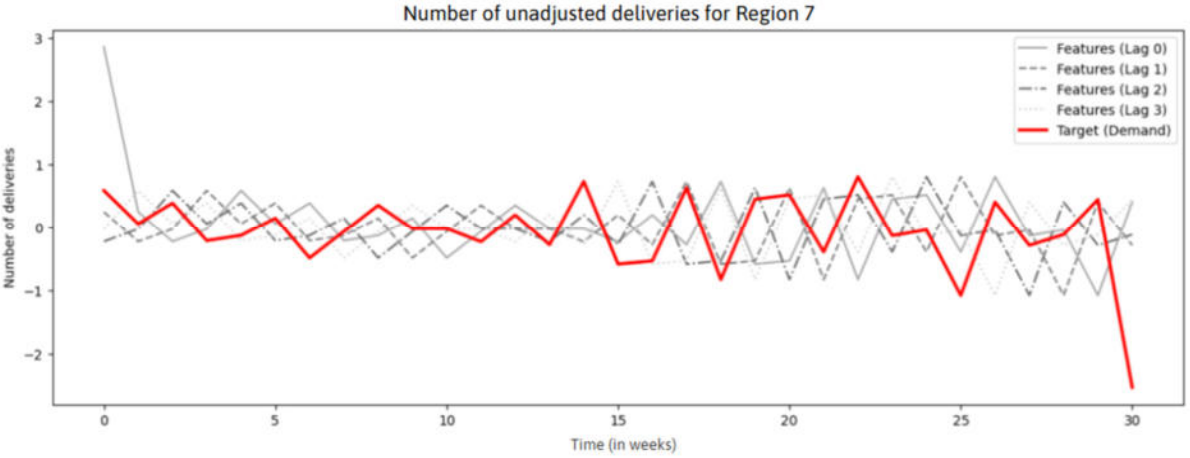



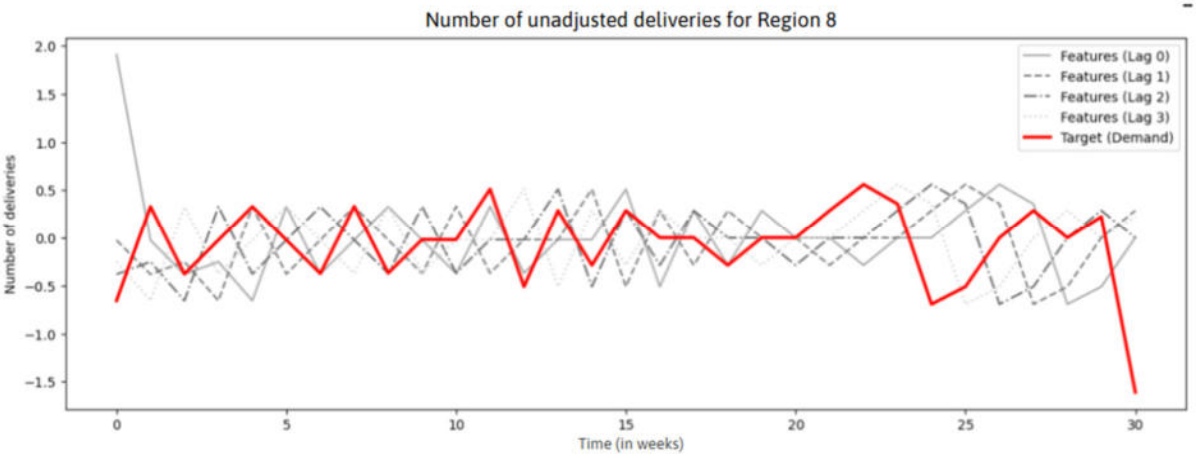

Number of unadjusted deliveries for Region 3
Features (Lag 0)
Features (Lag 1)
Features (Lag 2)
Features (Lag 3)
Target (Demand)
Number of deliveries
Time (in weeks)

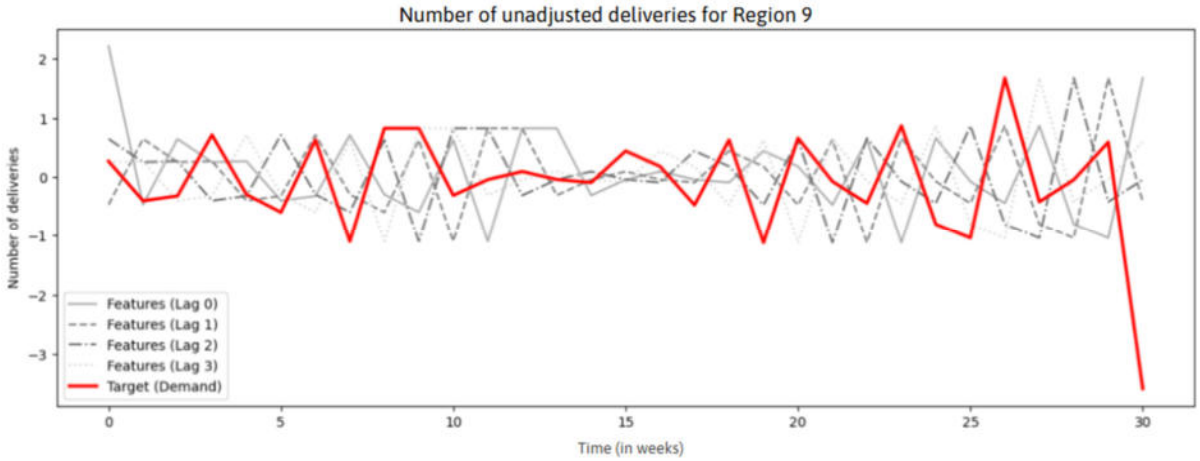

Number of unadjusted deliveries for Region 4
Features (Lag 0)
Features (Lag 1)
Features (Lag 2)
Features (Lag 3)
Target (Demand)
Number of deliveries
Time (in weeks)

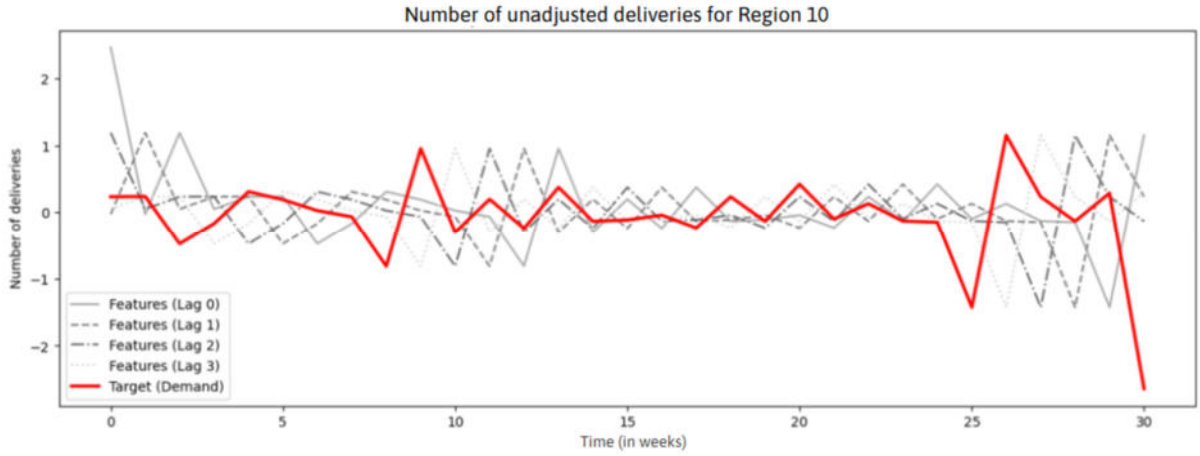

Number of unadjusted deliveries for Region 5
Features (Lag 0)
Features (Lag 1)
Features (Lag 2)
Features (Lag 3)
Target (Demand)
Number of deliveries
Time (in weeks)

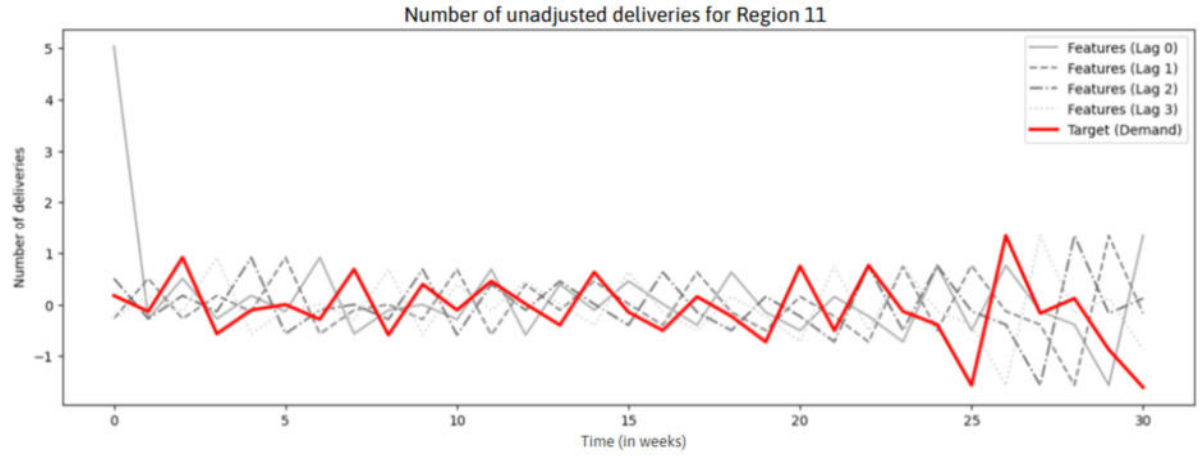

Number of unadjusted deliveries for Region 6
Features (Lag 0)
Features (Lag 1)
Features (Lag 2)
Features (Lag 3)
Target (Demand)
Number of deliveries
Time (in weeks)

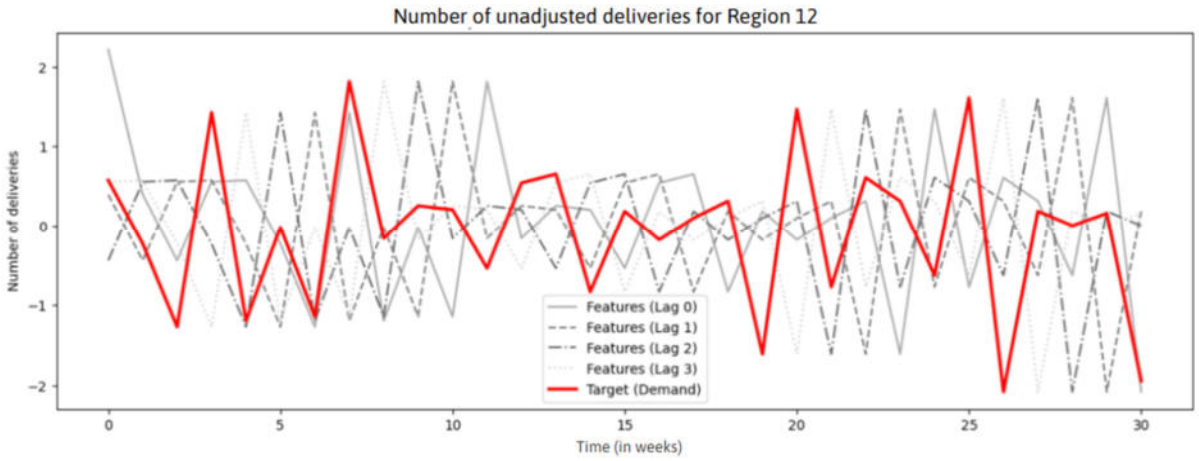

Number of unadjusted deliveries for Region 7
Features (Lag 0)
Features (Lag 1)
Features (Lag 2)
Features (Lag 3)
Target (Demand)
Number of deliveries
Time (in weeks)

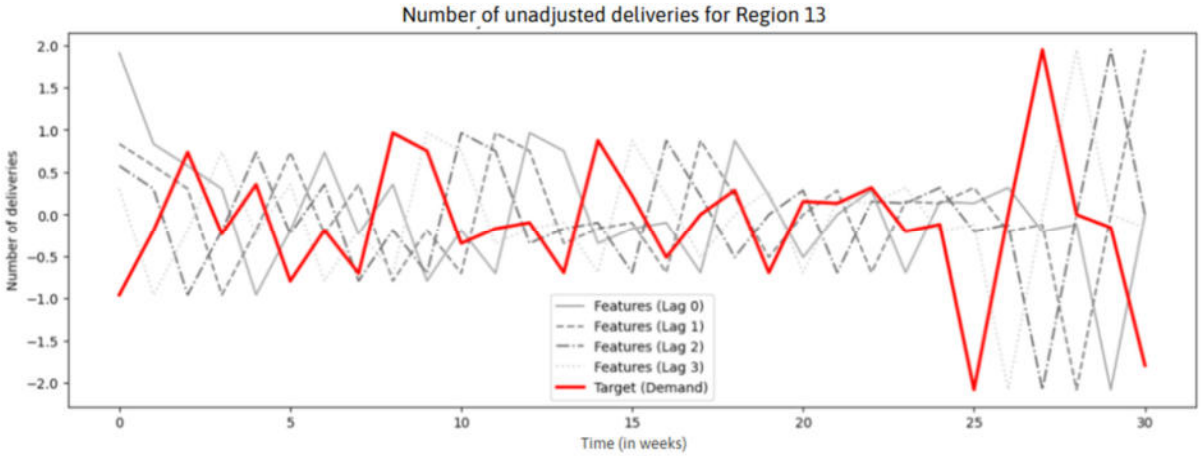

Number of unadjusted deliveries for Region 8
Features (Lag 0)
Features (Lag 1)
Features (Lag 2)
Features (Lag 3)
Target (Demand)
Number of deliveries
Time (in weeks)

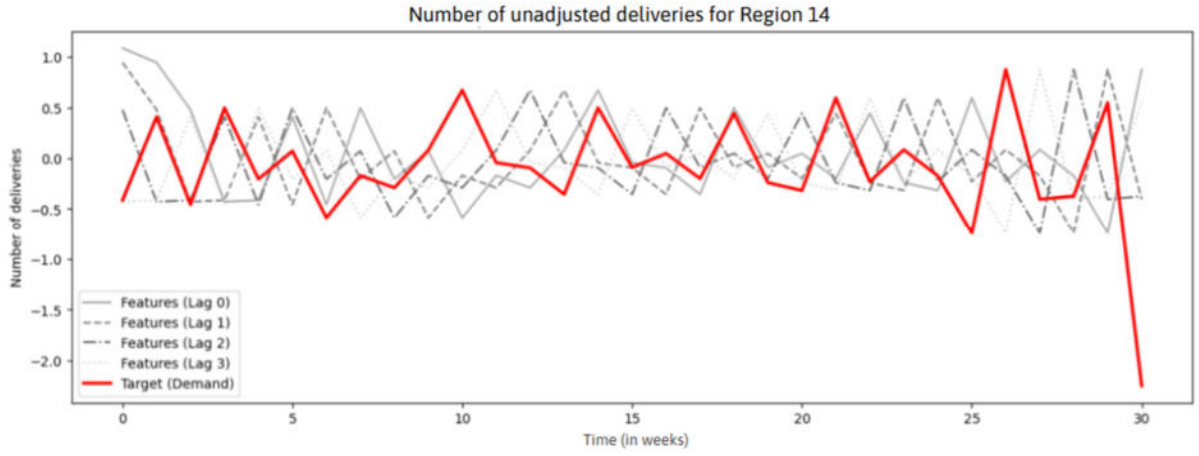

Number of unadjusted deliveries for Region 9
Number of deliveries
Time (in weeks)
Features (Lag 0)
Features (Lag 1)
Features (Lag 2)
Features (Lag 3)
Target (Demand)

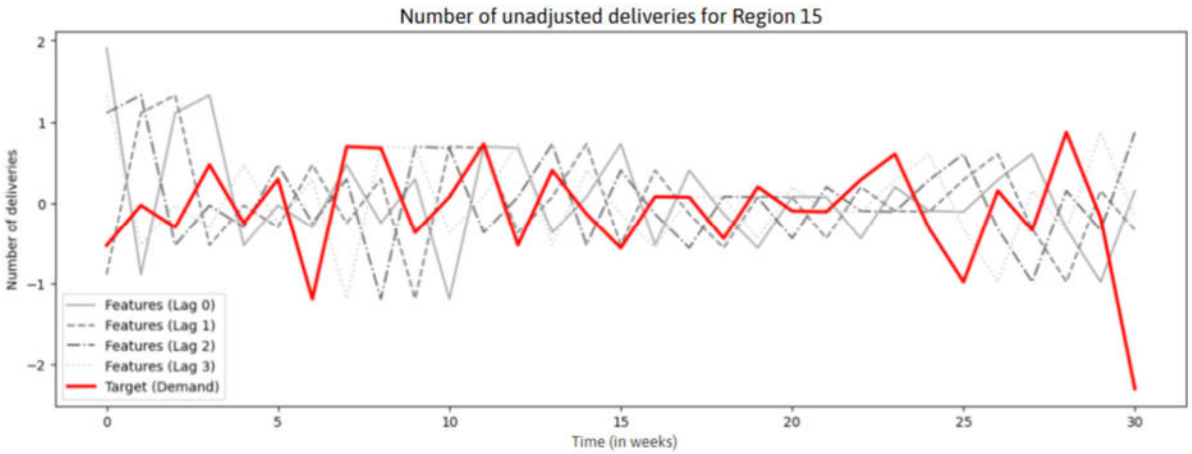

Number of unadjusted deliveries for Region 10
Number of deliveries
Time (in weeks)
Features (Lag 0)
Features (Lag 1)
Features (Lag 2)
Features (Lag 3)
Target (Demand)


Number of unadjusted deliveries for Region 11
Number of deliveries
Time (in weeks)
Features (Lag 0)
Features (Lag 1)
Features (Lag 2)
Features (Lag 3)
Target (Demand)

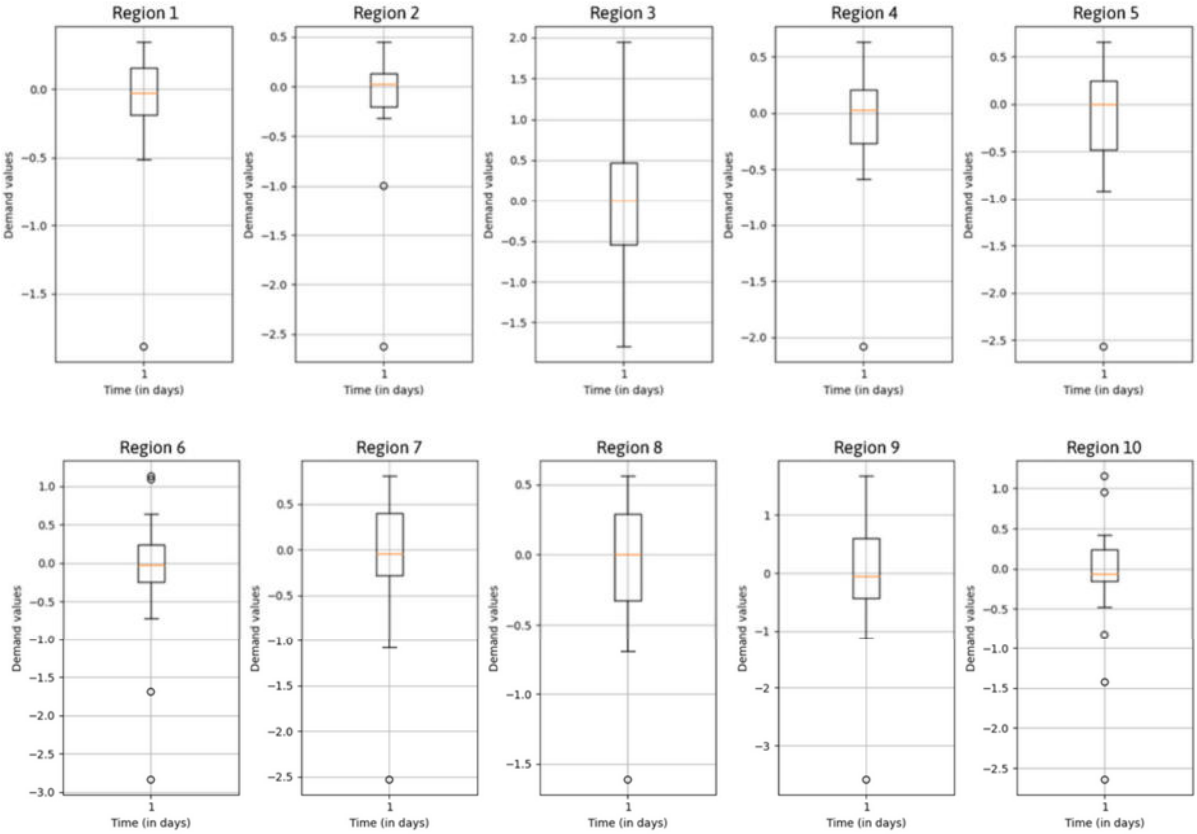

Number of unadjusted deliveries for Region 12
Number of deliveries
Time (in weeks)
Features (Lag 0)
Features (Lag 1)
Features (Lag 2)
Features (Lag 3)
Target (Demand)

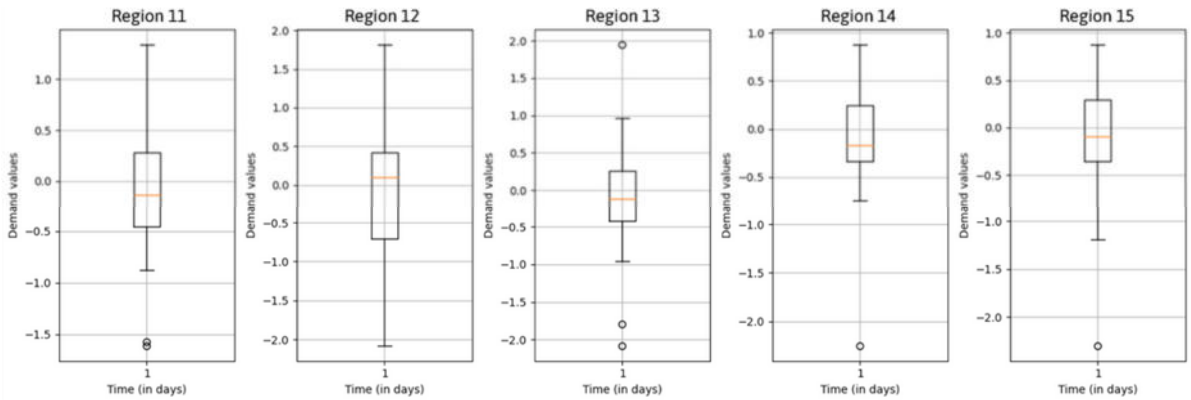

Number of unadjusted deliveries for Region 13
Number of deliveries
Time (in weeks)
Features (Lag 0)
Features (Lag 1)
Features (Lag 2)
Features (Lag 3)
Target (Demand)


Number of unadjusted deliveries for Region 14
Number of deliveries
Time (in weeks)
Features (Lag 0)
Features (Lag 1)
Features (Lag 2)
Features (Lag 3)
Target (Demand)




**Appendix D:** Unadjusted boxplot with target (demand) for each region






**Please note:** These boxplots were created before the data cleaning. The descriptives down below were created after the data cleaning

**Appendix E:** Descriptive Statistics for each region

| Region | Mean | Std | Min | 25% | 50% | 75% | Max |
|---|---|---|---|---|---|---|---|
| 1 | -0.026673 | 0.232701 | -0.513682 | -0.183332 | -0.028438 | 0.154431 | 0.345159 |
| 2 | -0.010460 | 0.285860 | -0.993252 | -0.204433 | 0.025001 | 0.134994 | 0.448950 |
| 3 | -0.019660 | 0.784824 | -1.791759 | -0.533861 | 0.000000 | 0.471908 | 1.945910 |
| 4 | -0.024031 | 0.305870 | -0.585890 | -0.263616 | 0.028881 | 0.202164 | 0.628609 |
| 5 | -0.049569 | 0.447436 | -0.916291 | -0.473778 | 0.000000 | 0.245656 | 0.655727 |
| 6 | -0.006441 | 0.529863 | -1.686399 | -0.242614 | -0.020203 | 0.241898 | 1.131402 |
| 7 | 0.000455 | 0.460751 | -1.077559 | -0.280144 | -0.041673 | 0.397109 | 0.806476 |
| 8 | -0.022415 | 0.342662 | -0.693147 | -0.308160 | 0.000000 | 0.287682 | 0.559616 |
| 9 | 0.008759 | 0.660591 | -1.122143 | -0.427482 | -0.048790 | 0.606746 | 1.673976 |
| 10 | 0.005146 | 0.460169 | -1.427116 | -0.153386 | -0.067127 | 0.233684 | 1.152680 |
| 11 | -0.050689 | 0.595501 | -1.568616 | -0.426676 | -0.131769 | 0.280720 | 1.335001 |
| 12 | -0.032368 | 0.925104 | -2.079442 | -0.664754 | 0.095310 | 0.424576 | 1.814265 |
| 13 | -0.057741 | 0.707461 | -2.079442 | -0.386021 | -0.117783 | 0.255413 | 1.945910 |
| 14 | -0.030156 | 0.408102 | -0.741937 | -0.328009 | -0.171897 | 0.245162 | 0.875469 |
| 15 | -0.033759 | 0.495257 | -1.190535 | -0.344107 | -0.095310 | 0.292510 | 0.875469 |

**Appendix F:** Adjacency Matrix of Pedal Me Dataset

Weighted Adjacency Matrix (15x15):

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1.000 | 0.425 | 0.157 | 0.207 | 0.564 | 0.519 | 0.400 | 0.027 | 0.225 | 0.501 | 0.645 | 0.302 | 0.221 | 0.291 | 0.295 |
| 1 | 0.425 | 1.000 | 0.068 | 0.262 | 0.378 | 0.304 | 0.496 | 0.013 | 0.474 | 0.215 | 0.644 | 0.141 | 0.120 | 0.237 | 0.516 |
| 2 | 0.157 | 0.068 | 1.000 | 0.058 | 0.124 | 0.182 | 0.084 | 0.130 | 0.039 | 0.314 | 0.104 | 0.273 | 0.265 | 0.085 | 0.049 |
| 3 | 0.207 | 0.262 | 0.058 | 1.000 | 0.128 | 0.299 | 0.494 | 0.019 | 0.364 | 0.144 | 0.272 | 0.066 | 0.182 | 0.069 | 0.137 |
| 4 | 0.564 | 0.378 | 0.124 | 0.128 | 1.000 | 0.293 | 0.258 | 0.018 | 0.180 | 0.361 | 0.467 | 0.347 | 0.131 | 0.516 | 0.376 |
| 5 | 0.519 | 0.304 | 0.182 | 0.299 | 0.293 | 1.000 | 0.459 | 0.041 | 0.210 | 0.479 | 0.455 | 0.209 | 0.393 | 0.152 | 0.175 |
| 6 | 0.400 | 0.496 | 0.084 | 0.494 | 0.258 | 0.459 | 1.000 | 0.020 | 0.453 | 0.243 | 0.550 | 0.122 | 0.201 | 0.140 | 0.256 |
| 7 | 0.027 | 0.013 | 0.130 | 0.019 | 0.018 | 0.041 | 0.020 | 1.000 | 0.009 | 0.050 | 0.019 | 0.036 | 0.100 | 0.011 | 0.008 |
| 8 | 0.225 | 0.474 | 0.039 | 0.364 | 0.180 | 0.210 | 0.453 | 0.009 | 1.000 | 0.120 | 0.347 | 0.069 | 0.092 | 0.114 | 0.302 |
| 9 | 0.501 | 0.215 | 0.314 | 0.144 | 0.361 | 0.479 | 0.243 | 0.050 | 0.120 | 1.000 | 0.331 | 0.421 | 0.325 | 0.209 | 0.152 |
| 10 | 0.645 | 0.644 | 0.104 | 0.272 | 0.467 | 0.455 | 0.550 | 0.019 | 0.347 | 0.331 | 1.000 | 0.199 | 0.179 | 0.254 | 0.383 |
| 11 | 0.302 | 0.141 | 0.273 | 0.066 | 0.347 | 0.209 | 0.122 | 0.036 | 0.069 | 0.421 | 0.199 | 1.000 | 0.147 | 0.295 | 0.132 |
| 12 | 0.221 | 0.120 | 0.265 | 0.182 | 0.131 | 0.393 | 0.201 | 0.100 | 0.092 | 0.325 | 0.179 | 0.147 | 1.000 | 0.071 | 0.069 |
| 13 | 0.291 | 0.237 | 0.085 | 0.069 | 0.516 | 0.152 | 0.140 | 0.011 | 0.114 | 0.209 | 0.254 | 0.295 | 0.071 | 1.000 | 0.330 |
| 14 | 0.295 | 0.516 | 0.049 | 0.137 | 0.376 | 0.175 | 0.256 | 0.008 | 0.302 | 0.152 | 0.383 | 0.132 | 0.069 | 0.330 | 1.000 |

**Appendix G:** Code for data cleaning and inspection

Please note that regions were analyzed but “station” as a variable were used because it simplified the thought process

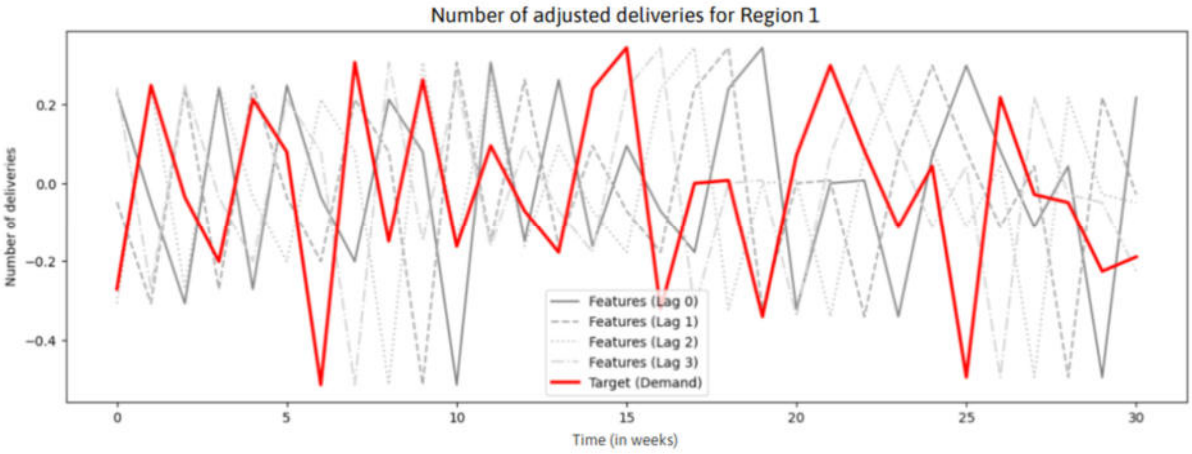
```
######################## Data inspection and cleaning ############

# Plot boxplots and calculate descriptive statistics for each station
def plot_boxplot_and_descriptives(dataset, station_index):

    station_features = np.array([features[station_index] for features in
dataset.features])
     station_targets = np.array([targets[station_index] for targets in
dataset.targets])

    # Combining features and target for boxplot
    combined_data = np.concatenate([station_features, station_targets[:,
np.newaxis]], axis=1)
     column_names = [f"Feature (Lag {i})" for i in range(station_fea-
tures.shape[1])] + ["Target (Demand)"]

    # Create DataFrame for easier manipulation
    df = pd.DataFrame(combined_data, columns=column_names)

    # Plot boxplot using seaborn
    plt.figure(figsize=(15, 7))
    sns.boxplot(data=df)
    plt.title(f"Boxplot for Station {station_index + 1}")
    plt.ylabel("Value")
    plt.xticks(rotation=45)
    plt.show()

    # Calculate descriptive statistics
    descriptives = df.describe(percentiles=[.25, .50, .75])
    print(f"Descriptive Statistics for Station {station_index + 1}:\n")
```

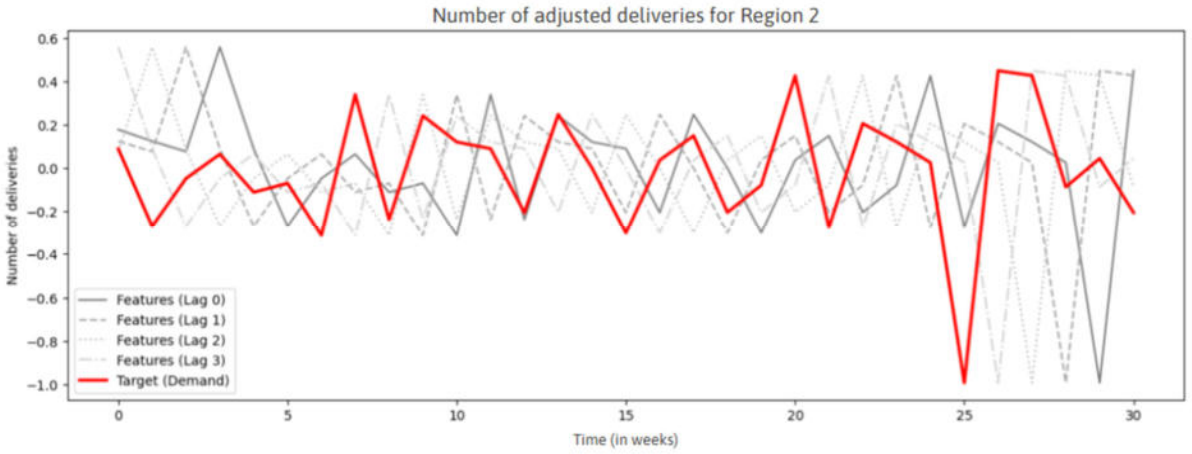
```
    print(descriptives)

    # Calculate whiskers
    whiskers = []
    for col in df.columns:
        q1 = df[col].quantile(0.25)
        q3 = df[col].quantile(0.75)
        iqr = q3 - q1
        lower_whisker = q1 - 1.5 * iqr
        upper_whisker = q3 + 1.5 * iqr
        whiskers.append((lower_whisker, upper_whisker))
        print(f"{col}: Lower whisker = {lower_whisker}, Upper whisker =
{upper_whisker}")

    return descriptives, whiskers

# Function for trim quartiles
def trim_to_quartiles(data, lower_q=0.25, upper_q=0.75):
    lower_quartile = np.quantile(data, lower_q)
    upper_quartile = np.quantile(data, upper_q)
    trimmed_data = np.clip(data, lower_quartile, upper_quartile)
    return trimmed_data

# Function which applies trim to quartiles to respective feature and-
timesteps
def correct_specific_outliers(features, targets):
    # The last target is trimmed to lower quartile
      trimmed_target  =  trim_to_quartiles(targets,  lower_q=0.25,  up-
per_q=0.25)
    targets[-1] = trimmed_target[-1]

    # The first feature is trimmed to upper quartile
     trimmed_features = trim_to_quartiles(features[:, 0], lower_q=0.75,
upper_q=0.75)
    features[0, 0] = trimmed_features[0]

    return features, targets

# Final function for "for" loop over all functions
def correct_dataset_outliers(dataset):
    number_of_stations = len(dataset.features[0])
    for station_index in range(number_of_stations):
       station_features = np.array([features[station_index] for features
in dataset.features])
        station_targets = np.array([targets[station_index] for targets
in dataset.targets])
```

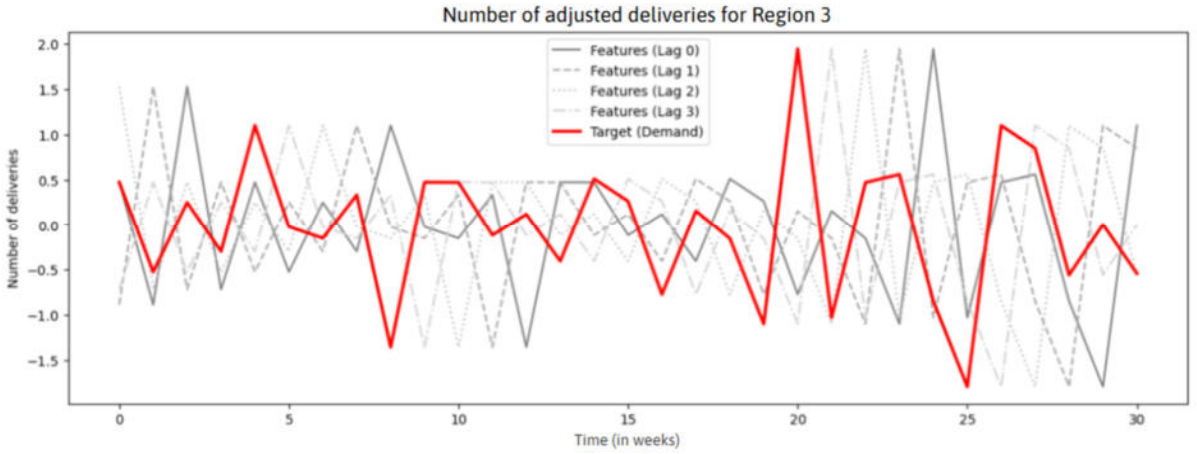
```
        corrected_features, corrected_targets = correct_specific_outli-
ers(station_features, station_targets)

        for idx in range(len(dataset.features)):
                dataset.features[idx][station_index]  =  corrected_fea-
tures[idx]
            dataset.targets[idx][station_index] = corrected_targets[idx]
    return dataset

# plot style attributes
linestyles = ['-', '--', ':', '-.']
greyscales = ['#808080', '#A9A9A9', '#C0C0C0', '#D3D3D3']

#plot station function
def plot_station_data(dataset, station_index, linestyles, greyscales):
    station_features = np.array([features[station_index] for features in
dataset.features])
     station_targets  =  np.array([targets[station_index]  for  targets  in
dataset.targets])
    plt.figure(figsize=(15, 5))
    for lag_index in range(station_features.shape[1]):
         plt.plot(station_features[:, lag_index], label=f"Features (Lag
{lag_index})",
                  linestyle=linestyles[lag_index  %  len(linestyles)],
color=greyscales[lag_index % len(greyscales)])
    plt.plot(station_targets, label="Target (Demand)", color='red', lin-
ewidth=2.5)
    plt.title(f"Time series of demand for Station {station_index + 1}")
    plt.xlabel("Time (in days)")
    plt.ylabel("Number of deliveries")
    plt.legend()
    plt.show()

# main process
loader = PedalMeDatasetLoader()
dataset = loader.get_dataset()
dataset = correct_dataset_outliers(dataset)
#for station_index in range(number_of_stations):
#    plot_station_data(dataset, station_index, linestyles, greyscales)

# checking number of stations
number_of_stations = len(dataset.features)

# Iteration over all stations
#for station_index in range(number_of_stations):
#    print(f"Analyzing Station {station_index + 1}:\n")
#    descriptives, whiskers = plot_boxplot_and_descriptives(dataset, sta-
tion_index)
```

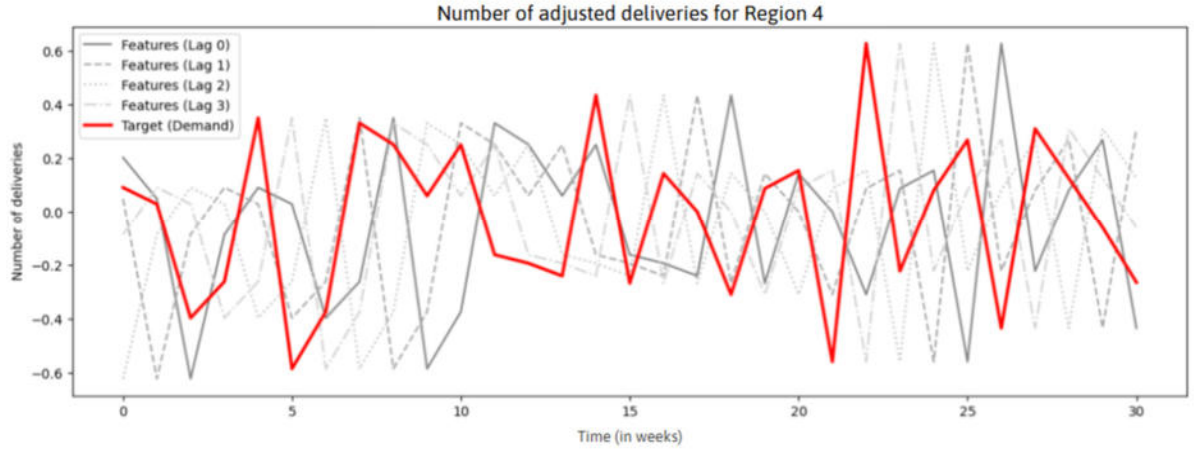
```
#     print("Descriptive Statistics:\n", descriptives)
#     print("Whiskers:\n", whiskers)
#     print("\n" + "="*50 + "\n")
```

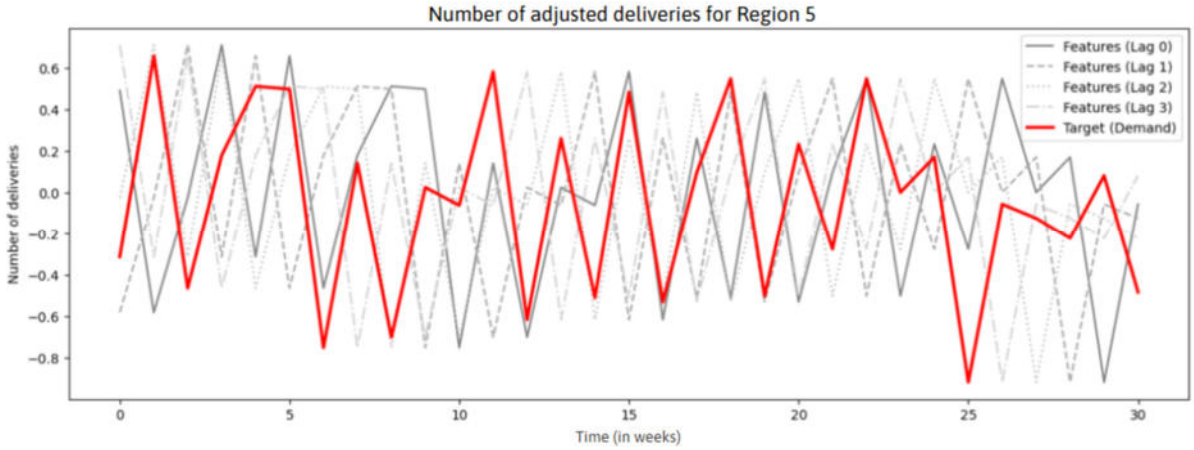
**Appendix H:** Adjusted time series with target (demand) and features (lagged demand) for each region

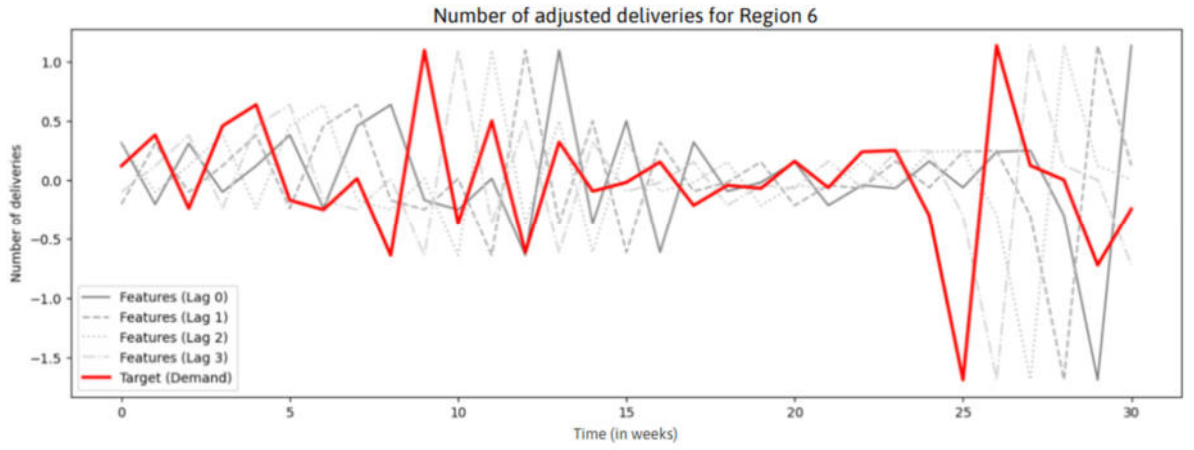


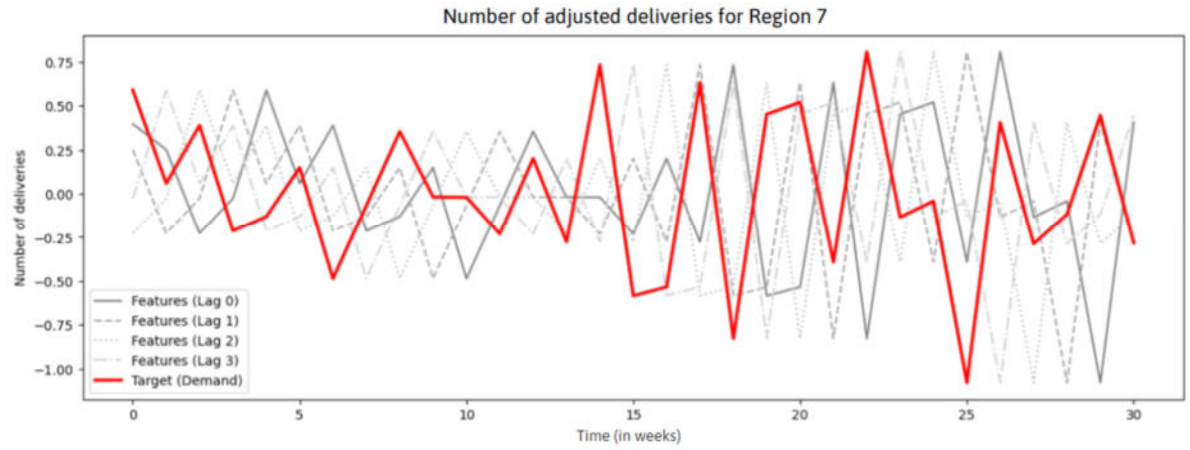



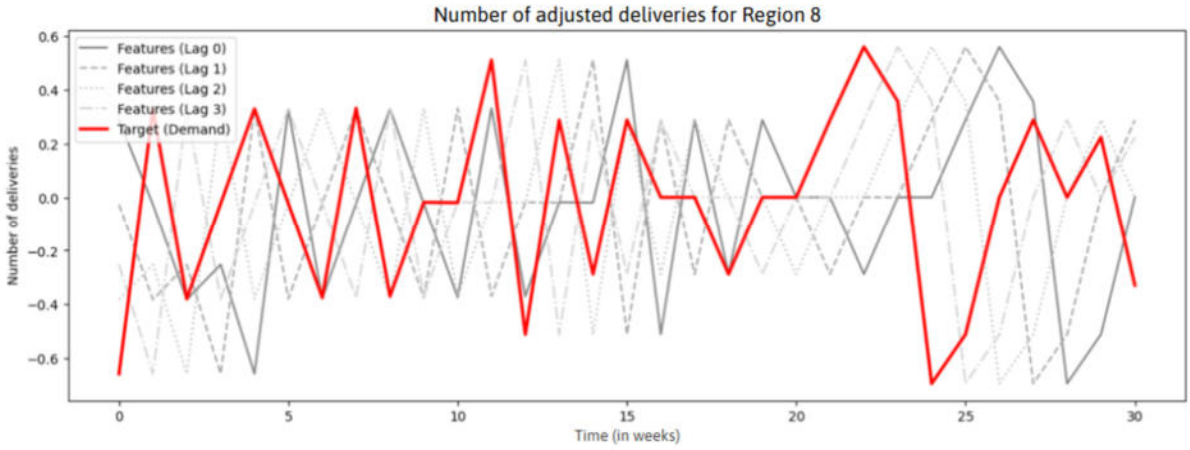



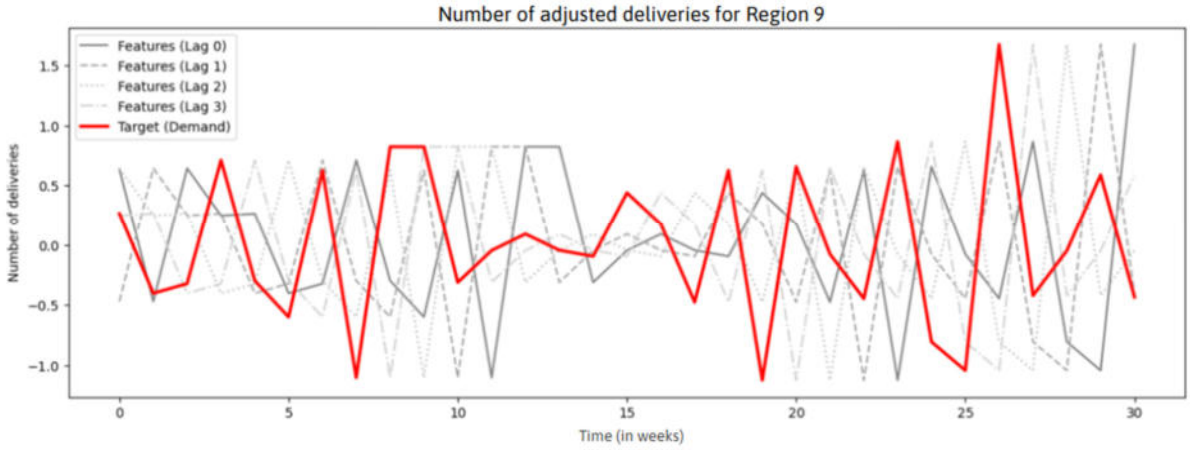

Number of adjusted deliveries for Region 4
Features (Lag 0)
Features (Lag 1)
Features (Lag 2)
Features (Lag 3)
Target (Demand)
Number of deliveries
Time (in weeks)

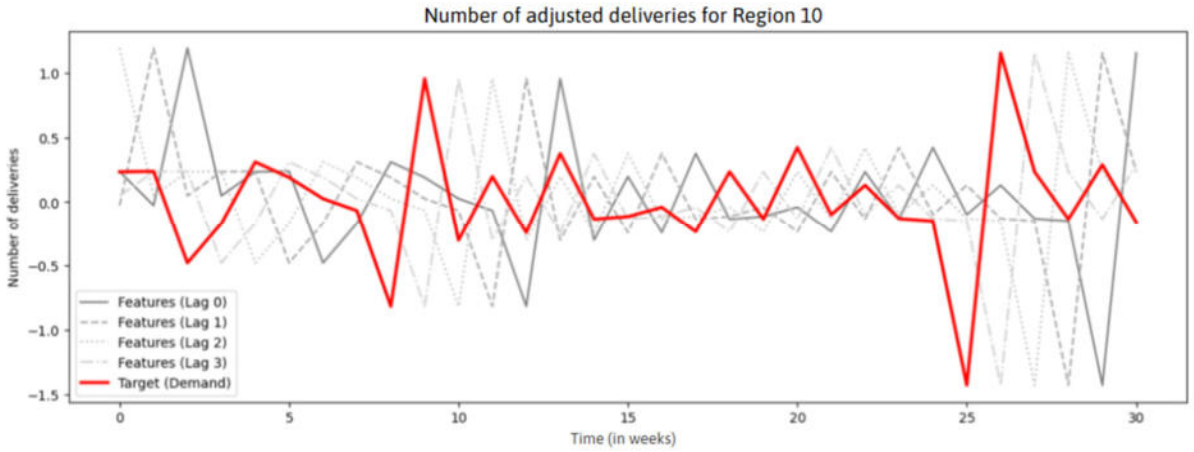

Number of adjusted deliveries for Region 5
Features (Lag 0)
Features (Lag 1)
Features (Lag 2)
Features (Lag 3)
Target (Demand)
Number of deliveries
Time (in weeks)

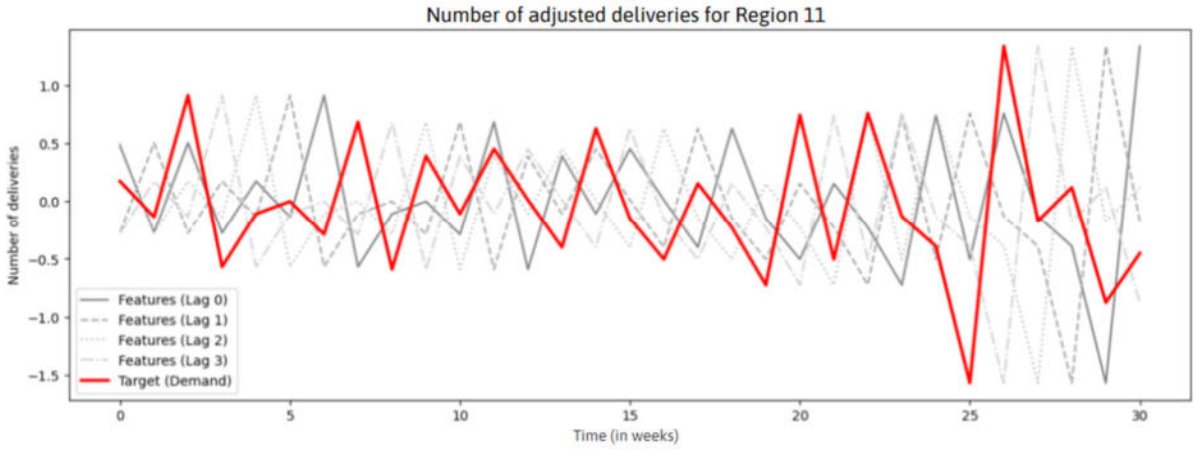

Number of adjusted deliveries for Region 6
Features (Lag 0)
Features (Lag 1)
Features (Lag 2)
Features (Lag 3)
Target (Demand)
Number of deliveries
Time (in weeks)

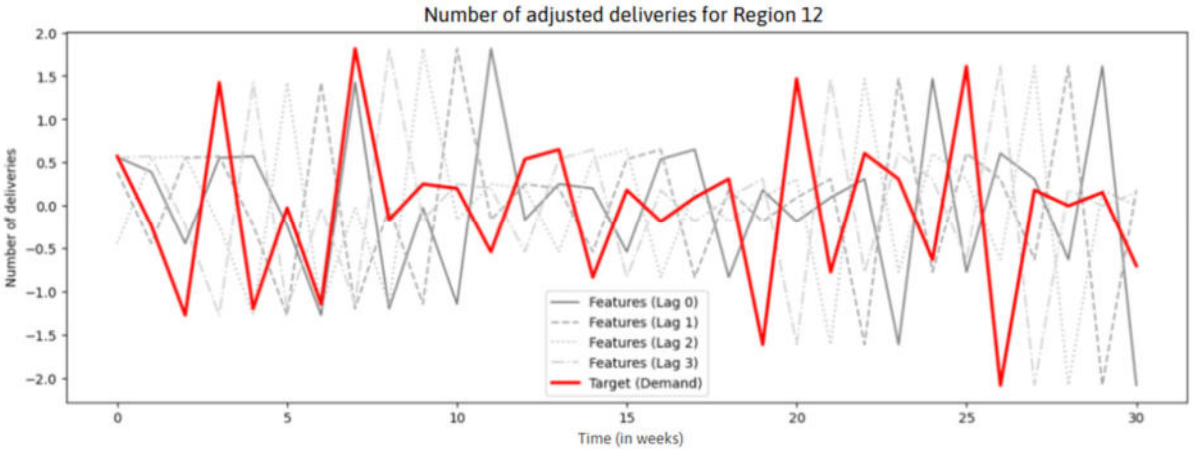

Number of adjusted deliveries for Region 7
Number of deliveries
Time (in weeks)
Features (Lag 0)
Features (Lag 1)
Features (Lag 2)
Features (Lag 3)
Target (Demand)

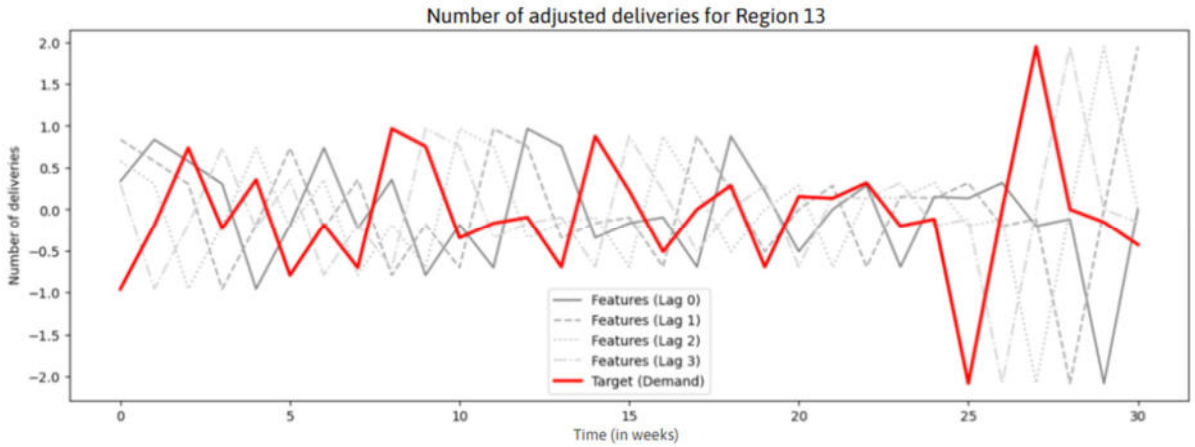

Number of adjusted deliveries for Region 8
Number of deliveries
Time (in weeks)
Features (Lag 0)
Features (Lag 1)
Features (Lag 2)
Features (Lag 3)
Target (Demand)

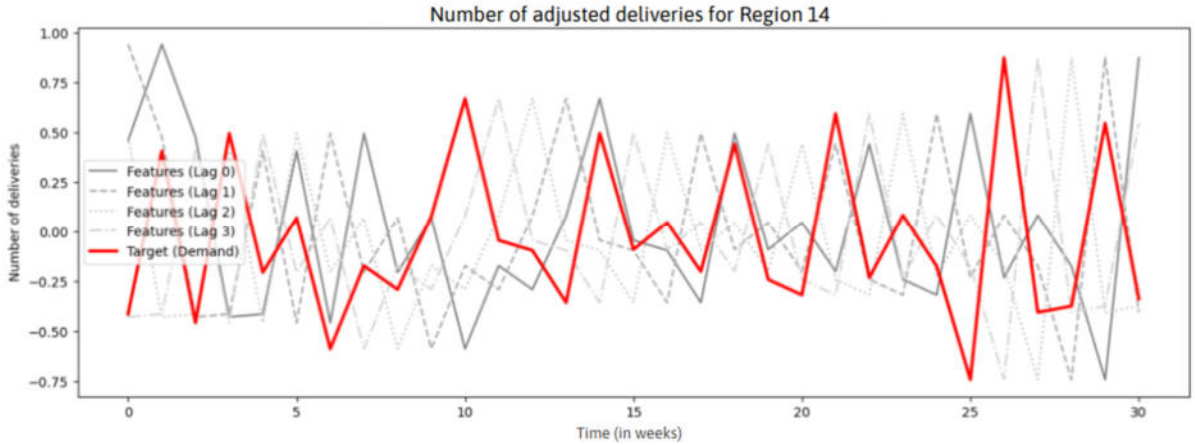

Number of adjusted deliveries for Region 9
Number of deliveries
Time (in weeks)
Features (Lag 0)
Features (Lag 1)
Features (Lag 2)
Features (Lag 3)
Target (Demand)

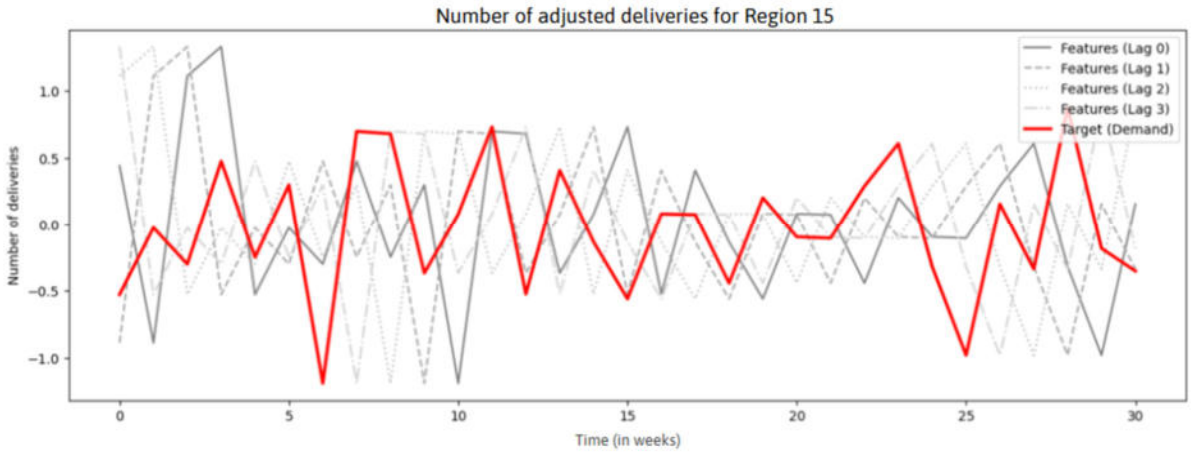

Number of adjusted deliveries for Region 10
Number of deliveries
Time (in weeks)
Features (Lag 0)
Features (Lag 1)
Features (Lag 2)
Features (Lag 3)
Target (Demand)


Number of adjusted deliveries for Region 11
Number of deliveries
Time (in weeks)
Features (Lag 0)
Features (Lag 1)
Features (Lag 2)
Features (Lag 3)
Target (Demand)


Number of adjusted deliveries for Region 12
Number of deliveries
Time (in weeks)
Features (Lag 0)
Features (Lag 1)
Features (Lag 2)
Features (Lag 3)
Target (Demand)


Number of adjusted deliveries for Region 13
Number of deliveries
Time (in weeks)
Features (Lag 0)
Features (Lag 1)
Features (Lag 2)
Features (Lag 3)
Target (Demand)


Number of adjusted deliveries for Region 14
Number of deliveries
Time (in weeks)
Features (Lag 0)
Features (Lag 1)
Features (Lag 2)
Features (Lag 3)
Target (Demand)


Number of adjusted deliveries for Region 15
Number of deliveries
Time (in weeks)
Features (Lag 0)
Features (Lag 1)
Features (Lag 2)
Features (Lag 3)
Target (Demand)

**Appendix I:** Code for Auto ARIMA

Please note that regions were analyzed but "station" as a variable were used because it simplified the thought process for coding

```
###################### ARIMA ######################

# Preparation of time series data for each station for ARIMA by breaking up the graph data into single time series data
station_series_dfs = []
for station_index in range(len(dataset.targets[0])):
    station_series = [target[station_index] for target in dataset.targets]
    df = pd.DataFrame(station_series, columns=['Demand'])
    station_series_dfs.append(df)

# Definition of the function for checking stationarity and application of differentiation
def check_stationarity_and_difference(series):
    result = adfuller(series.dropna())
    if result[1] > 0.05:  # Non-stationary
        series = series.diff().dropna()
    return series

# Prepare dictionaries to store predictions and actuals per station
arima_train_predictions = {}
arima_test_predictions = {}

# Metrics per Station
arima_train_metrics = {}
arima_test_metrics = {}

#Overall metrics
total_arima_train_actuals = []
total_arima_train_predictions = []
total_arima_test_actuals = []
total_arima_test_predictions = []

#Defining number of timesteps for testset
test_size = 5

for i, df in enumerate(station_series_dfs):
    print(f"Station {i+1}:")
    time_series = df['Demand']
    stationary_series = check_stationarity_and_difference(time_series)

    # Split in training and testing
    train = stationary_series.iloc[:len(station_series_dfs[0]) - test_size]
```

```
    test = stationary_series.iloc[len(station_series_dfs[0]) -
test_size:]
    print(f"Station {i+1}: Number of train data for ARIMA:
{len(train)}")
    print(f"Station {i+1}: Number of test data for ARIMA: {len(test)}")

    # Fit the ARIMA model
    model = auto_arima(train, seasonal=False, start_p=0, start_q=0,
max_p=5, max_q=5, stepwise=True, trace=True)
    train_predictions = model.predict_in_sample()
    test_predictions = model.predict(n_periods=test_size)

    # Store predictions per station
    arima_train_predictions[f"Station_{i+1}"] = train_predictions
    arima_test_predictions[f"Station_{i+1}"] = test_predictions

    # Overall performance metrics
    total_arima_train_actuals.extend(train)
    total_arima_train_predictions.extend(train_predictions)
    total_arima_test_actuals.extend(test)
    total_arima_test_predictions.extend(test_predictions)

    # Calculate and store metrics
    train_mse = mean_squared_error(train, train_predictions)
    train_rmse = np.sqrt(train_mse)
    train_mae = mean_absolute_error(train, train_predictions)
    arima_train_metrics[f"Station_{i+1}"] = (train_mse, train_rmse,
train_mae)

    test_mse = mean_squared_error(test, test_predictions)
    test_rmse = np.sqrt(test_mse)
    test_mae = mean_absolute_error(test, test_predictions)
    arima_test_metrics[f"Station_{i+1}"] = (test_mse, test_rmse,
test_mae)

    print(f"Train MSE ARIMA: {train_mse}, Train RMSE ARIMA:
{train_rmse}, Train MAE ARIMA: {train_mae}")
    print(f"Test MSE ARIMA: {test_mse}, Test RMSE ARIMA: {test_rmse},
Test MAE ARIMA: {test_mae}")

# Calculate overall train metrics
overall_train_mse = mean_squared_error(total_arima_train_actuals, to-
tal_arima_train_predictions)
overall_train_rmse = np.sqrt(overall_train_mse)
overall_train_mae = mean_absolute_error(total_arima_train_actuals, to-
tal_arima_train_predictions)
print(f"Overall ARIMA - Train MSE: {overall_train_mse}, Train RMSE:
{overall_train_rmse}, Train MAE: {overall_train_mae}")
```

```
# Calculate overall test metrics
overall_test_mse = mean_squared_error(total_arima_test_actuals, to-
tal_arima_test_predictions)
overall_test_rmse = np.sqrt(overall_test_mse)
overall_test_mae = mean_absolute_error(total_arima_test_actuals, to-
tal_arima_test_predictions)
print(f"Overall ARIMA - Test MSE: {overall_test_mse}, Test RMSE: {over-
all_test_rmse}, Test MAE: {overall_test_mae}")
```

**Appendix J:** Code for the GCRN models

Please note that regions were analyzed but "station" as a variable were used because it simplified the thought process for coding

```
##################### Graph based data split ###############

# Splitting of data using prebuild function of pytorch spatio temporal
train_dataset, test_dataset = temporal_signal_split(dataset, train_ra-
tio=0.85)
num_test_steps = len(test_dataset.features)
print(f"Number of test steps for GCRN models: {num_test_steps}")
num_train_timesteps = len(train_dataset.features)
print(f"Number of timesteps in train dataset: {num_train_timesteps}")

####################GConvGRU############################

# Define the RecurrentGCN model
class RecurrentGCN_GRU(nn.Module):
    def __init__(self, node_features, filters):
        super(RecurrentGCN_GRU, self).__init__()
        self.recurrent1 = GConvGRU(node_features, filters, 2)
        self.dropout = nn.Dropout(p=0.3)  # Adjust Dropout here!
        self.linear = nn.Linear(filters, 1)

    def forward(self, x, edge_index, edge_weight):
        h = self.recurrent1(x, edge_index, edge_weight)
        h = F.relu(h)
        h = self.dropout(h)
        h = self.linear(h)
        return h

# Initialize the model and optimizer
model_gru = RecurrentGCN_GRU(node_features=4, filters=32)
optimizer_gru = torch.optim.Adam(model_gru.parameters(), lr=0.01)

# Prepare to store loss data for plotting
train_losses_gru = []
```

```
test_losses_gru = []

# Initialize lists to store metrics for each station
train_actuals_gru = {i: [] for i in range(len(dataset.targets[0]))}
train_predictions_gru = {i: [] for i in range(len(dataset.targets[0]))}
test_actuals_gru = {i: [] for i in range(len(dataset.targets[0]))}
test_predictions_gru = {i: [] for i in range(len(dataset.targets[0]))}

# Initialize lists to store overall metrics
total_train_actuals_gru = []
total_train_predictions_gru = []
total_test_actuals_gru = []
total_test_predictions_gru = []

# Train the model
for epoch in tqdm(range(4)):
    epoch_train_loss = 0
    num_snapshots = 0  # Initialize snapshot counter

    model_gru.train() #start training mode

    # Clear the lists for each epoch
    for i in range(len(dataset.targets[0])):
        train_actuals_gru[i].clear()
        train_predictions_gru[i].clear()

    for time, snapshot in enumerate(train_dataset):
        optimizer_gru.zero_grad()
        y_hat = model_gru(snapshot.x, snapshot.edge_index, snap-
shot.edge_weight).squeeze()
        cost = torch.mean((y_hat - snapshot.y)**2)
        cost.backward()
        optimizer_gru.step()
        epoch_train_loss += cost.item()
        num_snapshots += 1

        # Store train predictions and actuals
        for i in range(len(dataset.targets[0])):
            train_predictions_gru[i].append(y_hat[i].item())
            train_actuals_gru[i].append(snapshot.y[i].item())

    train_losses_gru.append(epoch_train_loss / num_snapshots if
num_snapshots > 0 else 0)

    # Evaluate the model
    model_gru.eval()
    epoch_test_loss = 0
    num_test_snapshots = 0  # Initialize snapshot counter for testing
```

```
    # Clear the lists for each epoch
    for i in range(len(dataset.targets[0])):
        test_actuals_gru[i].clear()
        test_predictions_gru[i].clear()

    with torch.no_grad():
        for time, snapshot in enumerate(test_dataset):
            y_hat = model_gru(snapshot.x, snapshot.edge_index, snap-
shot.edge_weight).squeeze()
            if y_hat.dim() > 1:
                y_hat = y_hat.flatten()
            if snapshot.y.dim() > 1:
                snapshot.y = snapshot.y.flatten()
            for i in range(len(dataset.targets[0])):
                test_predictions_gru[i].append(y_hat[i].item())
                test_actuals_gru[i].append(snapshot.y[i].item())
            cost = torch.mean((y_hat - snapshot.y)**2)
            epoch_test_loss += cost.item()
            num_test_snapshots += 1

    test_losses_gru.append(epoch_test_loss / num_test_snapshots if
num_test_snapshots > 0 else 0)
    model_gru.train() #jump back to training mode

for i in range(len(dataset.targets[0])):
    total_train_actuals_gru.extend(train_actuals_gru[i])
    total_train_predictions_gru.extend(train_predictions_gru[i])
    total_test_actuals_gru.extend(test_actuals_gru[i])
    total_test_predictions_gru.extend(test_predictions_gru[i])

# Calculate train metrics for each station for GConvGRU
gru_train_metrics = {}
for i in range(len(dataset.targets[0])):
    train_mse = mean_squared_error(train_actuals_gru[i], train_predic-
tions_gru[i])
    train_rmse = np.sqrt(train_mse)
    train_mae = mean_absolute_error(train_actuals_gru[i], train_predic-
tions_gru[i])
    gru_train_metrics[f"Station_{i+1}"] = (train_mse, train_rmse,
train_mae)
    print(f"Station {i+1} GConvGRU - Train MSE: {train_mse}, Train
RMSE: {train_rmse}, Train MAE: {train_mae}")

# Calculate test metrics for each station for GConvGRU
gru_test_metrics = {}
for i in range(len(dataset.targets[0])):
```

```
    mse = mean_squared_error(test_actuals_gru[i], test_predic-
tions_gru[i])
    rmse = np.sqrt(mse)
    mae = mean_absolute_error(test_actuals_gru[i], test_predic-
tions_gru[i])
    gru_test_metrics[f"Station_{i+1}"] = (mse, rmse, mae)
    print(f"Station {i+1} GConvGRU - Test MSE: {mse}, Test RMSE:
{rmse}, Test MAE: {mae}")

# Calculate overall train metrics across all stations
overall_train_mse = mean_squared_error(total_train_actuals_gru, to-
tal_train_predictions_gru)
overall_train_rmse = np.sqrt(overall_train_mse)
overall_train_mae = mean_absolute_error(total_train_actuals_gru, to-
tal_train_predictions_gru)
print(f"Overall GConvGRU - Train MSE: {overall_train_mse}, Train RMSE:
{overall_train_rmse}, Train MAE: {overall_train_mae}")

# Calculate overall test metrics across all stations
overall_test_mse = mean_squared_error(total_test_actuals_gru, to-
tal_test_predictions_gru)
overall_test_rmse = np.sqrt(overall_test_mse)
overall_test_mae = mean_absolute_error(total_test_actuals_gru, to-
tal_test_predictions_gru)
print(f"Overall GConvGRU - Test MSE: {overall_test_mse}, Test RMSE:
{overall_test_rmse}, Test MAE: {overall_test_mae}")

##################GConvLSTM#########################

# Define the RecurrentGCN model
class RecurrentGCN_LSTM(nn.Module):
    def __init__(self, node_features, filters):
        super(RecurrentGCN_LSTM, self).__init__()
        self.recurrent1 = GConvLSTM(node_features, filters, 2)
        self.dropout = nn.Dropout(p=0.3)  # Adjust Dropout here!
        self.linear = nn.Linear(filters, 1)

    def forward(self, x, edge_index, edge_weight):
        h, c = self.recurrent1(x, edge_index, edge_weight)
        h = F.relu(h)
        h = self.dropout(h)
        h = self.linear(h)
        return h

# Initialize the model and optimizer
model_lstm = RecurrentGCN_LSTM(node_features=4, filters=32)
optimizer_lstm = torch.optim.Adam(model_lstm.parameters(), lr=0.01)
```

```
# Prepare to store loss data for plotting
train_losses_lstm = []
test_losses_lstm = []

# Initialize lists to store metrics for each station
train_actuals_lstm = {i: [] for i in range(len(dataset.targets[0]))}
train_predictions_lstm = {i: [] for i in range(len(dataset.tar-
gets[0]))}
test_actuals_lstm = {i: [] for i in range(len(dataset.targets[0]))}
test_predictions_lstm = {i: [] for i in range(len(dataset.targets[0]))}

# Initialize lists to store overall metrics
total_train_actuals_lstm = []
total_train_predictions_lstm = []
total_test_actuals_lstm = []
total_test_predictions_lstm = []

# Train the model
for epoch in tqdm(range(4)):
    epoch_train_loss = 0
    num_snapshots = 0  # Initialize snapshot counter

    model_lstm.train() #start training mode

    for i in range(len(dataset.targets[0])):
        train_actuals_lstm[i].clear()
        train_predictions_lstm[i].clear()

    for time, snapshot in enumerate(train_dataset):
        optimizer_lstm.zero_grad()
        y_hat = model_lstm(snapshot.x, snapshot.edge_index, snap-
shot.edge_weight).squeeze()
        cost = torch.mean((y_hat - snapshot.y)**2)
        cost.backward()
        optimizer_lstm.step()
        epoch_train_loss += cost.item()
        num_snapshots += 1

        for i in range(len(dataset.targets[0])):
            train_predictions_lstm[i].append(y_hat[i].item())
            train_actuals_lstm[i].append(snapshot.y[i].item())

    train_losses_lstm.append(epoch_train_loss / num_snapshots if
num_snapshots > 0 else 0)

    # Evaluate the model
    model_lstm.eval()
    epoch_test_loss = 0
```

```
    num_test_snapshots = 0  # Initialize snapshot counter for testing

    for i in range(len(dataset.targets[0])):
        test_actuals_lstm[i].clear()
        test_predictions_lstm[i].clear()

    with torch.no_grad():
        for time, snapshot in enumerate(test_dataset):
            y_hat = model_lstm(snapshot.x, snapshot.edge_index, snap-
shot.edge_weight).squeeze()
            if y_hat.dim() > 1:
                y_hat = y_hat.flatten()
            if snapshot.y.dim() > 1:
                snapshot.y = snapshot.y.flatten()
            for i in range(len(dataset.targets[0])):
                test_predictions_lstm[i].append(y_hat[i].item())
                test_actuals_lstm[i].append(snapshot.y[i].item())
            cost = torch.mean((y_hat - snapshot.y)**2)
            epoch_test_loss += cost.item()
            num_test_snapshots += 1

    test_losses_lstm.append(epoch_test_loss / num_test_snapshots if
num_test_snapshots > 0 else 0)
    model_lstm.train()

for i in range(len(dataset.targets[0])):
    total_train_actuals_lstm.extend(train_actuals_lstm[i])
    total_train_predictions_lstm.extend(train_predictions_lstm[i])
    total_test_actuals_lstm.extend(test_actuals_lstm[i])
    total_test_predictions_lstm.extend(test_predictions_lstm[i])

# Calculate train metrics for each station for GConvLSTM
lstm_train_metrics = {}
for i in range(len(dataset.targets[0])):
    train_mse = mean_squared_error(train_actuals_lstm[i], train_predic-
tions_lstm[i])
    train_rmse = np.sqrt(train_mse)
    train_mae = mean_absolute_error(train_actuals_lstm[i], train_pre-
dictions_lstm[i])
    lstm_train_metrics[f"Station_{i+1}"] = (train_mse, train_rmse,
train_mae)
    print(f"Station {i+1} GConvLSTM - Train MSE: {train_mse}, Train
RMSE: {train_rmse}, Train MAE: {train_mae}")

# Calculate test metrics for each station for GConvLSTM
lstm_test_metrics = {}
for i in range(len(dataset.targets[0])):
```

```
    mse = mean_squared_error(test_actuals_lstm[i], test_predic-
tions_lstm[i])
    rmse = np.sqrt(mse)
    mae = mean_absolute_error(test_actuals_lstm[i], test_predic-
tions_lstm[i])
    lstm_test_metrics[f"Station_{i+1}"] = (mse, rmse, mae)
    print(f"Station {i+1} GConvLSTM - Test MSE: {mse}, Test RMSE:
{rmse}, Test MAE: {mae}")

# Calculate overall train metrics across all stations
overall_train_mse = mean_squared_error(total_train_actuals_lstm, to-
tal_train_predictions_lstm)
overall_train_rmse = np.sqrt(overall_train_mse)
overall_train_mae = mean_absolute_error(total_train_actuals_lstm, to-
tal_train_predictions_lstm)
print(f"Overall GConvLSTM - Train MSE: {overall_train_mse}, Train RMSE:
{overall_train_rmse}, Train MAE: {overall_train_mae}")

# Calculate overall test metrics across all stations
overall_test_mse = mean_squared_error(total_test_actuals_lstm, to-
tal_test_predictions_lstm)
overall_test_rmse = np.sqrt(overall_test_mse)
overall_test_mae = mean_absolute_error(total_test_actuals_lstm, to-
tal_test_predictions_lstm)
print(f"Overall GConvLSTM - Test MSE: {overall_test_mse}, Test RMSE:
{overall_test_rmse}, Test MAE: {overall_test_mae}")

################### Plotting of selected stations ###################

selected_stations = [0,1,14]
num_historical_data = 10

# Predictions and actual values are correctly retrieved and plotted
fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(18, 5),
sharex=False, sharey=True)
fig.suptitle('Comparison of Actual Targets and Predictions for Selected
Stations')

for i, station_index in enumerate(selected_stations):
    historical_targets = [target[station_index] for target in da-
taset.targets[-(num_test_steps + num_historical_data):-num_test_steps]]
    test_targets = [target[station_index] for target in dataset.tar-
gets[-num_test_steps:]]



# Calculates the training steps and creates a list from 0 to
num_train_steps
```

```
    num_train_steps = num_train_timesteps - num_test_steps - num_his-
torical_data
    train_steps = list(range(num_train_steps))

# Predictions are retrieved correctly
    arima_preds = arima_test_predictions.get(f"Station_{station_index +
1}", [None] * num_test_steps)
    gconvgru_preds = test_predictions_gru[station_index][-
num_test_steps:]
    gconvlstm_preds = test_predictions_lstm[station_index][-
num_test_steps:]

    historical_steps = list(range(num_train_steps, num_train_steps +
len(historical_targets)))
    test_steps = list(range(num_train_steps + len(historical_targets),
num_train_steps + len(historical_targets) + len(test_targets)))

# Plots actual values and predictions for different models
    ax = axes[i]
    ax.plot(train_steps + historical_steps + test_steps,
            [None]*num_train_steps + historical_targets + test_targets,
label='Actual Values', marker='o')
    ax.plot(test_steps, arima_preds, label='ARIMA Predictions',
marker='x', linestyle="--")
    ax.plot(test_steps, gconvgru_preds, label='GConvGRU Predictions',
marker='*', linestyle="--")
    ax.plot(test_steps, gconvlstm_preds, label='GConvLSTM Predictions',
marker='^', linestyle="--")

    # Set the title to the current station
    ax.set_title(f'Region {station_index + 1}')

    # Set x-axis ticks in increments of 5
    ax.xaxis.set_major_locator(ticker.MultipleLocator(5))

    # Add legend to each subplot
    ax.legend()

    # Set labels
    ax.set_xlabel('Time Steps')
    ax.set_ylabel('Number of Requests')

# Print the plots
plt.tight_layout(rect=[0, 0, 1, 0.96])
plt.show()

##################### Plotting of Learning Curves ############
```

```
# GConvGRU Learning Curve
plt.figure(figsize=(10, 5))
plt.plot(train_losses_gru, label='Train Loss GConvGRU', marker='o')
plt.plot(test_losses_gru, label='Test Loss GConvGRU', marker='x')
plt.xlabel('Epoch')
plt.ylabel('Mean Squared Error')
plt.title('GConvGRU Learning Curve')
plt.legend()
plt.show()

# GConvLSTM Learning Curve
plt.figure(figsize=(10, 5))
plt.plot(train_losses_lstm, label='Train Loss GConvLSTM', marker='o')
plt.plot(test_losses_lstm, label='Test Loss GConvLSTM', marker='x')
plt.xlabel('Epoch')
plt.ylabel('Mean Squared Error')
plt.title('GConvLSTM Learning Curve')
plt.legend()
plt.show()

###################### Training Metrics Summary ######################

print("\n================== Metrics Summary for selected Regions
==================\n")

# Selected stations (in zero-based indexing)
selected_stations = [0, 1, 14]  # Indices for stations 1, 3 and 15

# Function for formatting the station name
def format_station_name(index):
    return f"Station_{index + 1}"

# ARIMA Training and Testing Metrics for selected stations
print("ARIMA Metrics:")
for station_index in selected_stations:
    station = format_station_name(station_index)
    train_mse, train_rmse, train_mae = arima_train_metrics[station]
    test_mse, test_rmse, test_mae = arima_test_metrics[station]
    print(f"{station}:")
    print(f"  Train MSE: {train_mse:.4f}, Train RMSE: {train_rmse:.4f},
Train MAE: {train_mae:.4f}")
    print(f"  Test MSE: {test_mse:.4f}, Test RMSE: {test_rmse:.4f},
Test MAE: {test_mae:.4f}")

print("\nGConvGRU Metrics:")
for station_index in selected_stations:
    station = format_station_name(station_index)
    train_mse, train_rmse, train_mae = gru_train_metrics[station]
```

```
    test_mse, test_rmse, test_mae = gru_test_metrics[station]
    print(f"{station}:")
    print(f"  Train MSE: {train_mse:.4f}, Train RMSE: {train_rmse:.4f},
Train MAE: {train_mae:.4f}")
    print(f"  Test MSE: {test_mse:.4f}, Test RMSE: {test_rmse:.4f},
Test MAE: {test_mae:.4f}")

print("\nGConvLSTM Metrics:")
for station_index in selected_stations:
    station = format_station_name(station_index)
    train_mse, train_rmse, train_mae = lstm_train_metrics[station]
    test_mse, test_rmse, test_mae = lstm_test_metrics[station]
    print(f"{station}:")
    print(f"  Train MSE: {train_mse:.4f}, Train RMSE: {train_rmse:.4f},
Train MAE: {train_mae:.4f}")
    print(f"  Test MSE: {test_mse:.4f}, Test RMSE: {test_rmse:.4f},
Test MAE: {test_mae:.4f}")

print("\n============================================================\
n")
```

**Appendix K:** Detailed Results for Region-Level Analysis

## Region 1

| Model | Epochs | Dropout | Train MSE | Test MSE |
|---|---|---|---|---|
| GConvGRU | 4 Epochs | Dropout 0,3 | 0,0376 ± 0,0074 | 0,0506 ± 0,0084 |
| GConvLSTM | 4 Epochs | Dropout 0,3 | 0,0399 ± 0,0034 | 0,0362 ± 0,0062 |
| GConvGRU | 4 Epochs | no Dropout | 0,0325 ± 0,0013 | 0,0572 ± 0,0078 |
| GConvLSTM | 4 Epochs | no Dropout | 0,0346 ± 0,0015 | 0,0519 ± 0,0097 |
| GConvGRU | 10 Epochs | Dropout 0,3 | 0,0394 ± 0,0026 | 0,0652 ± 0,0147 |
| GConvLSTM | 10 Epochs | Dropout 0,3 | 0,0391 ± 0,0039 | 0,0558 ± 0,0173 |
| GConvGRU | 10 Epochs | no Dropout | 0,0388 ± 0,0029 | 0,0862 ± 0,0313 |
| GConvLSTM | 10 Epochs | no Dropout | 0,0354 ± 0,0018 | 0,0711 ± 0,0127 |
| GConvGRU | 30 Epochs | Dropout 0,3 | 0,0455 ± 0,0094 | 0,2304 ± 0,0863 |
| GConvLSTM | 30 Epochs | Dropout 0,3 | 0,0495 ± 0,0078 | 0,1616 ± 0,0524 |
| GConvGRU | 30 Epochs | no Dropout | 0,0525 ± 0,0036 | 0,2715 ± 0,1003 |
| GConvLSTM | 30 Epochs | no Dropout | 0,0551 ± 0,0051 | 0,3029 ± 0,0536 |
| ARIMA | - | - | 0,0449 | 0,0242 |

## Region 3

| Model | Epochs | Dropout | Train MSE | Test MSE |
|---|---|---|---|---|
| GConvGRU | 4 Epochs | Dropout 0,3 | 0,4673 ± 0,0332 | 0,3383 ± 0,0283 |
| GConvLSTM | 4 Epochs | Dropout 0,3 | 0,5024 ± 0,0191 | 0,3475 ± 0,0693 |
| GConvGRU | 4 Epochs | no Dropout | 0,4553 ± 0,0043 | 0,3096 ± 0,0315 |
| GConvLSTM | 4 Epochs | no Dropout | 0,4694 ± 0,0098 | 0,3196 ± 0,0615 |
| GConvGRU | 10 Epochs | Dropout 0,3 | 0,3768 ± 0,0133 | 0,7227 ± 0,0842 |
| GConvLSTM | 10 Epochs | Dropout 0,3 | 0,3850 ± 0,0236 | 0,6558 ± 0,1179 |
| GConvGRU | 10 Epochs | no Dropout | 0,3644 ± 0,0116 | 0,5774 ± 0,1184 |
| GConvLSTM | 10 Epochs | no Dropout | 0,3701 ± 0,0077 | 0,8481 ± 0,0866 |
| GConvGRU | 30 Epochs | Dropout 0,3 | 0,2759 ± 0,0458 | 1,3176 ± 0,2450 |
| GConvLSTM | 30 Epochs | Dropout 0,3 | 0,2743 ± 0,0414 | 1,2314 ± 0,1274 |
| GConvGRU | 30 Epochs | no Dropout | 0,1912 ± 0,0135 | 1,4751 ± 0,2689 |
| GConvLSTM | 30 Epochs | no Dropout | 0,1984 ± 0,0128 | 1,3646 ± 0,3315 |
| ARIMA | - | - | 0,4720 | 0,3168 |

**Region 15**

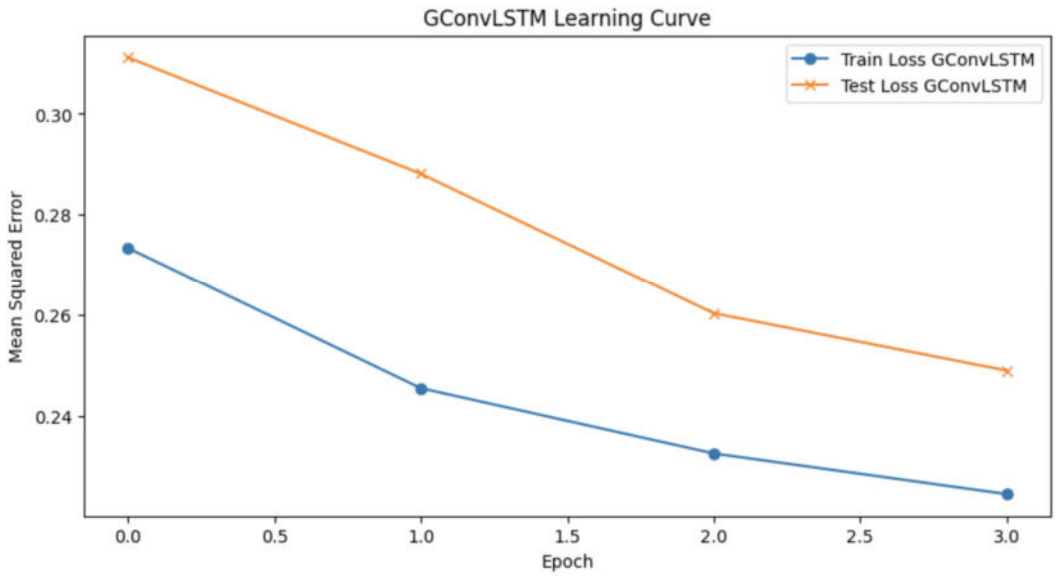
| Model | Epochs | Dropout | Train MSE | Test MSE |
|---|---|---|---|---|
| GConvGRU | 4 Epochs | Dropout 0,3 | 0,1912 ± 0,0079 | 0,1117 ± 0,0271 |
| GConvLSTM | 4 Epochs | Dropout 0,3 | 0,2008 ± 0,0132 | 0,1098 ± 0,0176 |
| GConvGRU | 4 Epochs | no Dropout | 0,1808 ± 0,0050 | 0,1252 ± 0,0357 |
| GConvLSTM | 4 Epochs | no Dropout | 0,1891 ± 0,0031 | 0,1159 ± 0,0265 |
| GConvGRU | 10 Epochs | Dropout 0,3 | 0,1690 ± 0,0052 | 0,1721 ± 0,0563 |
| GConvLSTM | 10 Epochs | Dropout 0,3 | 0,1678 ± 0,0123 | 0,1323 ± 0,0229 |
| GConvGRU | 10 Epochs | no Dropout | 0,1465 ± 0,0047 | 0,1900 ± 0,0548 |
| GConvLSTM | 10 Epochs | no Dropout | 0,1589 ± 0,0071 | 0,1798 ± 0,0273 |
| GConvGRU | 30 Epochs | Dropout 0,3 | 0,1582 ± 0,0093 | 0,4108 ± 0,0717 |
| GConvLSTM | 30 Epochs | Dropout 0,3 | 0,1469 ± 0,0146 | 0,3343 ± 0,046 |
| GConvGRU | 30 Epochs | no Dropout | 0,0937 ± 0,0149 | 0,4878 ± 0,1246 |
| GConvLSTM | 30 Epochs | no Dropout | 0,0945 ± 0,0160 | 0,4165 ± 0,0935 |
| ARIMA | - | - | 0,1932 | 0,3992 |

**Appendix L:** Global performance metrics with varying epochs. Dropout value: 0.3 in all configurations

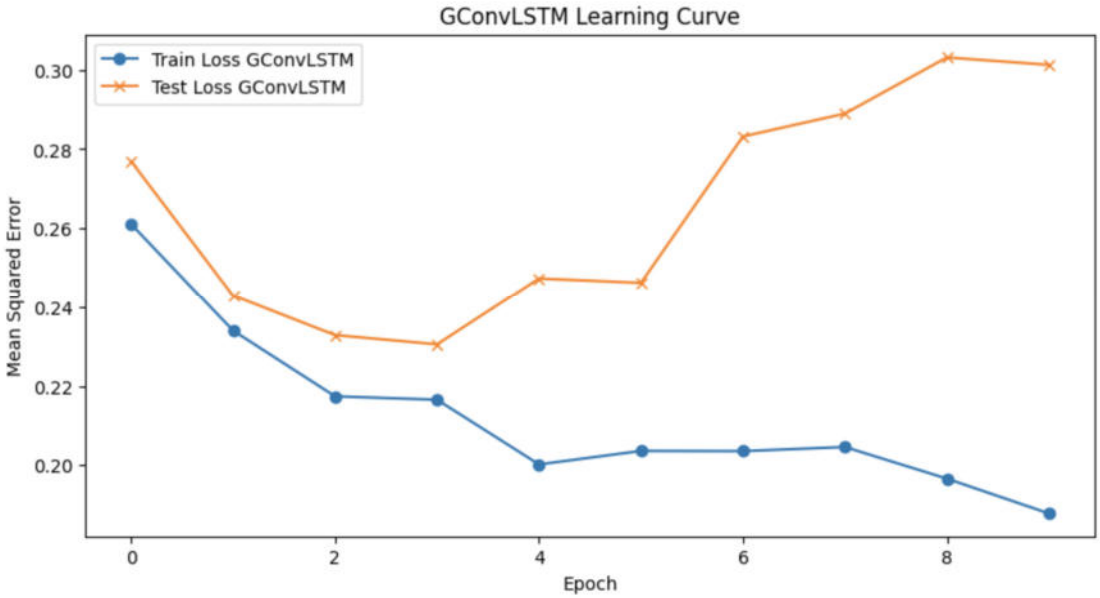
| | Epochs | Train MSE | Test MSE |
|---|---|---|---|
| GConvGRU | 4 | 0,2117 ± 0,0035 | 0,2326 ± 0,0013 |
| GConvLSTM | 4 | 0,2216 ± 0,0038 | 0,2339 ± 0,0031 |
| GConvGRU | 10 | 0,1845 ± 0,0012 | 0,3161 ± 0,0328 |
| GConvLSTM | 10 | 0,1879 ± 0,0036 | 0,3279 ± 0,0322 |
| GConvGRU | 30 | 0,1509 ± 0,0085 | 0,5968 ± 0,1342 |
| GConvLSTM | 30 | 0,1516 ± 0,0057 | 0,5502 ± 0,0543 |

**Appendix M:**

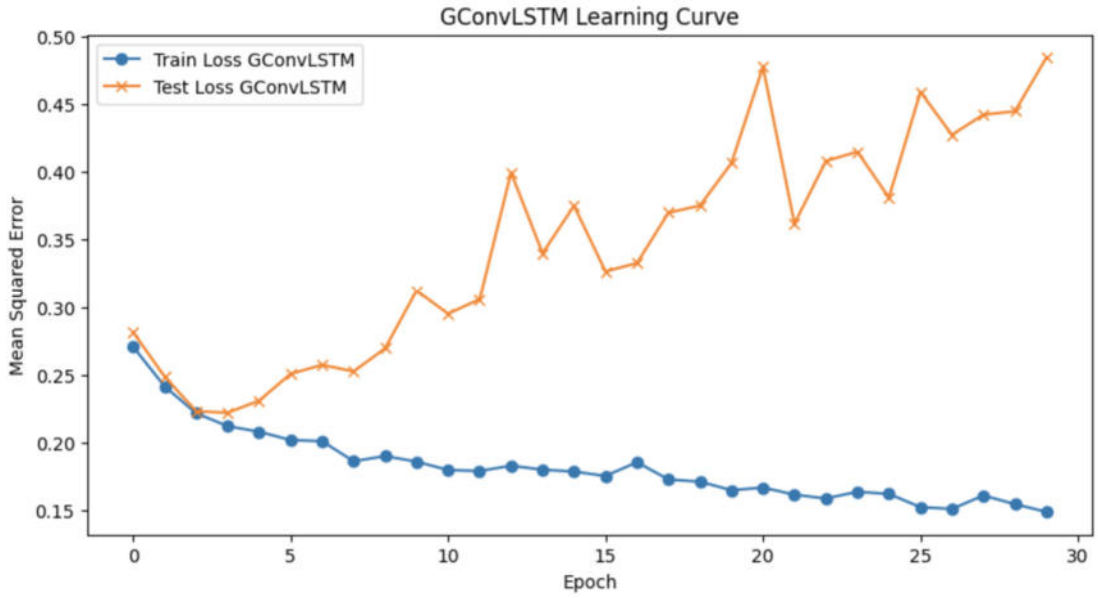
Learning Curves of GConvLSTM with 4 epochs and dropout




Learning Curves of GConvLSTM with 10 epochs and dropout



Learning Curves of GConvLSTM with 30 epochs and dropout



**Statement of Originality**

By signing this statement, I hereby acknowledge the submitted thesis (hereafter mentioned as "product"), titled:

**Comparative study of Graph Convolutional Recurrent Network models in traffic forecasting using the PedalMe's bike transport demand example of London**

to be produced independently by me, without external help. Wherever I paraphrase or cite literally, a reference to the original source (journal, book, report, internet, etc.) is given.

By signing this statement, I explicitly declare that I am aware of the fraud sanctions as stated in the Education and Examination Regulations (EERs) of the SBE.

Place: Maastricht, Netherlands
Date: 21.06.2024
First and last name: Leon Reiß
Study programme: Business Intelligence and Smart Services (MSc)
Course/skill: Master Thesis
ID number: I6337206

Signature: 

**Sustainable Development Goals (SDG) Statement**

| | |
|---|---|
| Name | Leon Reiß |
| ID | I6337206 |
| Supervisor | Prof. Dr. Roselinde Kessels |
| Date | 21.06.2024 |

Through the research conducted for this master's thesis, I seek to contribute to one or more of the 17 SDG(s) set forth by the United Nations (https://www.undp.org/sustainable-development-goals). Specifically:


SDG Code(s): 9, 11, 13
Explanation (max. 300 words):

My research on Intelligent Transport Systems (ITS) and their impact on traffic dynamics, contributes significantly to the achievement of several Sustainable Development Goals (SDGs). In particular, by developing and analyzing advanced traffic forecasting models and integrating modern technologies such as artificial intelligence and machine learning, I support the following SDGs:

**SDG 9: Industry, innovation and infrastructure**

My work contributes to improving transport infrastructure by evaluating models for decision support systems (DSS) that make transport more efficient and sustainable.

**SDG 11: Sustainable cities and communities**

ITS solutions improve urban mobility and contribute to the creation of safe, inclusive and sustainable cities. By reducing traffic congestion and promoting green transport options, my research improves the quality of life in urban areas and supports the reduction of air pollution and emissions.

**SDG 13: Climate action**

By promoting sustainable transport options and optimizing traffic management, my research contributes to climate change mitigation and supports the Sustainable Development Goals related to climate change mitigation.