UNIVERSITY OF SZEGED

FACULTY OF SCIENCE AND INFORMATICS

BOLYAI INSTITUTE

Boróka Jankus

GRAPH NEURAL NETWORKS FOR TIME SERIES FORECASTING IN WATER MANAGEMENT

Master Thesis

Applied Mathematics MSc, technical mathematics specialization

Supervisor:
Zsolt Vizi, Ph.D.

Szeged, 2023

Acknowledgements

I would like to express my thanks to my theme supervisor, Zsolt Vizi, from whom through the last two years I have received a lot of support, opportunities for improvement, and guidance. I'm thankful for his encouraging words, and that he believed in me. Thank you for your patience, and for always being available when I needed help.

Contents

1 Introduction **5**

2 Machine learning **7**

2.1 Supervised learning - the basics 7

2.2 Artifical Neural Network (ANN) 9

2.3 Recurrent Neural Network (RNN) 10

2.3.1 Computing the gradient in RNN 11

2.4 Long Short Term Memory (LSTM) 11

2.4.1 LSTM with "Peephole Connections" 14

3 Graphs in the machine learning **16**

3.1 Convolutional Graph Neural Networks (ConvGNN) 16

3.1.1 Convolution 17

3.1.2 Laplacian matrix 18

3.1.3 Spectral-based graph convolution 19

3.1.4 ChebNet 21

4 The LSTM and graphs meeting **25**

4.1 Road speed prediction 25

4.2 PM2.5 forcasting 30
4.3 Graph Convolutional Recurrent Network (GCRN) 33
4.3.1 Model 1. 33
4.3.2 Model 2 34
4.3.3 The performance of the models 35

5 The programming framework used **37**
5.1 PyTorch Geometric 37
5.1.1 `MessagePassing` 38
5.1.2 `torch.geometric.data.Data` 39
5.2 PyTorch Geometric Temporal 40

6 Water level forecast for Szeged **41**
6.1 Data, data processing 41
6.2 Model 44
6.3 Training 46
6.4 Results and evaluation 47
6.5 Open issues, directions for improvement 54

Chapter 1

Introduction

Although the foundations of machine learning and neural networks were laid in the 1940s, their explosive take-up only occurred around the 2000s. With the rapid development of machine learning, the use of deep learning has spread to more and more disciplines. We see it every day in facial recognition systems on our phones, stock market forecasts, or through YouTube's recommendation system.

Machine learning algorithms can be divided into two large groups. These are the unsupervised learning algorithms and the supervised learning algorithms. Unsupervised learning data is unlabelled and explores similarities and differences in information. In contrast, supervised learning data are labelled. In this case, given an input data point, the algorithm is tasked with learning to predict a target output (label).

For some topics, there is more correlation between the data than classical supervised learning models can handle. In the case of speech recognition, as opposed to image recognition, or price prediction, the succession of data carries important information. We also need to take into account the effect of temporally adjacent data. Therefore, when modelling time series, we cannot assume that successive patterns are independent. In this thesis we deal with the prediction of water levels, which is also a time series based task.

Water level forecasting is of great importance for water management and flood protection. Since most of our country is lowland, flood control has always played an important role in our lives throughout history. Centuries of records and then digitisation have accumulated a wealth of data. Until today, physical and hydrological models have been used for forecasting.

As the concept of artificial intelligence becomes more widely known, so the need for data-driven methods is emerging in more and more disciplines. This has also happened in the field of water management.

In the spring of 2021, the Lower Tisza Water Management Directorate (the Hungarian acronym is ATIVIZIG) approached the Bolyai Institute at University of Szeged with a cooperation opportunity. The aim was to test modern forecasting approaches on available water time series. Some results of the collaboration are presented in the [10] thesis, which I used as a first starting point. During this collaboration, the idea of developing models that take into account the relative placement of water gauges with respect to each other emerged.

We aim to develop such a model, based on water data sets. Due to the time series nature of the data, we started from the Recurrent Neural Network framework. We investigated its combinability with graphs to exploit the geometry of the gauging stations. In this thesis, we first give an introduction to the architecture of RNNs and present the so-called LSTM cell, which was the basis for the models we investigated. Then, in Chapter 3, we show applicability of machine learning models on graph-based data. Chapter 4 presents the models reviewed in the literature search, until we arrive at the selected model. Chapter 5 presents the main Python libraries and classes we use, and Chapter 6 contains the description and evaluation of the developed model.

Chapter 2

Machine learning

2.1 Supervised learning - the basics

Machine learning algorithms can be divided into two large groups. These are the unsupervised learning algorithms and the supervised learning algorithms. In the case of supervised learning the data set contains features, and each data has a target value.

Features are some descriptive properties in the data that are used as input to the model. The target is an output value that we want to train the model to predict. The target can be a continuous or a discrete value. We talk about regression when the target value is continuous, for example, when we want to predict a person's income or the probability of raining tomorrow. A discrete target value is a way of labeling or categorizing data, for example, defining the blood group or grouping scientific articles by discipline. This task is called classification. So, since the target value is the determinant, we can formulate both regression and classification tasks within the same topic. In our case, if we were interested in the degree of flood preparedness required, we would talk about classification, and if we were interested in specific water levels, we would talk about regression.

The task of supervised learning is to learn a mapping that assigns a dependent variable (output value) to the independent variables (input variables). So we look for a relationship between the input features and target values. This mapping is a model that contains a model architecture, internal parameters and external also known as hyperparameters. The architecture of the

model determines the course of the calculation. The hyperparameters are free parameters in the architecture, which are defined by the modeler. The internal parameters are determined by the computer during the training procedure. First we look for the best internal parameters for the given model and hyperparameters. Then we can choose the best hyperparameters for the model iteratively, finding best internal parameters for a specific hyperparameter setting. Usually the second step can also is automated, but in our case it is done manually due to (hardware) resource limitations.

To find the internal parameters, we need a loss function that can measure the goodness of the prediction. In general, it is a function of the deviation between the predicted and the target value. We can formally write the loss function in the form below:

$$L(\theta) = \sum_{i=1}^{N} l\big(y_i, F(X_i;\theta)\big),$$

where X_i and y_i are the input features and the target belonging to the ith data point, respectively; N is the number of data points; F is the model and θ is the vector of internal parameters; l is a function measures the difference between the values; $L(\theta)$ means the loss function. Note that l follows different logic for classification and regression. An important requirement for the loss function is that it is easy to calculate and differentiate.

We need to find the global minimum of the loss function, and those will be the best internal parameter setting. Minimisation algorithms in machine learning use the gradient descent (GD) [7, 18] technique most frequently. As it is known, the gradient tends to the steepest growth. Since we want to decrease the value of the loss function, not increase it, we go in the opposite direction. The step size is determined by the so-called learning rate, thus we take $L(\theta)$ and calculate the its gradient. To decrease $L(\theta)$, move in the direction of the negative gradient. Thus, we can update θ by $\theta - \varepsilon\nabla_\theta L(\theta)$, where ε is the learning rate. It is important to mention that for calculating the gradient we use backpropagation from area of automatic differentiation, which is the most popular algorithm for this purpose in machine learning [7].

The gradient descent method presented above takes a single step after examining the entire training set. For big data sets, this makes the update of the weights very slow. Therefore, a relaxation of the gradient descent is used in most deep-learning models called the stochastic gradient descent (SGD). At each step, we estimate using a only small sample set, i.e. we take a

random mini-batch of m instances, uniformly sampled from the training set and then perform the gradient descent steps on this mini-batch.

2.2 Artifical Neural Network (ANN)

In the 1940s, researchers began investigate the human brain's ability to recognise patterns. The Artificial Neural Network (ANN, shortly)[7, 2, 13] was inspired by the brain's neural network signal transmission. The ANNs build up of layers of nodes called neurons: there is always an input layer and an output layer, with one or more hidden layers in between. An ANN with several hidden layers is called a Deep Neural Network (DNN), where the depth of the network means the number of intermediate layers. Each neuron is connected to at least one other neuron from the adjacent layers and can send and receive signals through the edges between them, similar to brain synapses. The edges have some weight and a neuron aggregates the input signals based on the weight and then uses a nonlinear function to calculate the output signal, which transmits to its neighbours in the next layer. During the learning process, these weights are continuously updated. When using DNN, it is possible to have completely different intermediate layers.

Formally, this means that for input x and target y we define a

$$y = f(x, \theta)$$

where, for example, for a network with three intermediate layers

$$f(x, \theta) = f^{(3)}\left(f^{(2)}\left(f^{(1)}\left(x, \theta_1\right), \theta_2\right), \theta_3\right),$$

where $f^{(1)}$ is the first layer, $f^{(1)}$ is the second layer and $f^{(3)}$ is the third layer.

In most cases, DNNs are feedforward networks, which means information flows only forward, from the input through the intermediate layers to the output. So there is no feedback from the model output to the model.

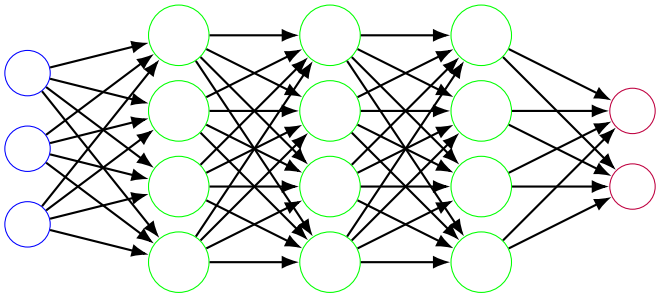


Figure 2.1: A Deep Neural Network (DNN). The blue circles are the input layer, the green circles make up the hidden layers and the purple circles are the output layer.

2.3 Recurrent Neural Network (RNN)

The recurrent neural network [7, 13] (RNN) is a type of deep neural networks, which is used to process a sequence of $x^{(1)}, \cdots, x^{(t)}$ data, in other words, sequential data.

In the 1980s, a new idea emerged in machine learning where parameters are shared between the recursion steps of a single level of the model. RNNs exploit this to allow the handling of sequences of different lengths.

Unlike previous deep learning algorithms, RNNs consider the temporal dependency of data. The output always depends on the previous inputs. The basic idea is to add memory to the neurons (hidden state), which is an input at each step along with the current data. A neuron usually consists of a single (tanh) layer, which is used to store the information.

The most of RNNs uses a similar equation as

$$h^{(t)} = f\left(h^{(t-1)}, x^{(t)}; \theta\right)$$

to compute the values of hidden state. As shown in the Figure 2.2. this process can be represented as a circuit diagram and an unfolded computational graph.

The unfolded structure has two important results.

- It is not necessary to create a new function g for the entire past sequence at each time step, i.e. we can use the same function and parameters at each time step.
- The input size of the model is constant and independent of the length of the sequence.

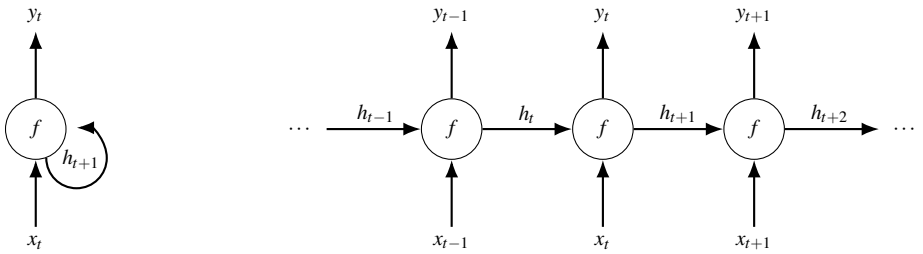


Figure 2.2: One hidden layer RNN, as folded and unfolded structure. As time goes on, the output always depends on the previous inputs. If we want to build a multilayer model like the DNN, we can append the next layer to the output layer. A specific feature of RNN is that it uses the same weight parameter in a recursive loop.

Since the model works for all time steps and sequences, less learning data is sufficient for teaching. Furthermore, most recurrent networks can process variable-length sequences.

2.3.1 Computing the gradient in RNN

The backpropagation through time [7] (BPTT) algorithm is applied to compute the gradient. First must be built the computational graph along which the information flows through the network. Then the information from loss function flows backwards, and we compute the gradient. The difference between the BPTT algorithm and the traditional backpropagation is that BPTT sums the errors in each time step because of the shared parameters. At this point we should mention one problem with RNNs: backpropagation on the unfolded structure means the repetition of matrix multiplications with the same matrix. This can cause the gradient to disappear or explode, leading to unstable training procedure for an RNN.

2.4 Long Short Term Memory (LSTM)

Another shortcoming of RNN is that it cannot learn long-term relationships, i.e. it does not "remember" well the data it has seen long ago in the input sequence. RNNs using Long short-term memory cells (LSTMs)[1, 11, 13], a subclass of RNNs, are well suited to deal with the problems that arise. They use a new internal state (cell state) as memory: the cell state is transmitted through iterative layers and updated from cell to cell. LSTM allows previous

information to be forgotten or retained and new information to be added to the cell state. The (recursively applied) module consists of 4 parts: deleting redundant memory, inserting new information, updating memory, and generating output.

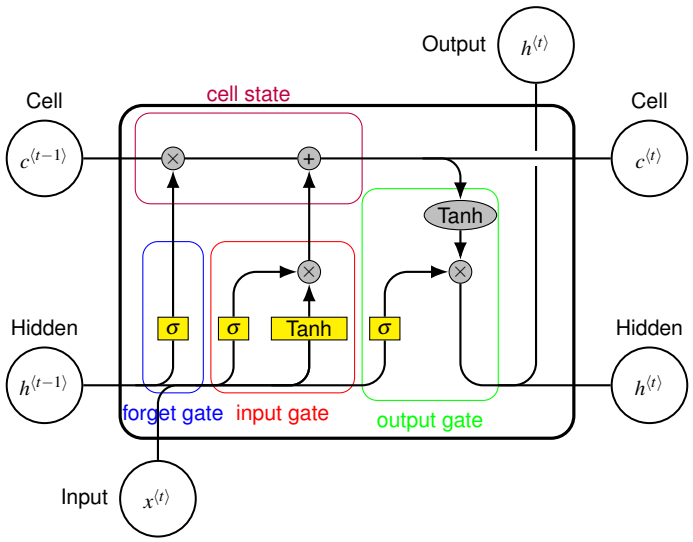


Figure 2.3: The LSTM cell uses two internal states: at the top of the figure you can follow the cell update, and at the bottom you can see the hidden state path. The differently colored forget, input, and output gates ensure that redundant information is forgotten, new and useful information is stored in memory, and information is passed between cells. The grey circles and ellipses represent element-by-element operations, and the yellow rectangles are the so-called activation functions.

Figure 2.3 shows an LSTM cell consisting of three gates. Information flows through them. The gates may contain a multiplication by a weight matrix, an activation function ($\tanh(x)$ or sigmoid function: $\sigma(x) = \frac{1}{1+e^{-x}}$), and some elementwise operations ($\otimes$, $\oplus$). In the learning process, we want to determine the optimal weight values.

Let us denote the input by $x_t \in \mathbb{R}^{d_x}$, the hidden state by $h_t \in [-1,1]^{d_h}$ and cell state by $c_t \in \mathbb{R}^{d_h}$. The weight matrices are $W_{x\cdot} \in \mathbb{R}^{d_h \times d_x}$, $W_{h\cdot} \in \mathbb{R}^{d_h \times d_h}$, and biases are denoted by $b_f, b_i, b_c, b_o \in \mathbb{R}^{d_h}$. First, the forget gate decides which information from the previous cell state to forget. Based on the previous hidden state, and the new input, a sigmoid layer returns a number between 0 and 1 for each value of the cell state enabling to forget the unnecessary information:

$$f_t = \sigma\left(W_{xf}x_t + W_{hf}h_{t-1} + b_f\right) \quad \in [0,1]^{d_h}.$$

In the next step, the input gate decides what it remembers from the incoming data based on the previous hidden state. First again, a sigmoid layer decides how much of each piece of information to remember. Then a vector of new possible values is created: $\tilde{c}_t$. The product of the them, weighted by importance, determines what to add to the cell memory:

$$\begin{aligned} i_t &= \sigma\left(W_{xi}x_t + W_{hi}h_{t-1} + b_i\right) && \in [0,1]^{d_h}, \\ \tilde{c}_t &= \tanh\left(W_{xc}x_t + W_{hc}h_{t-1} + b_c\right) && \in [-1,1]^{d_h}. \end{aligned}$$

Using these vectors we update the cell state, i.e.

$$c_t = f_t \otimes c_{t-1} \oplus i_t \otimes \tilde{c}_t \quad \in \mathbb{R}^{d_h}$$

The multiplication $f_t \otimes c_{t-1}$ forgets the information from the previous cell state and masks the information specified by the forget gate. Similarly, the $i_t \otimes \tilde{c}_t$ multiplication keeps the new values that the input gate considers important. The sum of the two gives the new cell state, i.e. the updated memory state.

In the last step, the output gate creates the output: a sigmoid layer decides (based on the previous hidden state and the current input) which data from the cell state to return, the tanh layer projects the updated cell state to $[-1,1]$, then the product of them becomes the new hidden state, which passes to the next cell:

$$\begin{aligned} o_t &= \sigma\left(W_{xo}x_t + W_{ho}h_{t-1} + b_o\right) && \in [0,1]^{d_h}, \\ h_t &= o_t \otimes \tanh\left(C_t\right) && \in [-1,1]^{d_h}. \end{aligned}$$

We note that the LSTM cells use shared parameters, i.e. in all time steps we use the same weights and biases. It can therefore handle sequences of different lengths, since as we have seen, the cell state and hidden state are updated every time step.

One of the advantages of LSTM is that because it projects the information to $[-1,1]$ at each step, a stable gradient calculation can be obtained, thus vanishing or exploding gradient prob-

lem can be omitted. On the other hand, due to the input and forget gates, only the important data is always stored, so less memory is needed than if all data were stored. However, the disadvantage is that since each cell needs the output of the previous cell, the computations cannot be parallelised, which can make learning very slow and has been shown to be unusable for huge amount of data, for example in natural language processing.

2.4.1 LSTM with "Peephole Connections"

In 2000, Gers & Schmidhuber [5] proposed an addition to the LSTM cells. The limitation of LSTM is that the gates have no direct connection to the cell state they are supposed to control. They introduced so-called peephole connections adding $w_{c.} \otimes c_t$ to the input, forget, and output gates: The new model:

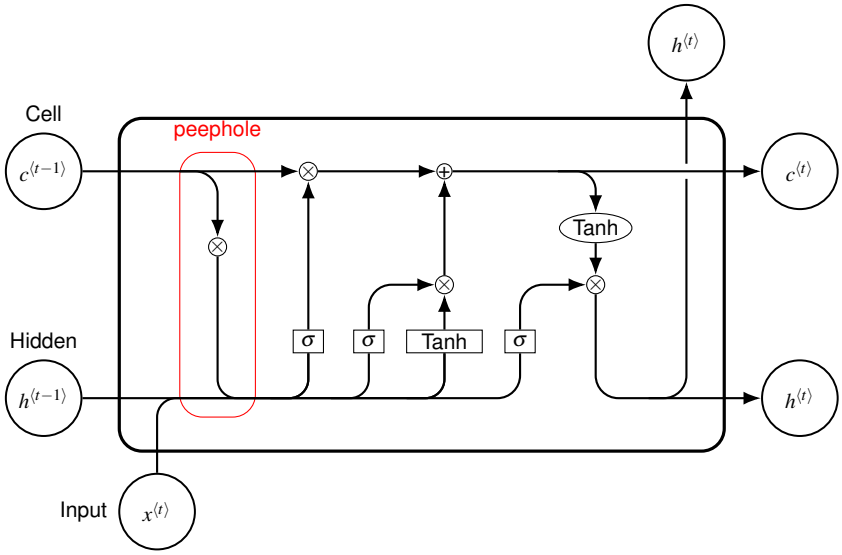


Figure 2.4: The peephole LSTM cell, i.e. the extension of the calssic LSTM with a window through which the cell can take the current cell state into account.

$$
\begin{aligned}
f_t &= \sigma\left(W_{xf}x_t + W_{hf}h_{t-1} + w_{cf} \otimes c_{t-1} + b_f\right) && \in [0,1]^{d_h}, \\
i_t &= \sigma\left(W_{xi}x_t + W_{hi}h_{t-1} + w_{ci} \otimes c_{t-1} + b_i\right) && \in [0,1]^{d_h}, \\
\tilde{c}_t &= \tanh\left(W_{xc}x_t + W_{hc}h_{t-1} + b_c\right) && \in [-1,1]^{d_h}, \\
c_t &= f_t \otimes c_{t-1} \oplus i_t \otimes \tilde{c}_t && \in \mathbb{R}^{d_h}, \\
o_t &= \sigma\left(W_{xo}x_t + W_{ho}h_{t-1} + w_{co} \otimes c_t + b_o\right) && \in [0,1]^{d_h}, \\
h_t &= o_t \otimes \tanh\left(c_t\right) && \in [-1,1]^{d_h},
\end{aligned}
$$

where $w_{c\cdot} \in \mathbb{R}^{d_h}$.

The previous LSTM cell could not extract information from the size of the time lags, however the newly introduced peephole LSTM cells can look at the current cell states allowing the LSTM cell to learn to measure the size of time shifts so it can be used to classify musical material and rhythms.

Chapter 3

Graphs in the machine learning

Graphs are the easiest way to model many things in the world around us. The information that can be described in this way is called unstructured data. This name refers to the fact that they are not stored in the traditional row-column tabular data structure. This structural difference makes the application of deep learning to graphs non-trivial. About a decade ago, a type of neural network that works well on graph data, started to be developed. The class of NNs that can process data represented in a graph is called Graph Neural Networks (GNN). One of the important properties about GNNs is that the nodes of graphs receive information from their neighbours, and the graph is updated based on this information.

To avoid misunderstanding, we point out that although neural networks are also represented as graphs, this is not what the current name refers to. The term graph neural networks, which we have just introduced, means that the input data has a graph structure.

3.1 Convolutional Graph Neural Networks (ConvGNN)

Convolutional Graph Neural Networks are one subclass of GNN, which can be understood as a generalization of Convolutional Neural Networks (CNN) on graph-structured data.

Machine learning models usually use data in matrix format as input, so we need to formulate our graph-based task in a way that is compatible with these models. The adjacency matrix looks like a good starting point, but since there is no basic ordering of the nodes, so we can

encode the relations of a graph with multiple adjacency matrices, but has not guarantee that different matrices will give the same result in a deep learning task. Furthermore, the structure of each graph can vary differently in the number and location of nodes and edges. Given this, we need models, that are permutation invariant with respect to the nodes and work well for data of very different sizes.

3.1.1 Convolution

In the case of 2-dimensional convolution, a sliding window is used to scroll through the data set in tabular form. At each point, the procedure takes the value from the point and its neighbors, and then calculates their weighted average. As shown in Figure 3.1., the size of the window determines the distance from which the information is taken.

One approach for the graph convolution is to generalize this idea. An image can be considered as a specific graph where the adjacent pixels are connected, thus analogously to the 2-dimension, graph convolution can provide information about a node by taking the weighted average of the data of its neighboring nodes. The difficulty can be that since in 2D the neighbors of a point are ordered and have a fixed number, in graphs they can have a variable number of neighbors and no clearly defined order.

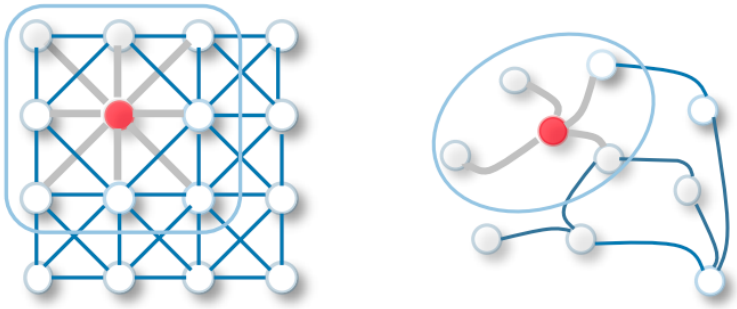


Figure 3.1: The origin of graph convolution. On the left, is the classical convolution used in image processing. A sliding window of fixed size is used to traverse all points, taking into account the neighbors. On the right is the graph convolution defined analogously to the previous one, where the neighborhood is interpreted along the edges. Source: [17]

In recent years, several methods have been developed to determine graph convolution. Two main categories are distinguished, spectral and spatial approaches. The main difference be-

tween them is that Spectral ConvGNN can focus on global patterns from the graph, but Spatial ConvGNN needs more layers to do so. In this thesis, we focus on the spectral-based approach.

3.1.2 Laplacian matrix

In the spectral approach, we assume that $\mathscr{G} = (V,E)$ is an undirected, simple graph, where V denotes the set of vertices with $|V| = n$ and E is the set of edges with $|E| = m$. The distance between two vertices is the length of the shortest path between them. If the edges of the graph are weighted, then $w_{ij} \geq 0$ denotes the weight of the edge $(v_i, v_j) \in E$, where $w : E \to \mathbb{R}_+$.

Let $W \in \mathbb{R}^{n\times n}$ be the adjacency matrix of the graph, i.e.

$$W_{ij} = \begin{cases} w_{ij}, & \text{if } (i,j) \in E, \\ 0, & \text{otherwise.} \end{cases}$$

Let $D \in \mathbb{R}^{n\times n}$ be a diagonal matrix containing the weight sums of the edges connected to the specific vertex, i.e., $D = \operatorname{diag}(d_1, d_2, \ldots, d_n)$, where

$$d_i = \sum_{j=1}^{n} W_{ij}.$$

The resulting matrix D is called the generalized degree matrix.

Now we can define the graph Laplacian $\mathscr{L} = D - W \in \mathbb{R}^{n\times n}$, i.e.

$$\mathscr{L}_{ij} = \begin{cases} d_i & \text{if } i = j, \\ -w_{ij}, & \text{if } (i,j) \in E, \\ 0, & \text{otherwise,} \end{cases}$$

and the normalized Laplacian matrix $\mathscr{L}_n = D^{-\frac{1}{2}}\mathscr{L}D^{-\frac{1}{2}} = I - D^{-\frac{1}{2}}WD^{-\frac{1}{2}}$, where $D^{-\frac{1}{2}}$ is evaluated elementwise.

$$\left(\mathscr{L}_n\right)_{ij} = \begin{cases} 1, & \text{if } i = j, \\ -\frac{w_{ij}}{\sqrt{d_i d_j}}, & \text{if } (i,j) \in E, \\ 0, & \text{otherwise.} \end{cases}$$

The resulting normalized Laplace matrix $\mathscr{L}_n \in \mathbb{R}^{n \times n}$ is a symmetric positive semidefinite matrix, hence the following is true:

- all eigenvalues are real and non-negative: $0 = \lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$,
- it has n different eigenvectors: $u_1 \neq u_2 \neq \cdots \neq u_n$,
- the eigenvectors built an orthonormal basis.

The eigenvalues of the Laplace matrix is called the spectrum of the graph, and it is denoted by $\sigma(\mathscr{L}_n)$. Since $\mathscr{L}_n$ is symmetric, it is orthogonally diagonalizable, i.e., we can take $\mathscr{L}_n = U\Lambda U^T$, where $U = [u_1, \cdots, u_n] \in \mathbb{R}^{n \times n}$ contains the eigenvectors, and $\Lambda = \text{diag}\Big(\big[\lambda_1, \cdots, \lambda_n\big]\Big) \in \mathbb{R}^{n \times n}$ is a diagonal matrix containing the eigenvalues associated with the eigenvectors.

3.1.3 Spectral-based graph convolution

The convolution of the integrable $f, g : \mathbb{R} \to \mathbb{R}$ functions is the function

$$(f * g)(x) = \int_{-\infty}^{\infty} f(t)g(x-t)dt.$$

Suppose that f and g are functions on the vertices of the graph, $f, g : V \to \mathbb{R}$. Then we cannot apply the above definition to the functions f and g since we cannot define the shift $g(x-t)$. We search for an approach that makes the convolution applicable to graphs. From the convolution theorems we know that Fourier and Laplace transformations can convert the convolution operation into a multiplication and we want to transfer this idea to the graph. So we need a mapping acting on a graph, having an inverse, and bringing the convolution into some form of multiplication.

Take the orthogonal eigenvectors of the normalized Laplacian matrix (U), also called as graph Fourier-modes. We denote the single feature vector containing values for all nodes by $x \in \mathbb{R}^n$. Then, by definition, the graph Fourier transform of the input signal x is

$$\mathscr{F}[x] := \hat{x} := U^T x \quad \in \mathbb{R}^n,$$

and the inverse graph Fourier transform is

$$\mathscr{F}^{-1}[x] := U\hat{x} \quad \in \mathbb{R}^n.$$

By definition of the inverse transform, we can write $x = \sum_{i=1}^{n} \hat{x}_i u_i$. We can now define the graph convolution of the functions $x, g : V \to \mathbb{R}$ as

$$x *_{\mathscr{G}} g = \mathscr{F}^{-1}\left(\mathscr{F}(x) \otimes \mathscr{F}(g)\right) = U\left(\left(U^T x\right) \otimes \left(U^T g\right)\right) \in \mathbb{R}^n,$$

where $\otimes$ denotes elementwise multiplication. Let us put the elements of $U^T g$ to the main diagonal of a diagonal matrix, i.e.

$$\varphi = \varphi(g, U) = \operatorname{diag}\left(U^T g\right) \in \mathbb{R}^{n \times n},$$

to rewrite the previous definition to

$$x *_{\mathscr{G}} \varphi = U \cdot \varphi \cdot U^T x \in \mathbb{R}^n.$$

The function φ is called a filter. In graph convolution, the input signal x is projected into the space spanned by the eigenvectors, a filter φ is applied, and the result is transformed back into the original space. In the following, we use the notation $*_{\mathscr{G}}$ for the graph convolution notation.

If more than one feature is interpreted on the vertices, the dimensions are as follows. Let d_x be the number of features, then we stack the feature vectors for all nodes into rows of a matrix to get $x \in \mathbb{R}^{n \times d_x}$, moreover U and φ are unchanged so $x *_{\mathscr{G}} \varphi \in \mathbb{R}^{n \times d_x}$. This operation will give the calculation in a layer of the convGNN.

By stacking layers on top of each other, we also get information for each node from its distant neighbors and it is allowed to have different numbers of incoming and outgoing features in each layer. A layer of graph convolution is defined as follows: at the k-th step (layer), let

- d_{k-1} the number of input features,
- d_k the number of output features,
- $\Theta_{i,j}^{(k)} = \text{diag}(\theta_1, \theta_2, \cdots, \theta_n) \in \mathbb{R}^{n \times n}$ the filter. It is a diagonal matrix containing the learning parameters,
- $H^{(k)} \in \mathbb{R}^{n \times f_k}$ the graph signal,
- $H_i^{(k)} \in \mathbb{R}^{n \times 1}$ the i-th column of the graph signal.

Then

$$H_j^{(k)} = f \sum_{i=1}^{d_{k-1}} \left(U \cdot \Theta_{i,j}^{(k)} \cdot U^T H_i^{(k-1)} \right) \qquad (j = 1, 2, \cdots, d_k),$$

i.e., the j-th column of the graph signal is obtained by performing graph convolution on all columns of the input signal, taking the sum of them, and then applying the activation function f to the resulting vector. Thus the output of the convolution layer is a matrix $(n \times d_k)$. Note that $H^{(0)} = X \in \mathbb{R}^{n \times d_0}$, i.e., in the first step we consider the input signal ($d_0 = d_x$). If multiple layers are built in sequence, a nonlinear transformation is applied to the outputs between each layer, and then this becomes the input to the next layer.

Spectral-based graph convolutions in the literature follow the definition above, the only difference is the method of choosing the filter φ.

3.1.4 ChebNet

Multiplication by the matrix of eigenvectors and eigenvalues is very expensive, having the complexity of $\mathcal{O}\left(n^3\right)$. Using the filter shown below, the computation of a layer can be reduced to $\mathcal{O}\left(|E|\right)$.

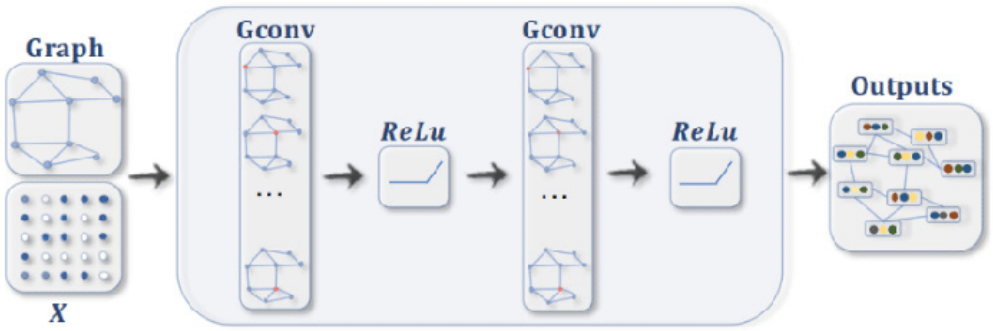


Figure 3.2: A ConvGNN with two graph convolutional layers. The calculation runs from left to right. Given the graph signal X, the convolution step is performed parallel to each vertex of the graph. It is an aggregation of information from all neighbors. An activation function (nonlinear transformation) is performed on the resulting values, yielding the hidden state $H^{(1)}$. This becomes the input of a second layer, on which the same process is performed. Thus by stacking several layers, the vertices indirectly receive messages from distant neighbors. Source: [17]

Chebyshev polynomials are defined by the following recursion.

$$\begin{aligned} T_0(x) &= 1, \\ T_1(x) &= x, \\ T_{k+1}(x) &= 2xT_k(x) - T_{k-1}(x). \end{aligned}$$

Since the Chebyshev polynomials are the basis of the polynomials defined on $[-1,1]$, any k-degree polynomial p can be written as $p(x) = \sum_{i=0}^{k} \alpha_i T_i(x)$. Furthermore, by Weierstrass's approximation theorem, we know that a function can be approximated by polynomials by an arbitrary precision. These imply that the approximation of a function can always be written as a sum of Chebyshev polynomials.

The ChebNet method, which we present now, exploits this and approximates the φ filter by the Chebyshev polynomial of the diagonal matrix of eigenvalues. We perform an element-by-element evaluation of the polynomial on the matrix. As a first step, we need to shift the spectrum of the Laplacian matrix to $[-1,1]$, so take the scaled eigenvalue matrix

$$\tilde{\Lambda} = \frac{2}{\lambda_{\max}} \Lambda - I_n \in \mathbb{R}^{n \times n},$$

where I_n is the identity matrix of size n and $\lambda_{\max}$ is the largest eigenvalue of the Laplacian. Then

$$g_\theta = \sum_{i=0}^{K} \theta_i T_i\left(\tilde{\Lambda}\right) \in \mathbb{R}^{n\times n}$$

is a good approximation of order K ($\theta_i \in \mathbb{R}$). The task is to learn the parameter vector $\theta \in \mathbb{R}^{K+1}$, i.e., to determine the Chebyshev coefficients. Based on the section on spectral graph convolution, the convolution of the graph $x \in \mathbb{R}^n$ with the filter g_θ is

$$x *_{\mathcal{G}} g_\theta = U \cdot \left(\sum_{i=0}^{K} \theta_i T_i\left(\tilde{\Lambda}\right) \right) \cdot U^T x.$$

Assuming $\tilde{\mathcal{L}} = \frac{2}{\lambda_{\max}}\mathcal{L} - I_n$, we can prove by induction that $T_i\left(\tilde{\mathcal{L}}\right) = U T_i\left(\tilde{\Lambda}\right) U^T \in \mathbb{R}^{n\times n}$, thus ChebNet searches for the parameter vector $\theta \in \mathbb{R}^{K+1}$ of the convolution

$$x *_{\mathcal{G}} g_\theta = \left(\sum_{i=0}^{K} \theta_i T_i\left(\tilde{\mathcal{L}}\right) \right) x \qquad \in \mathbb{R}^n.$$

This definition gives a localized filter, i.e., it can provide local information regardless of the size of the graph. Since we have used a polynomial of order K, the value of a point depends only on points at a distance up to K.

Chebyshev Convolution (ChebConv)

Note also here that in case of more features, the dimensions change, i.e. $x \in \mathbb{R}^{n\times d_x}$. In this case we define the Chebyshev Convolution (ChebConv) as follows.

$$x *_{\mathcal{G}} \Theta = \sum_{i=0}^{K} P_i(x) \cdot \Theta_i,$$

where

- $\Theta \in \mathbb{R}^{K\times d_x \times l}$, so $\Theta_i \in \mathbb{R}^{d_x \times l}$,
- $x \in \mathbb{R}^{n\times d_x}$,

- $P_i(x)$ is recursively computed as follows:

$$\begin{aligned} P_o(x) &= x && \in \mathbb{R}^{n \times d_x}, \\ P_1(x) &= \tilde{\mathscr{L}} \cdot x && \in \mathbb{R}^{n \times d_x}, \\ P_i(x) &= 2 \cdot \tilde{\mathscr{L}} \cdot P_{i-1}(x) - P_{i-2}(x) && \in \mathbb{R}^{n \times d_x}, \end{aligned}$$

  where $\tilde{\mathscr{L}} = \frac{2}{\lambda_{\max}} \mathscr{L} - I_n \in \mathbb{R}^{n \times n}$ is the scaled Laplacian.

This implies that $x *_{\mathscr{G}} \Theta \in \mathbb{R}^{n \times l}$, so by choosing l we can influence the result of the convolution.

Chapter 4

The LSTM and graphs meeting

During the literature research, I looked at several models to choose the right one for our problem. In this chapter, I describe some of the papers, where interesting ideas appeared for developing recurrent graph neural networks.

4.1 Road speed prediction

The first framework I studied was the so-called graph LSTM, which was designed for road speed prediction. The research in [16] proposes a combination of GNNs and LSTMs. The presented model takes a graph as an input and output is also given as a graph. They are called linkage graphs which means that they are a directed graph with vertices representing the road segments and edges representing the accessibility of the road segments. If there is a connection from road a with road b, then there is an edge from vertex a to vertex b. In the model, we distinguish features for the vertices, for the edges and for the whole graph.

The graph can be described as follows. Let A_v, A_e, A_u denote the current number of features on vertices, edges, and graphs, respectively. In addition, we use the notations $|V| = n$ and $|E| = m$ as introduced earlier. The graph is a tuple

$$G = \left(V, E, V^{\text{attr}}, E^{\text{attr}}, U^{\text{attr}}\right)$$

where:

- V denotes the vertices of the graph,
- $E = \{(l, s, r)\}$ stores the edges of the graph, where s is the vertex from which the edge l originates, and r is the vertex where the edge arrives,
- $V^{\text{attr}} \in \mathbb{R}^{n \times A_v}$ the node features matrix,
- $E^{\text{attr}} \in \mathbb{R}^{m \times A_e}$ the edge features matrix,
- $U^{\text{attr}} \in \mathbb{R}^{1 \times A_u}$ the graph level features matrix.

During the time steps, the structure of the graph does not change, only the attributes are updated. For the proposed model, first we have to define a so-called GN block, in which the feature matrices are updated in three phases. The dimensionalities of the matrices in the different update steps are summarized in the tables.

1. Update edge features based on start and end vertex. In the second line of the algorithm, [] means that by writing the matrices consecutively after each other to create a larger matrix (i.e. stacking them next to each other).

Algorithm 1 Edge update

1: **for** i in range(m) **do**
2: $\quad \tilde{E}_i^{\text{attr}} = [E_i^{\text{attr}}, V_r^{\text{attr}}, V_s^{\text{attr}}, U^{\text{attr}}]$
3: **end for**
4: $E^{\text{attr}'} = \tanh\left(W_E \cdot \left(\tilde{E}^{\text{attr}}\right)^T + b_E\right)$

$\tilde{E}_i^{\text{attr}} \in \mathbb{R}^{1 \times \overbrace{(A_e + A_v + A_v + A_u)}^{P}}$	$\tilde{E}^{\text{attr}} \in \mathbb{R}^{m \times P}$	$W_E \in \mathbb{R}^{N_E \times P}$
	$E^{\text{attr}'} \in \mathbb{R}^{N_E \times m}$	$\text{agg}^i(E^{\text{attr}'}) \in \mathbb{R}^{1 \times N_E}$

2. Update nodes features based on the edges they receive. In the second row, the rows of the transposed matrix contain the information associated with the edges. So it sums the rows of edges whose endpoint is vertex i. We use the notation $E_k.r$ referring to the arriving vertex r of edge E_k.

Algorithm 2 Node update

1: **for** i in range(n) **do**

2:
$$\text{agg}^i\left(E^{\text{attr}'}\right) = \sum_{j\in\{k|E_k.r=V_i\}} \left(E_j^{\text{attr}'}\right)^T$$

3:
$$\tilde{V}_i^{\text{attr}} = \left[\text{agg}^i\left(E^{\text{attr}'}\right), V_i^{\text{attr}}, U^{\text{attr}}\right]$$

4: **end for**

5: $V^{\text{attr}'} = \tanh\left(W_V \cdot \left(\tilde{V}^{\text{attr}}\right)^T + b_V\right)$

$\tilde{V}_i^{\text{attr}} \in \mathbb{R}^{1\times\overbrace{(N_E+A_v+A_u)}^{Q}}$	$\tilde{V}^{\text{attr}} \in \mathbb{R}^{n\times Q}$	$W_v \in \mathbb{R}^{N_V\times Q}$
	$V^{\text{attr}'} \in \mathbb{R}^{N_V\times n}$	

3. Updates the graph-level features based on all vertices and edges. As in the previous step, the rows of the transposed matrices contain the information from the edges or vertices. The sum is used to sum the rows of the matrix, i.e. it sums all the edge/peak information.

Algorithm 3 Global update

1: $\tilde{U}^{\text{attr}} = \left[\sum_{j=1}^{m}\left(E_j^{\text{attr}'}\right)^T, \sum_{j=1}^{n}\left(V_j^{\text{attr}'}\right)^T, U^{\text{attr}}\right]$

2: $U^{\text{attr}'} = \tanh\left(W_u \cdot \left(\tilde{U}^{\text{attr}}\right)^T + b_U\right)$

$\tilde{U}^{\text{attr}} \in \mathbb{R}^{1\times\overbrace{(N_E+N_V+A_u)}^{R}}$	$W_U \in \mathbb{R}^{N_U\times R}$
$U^{\text{attr}'} \in \mathbb{R}^{N_U\times 1}$	

So the algorithm of the complete GN block can be described as follows.

Algorithm 4 GN block

Input $G = (V, E, V^{\text{attr}}, E^{\text{attr}}, U^{\text{attr}})$

1: Edge update

2: Node update

3: Global update

Output $G' = \left(V, E, \left(V^{\text{attr}'}\right)^T, \left(E^{\text{attr}'}\right)^T, \left(U^{\text{attr}'}\right)^T\right)$

It is observed that although concatenate increases the size of the matrices, the GN block reduces it. From the above, by choosing N_E, N_V, N_U appropriately, we can control the number of output features.

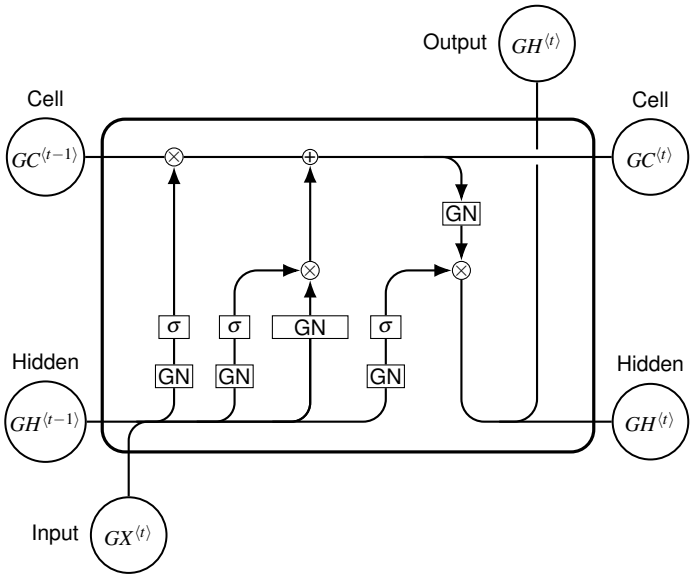


Figure 4.1: The GN-LSTM cell. In each gate we concatenate the given graphs and then execute a GN-block on them. By concatenation of graphs we mean the new graph created by writing the corresponding feature matrices side by side. The ather operations are done in the same way as in classical LSTM, i.e., element by element.

The model proposed in [16] combines the GN block with the classical LSTM as shown in Fig.4.1. The resulting cell is called GN-LSTM. It can be seen that there is one hidden and one cell state, as in the classical LSTM: GH_t and GC_t. These states and the input are all represented as graphs. The GN-LSTM cell formally looks like this:

$$
\begin{aligned}
f_t &= \sigma\Big(\mathrm{GN}_f\big([\mathrm{GH}_{t-1}, \mathrm{GX}_t]\big)\Big), \\
i_t &= \sigma\Big(\mathrm{GN}_i\big([\mathrm{GH}_{t-1}, \mathrm{GX}_t]\big)\Big), \\
\tilde{\mathrm{GC}}_t &= \mathrm{GN}_c\big([\mathrm{GH}_{t-1}, \mathrm{GX}_t]\big), \\
\mathrm{GC}_t &= f_t \otimes \mathrm{GC}_{t-1} \oplus i_t \otimes \tilde{\mathrm{GC}}_t, \\
o_t &= \sigma\Big(\mathrm{GN}_o\big([\mathrm{GH}_{t-1}, \mathrm{GX}_t]\big)\Big), \\
\mathrm{GH}_t &= o_t \otimes \mathrm{GN}_o\big([\mathrm{GC}_t]\big),
\end{aligned}
$$

where

- $G_M = [G_1, G_2]$ means concatenating two graphs, i.e. stacking their feature matrices after each other, e.g.: $G_M.V^{\text{attr}} = [G_1.V^{\text{attr}}, G_2.V^{\text{attr}}]$,
- $\hat{G} = \sigma(G)$ the sigmoid function on the graph means the sigmoid function of features matrices, i.e. $\hat{G}.V^{\text{attr}} = \sigma\left(G.V^{\text{attr}}\right)$.

In summary, a single GN-LSTM cell execution from the input graph based on the hidden and cell graphs yields an output graph, i.e.

input:

$$
\mathrm{GX}_t = \left(V, E, \mathbb{R}^{n \times A_v}, \mathbb{R}^{m \times A_e}, \mathbb{R}^{1 \times A_u}\right),
$$

hidden:

$$
\mathrm{GH}_{t-1} = \left(V, E, \mathbb{R}^{n \times N_V}, \mathbb{R}^{m \times N_E}, \mathbb{R}^{1 \times N_U}\right),
$$

output:

$$
\mathrm{GH}_t = \left(V, E, \mathbb{R}^{n \times N_V}, \mathbb{R}^{m \times N_E}, \mathbb{R}^{1 \times N_U}\right).
$$

The cell graph is not written separately, because its dimensionality is the same as the hidden ones. As mentioned before, the graph structure is not changed, only the feature matrices are updated.

4.2 PM2.5 forcasting

Now we see a graph-based Long Short-Term Memory (GLSTM) model, which was proposed in [14] to predict PM2.5 concentration in air. The aim was to create a model that could predict all stations together, without the need to build separate models for each station.

In this model, we are not working with a known graph structure. We are given the monitoring stations, which are treated as if they were a complete, directed graph. The nodes of the graph are the air quality stations, and the edges represent relations between the stations. The exact relations, A_{adj} the matrix, are discovered during the training of the model. It is also an internal parameter that we want to train the model to determine.

The matrix A_{adj} describes the influence of the vertices on each other. The experiences have shown that the adjacency matrix is not symmetric, i.e., the effect of v_j on vertex v_i is not the same as the effect of v_i on vertex v_j, but this can be handled with directed graphs.

In the following, $X_t \in \mathbb{R}^{d_x \times n}$ denote the air quality features d_x measured at station n at time t, and denote $a_{k,v} = A_{\text{adj}}(k,v)$ the effect of node k on node v. For each vertex, we assign a new LSTM cell, but we also use information from the other vertices, based on the matrix A_{adj}. Note that if we decrease the number of vertices to one, we get back the classical LSTM. The forget gate only makes a decision based on the station's hidden state, but the other gates all take into account the $a_{k,v}$ effects. Formally, a GLSTM cell for v at time t is the following:

$$
\begin{aligned}
f_t^v &= \sigma\left(W^f x_t^v + U^f h_{t-1}^v + b^f\right) && \in [0,1]^{d_h},\\
i_t^v &= \sigma\left(W^i x_t^v + U^i \sum_{k=1}^{n} a_{k,v} h_{t-1}^k + b^i\right) && \in [0,1]^{d_h},\\
\tilde{c}_t^v &= \tanh\left(W^c x_t^v + U^c \sum_{k=1}^{n} a_{k,v} h_{t-1}^k + b^c\right) && \in [-1,1]^{d_h},\\
c_t^v &= f_t^v \otimes \sum_{k=1}^{n} a_{k,v} c_{t-1}^k + i_t^v \otimes \tilde{c}_t^v && \in \mathbb{R}^{d_h},\\
o_t^v &= \sigma\left(W^o x_t^v + U^o \sum_{k=1}^{n} a_{k,v} h_{t-1}^k + b^o\right) && \in [0,1]^{d_h},\\
h_t^v &= o_t^v \otimes \tanh(c_t^v) && \in [-1,1]^{d_h},
\end{aligned}
$$

where

- $x_t^v \in \mathbb{R}^{d_x}$ is the input vector,
- $W^f, W^i, W^o, W^c \in \mathbb{R}^{d_h \times d_x}$ are the parameter matrices for controlling the input vector,
- $U^f, U^i, U^o, U^c \in \mathbb{R}^{d_h \times d_h}$ are the parameter matrices for controlling the hidden state,
- $b^f, b^i, b^o, b^c \in \mathbb{R}^{d_h}$ are the bias vectors.

The complete network from unrolling the above described GLSTM cell is shown in Figure 4.2. This is a fully connected network, where the number of layers is determined by the time period we want to learn over, and the number of cells within a layer is determined by the number of stations. Within the cells, the aggregation of a cell and hidden states is done based on their corresponding node effects. This step is marked in red in the description of GLSTM and is the reason for the fully connectedness. Then, as in the classical LSTM-RNN, the new cell and hidden states coming out of the cells are passed on to the next layer, where the same process takes place again.

The $W^{\cdot}, U^{\cdot}, b_{\cdot}$ parameters are shared in time, as in LSTM-RNN, and now they are also shared in space. Furthermore, all GLSTM cell has one A_{adj} matrix, which is learned over time, so A_{adj} is also shared in time.

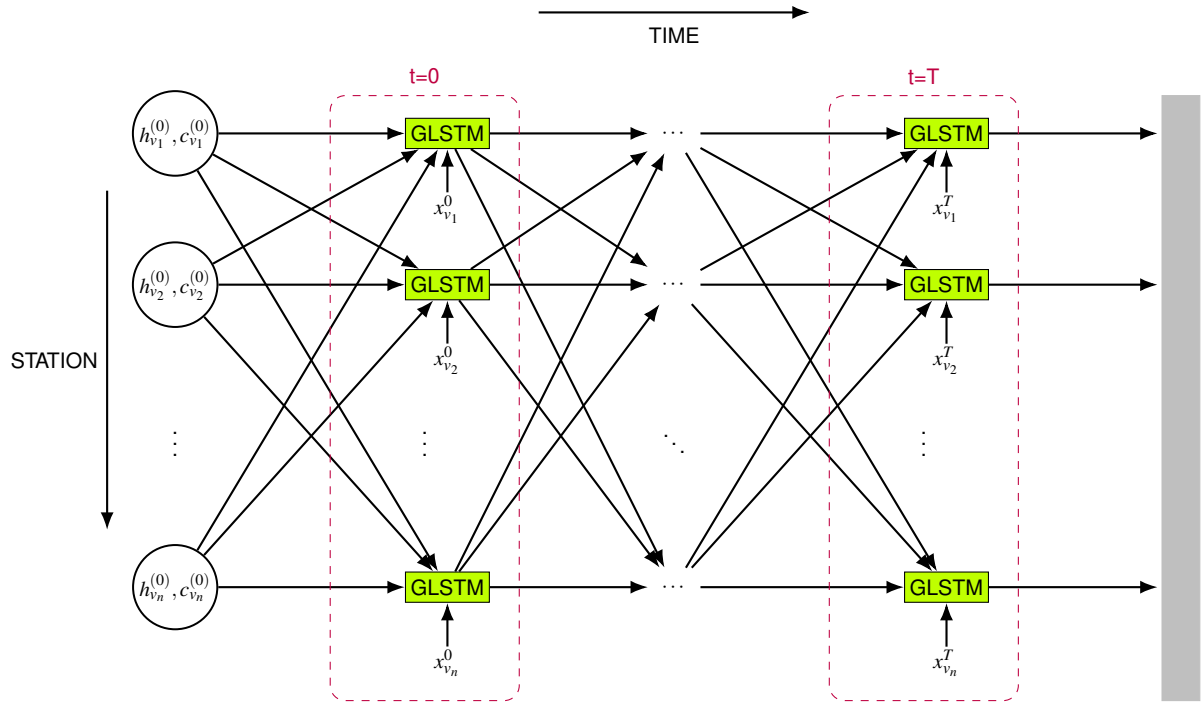


Figure 4.2: The GLSTM is a time-space model built from cells. It is a fully connected network, where the number of layers is determined by the learning time interval and the number of cells within each layer is determined by the number of stations. Information is transmitted between layers by cells and hidden states.

As we discussed earlier, this model does not know the structure of the graph in advance, and learns this by learning the effects of the vertices on each other. From a point of view of the air quality, this is understandable, since the spread of pollutants can happen in any direction and is always influenced by current environmental factors. It is also possible that the pollutant bypasses a closer station because of a high mountain and only reappears at a distant station.

In our task, however, the riverbed pre-fixes the structure of the graph, and such cases of a flood wave skipping a station should not exist. Therefore we decided to investigate further models.

4.3 Graph Convolutional Recurrent Network (GCRN)

The Graph Convolutional Recurrent Network connects the convolutional neural networks (CNN) on graphs and the classical recurrent neural networks (RNN). It makes it possible to make a temporal prediction based on spatial data.

Two models were proposed by Seo et al. in [15], both uses peephole LSTM and the Chebyshev graph convolution. The only difference is how the graph convolution applied in the calculations.

4.3.1 Model 1.

The first model proposed simply places the LSTM on the graph CNN. As shown in Figure 4.3a, it first performs a graph convolution on the input data, and the result is placed in the peephole LSTM. In this way, it extracts the spatial information first, and then calculates on the temporal information. Formally we have

$$
\begin{aligned}
x_t^{\mathrm{CNN}} &= x_t *_{\mathscr{G}} W^{\mathrm{CNN}}, \\
f_t &= \sigma\left(x_t^{\mathrm{CNN}} W_{xf} + h_{t-1} W_{hf} + w_{cf} \otimes c_{t-1} + b_f\right) && \in [0,1]^{n \times d_h}, \\
i_t &= \sigma\left(x_t^{\mathrm{CNN}} W_{xi} + h_{t-1} W_{hi} + w_{ci} \otimes c_{t-1} + b_i\right) && \in [0,1]^{n \times d_h}, \\
\tilde{c}_t &= \tanh\left(x_t^{\mathrm{CNN}} W_{xc} + h_{t-1} W_{hc} + b_c\right) && \in [-1,1]^{n \times d_h}, \\
c_t &= f_t \otimes c_{t-1} + i_t \otimes \tilde{c}_t && \in \mathbb{R}^{n \times d_h}, \\
o_t &= \sigma\left(x_t^{\mathrm{CNN}} W_{xo} + h_{t-1} W_{ho} + w_{co} \otimes c_t + b_o\right) && \in [0,1]^{n \times d_h}, \\
h_t &= o_t \otimes \tanh\left(c_t\right) && \in [-1,1]^{n \times d_h},
\end{aligned}
$$

where

- $W^{\mathrm{CNN}} \in \mathbb{R}^{K \times d_x \times d_x}$ is the Chebishev coefficient in case of K supported model,
- $x_t \in \mathbb{R}^{n \times d_x}$ contain the measured values of the features of the nodes (assuming n nodes) at time t,

- $x_t *_{\mathcal{G}} W^{\text{CNN}}$ means the Chebyshev Convolution of x_t with W^{CNN} filter, so

$$x_t *_{\mathcal{G}} W^{\text{CNN}} = \sum_{i=0}^{K} P_i(x_t) \cdot W_i^{\text{CNN}} \qquad \in \mathbb{R}^{n \times d_x}$$

  since

  - $P_i(x_t) \in \mathbb{R}^{n \times d_x}$ is computed recursivly, as seen in ChebConv definition (3.1.4),
  - $W_i^{\text{CNN}} \in \mathbb{R}^{d_x \times d_x}$ is the convolutional kernel,

- the weights are $W_{h.} \in \mathbb{R}^{d_h \times d_h}$, $W_{x.} \in \mathbb{R}^{d_x \times d_h}$, $w_{c.} \in \mathbb{R}^{n \times d_h}$, $b_. \in \mathbb{R}^{n \times d_h}$.

4.3.2 Model 2

We now use the peephole LSTM as a basis and replace the multiplication by W matrices by graph convolution with the Chebyshev polynomial. Hereafter I will refer to the model as Cheb-convLSTM:

$$\begin{aligned}
f_t &= \sigma(x_t *_{\mathcal{G}} W_{xf} + h_{t-1} *_{\mathcal{G}} W_{hf} + w_{cf} \otimes c_{t-1} + b_f) && \in [0,1]^{n \times d_h}, \\
i_t &= \sigma(x_t *_{\mathcal{G}} W_{xi} + h_{t-1} *_{\mathcal{G}} W_{hi} + w_{ci} \otimes c_{t-1} + b_i) && \in [0,1]^{n \times d_h}, \\
\tilde{c}_t &= \tanh(x_t *_{\mathcal{G}} W_{xc} + h_{t-1} *_{\mathcal{G}} W_{hc} + b_c) && \in [-1,1]^{n \times d_h}, \\
c_t &= f_t \otimes c_{t-1} + i_t \otimes \tilde{c}_t && \in \mathbb{R}^{n \times d_h}, \\
o_t &= \sigma(x_t *_{\mathcal{G}} W_{xo} + h_{t-1} *_{\mathcal{G}} W_{ho} + w_{co} \otimes c_t + b_o) && \in [0,1]^{n \times d_h}, \\
h_t &= o_t \otimes \tanh(c_t) && \in [-1,1]^{n \times d_h},
\end{aligned}$$

where $W_{h.} \in \mathbb{R}^{K \times d_h \times d_h}$ and $W_{x.} \in \mathbb{R}^{K \times d_x \times d_h}$ are the Chebishev coefficients in the case of K supported model. $x_t *_{\mathcal{G}} W_{xf}$ means the the Chebyshev Convolution of x_t with W_{xf} filter, so like the previous one

$$x_t *_{\mathcal{G}} W_{xf} = \sum_{i=0}^{K} P_i(x_t) \cdot W_{xf}^i \qquad \in \mathbb{R}^{n \times d_h}$$

where, $P_i(x_t) \in \mathbb{R}^{n \times d_x}$ and $W_{xf}^i \in \mathbb{R}^{d_x \times d_h}$.

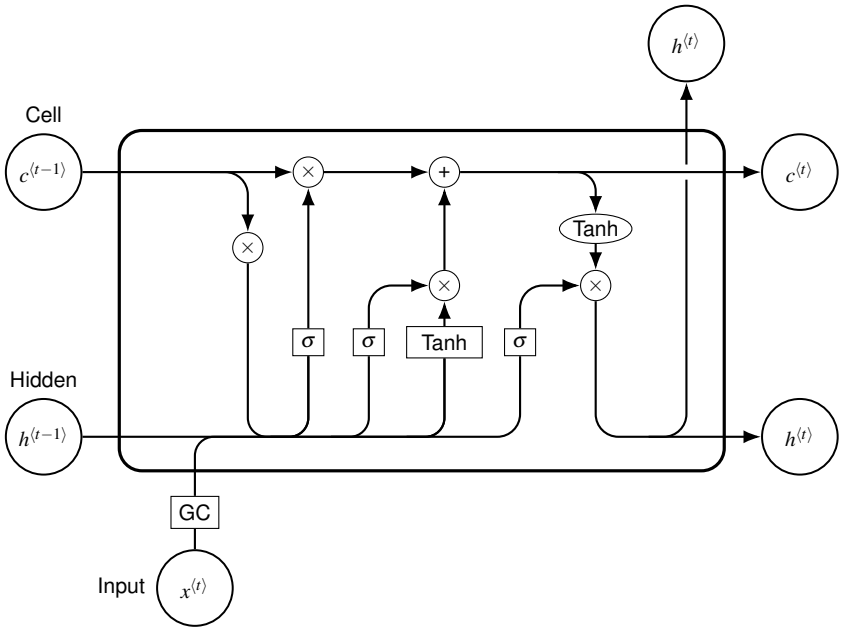
As you can see in Figure 4.3b. graph convolution happens inside each gate. It uses both spatial and temporal information.

4.3.3 The performance of the models

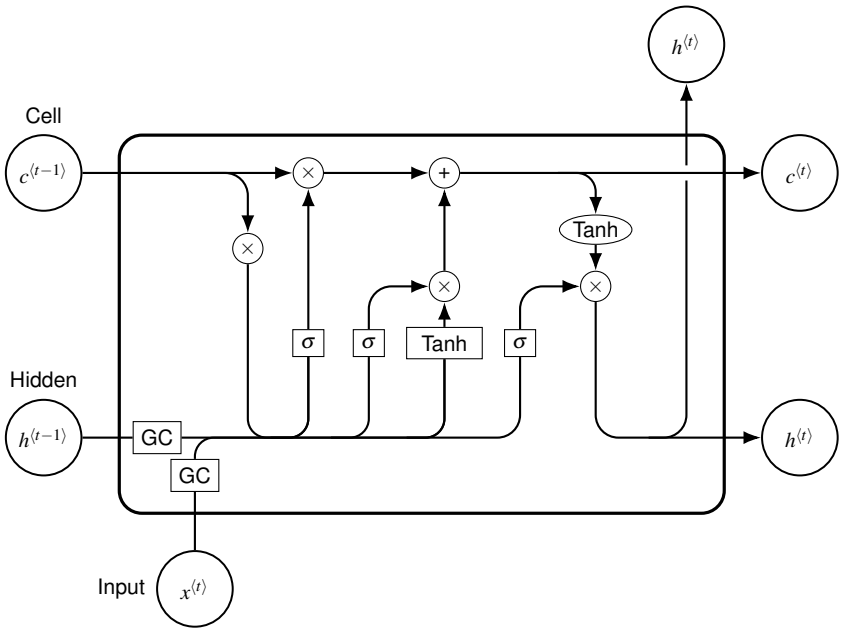
The authors of the paper tested the models on two problems. The first dataset is moving-MNIST, which is a video sequence of handwritten digits. Each video is a sequence of 20 frames representing the movement of two digits. The task here is to predict the second 10 frame based on the first 10 frame. The other data set is the Penn Treebank (PTB) corpus, which contains millions of words and text fragments. The task was to tag each word with part-of-speech tags. On both data sets, graphs were created using k-nearest neighbors algorithm (shortly k-NN). For moving-MNIST, Model 2 performed better than the combination of LSTM with CNN. Moreover, even in runtime, a significant improvement was observed. It may be surprising given that the data are frames for which the CNNs were developed. However, this is where the advantage of graph spectral filters is seen, in that orders of magnitude less parameter learning is required. For this data set, Model 1 was not tested.

However, for linguistic modeling, Model 1 significantly outperforms Model 2. The authors attribute this to the large increase in the size of the hidden state due to the vocabulary size.

Since Cheb-convLSTM exploits both temporal and spatial data in parallel, and we do not expect dimensions as large as the vocabulary, we decided to use this model for the prediction of flood waves.



(a) Model 1 cell



(b) Model 2 cell

Figure 4.3: As shown the Model 1 first performs a graph convolution on the input data and the result goes into the peephole LSTM. This means that it first extracts the spatial information and then the temporal information. In contrast, Model 2 performs the graph convolution inside the gates. Thus, spatial and temporal information are used simultaneously.

Chapter 5

The programming framework used

5.1 PyTorch Geometric

PyTorch is a machine learning framework originally developed in Python and released in 2016. It is well suited for applications such as language processing but is now also the basis for many deep learning softwares. The `torch.Tensor` class defined here, made it possible to learn on the GPU. Similar to NumPy Arrays, it can be used to handle multidimensional matrices. Note that here the matrix is a rectangular array of numbers.

PyTorch Geometric (PyG) is a PyTorch subclass library. It is useful for building and teaching Graph Neural Networks (GNNs). It contains many published and applied methods and convolution layers. For example, the previously described ChebNet method, based on Chebishev polynomials, is available as `ChebConv`. It is important to note that due to Python's summarization, the K supported filter uses nodes with distance at most $K-1$. The `ChebConv` is a class based on `MessagePassing`, which is described below.

In PyG, there are several classes for storing and manipulating data. Among them, we are introduced to the `torch_geometric.data.Data` class. This is used by `ChebConv` and we will see it pop up elsewhere later.

5.1.1 `MessagePassing`

Previously, the properties of graphs were described by matrices. However, there is no guarantee that these matrices have good properties, e.g. for a directed graph, the adjacency matrix may not be symmetric, then the normalized Laplace matrix is not symmetric either, thus we cannot take an eigenvalue decomposition. For pairwise graphs, it is usual to plot the adjacency matrix not for all pairs of points, but only for those of different classes.

Let us look at the problem from a different approach. The neighbours of each vertex can be easily read off in the above cases. Let's go through the list of vertices and update each vertex feature based on its neighbors. Our algorithm consists of the following steps:

1. The feature vector for vertex i is given the information from the neighboring j vertex based on the feature vectors of the vertices. Optionally, the edge feature vector between them also plays a role. We take a function of the feature vectors.
2. We aggregate the information obtained from all neighbors of vertex i.
3. We update the feature vector of vertex i: a function of the previous feature vector of i and the new information vector.

The above three steps are applied to all vertices of the graph. This is an update of one layer.

This can be formally described as follows:

$$x_i^{(k)} = \gamma^{(k)}\left(x_i^{(k-1)}, \bigoplus_{j \in \mathcal{N}(i)} \phi^{(k)}\left(x_i^{(k-1)}, x_j^{(k-1)}, e_{ji}\right)\right),$$

where

- $x_i^{(k-1)} \in \mathbb{R}^F$ the feature vector of the i node on the $(k-1)$ layer,
- $e_{ji} \in \mathbb{R}^D$ an optional feature vector of the edge from vertex j to vertex i,
- $\bigoplus$ is a differentiable, permutation-invariant function, e.g. sum, mean, or max,
- $\phi^{(k)}$ and $\gamma^{(k)}$ are differentiable functions, which can be different for different layers,
- $\mathcal{N}(i)$ means the neighbours of i vertex.

A great advantage of the resulting algorithm is that it is permutation invariant, i.e., insensitive to the numbering order of the vertices. This was still a problem for neighborhood matrices. This procedure called `MessagePassing`, is a core class in PyTorch Geometric. Many graph convolutional layers in PyG are also implemented on this basis.

5.1.2 `torch.geometric.data.Data`

The class `Data` in the `torch_geometric.data` library can be used to describe graphs. It mimics the behavior of a Python dictionary. It can contain vertex, link, and graph attributes. It usually has the following parameters:

- `data.x`: Node feature matrix with shape `[num_nodes, num_node_features]`.
- `data.edge_index`: Graph adjacency matrix in COO (COOrdinate) format with shape `[2, num_edges]`. Take the coordinates of the 1's in the adjacency matrix. The first vector stores the row index of 1's, and the second vector is the column index.

	v_1	v_2	v_3
v_1	1	0	1
v_2	1	0	0
v_3	0	1	1

$$[[0,0,1,2,2],$$
$$[0,2,0,1,2]]$$

  Note: in case of undirected edges, both directions must be added to the edge description.

- `data.edge_attr`: Edge feature matrix with shape `[num_edges, num_edge_features]`.
- `data.y`: The target for which we are training the model to predict. The shape depends on the problem.

Depending on the task, there may be other parameters, such as the number of classes for classification.

5.2 PyTorch Geometric Temporal

In PyG, we dealt with the spatial location of data. This was extended with temporal variables in the PyTorch Geometric Temporal library. It is the first open-source library for temporal deep learning on geometric structures. It can be used on both static and dynamically varying graphs. It is now suitable for processing time series-type data taking into consideration geometric location.

In PyTorch Geometric Temporal, data sets are handled by so-called data set iterators. Spatio-temporal data sets are stored as a series of discrete-time snapshots. There are three types of `TemporalSignalIterators`:

- `StaticGraphTemporalSignal`,
- `DynamicGraphTemporalSignal`,
- `DynamicGraphStaticSignal`.

For us, only the first one is important, which is designed for with constant time difference, dynamically changing time signals interpreted on a static graph. The iterator returns a snapshot of the period (day, week) at each step. The type of snapshots are the `Data` from PyTorch Geometric. This means that it contains the node attributes, edge pairs and edge attributes, target values and additional optional attributes, at the given point in time. Since the graph is static, the edge and edge attributes are the same for all snapshots. Between two snapshots, the attributes and edge characteristics may also change.

The library contains more interesting datasets and several Graph Convolutional Layers that can be applied to them. For example, we can see an Epidemiological Prediction on the Hungarian Chickenpox Cases dataset to predict weekly cases. We can make a regression from the hourly energy production of windmills in a European country or we can read a case study of a recurrent graph neural network to predict daily views of Wikipedia pages. Reviewing these helped us choose the right model for us. In the end we decides to use the `GConvLSTM` which is the Chebyshev Graph Convolutional Long Short Term Memory Cell. It is the implementation of the Cheb-convLSTM model from the 4. chapter.

Chapter 6

Water level forecast for Szeged

In this chapter we present the structure and results of our model. The code for our work is available at `https://github.com/bjankus/water-management-GConvLSTM/`.

6.1 Data, data processing

The data provided to me includes water level data from Szeged and 11 more stations from 1951 to 2019. Most of the measurement stations are located along the Tisza, but we also have observations from stations on tributaries. The data of the water level measurements are available in centimetres and tabulated format. The rows are the measurement dates and the columns are the time series for each station. I had already obtained a cleaned-up data table with no missing cells resulted in the ongoing research.

My first task was to construct the graph of the measurement stations. For this, I only took into consideration the geographic location shown in the Figure 6.1. On the map, I marked Szeged by red, the stations along the Tisza by blue, the stations along the Körös by green, and the stations along the Maros by yellow. Note that the data include a station south of Szeged, which may seem surprising at first glance, but due to the phenomenon observed by water experts, whereby if the water flow at a downstream station increases significantly (either due to precipitation or a tributary connecting to the river), this can cause the water level at the

upstream station to swell. Therefore, Zenta cannot be ignored when making a forecast for Szeged.

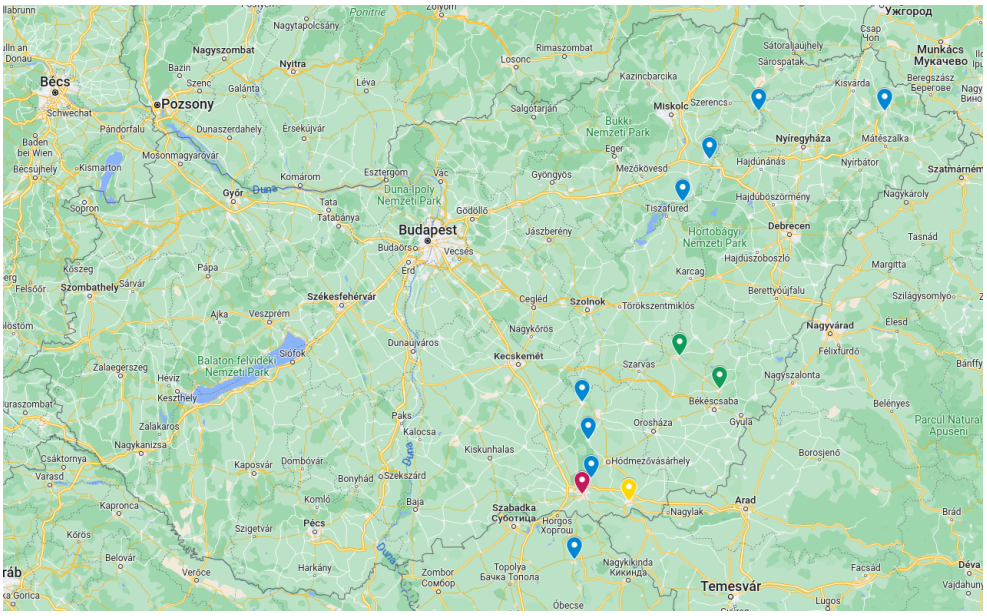


Figure 6.1: The geographical location of the measuring stations. Szeged is marked by red, the stations along the Tisza are marked by blue, the stations along the Körös by green, and the stations along the Maros by yellow.

I have considered two stations adjacent to the graph if they are adjacent along rivers, and the edges are oriented according to the streaming direction of the rivers. Thus I obtained the graph shown in 6.2. Because of the phenomenon of back-swelling discussed above, it might be worthwhile to introduce a back-edge at the junction of the tributaries and examine the resulting graph, but this will not considered now.

Before the data is loaded into the model, a preliminary data processing step is performed. At this point, we need to set the period we want to train on. We identify which column in the table belongs to which station and calculate the standard deviation and mean values per station, i.e., per column, on the training dataset. This will be needed to standardise the data. The mean and standard deviation values will be saved so that we can transform the predictions back into water level values after the training. Next, in `DatasetLoader`, a `self._dataset`, protected variable is created. This is a variable of type `dictionary`, i.e., it contains key-value pairs:

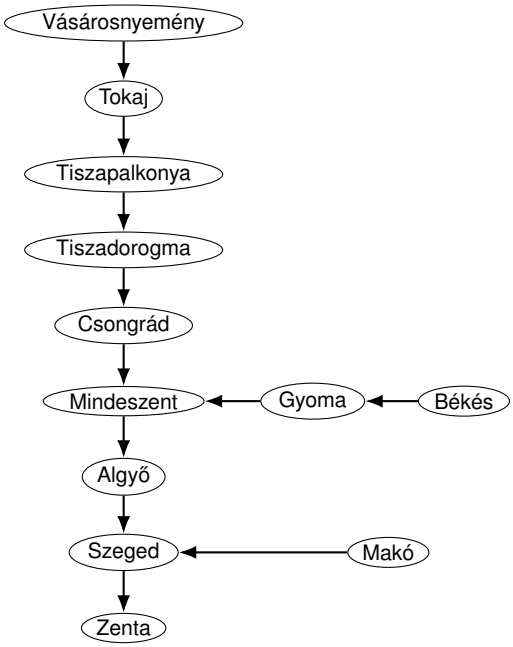


Figure 6.2: Graph of the measuring stations.

- `self._dataset["edges"]` is a list of edges based on the graph 6.2. An edge is a two-element list of names of neighbouring cities.
- `self._dataset["node_ids"]` is also a dictionary containing the pairs of city names and corresponding column numbers. It creates the link between the graph and the dataset.
- `self._dataset["FX"]` is the `numpy.array` type of data table containing the standardized data. Note that the standardization parameters are calculated for the train set only, but the test dataset will also be standardized with these values to avoid look-ahead bias.

The function `get_dataset` function generates a `dataset` of type `StaticGraphTemporal` based on the `self._dataset`, depending on the number of days in the past and the number of days to be forecast. As discussed earlier, the edges are already in COO format, and since the graph is unweighted, each edge is assigned 1 as weight identically. Since we are talking about a static graph, the `dataset._edges` and `dataset._edge_weights` are the same for all snapshots. And the snapshots are contained in `dataset.features` and

`dataset.targets`. Using Fig. 6.3, it is easier to understand what these snapshots look like.

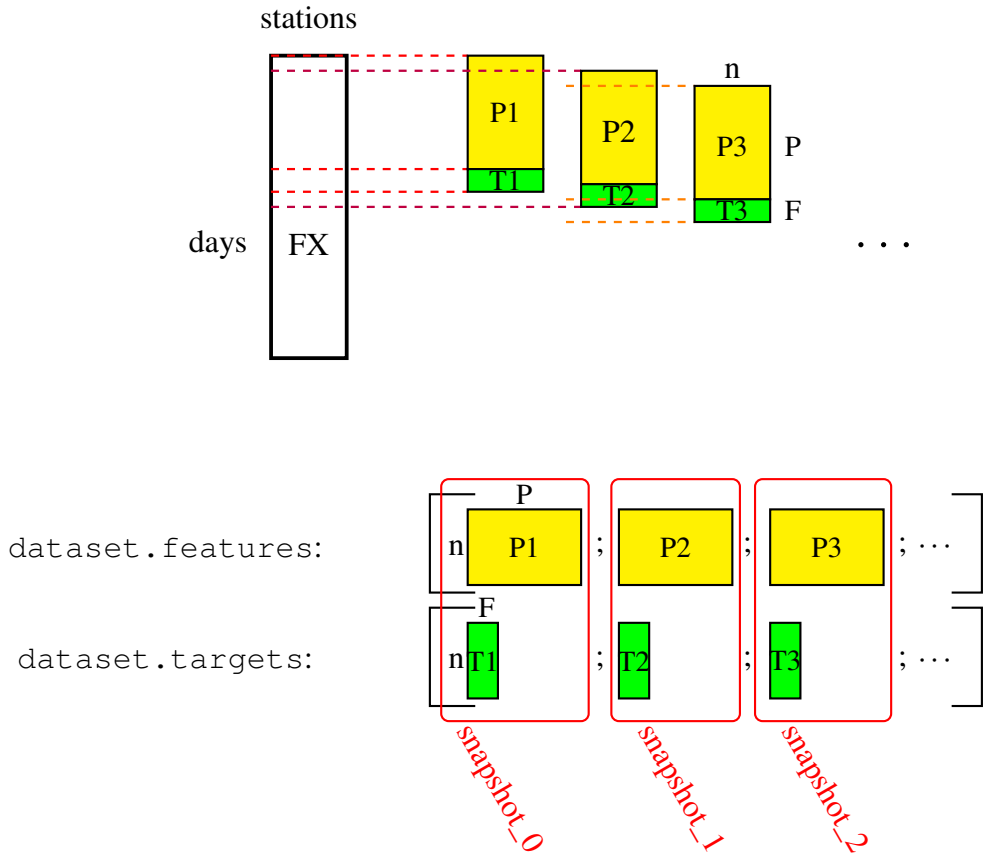


Figure 6.3: The `get_dataset` function takes snapshots. It first splits the `FX` table containing standardised data into pairs of past and target tables. Their dimensions are the number of days in the past (P) times the number of stations (n), and the number of days to be forecast (F) times n, respectively. The tables are not disjoint, we use all possible tables of size $(P+F)\times n$. The snapshots then store the transpose of the tables. The snapshots also include the `_edges` and the `_egde_weights`, but since these are the same for all snapshots, they are not shown in the figure.

6.2 Model

Similar to the Thesis [10], the model I created is based on the Encoder-Decoder architecture, which is a frequently used setting for RNNs to map an input sequence to an output sequence

of arbitrary size. The model consists of two parts. In the first phase, the encoder receives the past data as input to extract the most useful information. The resulting output is received by the decoder part, which aims to predict a sequence for the future values. In [10], both the encoder and the decoder part consist of a multi-layer stack of LSTMs. In our model, we exploit that the encoder and decoder can have different structures, since we want to extract relevant information from the graph structure and connections, thus we use a graph-based model for the encoder part. In the decoder, the graph structure is no longer needed, so we used a classical LSTM cell. In this thesis we used only one layer from each cells, but stacking can be considered in the follow-up of this research.

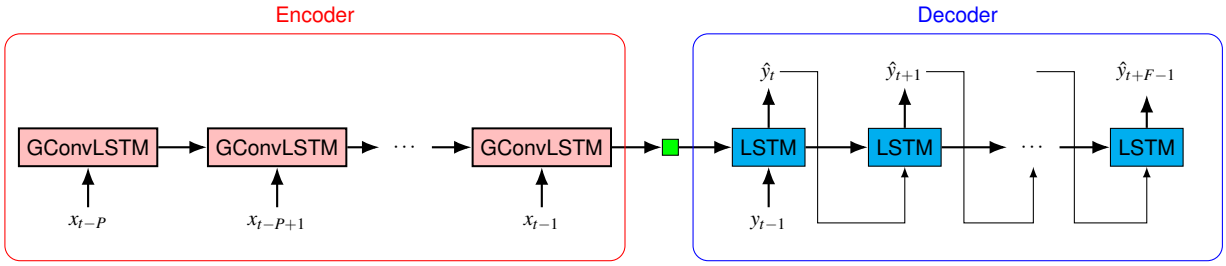


Figure 6.4: The Encoder-Decoder architecture of my model

Figure 6.4 shows the structure of the model. For the encoder part, we used the cells implemented in `PyTorch Geometric Temporal` defined as `GConvLSTM`, which was described in 4.3.2. We apply this P times, where P is the number of available past data.

The inputs are the feature vectors, in this case, the water level for each station for a given day, i.e. $x_{t-i} \in \mathbb{R}^n$. The outputs are the hidden and cell-state, of size $\mathbb{R}^{n \times d_h}$, which are passed to the decoder. Since we only want to predict Szeged in the decoder part, we need to reduce the dimension of the hidden and cell state. Here we considered two possible methods. In the first case we extract from the matrix the set of information associated with Szeged, on the other hand, as a second approach we use a `Linear` (i.e. fully connected) layer to compress all the information. In the end, we chose the first variant, i.e. the hidden and cell state of size $\mathbb{R}^{1 \times d_h}$, formed by a single row, are fed into the decoder. This transformation is indicated by the green square between the encoder and decoder. The input of the first step of the decoder is the last target value of the past for Szeged since it is available, i.e. $y_{t-1} \in \mathbb{R}$. In later steps, the input is always the value predicted in the previous time step and these steps are performed F times, where F is the number of forecast days.

6.3 Training

In this section, we introduce the details about searching for the best internal parameters for the model on the selected training data for a specific hyperparameter settings. After training, the internal parameters are extracted so that we can test the model performance on the test data with these parameter settings. In the model presented in the previous section, the hyperparameters were the number of days used from the past (P), the dimensionality of the hidden state (d_h), and the distance of the neighbors considered (K). For training, we used data from January 1, 1951, to April 26, 2004, and trained for a 7-day ahead forecast.

During training, the data are inputted several times (number of epochs) over and over again, and thus we search for the best parameter setting for a defined loss function. For regression models, the most commonly used loss function is the Root Mean Square Error (RMSE),

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

but our model uses the rootless Mean Square Error (MSE)

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

based on a model for predicting chickenpox cases, where $\hat{y}$ denotes the predicted value.

The Adam optimizer algorithm was used for the optimization procedure, which is an extension of SGD and it was introduced in 2015 [4]. It combined the advantages of two previous SGD algorithms to obtain an optimizer with low memory requirements that can be efficiently applied to big data. While classical SGD applies the same learning rate throughout the learning process, Adam computes a unique learning rate for different parameters and updates it continuously throughout the learning process.

Unfortunately, the `StaticGraphTemporalSignal` is not compatible with groupping into mini-batches during the training, so we cannot take enough advantage of the mini-batch stochastic gradient descent. The update of the weights, i.e. the gradient calculation, is always performed only after the entire training set has been run through, causing a slowdown in the learning of the model.

As mentioned earlier, we did not use a systematical tuning algorithm due to the V-RAM limitations at our disposal. We tested the different parameter settings manually one by one. The following table contains the parameters of the models I tested:

Name	number of days in the past (P)	hidden size (d_h)	K
Model 1	5	50	1
Model 2	5	50	2
Model 3	10	50	1
Model 4	10	50	2
Model 5	10	50	3
Model 11	5	60	1
Model 12	5	60	2
Model 13	10	60	1
Model 14	10	60	2
Model 21	5	70	1
Model 22	5	70	2
Model 24	10	70	2
Model 25	10	70	3
Model 34	10	100	2
Model 35	10	100	3

6.4 Results and evaluation

During the spring of 2006, the arriving flood wave broke a multi-decade record at Szeged. Since we aim to build a model that can predict similar emergencies, we used the period 10.02.2006 - 10.06.2006 for the evaluation.

The following measures were used to compare the different models:

- Root Mean Square Error (RMSE)

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

- Mean Absolute Error (MAE)

$$MAE(Y,\hat{Y}) = \frac{\sum_{i=1}^{N} |y_i - \hat{y}_i|}{N},$$

- R^2 correlation

$$R^2(Y,\hat{Y}) = 1 - \frac{\sum_{i=1}^{N} (y_i - \hat{y}_i)^2}{\sum_{i=1}^{N} (y_i - \bar{y})^2},$$

- Willmott's Index (WI)

$$WI(Y,\hat{Y}) = 1 - \frac{\sum_{i=1}^{N} (y_i - \hat{y}_i)^2}{\sum_{i=1}^{N} \left(|\hat{y}_i - \bar{y}| + |y_i - \bar{y}|\right)^2},$$

where $Y = (y_1, y_2, \ldots, y_n)$ are the target values of the test data, $\hat{Y} = (\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_n)$ the appropriate forecasting (different days are evaluated separately) and $\bar{y}$ is the mean of Y. As can be seen from the formulae, for RMSE and MAE closer to 0, the better the model, however for R^2 and WI, good models have values close to 1.

Note that the R^2 indicator is known in hydrology as the Nash-Sutcliffe Efficiency(NSE). Furthermore, $R^2 = 0$ means that the forecast is as efficient as the average of the time series, but if the average gives a better forecast than the model, the value can become negative. Willmott's Index, also known as the Index of Agreement, also comes from the field of hydrology.

The following pages show comparisons where the models differ on a single parameter. By examining these, we wanted to draw conclusions about the direction in which the parameters should be changed to obtain better results. The models compared in Figure 6.5. all have 5 long history data, $K = 1$ distance, with differences in hidden size. The hidden size determines how much memory is used to store important information. Looking at the charts, we can see that although they perform roughly the same for short forecasts (1-2 days), but as we increase the forecast distance, we need a larger memory size. One might naively conclude that simply increasing the hidden size continuously will improve the performance of the models. However, this is a naive idea, and the Figure 6.6. illustrates this phenomenon: the figure also shows a comparison of models with the same long history (10) and distance ($K = 2$). We can see that for increasing hidden sizes the models perform worse and worse. This may be due to a phenomenon called overfitting. Overfitting is defined as when, despite a continuous decrease

in the error on the train set, the error on the test set starts to increase. In this case, due to the large enough memory, the model learns too specific features that are only relevant to the current data, thus losing the general patterns. So it is very important to set the right memory size for the given distance and past parameters. The main question of this thesis was whether we can improve the performance by considering distant neighbours. Based on the Figure 6.7. and the previous results, our answer is that by setting the appropriate hidden size, we can achieve a definite improvement by increasing the distance.

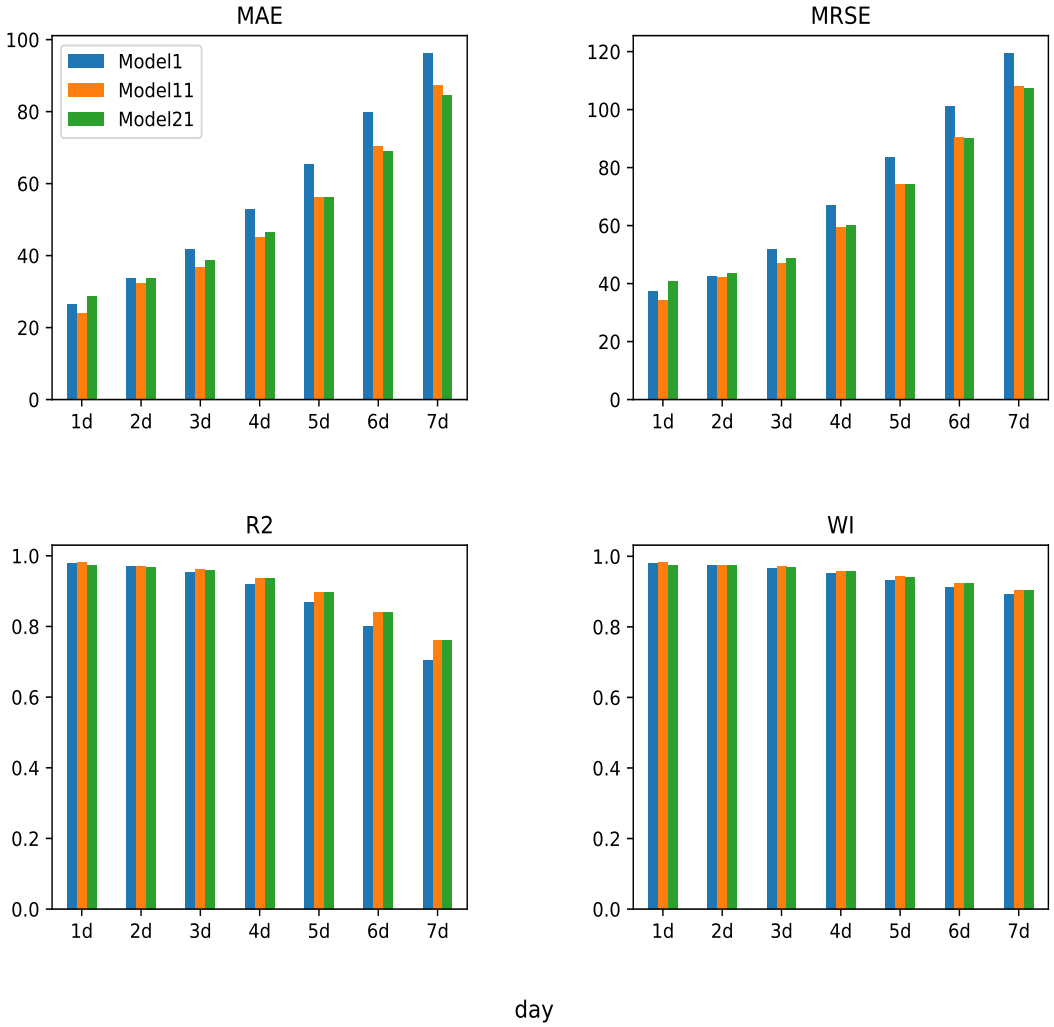


Figure 6.5: In each model the number of days in the past is 5, and use the $K = 1$. In addition, they are given increasingly larger hidden sizes as serial number increases. As we increase the prediction distance, it gives better and better results for larger and larger memory sizes.

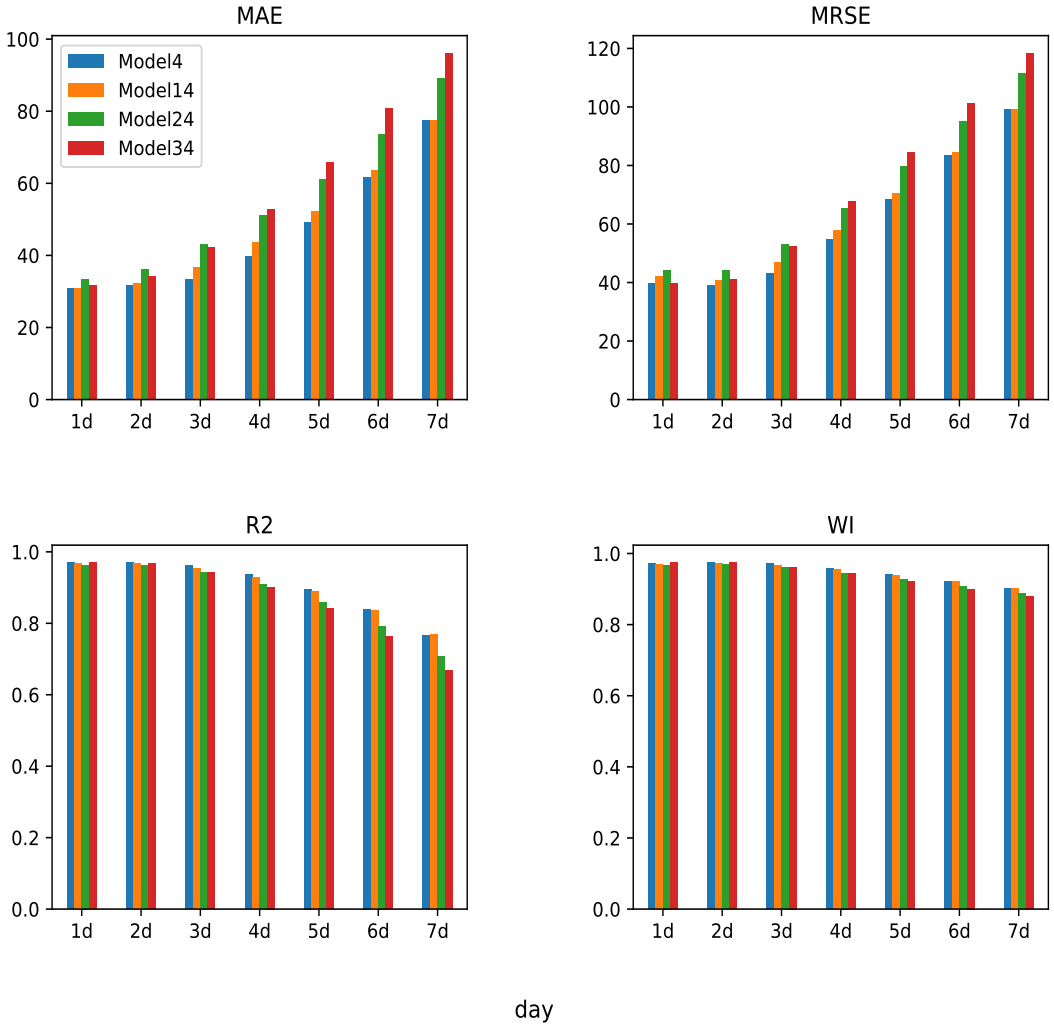


Figure 6.6: In each model the number of days in the past is 10, and use the $K = 2$. In addition, they are given increasingly larger hidden sizes as serial number increases. As the hidden size increases, we get worse and worse performance. It may be due to overfitting.

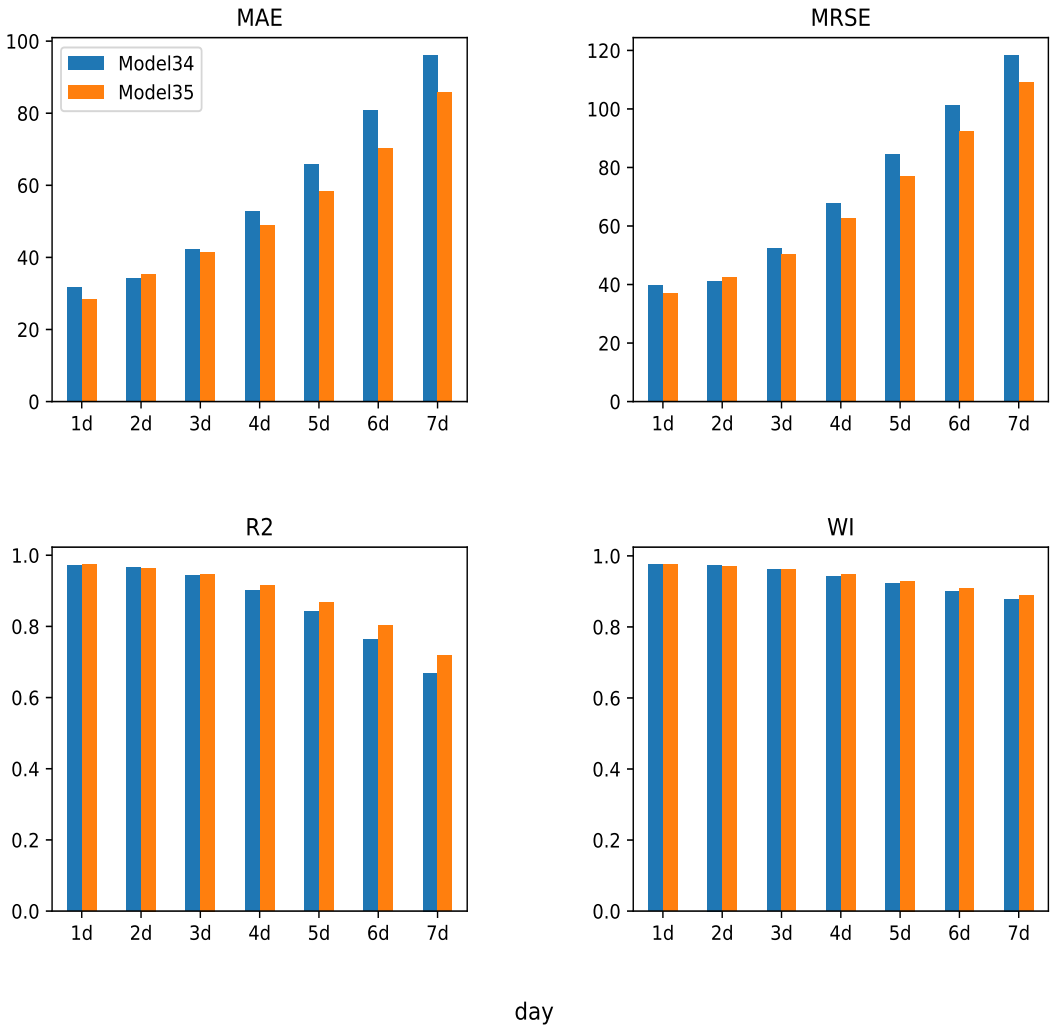


Figure 6.7: In both models the number of days in the past is 10, the hidden size is 100, and K is increased from 2 to 3. We observe that with appropriate parameter settings P and d_h, we can improve the predictions by increasing the distance.

As the final decision was based on the MAE, all its values are shown in the table below.

MAE

Name	1 day	2 day	3 day	4 day	5 day	6 day	7 day
Model 1	26.5	33.7	41,7	52.9	65,4	79,9	96,3
Model 2	22,9	26,7	36,8	49,7	65	81,2	98,6
Model 3	26,5	29,7	34,2	42,6	53,9	67,3	82,4
Model 4	31	31,7	33,5	39,9	49,2	61,8	77,6
Model 5	37,7	41	45,2	55,4	66,6	79,1	92,9
Model 11	24,1	32,4	36,9	45	56,3	70,4	87,3
Model 12	21,9	27,7	37	49,2	62,8	76,6	91,2
Model 13	22,4	32,9	46,9	61,7	77,9	93,6	109,3
Model 14	31	32,4	36,8	43,7	52,4	63,7	77,5
Model 21	28,7	33,7	38,6	46,5	56,1	69	84,6
Model 22	26,9	28,3	35,2	45,1	57,7	73,1	90,5
Model 24	33,4	36,3	43,2	51,1	61,1	73,8	89,1
Model 25	29,8	34	43,3	53,2	63,8	75,7	90
Model 34	31,8	34,4	42,3	53	65,9	80,8	96,2
Model 35	28,3	35,4	41,5	48,83	58,4	70,4	85,9

Based on MAE, the best performance was given by the Model4, and at this moment we consider it to be our best model. Figure 6.8 shows the predictions of this model compared to the actual measured data. The plots represent 1-day, 2-day, 5-day, and 7-day predictions, where the expected margins of error were 5, 10, 25, and 35 cm. In addition, a confidence interval is shown around the forecast, given for each day by the average of the previous 15 days' forecast absolute errors is calculated as

$$\sigma^{\pi}(t) = \frac{1}{15}\sum_{i=1}^{15} |y(t-i) - \hat{y}^{\pi}(t-i)|$$

where $\sigma^{\pi}(t)$ denotes the confidence of the π-day forecast at time t, and $\hat{y}^{\pi}(t)$ denotes the result of the π-day forecast for day t based on data available up to day $(t-\pi)$. Of course, this is different for different day forecasts, thus the π day confidence we use π-day forecasts.

6.5 Open issues, directions for improvement

One of our main questions - how to take geography into account when predicting flood waves - has been answered. Unfortunately, however, the current best model is still far behind the LSTM-LSTM model presented in [10]. Still, I think that this does not mean that we have reached a complete dead end. It is simply a need for further improvement of the model architecture.

For easier management and to avoid overfitting, it would be worth introducing early stopping. First, the data set is not split into two parts, but into three: train, validation, test. We will continue to train on the train set, but we will continuously evaluate the validation set. If the phenomenon of overfitting occurs, the early stopping condition stops the learning cycle. So, during learning, we do not load the data set with a predefined number of epochs, but the runtime depends on the performance. Early stopping can also be used in another case: when performance does not change significantly after a certain time. This way, we do not waste resources (time and memory) unnecessarily.

Also to reduce runtime, it would be good if the data and the model were compatible with the mini-batch concept, which can be solved via workarounds or joining the open source development of `Pytorch Geometric Temporal`.

Furthermore, I would like to draw your attention to the memory limitations mentioned several times. The local water management directorate (the collaborative partner in this research) does not have a GPU in their server machines (since they needed earlier only CPUs for high-performance computations) on which to do the training, so we used Google Colaboratory first, this is a free cloud service maintained by Google that has GPU support. However, here we had the problem that due to the long runtimes, Google regularly stopped training (the platform increments the limitation thresholds since the pandemic). Since we could not complete any of the trainings, we eventually dropped this option and continued working on our own hardware. Due to hardware limitations on our side, we were only able to train rather few and small hyperparameter settings, e.g. we could not get to the 15-day past of the model best chosen in [10] and the larger hidden sizes might have been required for it.

In building the model, we had to choose between several options, we are sure that the application of one (or several) of the following ideas could improve the accuracy of the models.

- At the very beginning of my work, the question of how to design the graph arose. If we add the back-edges mentioned above to the tributary connections, would this improve performance?
- In the literature research, I have repeatedly seen that finding the right loss function is as important as setting the right hyperparameters. It is possible that changing this could also result in improved model performance.
- We also had such a choice situation when we had to decide how to pass the information stored in the hidden state between the Encoder and Decoder steps.

I am hopeful that these paths will lead to further results that ATIVIZIG can exploit and further research projects will use this material as a basis in the follow-up development phases.

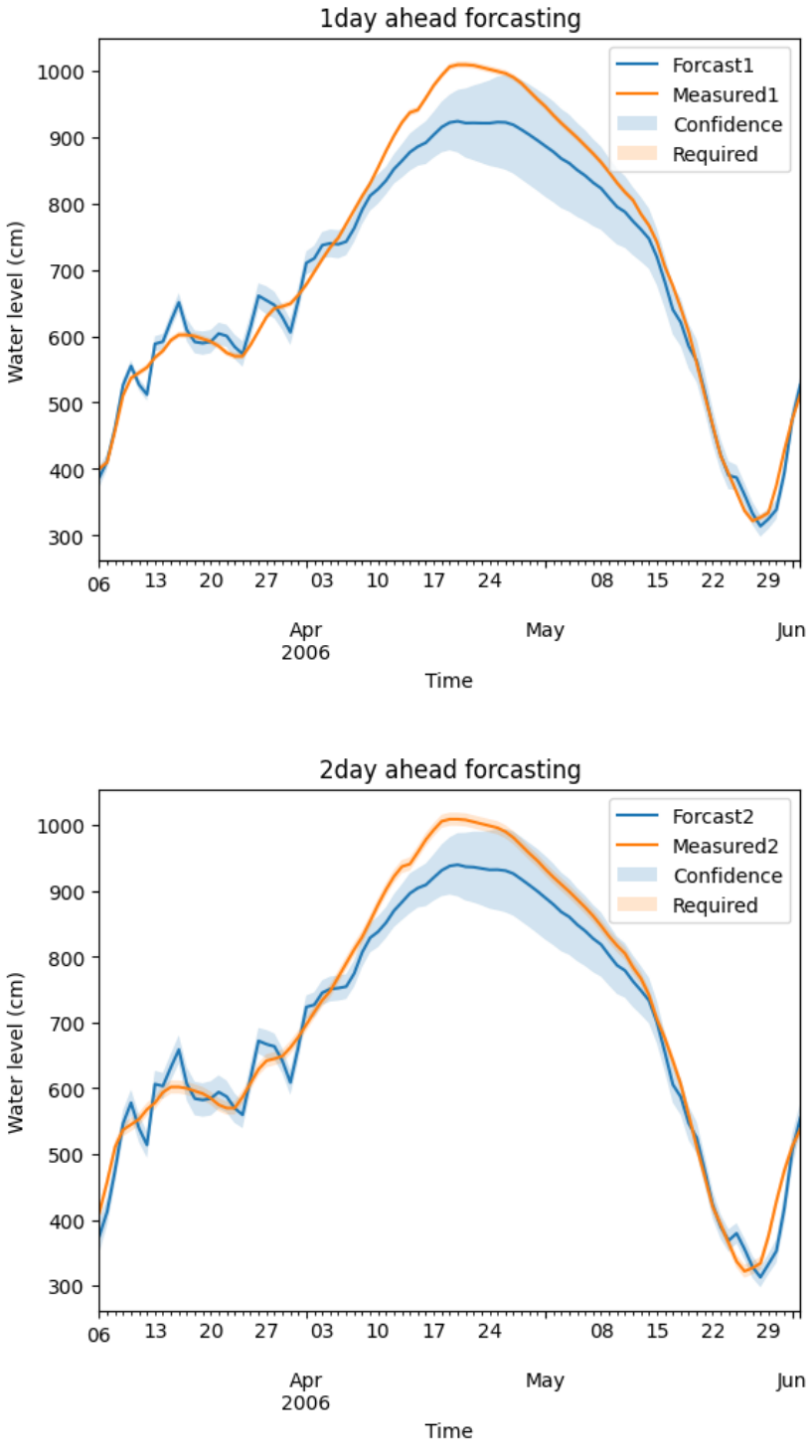


Figure 6.8: Predictions based on the selected model (Model4) for the period 10.02.2006 - 10.06.2006.

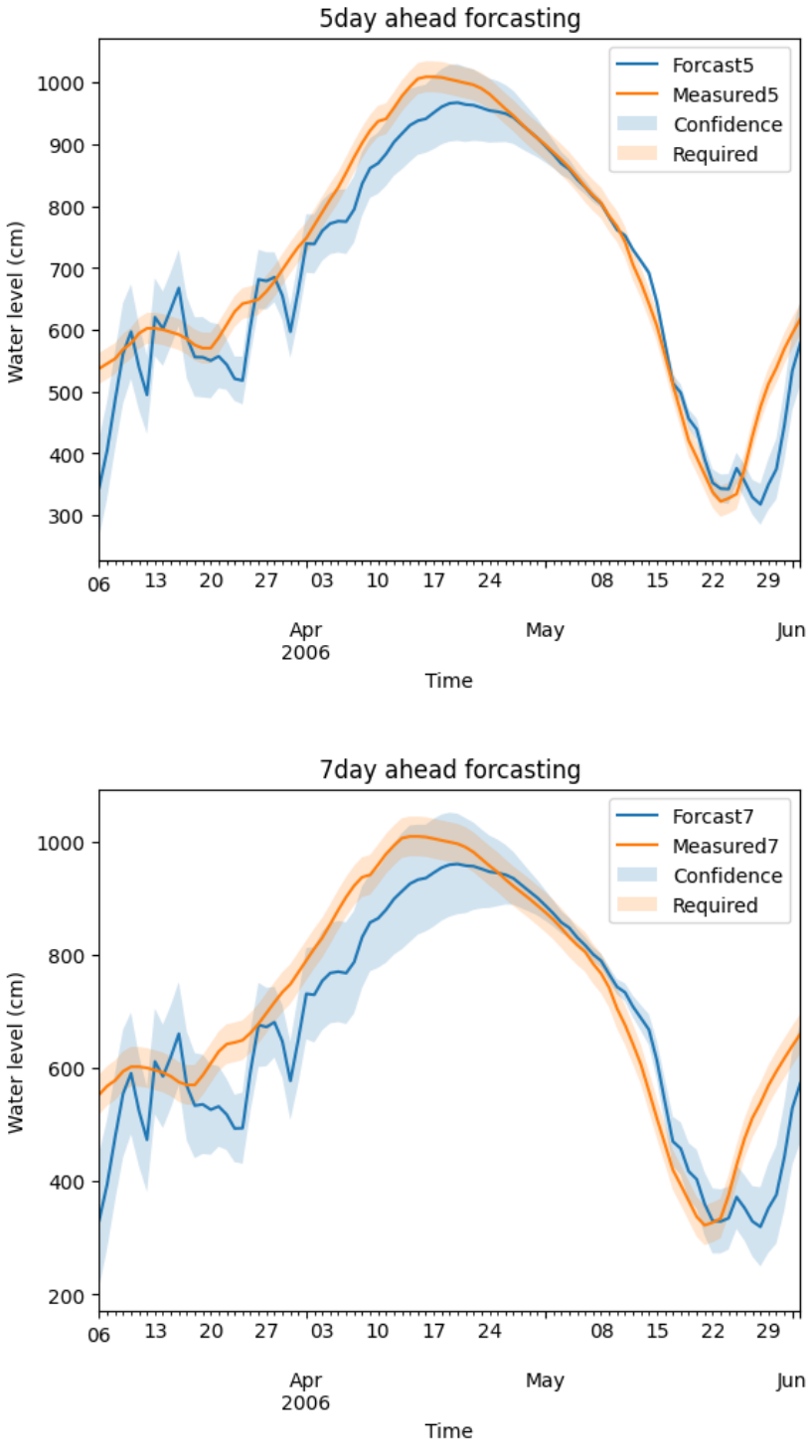


Figure 6.8: Predictions based on the selected model (Model4) for the period 10.02.2006 - 10.06.2006.

Bibliography

[1] Benyó Balázs, *Konvolúciós hálózatok*, educational material, Semmelweis University, Budapest, https://semmelweis.hu/kepalkotas/files/Convolut_network_Benyo.pdf

[2] Benyó Balázs *Mesterséges neurális hálózatok*, educational material, Semmelweis University, Budapest, https://semmelweis.hu/kepalkotas/files/ANN_BenyoB.pdf

[3] Bing Yu, Haoteng Yin, Zhanxing Zhu, "Spatio-Temporal Graph Convolutional Networks: A Deep Learning Framework for Traffic Forecasting", 2018. https://doi.org/10.48550/arXiv.1709.04875

[4] Diederik P. Kingma, Jimmy Ba, "Adam: A Method for Stochastic Optimization", *Published as a conference paper at the 3rd International Conference for Learning Representations*, San Diego, 2015. https://doi.org/10.48550/arXiv.1412.6980

[5] Felix Gers, Jurgen Schmidhuber, "Recurrent nets that time and count", *Proceedings of the IEEE-INNS-ENNS International Joint Conference on*, Volume: 3, 2000. https://www.researchgate.net/publication/3857862_Recurrent_nets_that_time_and_count

[6] Harsányi Benedek, *Gráf konvolúciós hálózatok és alkalmazásaik*, ELTE, Budapest, 2021.

[7] I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning*, MIT Press, 2016. https://www.deeplearningbook.org/

[8] Michaël Defferrard, Xavier Bresson, Pierre Vandergheynst, "Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering", *30th Conference on Neural Information Processing Systems*, 2017. https://doi.org/10.48550/arXiv.1606.09375

[9] Oliver Wieder, Stefan Kohlbacher, Mélaine Kuenemann, Arthur Garon, Pierre Ducrot, Thomas Seidel, Thierry Langer, "A compact review of molecular property prediction with graph neural networks", *Drug Discovery Today: Technologies*, Volume 37, Pages 1-12, ISSN 1740-6749, 2020. https://www.sciencedirect.com/science/article/pii/S1740674920300305

[10] Rátki Luca, *Flood Forecasting with Data Driven Model for river Tisza*, Szegedi Tudományegyetem, Szeged, 2022

[11] S. Hochreiter, J. Schmidhuber, "Long short-term memory", *Neural Computation*, 9(8):1735-1780, 1997. https://www.researchgate.net/publication/13853244_Long_Short-term_Memory

[12] Thomas N. Kipf, Max Welling: "Semi-Supervised Classification with Graph Convolutional Networks", *Conference paper at ICLR*, 2017. https://doi.org/10.48550/arXiv.1609.02907

[13] Tóth László, Grósz Tamás, *Mesterséges neuronhálók és alkalmazásaik*, e-learning material, University of Szeged, Szeged, 2019. https://eta.bibl.u-szeged.hu/2365/

[14] Xi Gao, Weide Li, "A graph-based LSTM model for PM2.5 forecasting", *Atmospheric Pollution Research*, Volume 12, Issue 9, ISSN 1309-1042, 2021. https://www.sciencedirect.com/science/article/pii/S1309104221002166

[15] Youngjoo Seo, Michaël Defferrard, Pierre Vandergheynst, Xavier Bresson, "Structured Sequence Modeling with Graph Convolutional Recurrent Networks", 2016. https://doi.org/10.48550/arXiv.1612.07659

[16] Zhilong Lu, Weifeng Lv, Yabin Cao, Zhipu Xie, Hao Peng, Bowen Du, "LSTM variants meet graph neural networks for road speed prediction", *Neurocomputing*, Volume

400, ISSN 0925-2312, Pages 34-45, 2020. https://www.sciencedirect.com/science/article/pii/S0925231220303775

[17] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, Philip S. Yu, "A Comprehensive Survey on Graph Neural Networks", *Journal of LaTeX class files*, Vol. XX, No. XX, August 2019. https://doi.org/10.48550/arXiv.1901.00596

[18] Zsolt Vizi, *Mathematics of Machine Learning*, course material, University of Szeged, Bolyai Institute, Szeged, 2022.

Bibliography

Websites used

PyTorch Documentation,

https://pytorch.org/docs/stable/index.html May 8, 2023 19:25

PyTorch Geometric Documentation,

https://pytorch-geometric.readthedocs.io/en/latest/ May 8, 2023 19:26

https://github.com/pyg-team/pytorch_geometric May 8, 2023 19:27

PyTorch Geometric Temporal Documentation,

https://pytorch-geometric-temporal.readthedocs.io/en/latest/ May 8, 2023 19:28

https://github.com/benedekrozemberczki/pytorch_geometric_temporal May 8, 2023 19:28

Summaries from IBM website

https://www.ibm.com/topics/artificial-intelligence?lnk=fle May 8, 2023 19:29

https://www.ibm.com/topics/machine-learning?lnk=fle May 8, 2023 19:29

https://www.ibm.com/topics/recurrent-neural-networks May 8, 2023 19:29

More useful websites

https://colah.github.io/posts/2015-08-Understanding-LSTMs/ May 8, 2023 19:29

https://www.mlbudapest.com/blog/lstm-modellek May 8, 2023 19:30

https://distill.pub/2021/gnn-intro/?fbclid=IwAR3_3x8pySiuQBqHFYZAnPXf1PpRixkaCVgsKJP4woont-MW4ibooRfJ0zM May 11, 2023 14:44

https://theaisummer.com/graph-convolutional-networks/#the-recurrent-chebyshev-expansion May 11, 2023 14:44

Statement

I, the undersigned, Boróka Jankus, declare that the results of my thesis are the results of my own work and that I have used only the sources cited (literature, tools, etc.). I acknowledge that my thesis has been University of Szeged Library will be available for loan and published on the Internet.