Technical University of Munich
Chair of Traffic Engineering and Control
Univ.-Prof. Dr.-Ing. Klaus Bogenberger

Interdisciplinary Project

Short-Term Traffic Flow Prediction Using Deep Learning

Author:
Askar Kolushev

Mentors:
Natalie Steinmetz, M.Sc.
Alexander Kutsch, M.Sc.

Date of submission: February 25, 2024

Abstract

In this project, we explore application of deep learning for short-term traffic flow prediction to real data collected from vehicle detectors in the German city of Ingolstadt. After describing the task, we provide an overview of various state-of-the-art methods developed in the area to select the most relevant approach for our problem. We find the DBN-KELM model (HAN and HUANG, 2020) as the most fitting for our purposes and describe all steps of this method in details. Then, we implement the procedure using PyTorch, fixing and adjusting the given data set. Finally, we tune the hyper parameters and test our implementation. Our experiments give mixed results indicating a room for improvement, but suggesting this area of research as highly promising. We conclude our report by interpreting different sides of the outcome and suggesting possible ways to enhance the method's accuracy.

Contents

1. Introduction

The modern economic development, followed by fast growth rate of the population imposes especially significant stress for both private and public transportation systems. This leads to an increase in the number of congestions and accidents on roads and has an impact on the everyday life of residents and commuters, efficiency of resource utilization and the state of the environment. To tackle said challenges, modern planning authorities utilize various aspects of *Intelligent Transporation Systems (ITS)*. It encompasses state-of-the-art technologies, mainly for communication and data analysis, to optimize and improve safety of the transportation systems.

The development of various types of sensors installed along the roads provides new types and amounts of recorded data, the analysis of which proves to be useful for the advancement of the modern ITS. An *Inductive loop detector (ILD)* consists of one or more coils embedded in the road pavement (LAMAS et al., 2016). As vehicles pass above the coils, a slight change in inductance can be observed through isolated cables that connect it to the control cabinet 1.1. Apart from using these signals for more efficient traffic management, recording these registrations allows to draw statistics on the usage level in specific time slots on road intersections where the detectors are installed.

One of the crucial components of ITS is short-term traffic flow prediction. It's purpose is to estimate the amount of vehicles present of passing through a point in the road system. We are given a data set representing the traffic density in the city of Ingolstadt, Germany. The set is organized into 115 folders. Each folder represents an intersection in the road network of the city of Ingolstadt and is named after that intersection using a 4-digit code. It contains

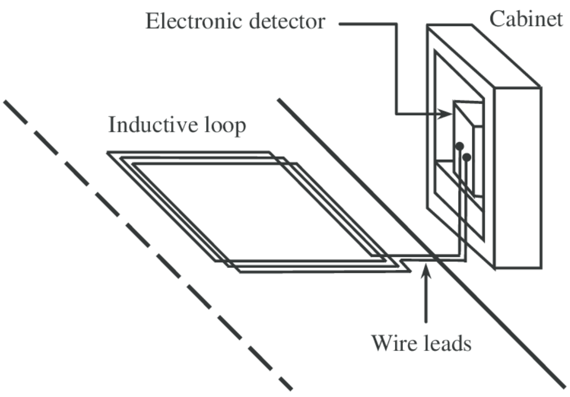


Figure 1.1: Elements of an inductive loop detector. (LAMAS et al., 2016)

365 *csv*-files - one for each day in the year 2021. Each file is a data table with readings from detectors as the number vehicles counted by the detectors in 15-minute intervals. The name of a file specifies the date and follows the format *DetCount_2021<month><day>.csv*. Apart from the exact time stamps of measurements (the *DATUM* column), the columns of the tables represent the names of the detectors. Each detector is installed on a separate lane approaching the intersection, and their names are unique within each intersection, i.e., folder. The exact mapping between the name of a detector within the target intersetion and the corresponding source of traffic for it's lane is given in an additional spreadsheet called *Look Up Table (Detectors Mapping on links).xlsx*, global for the whole data set. This file has a separate entry for each lane, including such data as the street names, starting and ending intersection codes and names of relevant detectors on the ending intersection. Some discrepancies between the Look-up table and actual data files, as well as fixes for them and other issues in the provided data set are given in Section 4.1.

The goal of this project is to study various methods for short-term traffic state prediction using *deep learning*, select the most fitting one, implement it if a ready realization is not available and test on the given data described above. These methods differ from each other by considering a city environment versus a freeway network, incorporating different types of input data and prediction model architectures.

As part of the efforts for development and integration of ITS, researchers usually approach the task of short-term traffic flow prediction in one of the two ways: statistic and neural networks (KARLAFTIS and VLAHOGIANNI, 2011).

Early statistics-based methods are usually based on probabilistic modelling of data. Assuming an underlying probability distribution of traffic conditions, they aim to capture parameters and patterns for generating the data. However, most of them assume a linear dependency between values, which makes them less efficient in a nonlinear setting.

Autoregressive integrated moving average (ARIMA) (MIN and WYNTER, 2011) uses historical univariate data to analyse and forecast the trend. The popularity of this method has made it a baseline for comparison in many research papers. To account for more factors and extreme conditions, ARIMAX (TSIRIGOTIS et al., 2012) was introduced as it's multivariate extension.

Support vector regression (SVR) (MÜLLER et al., 1997) applies the techniques of *Supprt vector machines (SVM)* (MUKHERJEE et al., 1997) to linearly separate data in a high-dimensional space after mapping to it from a low-dimensional space.

K-nearest neighbors(KNN) (ZHANG et al., 2013) has a very simple algorithm that is able to derive prediction directly through data analysis without involving complex functions. When designed with an adequate search mechanism, it can consider both temporal and spatial parameters, as well as incorporate other factors.

Other examples include *Kalman filtering* (GUO et al., 2014), *support vector machines (SVM)* (YANG et al., 2006), *Markov chains* (QI and ISHAK, 2014) and *Bayesian networks* (WANG et al., 2014).

Neural networks, on the other hand, are more flexible with input parameters thanks to their complex architectures encompassing tens of thousands neurons. Moreover, they are proven to handle nonlinear data much better than their more traditional counterparts (CHAN et al., 2012). The extensive flexibility of various architectures has allowed the researchers to not only adapt and apply neural networks in many different domains, but experiment with different approaches to solve each specific task. *Deep Belief Network (DBN)* is a deep learning method that relies on *Restricted Boltzmann machines (RBM)* (HINTON et al., 2006) for pre-training and was successfully applied by HUANG et al. (2014), showing much better results compared to shallower models. *Convolutional neural network (CNN)* (KRIZHEVSKY et al., 2012) use filters on neighboring inputs, making them more efficient in grid structures like images, thus being useful for capturing spatial relationships in road networks. *Long short-term memory network (LSTM)* (GRAVES et al., 2013) is a special type of a *recurrent neural network (RNN)* that solves the problem of vanishing gradients traditional for this type of models.

2. Literature

There are multiple works that aim at adapting widely used deep network architectures for the problem of traffic state prediction. CNNs rely on neighboring relationships, while LSTMs are good at preserving information across long sequences of data. YUANKAI and TAN (2016) combine both these models for traffic flow prediction using a method called *(CLTFP)*. As CNN filters inputs spatially, LSTM capture short-term variation and long-term periodicity to extract sptial-temporal features. The final prediction is then achieved with a linear prediction layer. Being one of the earlier attempts on applying deep learning techniques to the problem, CLTFP is claimed to show significant improvements over it's predecessors. However, the method was tested on values collected from 33 locations, making it wastly different from the grid structure observed in our data.

The idea of LSTM is based on the *gating* mechanism, which allows the model to learn additional parameters used to ignore data on specific steps of processing a recurrent sequence. This removes unnecessary operations with data and parameters, thus solving the problem of vanishing and exploding gradients. *Gated recurrent unit (GRU)* (CHUNG et al., 2014) is a primary example of executing this idea. ZHANG and KABUKA (2018) further explore and specify this idea by introducing a *deep GRU recurrent neural network (DGRNN)*. Integrating it into a fully-connected model allows them to discover both short and long term correlation in data series. Unfortunately, their data format is very different as well. While being collected from detectors placed in a grid structure in the state of California, their features are augmented with very detailed weather conditions. The traffic detector readings themselves are represented with vehicle miles travelled (VMT) sampled hourly, compared to our passing vehicle counts sampled each 15 minutes.

LIU et al. (2019) propose an *attentive traffic flow machine (ATFM)*. In it, the first *ConvLSTM* (SHI et al., 2015) unit takes input from the original traffic flow features, as well as the memorized representations of previous moments and uses an attention mechanism no build a weight map. The second unit adjusts the spatial dependencies with the attentional map to generate a spatial-temporal feature representation. For short-term prediction, they use a customized *ResNet* (HE et al., 2016) for the initial feature extraction, followed by *Sequential Representation Learning* and *Periodic Representation Learning*, both of which utilize ATFM in a recurrent manner - and finally get the prediction using a *Temporally-Varying Fusion* of their own design. Moreover, they further extend their method for a long-term traffic prediction. However, their approach was tested on TaxiBJ (ZHANG et al., 2017) and BikeNYC (ZHANG et al., 2016) data sets, both of which contain GPS trajectories rather than passage counts.

In an attempt to leverage the best of all worlds, HASSANNAYEBI et al. (2021) use a *1-*

dimensional convolution neural network (1DCNN) to extract traffic flow local trend features and two types of gated RNN (LSTM and GRU) for long temporal dependencies trend features. An attention-based *dynamic optimal weighted coefficient algorithm (DOWCA)* is then used to calculate dynamic weights of the outputs of CNN-LSTM and CNN-GRU. The resultant *combined deep learning prediction (CDLP)* model is optimized to minimize the sum of squared errors and shows better results than the baseline models. Unfortunately, it has been tested on a single intersection of two roads, rather than the whole network, which makes us question the relevance of this promising method for the task at hand.

Following a more traditional deep-learning based approach, HAN and HUANG (2020) propose to use the combination of a feature extractor based on a deep belief network with a *kernel extreme learning machine (KELM)* as a final prediction model. Using this combined architecture (DBN-KELM) for the prediction of residual noise and *singular spectrum analysis (SSA)* (LELES et al., 2017) for the trend forecast, they achieve an improved accuracy compared to base models. On top of that, they introduce a compression method that allows to greatly reduce the training time by applying the method to a subset of the global data. In the original work, DBN-KELM was tested on a freeway network of a much smaller magnitude, giving us no guarantee of success in a city environment. However, the data here still look the most similar compared to other listed approaches, and the method can clearly be adapted for our purposes. Therefore, it was selected for our project. The next chapter gives a more detailed description of all steps involved in the original work, and Section 4.2 dives deeper into our implementation.

3. Method description

The method presented in HAN and HUANG (2020) is a novel deep-learning based approach to short-term traffic state forecast. It includes an elaborate tool for time series prediction via separation into two components, as well as a data compression method aimed at reducing the necessary steps of the algorithm.

As an overview, the procedure starts with separating entrances (approaches) to intersections into groups with the highest internal degree of correlation. One representative approach is selected from each group for further processing. When predictions are obtained for each selected entrance, they are being extrapolated from each representative to it's respective group to produce a forecast for the whole network. Applying the method for each group is possible, however the grouping mechanism allows to decrease the number of times the prediction techniques are applied, saving the running time.

For each entrance, the data series is being split into two components (*trend* and *residual*) using the *fast Fourier transform (FFT)*. The prediction for the trend component is obtained via the *Overlap-SSA* (LELES et al., 2017), while the residual is fed into a deep architecture which is the main subject of the paper. The model uses a DBN for feature extraction and a KELM for the final prediction. When the predicted sample is obtained for both components, they are added back together to produce the final forecast.

The rest of the chapter describes each used technique in details and explains the place they hold in the general approach.

3.1. Expected input data format

The data as expected by all presented techniques consist of a set of time series, each containing integral values representing the numbers of vehicles that passed through a fixed point in space in equal time periods. In practice, an individual sequence of numbers is to be taken from all detectors located on the same *entrance* to an *intersection*, i.e., on all lanes entering it. Thus, each direct *connection* between two intersections contains one or two *sections* - for each direction - and each section provides one sequence of vehicle counts collected from all detectors installed on the relevant entrance.

Formally, if the network is represented as a directed graph $G = (N, E)$, N will represent the set of all intersections and $E = \{s_i, i = 1, 2, \dots, p\}$ - all p sections connecting them in various directions. Let d be the number of samples in the series. Then, for each section s_i, denote an individual reading from a counter in the period (t_{j-1}, t_j) as $z(s_i, t_j)$. The whole traffic flow from t_1 until t_d for s_i will be a vector $\mathbf{q}_i = \{z(s_i, t_1), z(s_i, t_2), \dots, z(s_i, t_d)\}$. Stacking these vectors for all p sections will produce a matrix $\mathbf{Q} = [\mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_p], \mathbf{Q} \in \mathbb{R}^{d \times p}$,which will act as an input for the rest of the procedure.

3.2. Data grouping and compression

The goal of data compression is to produce successful forecasting for the whole network by applying the actual prediction algorithm on a subset of all sequences. Apart from a compressed form of the data set, this step produces means for restoring the original data, which is eventually used to extrapolate the prediction onto the sequences omitted during compression. Ultimately, this means rewriting the input traffic flow matrix $\mathbf{Q}$ as

$$\mathbf{Q} = \mathbf{C}\mathbf{X}, \tag{3.1}$$

where $\mathbf{Q} \in \mathbb{R}^{d \times p}$, $\mathbf{C} \in \mathbb{R}^{d \times r}$ and $\mathbf{X} \in \mathbb{R}^{r \times p}$.

To construct the contraction matrix $\mathbf{C}$, we divide all sections into groups in a way that maximizes correlation between flows for sections within one group and minimizes it between sections in different groups. The paper uses the following formula for correlation between any two sections s_i and s_j (recognized as the Pearson product-moment correlation coefficient):

$$R(i,j) = \frac{\sum_{k=1}^{d}((z(s_i,t_k) - \bar{z}(s_i))((z(s_j,t_k) - \bar{z}(s_j))))}{\sqrt{\sum_{k=1}^{d}(z(s_i,t_k) - \bar{z}(s_i)^2)}\sqrt{\sum_{k=1}^{d}(z(s_j,t_k) - \bar{z}(s_j)^2)}}, \tag{3.2}$$

, where $\bar{z}(s_i) = \frac{1}{d}\sum_{k=1}^{d} z(s_i, t_k), \quad \bar{z}(s_j) = \frac{1}{d}\sum_{k=1}^{d}(z(s_j, t_k))$.

By calculating $R(i,j)$ for all pairs of sections in the network, we construct a correlation coefficient matrix $\mathbf{R} \in \mathbb{R}^{p \times p}$.

Then, we select a threshold α as a hyper parameter and group sections in such a way that if two sections have flows whose correlation exceeds α, then they end up in the same group. The number of groups r will then be proportional to α as a higher threshold will make it more likely that two sequences will not correlate enough and will be placed into different groups.

By selecting one representative sequence c_i from each group and stacking them as columns, we produce the contraction matrix

$$\begin{gathered}\mathbf{C} = \{c_1, c_2, \dots, c_r\}, \\ \{c_1, c_2, \dots, c_r\} \subseteq \{q_1, q_2, \dots, q_p\}.\end{gathered} \tag{3.3}$$

It is the matrix $\mathbf{C}$ that will be fed to the prediction mechanism further down the line. The predicted samples will only be available for the groups representative sections used for it's construction. From 3.1 we know that we require the relation matrix $\mathbf{X}$ to reconstruct the predictions for the whole network. Folowing the same equation, we can use the Moore-Penrose pseudo-inverse to obtain it:

$$\begin{gathered}\mathbf{X} = \mathbf{C}^{\dagger}\mathbf{Q} \\ \mathbf{X} = (\mathbf{C}^T\mathbf{C})^{-1}\mathbf{C}^T\mathbf{Q}.\end{gathered} \tag{3.4}$$

In short, the algorithm looks as follows:

Step 1: Calculate $R(i, j)$ for all sections in the network (all columns of Q) using Equation 3.2 and construct the cross-correlation coefficient matrix $\mathbf{R} \in \mathbb{R}^{p \times p}$.

Step 2: Set a threshold α.

Step 3: Select one section s_i from the network and put it. into one group with all sections s_j for which $R(i, j) > \alpha$.

Step 4: Remove all sections from the newly created group.

Step 5: Repeat the steps 3 and 4 until no ungrouped sections are left.

Step 6: Select a representative from each group and stack them as columns into the contraction matrix $\mathbf{C}$ - see Equation 3.3.

Step 7: Calculate the Relation matrix $\mathbf{X}$ using Equation 3.4.

3.3. Extraction of components

Observations show that the traffic flow data tend to show a high degree of periodicity. The sequence shows a clear main trend that repeats itself regularly. However, some daily deviations from it show as well. The DBN-KELM model presented in the paper performs well at processing these fluctuations, but does poorly on the general trend. Therefore, it is proposed to split the overall series into a *trend* and a *residual* components and process them individuallly via different methods.

The goal here is to take the original time series and extract two other series that would reconstruct the initial one when added together. Inherently, the relation of frequency and time functions that states that any time series can be represented as a sum of periodic functions that show different amplitudes on the whole range of the frequency domain. This allows us to represent any time series as a continuous set of coefficients for frequences of periodic functions, or in other words, represent the original time domain series on a frequency domain. This task is called a spectral decomposition and performed using the Fourier transform.

In it's discrete form, the Fourier transform distributes the frequences into a fixed number of frequency bands. Let $x(n)$ be a time series of N points, and the expression of discrete Fourier transform is as follows:

$$X(k) = \sum_{n=0}^{N} x(n) W_N^{kn}, k = 0, 1, \ldots, N-1; \quad W_N = \exp -j\frac{2\pi}{N} \tag{3.5}$$

The inverse of this transformation is then

$$X(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) W_N^{-nk}, \quad n = 0, 1, \ldots, N-1 \tag{3.6}$$

When the data shows a clear periodicity (like in the case of traffic flow), a stable periodic oscillation is observed on long time periods. Therefore, we see a high contribution in a low frequency band (the energy is high for a low frequency in the domain). Leveraging this, we can inverse the transformation separately for low and high frequencies to construct two independent time series (denoted A_t and E_t respectively) that would yield the original one when added together:

$$X_t = A_t + E_t \tag{3.7}$$

At the same time, the trend term A_t is expected to show a much higher degree of periodicity than the residuals E_t. This corresponds exactly to our goal.

To formalize the operation, we set a spectrum threshold P as a hyper parameter. The procedure including the future steps is described as follows:

Step 1: Having a time series X_t, perform a Fourier transform as described in 3.5 to obtain it's representation in a frequency domain:

$$Y_w = F(x_t) \tag{3.8}$$

Step 2: Set all frequencies whose energy is less than P to zero

Step 3: Obtain the trend term from the remaining frequencies:

$$A_t = F^{-1}(A_w), \tag{3.9}$$

where F^{-1} is the inverse Fourier tranform as described in 3.6.

Step 4: Obtain the residuals by subtracting the trend term from the original time series:

$$E_t = X_t - A_t \tag{3.10}$$

After applying prediction methods for both time series A_t and E_t, to obtain their extended versions ($\tilde{A}_t$ and $\tilde{E}_t$ respectively), we add them back together to output the final prediction for the original sequence:

$$\tilde{X} = \tilde{A}_t + \tilde{E}_t \tag{3.11}$$

3.4. Singular spectrum analysis

Singular Spectrum Analysis (SSA) is a method for processing and analysing a time series. HAN and HUANG (2020) doesn't provide a thorough description of if beyond shortly mentioning that a slightly improved version called Overlap-SSA (LELES et al., 2017) is used for forecasting the trend term. However, a concise and detailed description of the original SSA is given in GOLYANDINA and KOROBEYNIKOV (2014). The main purpose of this tool is separating a time series into an arbitrary number of components (similar to what is being done here via the Fourier tranform - see Section 3.3). However, we are more interested in another application, namely time series forecasting. The "Overlap" version of SSA (also known as *Ov-SSA*) provides an idea to decrease noise that arises naturally during the procedure. Unfortunately, while the mechanism of that improvement is quite clear for component separation, it is not obvious how to adapt those changes to the forecasting part. Therefore, keeping in mind that Ov-SSA is just a marginal upgrade upon the base method, we have decided to stick with the algorithm as described in in GOLYANDINA and KOROBEYNIKOV (2014). It was the only one used in the final implementation (Section 4.2) and we will not describe the advanced variant here.

SSA is a powerful tool for time series analysis that has been successfully applied to many non-conventional problems in various fields of science and engineering. As mentioned above, it's main feature is the ability to separate a data series into any number of components (trend+residuals, trend+oscillation+noise, etc.). We will describe this aspect of it's function first.

Let $\mathbb{X}_N = (x_1, x_2, \ldots, x_N)$ be a real-valued time series of length N. We pick two hyper parameters for the following procedure:

- Integer L is a *window length* used later in the contruction of a Hankel matrix;
- Grouping $\{I_1, I_2, \ldots, I_m\}$ defines the way to split the matrix into m disjoints groups.

The operation itself involves 4 steps:

Step 1: Embedding. We slide a window of length L along the original sequence $\mathbb{X}_N$ starting from the first element with a step size of 1. On each steps, it produces a lagged vectors of size L with $K = N - L + 1$ vectors in total:

$$X_i = (x_i, x_{i+1} \ldots, x_{i+L-1}), \quad i = 1, 2, \ldots, K \tag{3.12}$$

Stacking all these vectors as columns produces a *trajectory matrix* $\mathbf{X} \in \mathbb{R}^{L \times K}$:

$$\mathbf{X} = [X_1 : X_2 : \ldots : X_K] = (x_{ij})_{i,j=1}^{L,K} = \begin{bmatrix} x_1 & x_2 & x_3 & \cdots & x_K \\ x_2 & x_3 & x_4 & \cdots & x_{K+1} \\ x_3 & x_4 & x_5 & \cdots & x_{K+2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_L & x_{L+1} & x_{L+2} & \cdots & x_N \end{bmatrix} \tag{3.13}$$

Two important properties of $\mathbf{X}$ are

- both the rows and the columns are subseries of $\mathbb{X}_N$;
- the elements on anti-diagonals are the same, making $\mathbf{X}$ a *Hankel* matrix.

Step 2: Decomposition. Let $\{P_i\}_{i=1}^L$ be an orthonormal basis in $\mathbb{R}^L$. Consider the following decomposition of the trajectory matrix:

$$\begin{gathered} \mathbf{X} = \sum_{i=1}^{L} P_i Q_i^T = \mathbf{X}_1 + \mathbf{X}_2 + \ldots + \mathbf{X}_L, \\ Q_i = \mathbf{X}^T P_i \end{gathered} \tag{3.14}$$

Choosing $\{P_i\}_{i=1}^L$ as eigenvectors of $\mathbf{X}\mathbf{X}^T$ turns 3.14 into a *Singular Value Decomposition (SVD)* of $\mathbf{X}$:

$$\mathbf{X} = \sum_i \sqrt{\lambda_i} U_i, V_i^T, \tag{3.15}$$

The triple ($\sqrt{\lambda_i}$, U_i, V_i) is called ith *eigentriple*:

- λ_i are the eigenvalues of $\mathbf{X}\mathbf{X}^T$ in a non-increasing order - $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_L \geq 0$;

- U_i are left singular vectors;

- V_i are *factor vectors* or right singular vectors

Note: The paper also mentions an alternative choice of $\{P_i\}_{i=1}^L$, but states that it only fits for the analysis of stationary time series with zero mean, hence we omit it.

Step 3: Eigentriple grouping. Let d be the number of non-zero eigenvalues of $\mathbf{X}\mathbf{X}^T$ - $d = \max(j : \lambda_j \neq 0)$. Our next step is to group the indices $\{1, 2, \ldots, d\}$ into m disjoint subsets defined via the hyper parameters $\{I_1, I_2, \ldots, I_m\}$. If we define $\mathbf{X}_I = \sum_{i \in I} \mathbf{X}_i = \sum_{i \in I} \sqrt{\lambda_i} U_i V_i^T$, then 3.14 leads to

$$\mathbf{X} = \mathbf{X}_{I_1} + \mathbf{X}_{I_2} + \ldots + \mathbf{X}_{I_m} \tag{3.16}$$

The components $\mathbf{X}_{I_i}, i \in \{1, 2, \ldots, m\}$ are the output we seek from this step.

Step 4: Diagonal averaging. In this step, we transform each matrix $\mathbf{X}_I$ produced in the previous step via 3.16 into a new series of length N. Let $\mathbf{Y} \in \mathbb{R}^{L \times K}$ be a matrix with elements $y_{ij}, 1 \leq i \leq L, 1 \leq j \leq K$, and let $L \leq K$. *Diagonal averaging* transforms it into a series $(y_1, y_2, \ldots, y_N)$ using the formula

$$\tilde{y}_s = \sum_{(l,k) \in A_s} \frac{y_{lk}}{|A_s|}, \tag{3.17}$$

where $A_s = \{(l, k) : l + k = s_1, 1 \leq l \leq L, 1 \leq k \leq K\}$ is the sth "antidiagonal" and $|A_s|$ is the number of elements in it.

Another way to look at it that we take the original matrix $\mathbf{Y}$ and transform it into a Hankel matrix by iteratively replacing all elements in each antidiagonal with the average of all elements in it. After that, we take that matrix and concatenate it's first row with the last column without duplicating the common top-left entry. This is essentially an inverse of the operation 3.13 from the Embedding step.

Performing this operation on each of the group-specified matrices $\mathbf{X}_{I_i}, i \in \{1, 2, \ldots, m\}$ generates m series representing m components that form the original series $\mathbb{X}_N$ when added back together (see 3.16).

One remarkable property of this tool is the capability to capture periodicity in the series. Executing this power on multiple components before the reconstruction allows us to make predictions on the future trend, provided that the original data are indeed self-repeating. As a forecast method, this makes SSA unique as it has no trainable parameters, relying exclusively on the properties of data themselves and thus requiring to training before the inference.

The prediction process looks like this. Let

- I be the chosen set of eigentriples (the same hyper parameter as before);
- $P_i \in \mathbb{R}^L, i \in I$ the corresponding eigenvectors;
- $\underline{P_i}$ their first $L-1$ coordinates;
- π_i the last coordinate of P_i;
- $\nu^2 = \sum_{i \in I} \pi_i^2$;
- $\tilde{X}_N = (\tilde{x}_1, \tilde{x}_2, \dots \tilde{x}_N)$ the series reconstructed by I.

Then,

1. Define

$$R = \frac{1}{1-\nu^2} \sum_{i \in I} \pi_i \underline{P_i} \tag{3.18}$$

2. Recurrently calculate

$$y_i = \begin{cases} \tilde{x}_i & \text{for } i = 1, 2, \dots, N \\ \sum_{j=1}^{L-1} a_j y_{i-j} & \text{for } i = N+1, N+2, \dots, N+M \end{cases} \tag{3.19}$$

The numbers $y_{N+1}, y_{N+2}, \dots, y_{N+M}$ form the M terms of the recurrent forecast.

3.5. Deep belief network

A deep belief network is a deep architecture composed of multilayer random hidden variables. The first layer represents the input to the model, and several restricted Boltzmann machines are connected upward. HAN and HUANG (2020) use them for feature extraction as a part of the combined DBN-KELM architecture. While this paper covers the general principle of both RBM and DBN, in this project we relied on GHOJOGH et al. (2021), which give a much more thorough introduction.

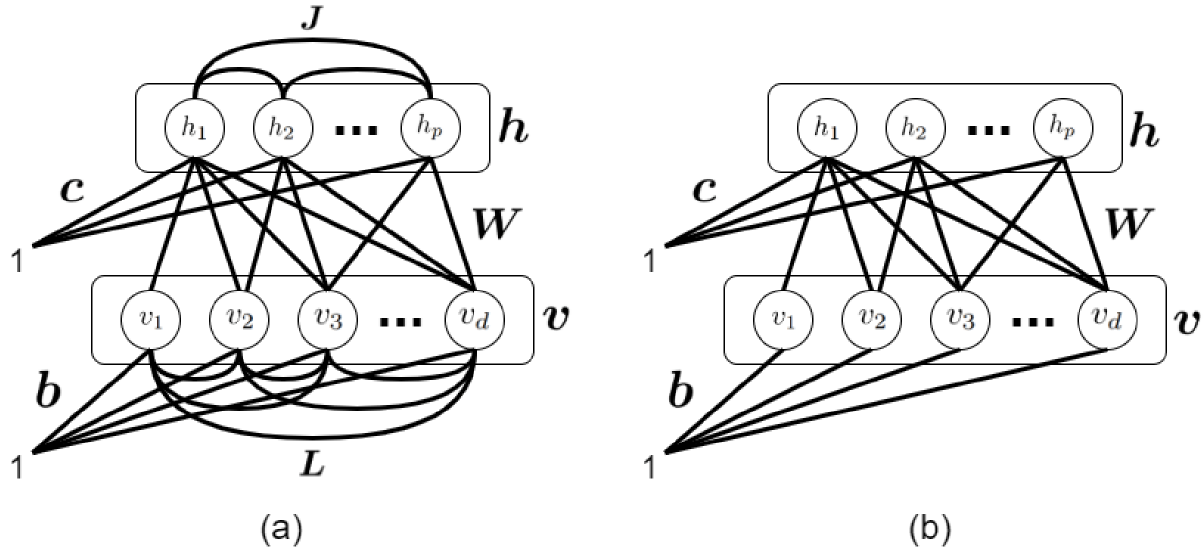


Figure 3.1: The structure of (a) a Boltzmann machine and (b) a restricted Boltzmann machine (GHOJOGH et al., 2021)

3.5.1. Restricted Boltzmann machine

Boltzmann machine (BM) is a generative model and a *probabilistic graphical model (PGM)* (BISHOP, 2006). It consists of a visible (or observation) layer $\mathbf{v} = [v_1, v_2, \ldots, v_d] \in \mathbb{R}^d$ and a hidden layer of latent variables $\mathbf{h} = [h_1, h_2, \ldots, h_p] \in \mathbb{R}^p$. The elements of both layers contain binary values, i.e., $\forall i, j$, $v_i \in \{0, 1\}$ and $h_j \in \{0, 1\}$. While the visible layer usually contains data (or other values we can see), the hidden layer represents latent variables. In the PGM of BM, there are connections between the elements $\mathbf{v}$ and $\mathbf{h}$ - both within and between the layers. Each of the elements of both $\mathbf{v}$ and $\mathbf{h}$ also has a bias. In a Restricted Boltzmann Machine (RBM), only inter-layer links are allowed. Figure 3.1 depicts both of them with their layers and links. Let w_{ij} denote a link between v_i and h_j, b_i be the bias link for v_i and c_j be the bias link for h_i. The whole model can then be described in a matrix form:

- $\mathbf{W} = [w_{ij}] \in \mathbb{R}^{d \times p}$
- $\mathbf{b} = [b_1, b_2, \ldots, b_d] \in \mathbb{R}^d$
- $\mathbf{c} = [c_1, c_2, \ldots, c_p] \in \mathbb{R}^p$

An RBM is what is called an Ising model (ISING, 1925), thus having an internal energy. According to HINTON and SEJNOWSKI (1983), the energy of an RBM based on interactions between linked units can be modelled as

$$\mathbb{R} \ni E(\mathbf{v}, \mathbf{h}) := -\mathbf{b}^T \mathbf{v} - \mathbf{c}^T \mathbf{h} - \mathbf{v}^T \mathbf{W} \mathbf{h} \tag{3.20}$$

Training an RBM amounts to minimizing this energy. This brings the links and biases of both layers to a state in which the hidden layer is capable of capturing meaningful features or embeddings for the visible data. Two commonly used training algorithms for RBMs are *Gibbs sampling* (Algorithm 1) and a *constrastive divergence* (Algorithm 2) which is based on it.

Algorithm 1: Gibbs sampling in RBM

```
Input: visible dataset $\mathbf{v}$, (intialization: optional), number of steps $n$
Get initialization or do random initialization of $\mathbf{v}$;
for $i$ from $1$ to $n$ do
    for $j$ from $1$ to $p$ do
        $h_j^{(v)} \sim \mathbb{P}(h_j|\mathbf{v}^{(v)})$
    end
    for $i$ from $1$ to $d$ do
        $v_i^{(v+1)} \sim \mathbb{P}(v_i|\mathbf{h}^{(v)})$
    end
end
```

Algorithm 2: Training RBM using contrastive divergence

```
Input: training data $\{x_i\}_{i=1}^n$, number of Gibbs sampling steps $n$
Randomly initialize $\mathbf{W}$, $\mathbf{b}$, $\mathbf{c}$;
while not converged do
    Sample a mini-batch $\{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_m\}$;
    from training dataset $\{\mathbf{x}_i\}_{i=1}^n$ (n.b. we may set $m = n$);
    // Gibbs sampling for each data point
    Initialize $\hat{\mathbf{v}}_i^{(0)} \leftarrow \mathbf{v}_i$ for all $i \in \{1, 2, \dots, m\}$;
    for $i$ from $1$ to $m$ do
        Algorithm 1 $\leftarrow \hat{\mathbf{v}}_i^{(0)}$;
        $\{h_i\}_{i=1}^p, \{v_i\}_{i=1^d} \leftarrow$ Last iteration of Algorithm 1;
        $\tilde{\mathbf{h}}_i \leftarrow [h_1, h_2, \dots, h_p]^T$;
        $\tilde{\mathbf{v}}_i \leftarrow [v_1, v_2, \dots, v_d]^T$;
        $\hat{\mathbf{h}}_i \leftarrow \mathbb{E}_{\sim \mathbb{P}(\mathbf{h}|\mathbf{v}_i)}[\mathbf{h}]$;
    end
    // gradients:
    $\nabla_{\mathbf{W}} \ell(\theta) \leftarrow \sum_{i=1}^m \mathbf{v}_i \hat{\mathbf{h}}_i^T - \sum_{i=1}^m \tilde{\mathbf{h}}_i \tilde{\mathbf{v}}_i^T$;
    $\nabla_{\mathbf{b}} \ell(\theta) \leftarrow \sum_{i=1}^m \mathbf{v}_i - \sum_{i=1}^m \tilde{\mathbf{v}}_i$;
    $\nabla_{\mathbf{c}} \ell(\theta) \leftarrow \sum_{i=1}^m \mathbf{h}_i - \sum_{i=1}^m \tilde{\mathbf{h}}_i$;
    // gradient descent for updating solution:
    $\mathbf{W} \leftarrow \mathbf{W} - \nu \nabla_{\mathbf{W}} \ell(\theta)$;
    $\mathbf{b} \leftarrow \mathbf{b} - \nu \nabla_{\mathbf{b}} \ell(\theta)$;
    $\mathbf{c} \leftarrow \mathbf{c} - \nu \nabla_{\mathbf{c}} \ell(\theta)$;
end
```

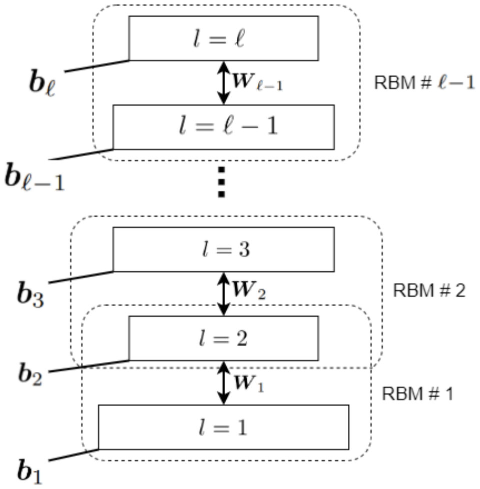


Figure 3.2: Pre-training a deep belief network by considering every pair of layers as an RBM (GHOJOGH et al., 2021)

.

3.5.2. Stacking RBM models

A DBN consists of several RBM models stacked together, as depicted on Figure 3.2. The first layer of the model accepts the input data as it's visible layer $\mathbf{v}_1$. Then, on each layer ℓ, an RBM model with the visible layer $\mathbf{v}_\ell$ has the hidden layer $\mathbf{h}_\ell$ that is acts as the visible layer $\mathbf{v}_{\ell+1}$ for the RBM above it. Training a DBN is described in Algorithm 13 and supposes optimizing all RBM components layer by layer via Algorithm 2 followed by error backpropagation.

Algorithm 3: Training a deep belief network

```
Input: training data $\{\mathbf{x}_i\}_{i=1}^n$
for $l$ from $1$ to $\ell - 1$ do
    if $l = 1$ then
        $\{\mathbf{v}_i\}_{i=1}^n \leftarrow \{\mathbf{x}_i\}_{i=1}^n$;
    else
        // generate $n$ hidden variables of previous RBM:
        $\{\mathbf{h}_i\}_{i=1}^n \leftarrow$ Algorithm 1 for $(l-1)$-th RBM$\leftarrow \{\mathbf{v}_i\}_{i=1}^n$;
        $\{\mathbf{v}_i\}_{i=1}^n \leftarrow \{\mathbf{h}_i\}_{i=1}^n$
    end
    $\mathbf{W}_l, \mathbf{b}_l, \mathbf{b}_{l+1} \leftarrow$ Algorithm 2 for $l$-th RBM $\leftarrow \{\mathbf{v}\}_{i=1}^n$
end
// fine-tuning using backpropagation:
Initialize network with weights $\{\mathbf{W}_l\}_{l=1}^{\ell-1}$ and biases $\{\mathbf{b}_l\}_{l=2}^n$;
$\{\mathbf{W}_l\}_{l=1}^{\ell-1}, \{\mathbf{b}_l\}_{l=1}^{\ell} \leftarrow$ Backpropagate the error of loss from several epochs
```

3.6. Kernel extreme learning machine

The kernel extreme learning machine is known to be dealing well with non-linear time series, which justifies it's use for traffic flow data.

Extreme learning machines (ELM) are feedforward neural networks whose input weights $\mathbf{W}_{in}$ and biases $\mathbf{b}_in$ for the hidden nodes are assigned randomly and fixed rather than being optimized iteratively (via gradient descent, annealing etc.). The output of each hidden node for the input $\mathbf{X}$ then becomes

$$\mathbf{H} = h(X) = f(\mathbf{X} \cdot \mathbf{W}_in + \mathbf{b}_in) \tag{3.21}$$

for the actifation function f. The output for the whole network is

$$\mathbf{Y} = \mathbf{H} \cdot \mathbf{W}_{out}. \tag{3.22}$$

The correct prediction is achieved via calculating the optimal output weights W_{out} analytically in one step (again, without gradual optimization):

$$\mathbf{W}_{out} = \mathbf{H}^{\dagger} = (\mathbf{H}^T \cdot \mathbf{H})^{-1} \cdot T \tag{3.23}$$

for the target values T.

Kernel extreme learning machines replace the random mapping of ELM with kernel mapping. Define the kernel learning mapping as

$$\mathbf{\Omega}_{ELM} = \mathbf{H}\mathbf{H}^T. \tag{3.24}$$

Since H is obtained by applying weights, biases and activation to the input, and it's outer product with itself is taken as the final result, we can replace the whole operation with a kernel function k:

$$\mathbf{\Omega}_{ELM} = h(x_i)h(x_j) = k(x_i, x_j). \tag{3.25}$$

Specifically, HAN and HUANG (2020) use a Gaussian kernel function here:

$$k(x, x_j) = \exp(-\gamma \cdot ||x - x_i||^2), \gamma > 0. \tag{3.26}$$

Regularization using a coefficient C, inversion and multiplication with the target values produces the relation matrix β:

$$\beta = (\frac{I}{C} + \mathbf{\Omega}_{ELM})^{-1}T. \tag{3.27}$$

This is then used with the kernel multiplication of each new unit and the original training data to predict the output for an unsees input x:

$$\begin{aligned} f(x) &= [k(x, x_1), k(x, x_2), \ldots, k(x, x_N)]\beta \\ &= [k(x, x_1), k(x, x_2), \ldots, k(x, x_N)](\frac{I}{C} + \mathbf{\Omega}_{ELM})^{-1}T. \end{aligned} \tag{3.28}$$

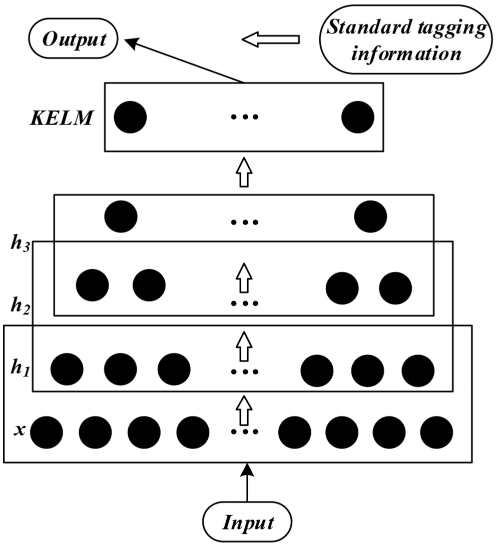


Figure 3.3: Structure of DBN-KELM (HAN and HUANG, 2020)

3.7. Complete prediction procedure

The DBN-KELM model is composed of DBN which is respondible for feature extraction and KELM for prediction. The architecture as depicted on Figure 3.3 consists of four hidden layers - three RBMs forming a DBN model and a single hidden layer of a kernel extreme learning machine. In other words, the output of DBN is embedded features that are then fed to the KELM model.

The DBN part of the model is trained using the *mean square error (MSE)* loss:

$$MSE = \frac{1}{N}\sqrt{\sum_{i=1}^{N}(y_i - \hat{y}_i)}. \tag{3.29}$$

This loss function is also used for the final evaulation of the direct prediction (applying DBN+KELM to the columns of $\mathbf{C}$). The prediction after reconstruction via Equation 3.1 (for sections not participating in direct prediction) a *mean absolute percentage error (MAPE)* function is used:

$$MAPE = \frac{1}{N}\sum_{i=1}^{N}\left|\frac{(y_i - \hat{y}_i)}{y_i}\right| \times 100\% \tag{3.30}$$

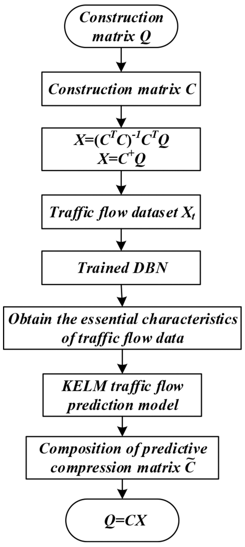

Construction matrix Q
Construction matrix C
X=(C^T C)^-1 C^T Q
X=C^+ Q
Traffic flow dataset X_t
Trained DBN
Obtain the essential characteristics of traffic flow data
KELM traffic flow prediction model
Composition of predictive compression matrix C̃
Q=CX

Figure 3.4: Processing steps of the whole prediction model (HAN and HUANG, 2020)

The prediction procedure is depicted in Figure 3.4:

Step 1: Construct the road network data matrix $\mathbf{Q}$ from the raw input data.

Step 2: Use the compression method to find a CX-decomposition of $\mathbf{Q}$. Select a subset of the data - the compression matrix $\mathbf{C}$, as well as the relation matrix $\mathbf{X}$ (see Equation 3.4) to extend the prediction to the whole network in the future.

Step 3: Preprocess the data from $\mathbf{C}$ by splitting it into the *trend* and the *residual* components via Fourier transform (see Section 3.3).

Step 4: Predict:

- the next sample in the trend term with the (Overlap-)SSA method;
- the next sample in the residuel term with a previously trained combined DBN-KELM model.

Step 5: Form the predicted compression matrix $\tilde{\mathbf{C}}$ by adding back the trend and residual terms after prediction - Equation 3.11.

Step 6: Obtain the prediction $\tilde{\mathbf{Q}}$ for the whole network by applying Equation 3.1 to $\tilde{\mathbf{C}}$ and the previously calculated relation matrix $\mathbf{X}$:

$$\tilde{\mathbf{Q}} = \tilde{\mathbf{C}}\mathbf{X} \tag{3.31}$$

4. Implementation

4.1. Data preparation and rectifying

On top of processing and merging some parts of the data to adapt it for the needs of the described method (see Section 3.1), analysis has revealed missing points and mistakes in the provided files, that needed to be solved first. For convenience, we wrote a Python script capable of resolving all these issues automatically given the original corrupt files.

1. Some entries in the *Look Up Table* contained spelling mistakes, i.e., extra or missing brackets, whitespaces, incorrect indices in detector names etc.
2. For some intersections, namely *2040*, *4170* and *8007*, the detector data are missing entirely, i.e., the tables are defined in corresponding csv-files, but are completely empty. Unfortunately, there were no alternative sources known for these data. Therefore, these intersections were ignored in further experiments.
3. Some files, for example, *DetCount_20210721.csv* for the intersection. *6010*, are missing the csv header. Assuming that the order of columns has to be the same for all files in the same folder, we copied the headers from files. This assumption was justified by the fact that this order was the same for all files having the header, and was alphabetical with regards to corresponding detector names.
4. Others, like *DetCount_20210429.csv* for the intersection *8403*, had only the *DATUM* column in them, with the names of detectors missing. Solved the same way as the completely missing headers by copying from other files in the folder.
5. This solution, however, would not work for the intersection *3050*, since none of it's files contained the header. We had to reconstruct it manually and insert it into one of the files before letting our script copy it to the other tables in the same folder.
6. Some files, e.g., *DetCount_20211215.csv* for *8403*, are missing values in several first rows. Our script fills them with zeros.
7. For intersection *6021*, there are files with two types of headers. Some, like *DetCount_20210101.csv*, contain columns for 9 detectors, while others, for example, *DetCount_20210416.csv* - for 11. For the first type of files, we inserted the missing 2 columns and filled them with zeros.
8. In all folders, the file *DetCount_20211031.csv* gives two conflicting readings on four time stamps from *02:00:00* to *02:45:00*. It is unclear what the problem has come from and which version is correct (since they are different), but we have decided to remove the first copy of these time stamps and keep the second one in each file.

9. Some 4-digit codes actually denoted groups of, rather than separate intersections. However, this was not reflected in the folder structure of the data set. Therefore, our script had to split the tables by extracting the detector columns from all csv-files and grouping them according the the real separation of these groups (as derived from the Look Up Table). These intersections and their splittings include:
    - *3060* - into *3060*, *3060_G* and *3060_L*;
    - *5090* - into *5090_Teilknoten 1* and *5090_Teilknoten 2*;
    - *6010* - into *6010_F1a* and *6010_F1b*.

4.2. Code structure and design

An official implementation for the approach authored by HAN and HUANG (2020) and described above is not provided online. For this project, we have resorted to implementing all methods by hand as much as the overview in the paper has allowed. Our code is written in *Python* using the *PyTorch* module for both data processing and model training. All essential steps were first realised and tested as separate *Jupyter Notebooks* - and then put together into a single module for running the complete procedure. This section provides details on how each portion of the method was implemented.

4.2.1. Procedure steps

This section explains how we have implemented all basic steps of the described procedure. The logic for all these steps is extracted to separate modules with step-related Jupyter notebooks merely calling their procedures for debugging. The last step uses all those modules combined to execute the whole method.

Step 1. Data extraction. As described in Section 4.1, our data provide the values separately for each detector on the entrance to the intersection. The expected traffic flow matrix $\mathbf{Q}$, on the other hand, has counts for each section combined, where the section is a connection between two intersections in one direction (see Section 3.1). Therefore, our first goal was to aggregate the lanes on each intersection by their sources and sum the values from their corresponding detectos. The information for this aggregation can be obtained from the provided Look Up Table.

Step 2. Data compression. As described in Section 3.2, the basis for the data compression step is computing the Pearson correlation coefficient via Equation 3.2 and building a matrix R from the computed values. PyTorch already provides an implementation for these two step in a one convenient function `torch.corrcoeff`. Whether the data are still corrupt or some sections of the network were completely unused in the analysed time period, but the input matrix $\mathbf{Q}$ contains columns filled entirely with zeros. This results in zero-division, breaking the computation of correlation coefficients. To tackle the issue, we exclude the zero-sequences on this step and work only with the remaining part of $\mathbf{Q}$. The rest of the steps is implemented more or less as supposed in the original paper, with the representative of each section group being selected as the first one indexed.

Step 3. Data preprocessing. The traditional way of using the discrete form of the Fourier tranform is *Fast Fourier Transform (FFT)*. PyTorch includes an implementation of this algortihm available as `torch.fft.fft`. Another function called `torch.fft.ifft` performes the inverse operation, i.e., converts a frequency-domain signal to a time-domain. Since the output of FFT is in complex numbers, we have to take their real part for further processing.

Step 4. Model training. The training process was not described in the paper explicitly, but could be derived from the models descriptions.

Before training and validating our methods on data, we have to normalize them to a fixed range. Since the sections vary a lot in their traffic activity, we deal with very different ranges in their raw forms. Therefore, we first find the largest value in each series to divide all entries by them and store them separately to multiply the predictions back after forecasting. This way, during the prediction process our methods have to only deal with values between 0 and 1. We can then turn to realizing the training procedure itself.

First of all, the adopted description of RBMs states that each node contains a binary value. This contradicts the fact that our data are given as integers of arbitrary size (or floating-point numbers after normalization). This discrepancy is never addressed in the original paper. Since pre-training via RBM optimization plays the same role as initialization in a traditional deep neural network, we assumed that the binary values used for RBM optimization are not related to the data used in the end, and performed this step with randomly initialized binary vectors.

Apart from that, it is mentioned in Algorithm 13 that the training procedure for DBN requires multiple refinement steps of gradient descent. KELM, on the other hand, is fitted in one single operation. At the same time, KELM expects features correctly extracted by a trained DBN, while DBN is trained using a loss function of an output of KELM. We need to propagate the DBN loss on each training step through a KELM which is trained instantly. The only consistent way to do so is to re-fit KELM after each iteration. Thus, our procedure looks like this:

1. Pre-train DBN as described in lines 1-10 of Algorithm 13.
2. Use the DBN to extract features from the training data.
3. Fit KELM to predict the next-step value from the extracted features.
4. Use the combined DBN+KELM model again to make predictions.
5. Calculate the loss using MSE (Equation 3.29), propagate it through the unified model and update DBN.
6. Repeat all the steps from 2 for 100 training epochs.

This way, we iterate between updating weights of DBN using an "old" KELM and fitting a "new" KELM for working with an updated DBN.

MSE (Equation 3.29) used for model training and per-section evaulation, as well as MAPE (Equation 3.30) for evaluation after reconstruction (via Equation 3.31) are both available in the `torchmetrics` module as `MeanSquaredError` and `MeanAbsolutePercentageError` respectively.

Han and Huang (2020) do not specify explicitly whether the same trained instance DBN-KELM is to be shared for all sections or a new one created for each of them. We have decided to try both approaches.

Finally, in the original paper, the authors highlight a difference in periodicity of the traffic flow on week days versus week ends. Hence, they consider both of them separately, making it necessary to create a data set that would consider only one type of days. We split the complete input matrix $\mathbf{Q}$ using a utility function that counts samples per day and uses the `datetime` library to separate the dates. Our final data class derived from `torch.utils.data.Dataset` uses a sliding window returning it's contents as an input and the next sample after it as the expected output. We During this sliding process, it also makes sure to not divide the window between consecutive regions (e.g., start on Sunday and end on Saturday of the next week). Our infrastructure also divides these data into a 80% to 20% proportion to evaluate the model right after training and select the best set of hyper parameters.

4.2.2. Prediction methods

Here we give brief details of our implementation for the prediction methods.

Restricted Boltzmann machine. For the RBM class, as well as the training algortihm, there is an available open-source algorithm (CONG, 2019). The only change we had to do to their code was extracting forward propagation step (the affine linear layers with `torch.sigmoid` activation) into a separate function. This was necessary as the original was intended as a standalone model, rather than a component for a larged DBN architecture.

Deep Belief Network. Since the DBN was to be fine-tuned using gradient descent, it was worth declaring it as a child class of `torch.nn.Module` with an override for the `forward` method using the forward propagation method added for the RBMs. The RBMs themselves are stored in a member `torch.nn.ModuleList`, which is a standard for implementing deep models in PyTorch.

Kernel extreme learning machine. Equation 3.28 shows that inputs x from the original training data for KELM have to be passed to the kernel function during inference on new inputs. Thus, they have to be stored as a class member when fitting the model. The same is true for the matrix β which depends on the training ground truth. Gradient descent would not be used in this model, but it would for fine-tuning of the DBN. And since KELM is to be attached directly to the DBN, both x and β have to support the gradient propagation performed by PyTorch's autograd. For this reason, we store them as instances of the `torch.nn.Parameter` class.

Singular spectrum analysis. Implementing the SSA, we closely followed GOLYANDINA and KOROBEYNIKOV (2014). Before the forecasting method, we tested the whole approach by using it for decomposition to check if adding extracted components back together yields the original signal. Since the hyper parameters I and L are shared between both applications of the method, it made sense to implement it as a whole new class, rather than a function, and store the I and L as it's members. On top of that, for successful forecasting we need to use the reconstraction as an intermediate step, while keeping an access to the eigentriples calculated in the process. For this purpose, we split the deconstruction into embedding, singular value decomposition and sequences generation implemented as three different methods of the class. On a lower level, like for the other methods, all matrix operations are done using PyTorch.

5. Experiments and results

5.1. Data analysis

Planning our experiments started with analysis of the data at hand. HAN and HUANG (2020) treat week days and week ends separately. The motivation behind this is the difference in periodicity. With data collected by the American Traffic Research Data Laboratory, each week they observe two peaks in traffic flow every day from Monday to Friday and only one on Saturday and Sunday. This does not correspond exactly to what we can see on the data from Ingolstadt. Or at the very least, not for all weeks and sections, as illustrated on Figure 5.1. Nevertheless, it is possible to see differences in data behaviour with an inclination towards what is indicated in the original work. While we cannot see clear peaks during week days, the most of activity is still spread more widely through the day. On week ends, however, the peak hours are concentrated in a narrower point with less traffic flow around it. We conclude that splitting data into week ends and week days is similarly justified in our case.

The next step in our analysis is checking the degree of correlation between sections of the network. This aspect of the data is relevant for the compression step described in Section 3.2. The outcome of that step is dependent on our choice of the α hyper parameter. Specifically, two sections are inserted into one group if their correlation coefficient exceeds the pre-defined α (i.e., $R(i, j) > \alpha$, as defined in Equation 3.2) or different ones otherwise. Therefore, raising the value of α decreases the probability of placing two sections into one group, thus potentially increasing the number of groups in the end. The exact dependency between the value of α and the resultant number of groups in our data specifically is given in Figure 5.2.

5.2. Outline

Our first step in the evaluation is to select a set of hyper parameters to minimize the total MSE loss. For that purpose, we select four whole consecutive weeks from *May 31 to June 27, 2021*.

The input window for each call to the model was 1 hour long in the original paper. For 5-minute sampling intervals, that amounts to 12 samples. However, since in our data the readings are sampled only every 15 minutes, that would produce 4 samples as the input. Thus, we have chosen to use a 3-hour window to keep the number of samples at 12. This

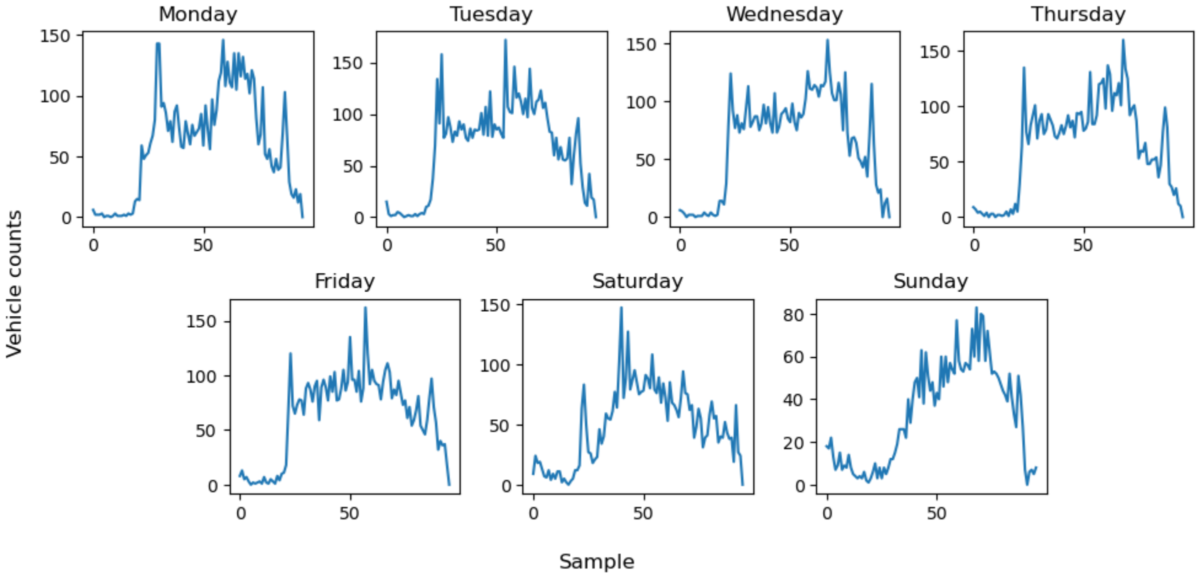
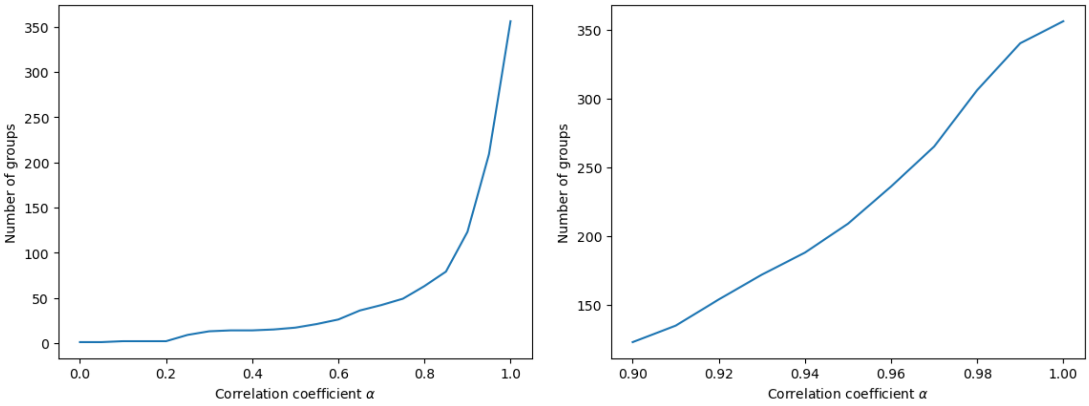


Figure 5.1: Traffic flow in vehicle counts per 15-minute sample taken on one section from the Ingolstadt road network during a week from *June 7* to *June 13, 2021*.



(a) $\alpha \in [0.00, 0.95]$

(b) $\alpha \in [0.90, 0.99]$

α	0.90	0.91	0.92	0.93	0.94	0.95	0.96	0.97	0.98	0.99
Number of groups	149	166	182	197	212	234	257	290	324	348
α	0.05	0.15	0.25	0.35	0.45	0.55	0.65	0.75	0.85	0.95
Number of groups	2	4	9	15	18	26	39	53	101	234

Figure 5.2: The number of groups formed from the sections in available data depending on the selected value of the hyper parameter α.

sliding procedure generates a sequence of 12-sample vectors and single-sample ground truths. We divide this sequence in a 80%/20% ratio to leave a part of data for validation. We use the resultant data set for training and validating DBN-KELM.

For trend prediction, it is not enough to use a 12-sample window. Since SSA is a non-parametric method, it's forecasting ability depends solely on data periodicity, which can be captured from a longer sequence only. Therefore, we extend the input window by shifting it's starting point to the beginning of the year. This approach could prove useful for the residual prediction as well, since it would give the model more data to train on. However, this would result in an excessive memory usage making it infeasible to fit KELM to the whole data. Therefore, we stick with applying both methods to different input periods ending at the same time point.

In the first session of runs, we determine the optimal values for the hyper parameters. There are three that we are interested in:

- spectral threshold P for splitting the series into two components as described in *Step 2* in the end of Section 3.3;
- output number of hidden nodes N of DBN, i.e., the dimensionality of extracted features to be passed to KELM;
- threshold α on correlation coefficient that defines the group splitting during the compression procedure (*Step 3* in the end of Section 3.2).

The first two parameters define per-section prediction, while the third is used only when utilizing the compression. Therefore, we adjust the parameters in two stages, using MSE loss to adjust P and N and MAPE for optimizing over α (Equations 3.29 and 3.30 respectively).

After that, we apply the procedure to all sections on the next week - from *June 28* until *July 4* - and report our results using the same losses.

With respect to the compression procedure, we first test the method's ability to preserve the original data in a compressed form without prediciton. For that, we observe the behaviour of the MSE on one of the sections for varying values of α. After that, we perform the same experiments with predicted values and compare both MSE and MAPE on a section that was not selected as a representative of it's group and whose trend was thus not predicted by the model directly.

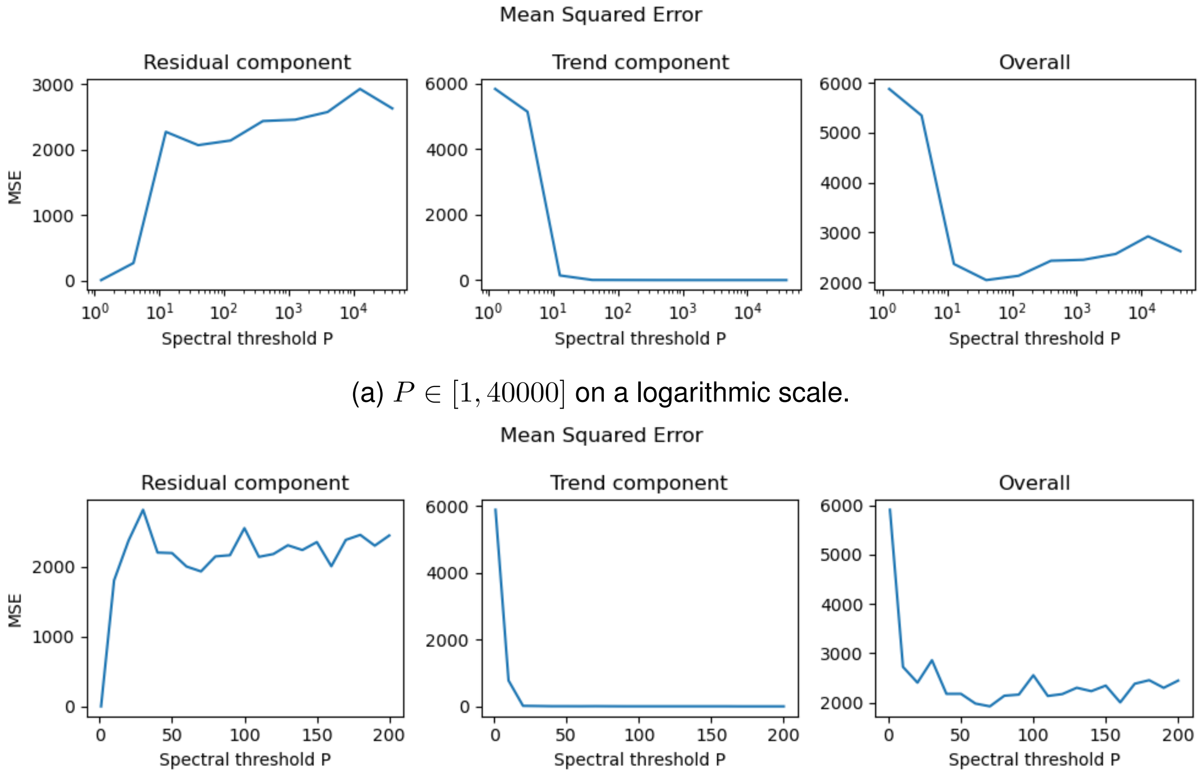
5.3. Results

Test runs of our method have proven infeasible to train one shared DBN-KELM on all sections data. Even if the model architecture is flexible enough to capture all intricacies in predicting the residuals, the memory requirements in that case could not be satisfied. This is due to the fact that fitting a KELM model requires storing the original data as a class member for further inference (see Equation 3.28). For 3-hour sliding windows shifted by 15 minutes each, this means $(24 - 3) \cdot 15 = 315$ input sequences per section per day. Storing them in an optimized form is not possible either since they are needed in calculation of β which is also stored. After unsuccessful attempts to run this configuration in various advanced machines, including the one with a *T4* GPU with *16 GB GPU RAM* available in *Google Colab*, we have resorted to training a separate model for each section. An alternative would be to train a model on a limited amount of data and still generalize it to all sections at once. This could be done by either training it on only one section, on just a handful of samples from multiple sections. Both approaches, however, are reasonable only under an assumption that the traffic flow is indeed uniform across all of them and the per-section prediction accuracy of our model is very high.

For the initial runs, we apply our method to six sections as a sanity check and select one of them with (*No. 160 in* $\mathbf{Q}$) more active traffic flow to optimize the hyper parameters. We ran the experiment through a grid search optimization (picking different pairs of values for hyper parameters and comparing the losses). The value of N does not seem to significantly affect the loss in the range we have tested it on ($N \in [4, 84, 164, 244]$). Hence, a further search was deemed unnecessary and we settled with the slightly better choice of $N = 84$. For P, we made an extensive search in various ranges from as low as $P = 1$ to as high as $P = 40000$. Figure 5.3 shows the reaction of the MSE loss on the adjustment in P. Increasing the value of P alters the way values are allocated between the two components, increasing the maximum value in the residuals and decreasing it in the trend. The overall minimum value is observed with $P = 70$. All three losses on the section are, however, very high - 5456.6, 2.95 and 5496.05 respectively. Averaged across all sections, the losses are 584.65, 126.20, 627.24.

Then, we apply the method with said parameters to in the whole testing week. The component separation achieved with the selected threshold is illustrated on Figure 5.4. It is worth mentioning that the mean value across all sections expected in the training portion is 4.89 times higher in the trend component than in the residuals. A prediction for one separate section is given on Figure 5.5.

Finally, we apply the proposed method for compressing network data. Figure 5.6 shows both errors on the *section 160* versus α. It illustrates the dependency for α in the whole meaningful range - $\alpha \in [0, 1]$. Unsurprisingly, both losses are very high when the value is close to 0, since this way we expect a complete reconstruction of the whole matrix $\mathbf{Q}$ from just a handful of groups. They, as expected, get much lower as α approaches 1 and more data are retained within the compressed form. Fusing the compression-reconstruction approach with prediction, we lose this clear descending trend and instead get a rise in MSE that peaks around $\alpha = 81$, as shown in Figure 5.7. Following the original paper, 5.7a the errors on the sections 161 which were not selected as a group representative, while 5.7b aggregates them for the whole network.



(a) $P \in [1, 40000]$ on a logarithmic scale.

(b) $P \in [1, 200]$ on a linear scale.

Figure 5.3: Values of MSE on two components and the combined data on the *160th section* compared with a varying value of the spectral threshold P.

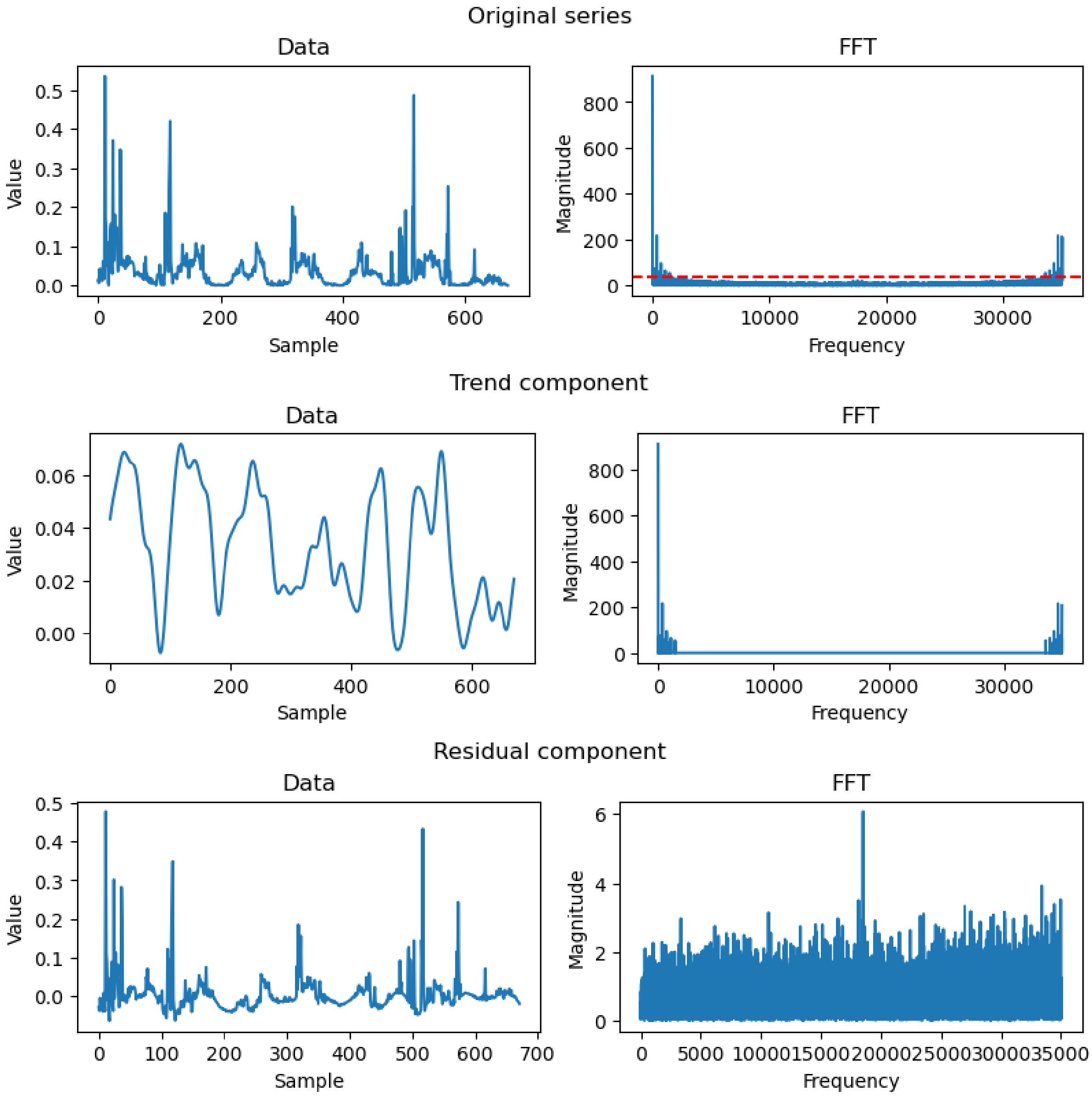
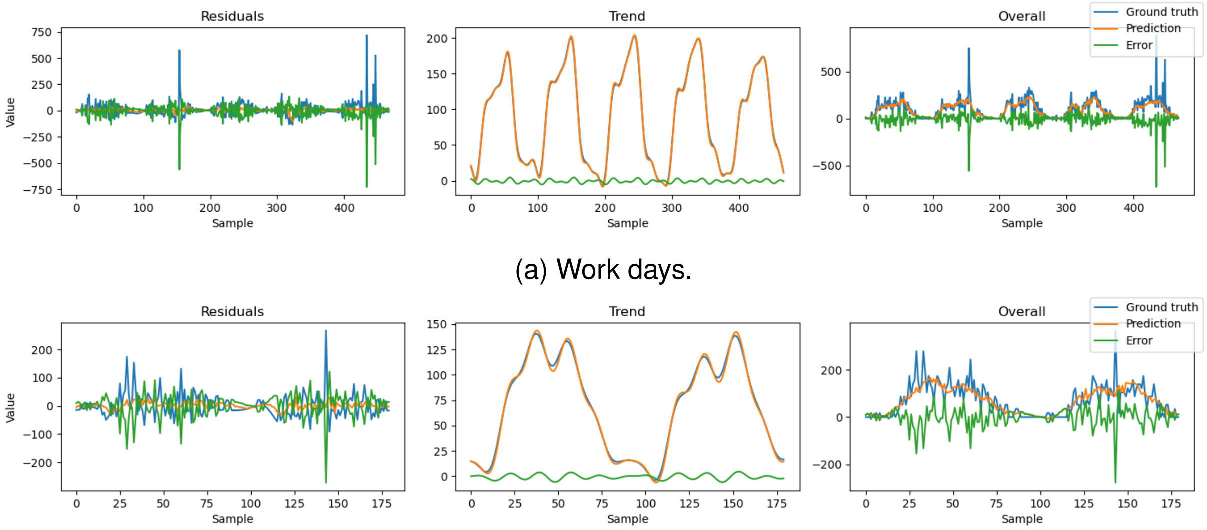


Figure 5.4: Spectral analysis of one week in data series and split into two components following the algorithm in Section 3.3. FFT values below the threshold $P = 40$ (dashed red line) are present in the residual component, but absent in the trend. The resultant extracted components on the left reflect this separation.



(a) Work days.

(b) Week ends.

Figure 5.5: Prediction results for the *160th section* during one week from *June 28* to *July 4, 2021*.

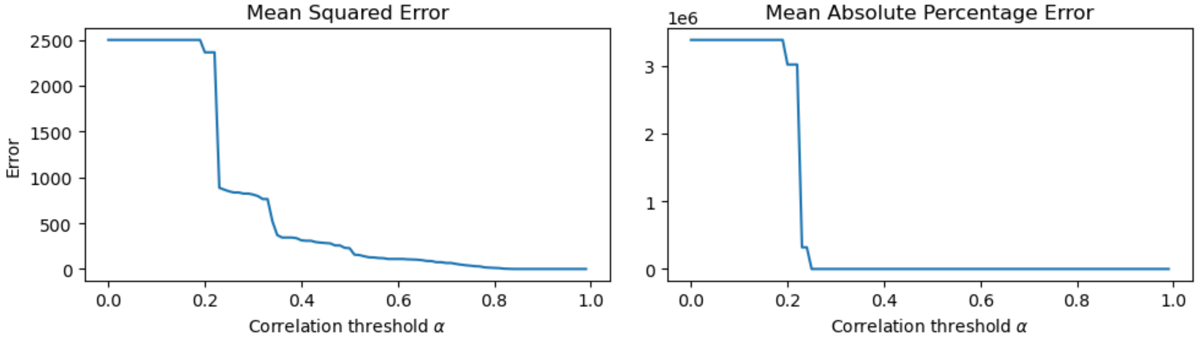
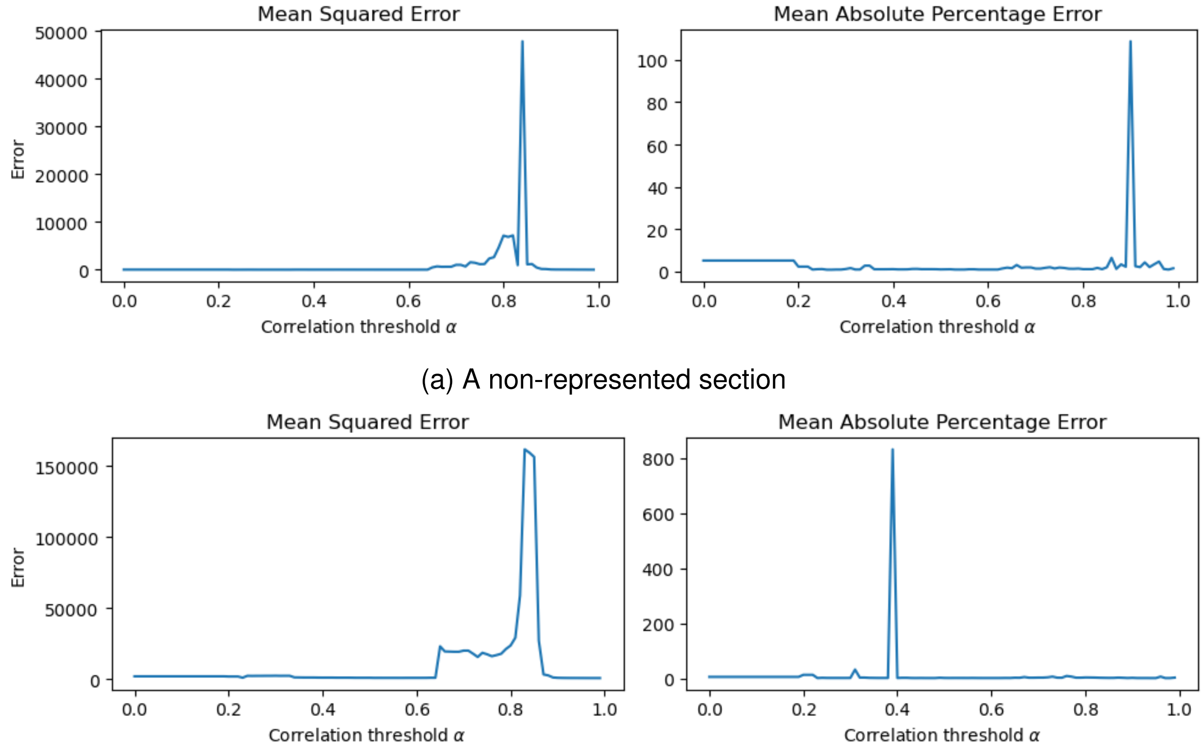


Figure 5.6: Reconstruction of network data from a compressed form without prediction.



(a) A non-represented section

(b) Whole network

Figure 5.7: Errors after reconstruction of the whole network from predicted sections with different values of α.

6. Discussion

The goal of this paper was to apply a deep learning-based technique for short-term prediction of traffic flow to data collected from detectors in a road network in a city of Ingolstadt in 2021. After researching various approaches and comparing them in relevance to our case, we have selected a method by HAN and HUANG (2020) called DBN-KELM used in conjunction with SSA. After studying the proposed method and implementing it in PyTorch, we adjusted hyper parameters empyrically and tested the procedures on the given data set.

Our findings show that the most significant hyper parameter for the resultant loss was the spectral threshold P used for splitting data into components, while adjusting the output number of nodes N gives only marginal and unpredictable improvements. However, even at the optimal configuration, DBN-KELM produces a very high loss on our data series, reaching up to 584.65 on average. SSA, on the other hand, shows surprisingly good results. There are still outliers within the sections that raise the overall loss, but averaged at 126.20, it is still 4.63 times lower than the residual component, exacerbated by the fact that an average value in the trend is almost five times greater than in the residual. For the compression part of the procedure, we observed that the loss of reconstruction went down as the correlation coefficient threshold α increased. Reconstructing after the prediction, we observed random fluctuation in both MSE and MAPE losses, as well as very high unexpected peaks around $\alpha = 0.81$.

The very high MSE loss that we have observed is drastically distinct from the values reported originally by HAN and HUANG (2020). Some ambiguous aspects of the method might serve as an explanation for our suboptimal results. The original method did not provide details for joint training procedures of both DBN and KELM models, leaving us with a gap for free interpretation. It was also unclear if a separate model was indeed to be used for each section, although doing it otherwise has proved infeasible with out computational resources. This point might not be the case for the original authors as they dealt with a much smaller data set, thus having to bear less struggle with processing power, as well as memory overhead. Finally, the section of the method related to using SSA for trend prediction was only mentioned briefly with all specifics left out, forcing us, among other things, to use the base version rather than a more refined Overlap-SSA.

For the method as a whole, the discrepancy in the observed results can be attributed to particulars of the data at hand. The first fact in that regard is a certain domain gap between our experiments. While our readings were collected from several hundreds of sections closely located in a densely populated city, the paper by HAN and HUANG (2020) dealt with a total of 19 criscrossed highways. Apart from aforementioned benefits this provides in term of computational resources, it is a known issue for many methods to perform differently in these two types of settings. Specifically, the dense area of dislocation introduces and amplifies additional variables, namely public events, accidents, business operations etc. This makes the behaviour of the flow predictable, requiring more powerful models and/or more training data. While all these peculiarities are practically random, the overall direction of the data remains mostly unchanged on a day-to-day basis. This is pointed to by the fact the SSA forecast that relies on the periodicity of the main trend component does perform much better than the deep part of the approach that deals with less consistent residuals.

Apart from that, our data lists readings collected over 15 minutes for each sample - 3 times less often, than for the original authors. To keep the data volume the same, we were forced to increase the width of the input time window. It is not unlikely that the given architecture is uncapable of dealing with a longer and coarser data that we have.

Last but not least, our set had a number of erroneous areas, demanding a rigorous rectifying process. Some of the issues with data required a certain extent of speculation about their properties, while others were completely intractable and forced us to omit the all together. These missing portions have a strong impact on indirect correlation between sections, and might have affected the compression procedure. Besides, this undermines our confidence in the rest of the data array that we have deemed valid.

Overall, despite mixed results, the model does seem to capture the behaviour of the series on the highest level. Specifically, the trend prediction by SSA follows a similar pattern as real data, capturing the periodicity of the everyday traffic flow and at the very least making this direction of research very promising. Our overview of other related works has supports this view, as well as proposes possible improvements to this experiment. These improvements may include, for example, incorporating additional data parameters (weather, social events, accidents etc.) to be considered by the deep learning part. For this, the learning-based DBN-KELM architecture has to be extended with modules of different architectures. LSTM, for example, is known to capture periodic data for long series. Considering the density of sections in a city setting, incorporating a CNN for capturing spatial dependencies of the additional parameters can be promising as well. In any case, both assertion and extention of the described procedure and our implementation of it requires new data with a higher sampling frequency and reliability.

To sum up, this project gave us an extensive overview on deep learning methods and how they are used for the short-term analysis of traffic flow on road networks, as well as a first-hand experience on implementing some of these methods and working with data. Despite mixed results, the project has proven very educational, and the selected method very promising at least.

List of Figures

List of Tables

Bibliography

BISHOP, CHRISTOPHER M. (2006). “Pattern Recognition and Machine Learning”. In: *Machine Learning* 128.9.

CHAN, KIT YAN; THARAM S. DILLON; JAIPAL SINGH; ELIZABETH CHANG (2012). “Neural-Network-Based Models for Short-Term Traffic Flow Forecasting Using a Hybrid Exponential Smoothing and Levenberg–Marquardt Algorithm”. In: *IEEE Transactions on Intelligent Transportation Systems* 13.2, pp. 644–654. DOI: 10.1109/TITS.2011.2174051.

CHUNG, JUNYOUNG; CAGLAR GULCEHRE; KYUNGHYUN CHO; Y. BENGIO (Dec. 2014). “Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling”. In.

CONG, BAC NGUYEN (2019). *rbm-pytorch*. https://github.com/bacnguyencong/rbm-pytorch.

GHOJOGH, BENYAMIN; ALI GHODSI; FAKHRI KARRAY; MARK CROWLEY (July 2021). “Restricted Boltzmann Machine and Deep Belief Network: Tutorial and Survey”. In.

GOLYANDINA, NINA; ANTON KOROBEYNIKOV (Mar. 2014). “Basic Singular Spectrum Analysis and Forecasting with R”. In: *Computational Statistics I& Data Analysis* 71, pp. 934–954. DOI: 10.1016/j.csda.2013.04.009.

GRAVES, ALEX; NAVDEEP JAITLY; ABDEL-RAHMAN MOHAMED (Dec. 2013). “Hybrid speech recognition with Deep Bidirectional LSTM”. In: pp. 273–278. DOI: 10.1109/ASRU.2013.6707742.

GUO, JIANHUA; WEI HUANG; BILLY WILLIAMS (June 2014). “Adaptive Kalman filter approach for stochastic short-term traffic flow rate prediction and uncertainty quantification”. In: *Transportation Research Part C: Emerging Technologies* 43. DOI: 10.1016/j.trc.2014.02.006.

HAN, LEI; YI-SHAO HUANG (2020). “Short-term traffic flow prediction of road network based on deep learning”. In: *IET Intelligent Transport Systems* 14.6, pp. 495–503. DOI: https://doi.org/10.1049/iet-its.2019.0133. eprint: https://ietresearch.onlinelibrary.wiley.com/doi/pdf/10.1049/iet-its.2019.0133. URL: https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/iet-its.2019.0133.

HASSANNAYEBI, ERFAN; CHUANXIANG REN; CHUNXU CHAI; CHANGCHANG YIN; HAOWEI JI; XUEZHEN CHENG; GE GAO; HENG ZHANG (July 2021). “Short-Term Traffic Flow Prediction: A Method of Combined Deep Learnings”. In: *Journal of Advanced Transportation* 2021, p. 9928073. ISSN: 0197-6729. DOI: 10.1155/2021/9928073. URL: https://doi.org/10.1155/2021/9928073.

HE, KAIMING; XIANGYU ZHANG; SHAOQING REN; JIAN SUN (June 2016). “Deep Residual Learning for Image Recognition”. In: pp. 770–778. DOI: 10.1109/CVPR.2016.90.

HINTON, GEOFFREY; SIMON OSINDERO; YEE-WHYE TEH (Aug. 2006). “A Fast Learning Algorithm for Deep Belief Nets”. In: *Neural computation* 18, pp. 1527–54. DOI: 10.1162/neco.2006.18.7.1527.

HINTON, GEOFFREY E.; TERRENCE J. SEJNOWSKI (1983). “Optimal Perceptual Inference”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. Vol. 448. IEEE.

HUANG, WENHAO; GUOJIE SONG; HAIKUN HONG; KUNQING XIE (Oct. 2014). “Deep Architecture for Traffic Flow Prediction: Deep Belief Networks With Multitask Learning”. In: *Intelligent Transportation Systems, IEEE Transactions on* 15, pp. 2191–2201. DOI: 10.1109/TITS.2014.2311123.

ISING, ERNST (1925). “Beitrag zur Theorie des Ferromagnetismus”. In: *Zeitschrift für Physik* 31.1, pp. 253–258.

KARLAFTIS, M.G.; E.I. VLAHOGIANNI (2011). “Statistical methods versus neural networks in transportation research: Differences, similarities and some insights”. In: *Transportation Research Part C: Emerging Technologies* 19.3, pp. 387–399. ISSN: 0968-090X. DOI: https://doi.org/10.1016/j.trc.2010.10.004. URL: https://www.sciencedirect.com/science/article/pii/S0968090X10001610.

KRIZHEVSKY, ALEX; ILYA SUTSKEVER; GEOFFREY HINTON (Jan. 2012). “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Neural Information Processing Systems* 25. DOI: 10.1145/3065386.

LAMAS, JOSE; PAULA-MARIA CASTRO-CASTRO; ADRIANA DAPENA; FRANCISCO VAZQUEZ-ARAUJO (Aug. 2016). “SiDIVS: Simple Detection of Inductive Vehicle Signatures with a Multiplex Resonant Sensor”. In: *Sensors* 16, p. 1309. DOI: 10.3390/s16081309.

LELES, MICHEL; J.P.H. SANSÃO; L.A. MOZELLI; H.N. GUIMARÃES (Nov. 2017). “Improving reconstruction of time-series based in Singular Spectrum Analysis: A segmentation approach”. In: *Digital Signal Processing* 77. DOI: 10.1016/j.dsp.2017.10.025.

LIU, LINGBO; JIAJIE ZHEN; GUANBIN LI; GENG ZHAN; LIANG LIN (2019). “Dynamic Spatial-Temporal Representation Learning for Traffic Flow Prediction”. In: *IEEE Transactions on Intelligent Transportation Systems* 22, pp. 7169–7183. URL: https://api.semanticscholar.org/CorpusID:210473164.

MIN, WANLI; LAURA WYNTER (2011). “Real-time road traffic prediction with spatio-temporal correlations”. In: *Transportation Research Part C: Emerging Technologies* 19.4, pp. 606–616. ISSN: 0968-090X. DOI: https://doi.org/10.1016/j.trc.2010.10.002. URL: https://www.sciencedirect.com/science/article/pii/S0968090X10001592.

MUKHERJEE, S.; E. OSUNA; F. GIROSI (1997). “Nonlinear prediction of chaotic time series using support vector machines”. In: *Neural Networks for Signal Processing VII. Proceedings of the 1997 IEEE Signal Processing Society Workshop*, pp. 511–520. DOI: 10.1109/NNSP.1997.622433.

MÜLLER, K. R.; A. J. SMOLA; G. RÄTSCH; B. SCHÖLKOPF; J. KOHLMORGEN; V. VAPNIK (1997). “Predicting time series with support vector machines”. In: *Artificial Neural Networks — ICANN’97*. Ed. by WULFRAM GERSTNER; ALAIN GERMOND; MARTIN HASLER; JEAN-DANIEL NICOUD. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 999–1004. ISBN: 978-3-540-69620-9.

QI, YAN; SHERIF ISHAK (June 2014). “A Hidden Markov Model for short term prediction of traffic conditions on freeways”. In: *Transportation Research Part C: Emerging Technologies* 43. DOI: 10.1016/j.trc.2014.02.007.

Shi, Xingjian; Zhourong Chen; Hao Wang; Dit-Yan Yeung; Wai Kin Wong; Wang-chun WOO (June 2015). “Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting”. In.

Tsirigotis, Lykourgos; Eleni Vlahogianni; Matthew Karlaftis (Jan. 2012). “Does Information on Weather Affect the Performance of Short-Term Traffic Forecasting Models?” In: *International Journal of Intelligent Transportation Systems Research* 10, pp. 1–10. doi: 10.1007/s13177-011-0037-x.

Wang, Jian; Wei Deng; Yuntao Guo (2014). “New Bayesian combination method for short-term traffic flow forecasting”. In: *Transportation Research Part C: Emerging Technologies* 43. Special Issue on Short-term Traffic Flow Forecasting, pp. 79–94. issn: 0968-090X. doi: https://doi.org/10.1016/j.trc.2014.02.005. url: https://www.sciencedirect.com/science/article/pii/S0968090X14000370.

Yang, Z.; Y. Wang; Q. Guan (2006). “Short-term Traffic Flow Prediction Method Based on SVM”. In: *Journal of Jilin University (Engineering and Technology Edition)* 36, pp. 881–884.

Yuankai, Wu; Huachun Tan (Dec. 2016). “Short-term traffic flow forecasting with spatial-temporal correlation in a hybrid deep learning framework”. In.

Zhang, Da; Mansur Kabuka (Mar. 2018). “Combining Weather Condition Data to Predict Traffic Flow: A GRU Based Deep Learning Approach”. In: *IET Intelligent Transport Systems* 12. doi: 10.1049/iet-its.2017.0313.

Zhang, Junbo; Yu Zheng; Dekang Qi (Feb. 2017). “Deep Spatio-Temporal Residual Networks for Citywide Crowd Flows Prediction”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 31.1. doi: 10.1609/aaai.v31i1.10735. url: https://ojs.aaai.org/index.php/AAAI/article/view/10735.

Zhang, Junbo; Yu Zheng; Dekang Qi; Ruiyuan Li; Xiuwen Yi (2016). “DNN-based prediction model for spatio-temporal data”. In: *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. SIGSPACIAL ’16. Burlingame, California: Association for Computing Machinery. isbn: 9781450345897. doi: 10.1145/2996913.2997016. url: https://doi.org/10.1145/2996913.2997016.

Zhang, Lun; Qiuchen Liu; Wenchen Yang; Wei Nai; Decun Dong (Nov. 2013). “An Improved K-nearest Neighbor Model for Short-term Traffic Flow Prediction”. In: *Procedia - Social and Behavioral Sciences* 96, pp. 653–662. doi: 10.1016/j.sbspro.2013.08.076.