HOCHSCHULE
HANNOVER
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS
–
Fakultät IV
Wirtschaft und
Informatik

Training and evaluating deep learning models on road graphs for traffic prediction using SUMO

Vogel, Erik

Bachelorarbeit im Studiengang "Angewandte Informatik"

7. August 2023

Autor: Erik Vogel
1630217
erik.vogel@stud.hs-hannover.de

Erstprüfer: Prof. Dr. Carsten Kleiner
Abteilung Informatik, Fakultät IV
Hochschule Hannover
carsten.kleiner@hs-hannover.de

Zweitprüfer: Robin Buchta, M.Sc.
Abteilung Informatik, Fakultät IV
Hochschule Hannover
robin.buchta@hs-hannover.de

Selbstständigkeitserklärung

Mit der Abgabe der Ausarbeitung erkläre ich, dass ich die eingereichte Bachelorarbeit selbständig und ohne fremde Hilfe verfasst, andere als die von uns angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Werken wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht haben.

Hannover, den 7. August 2023

Contents

1 Introduction

With more and more traffic each year, the risk of accidents, congestion, longer travel times, and other factors caused by an increased traffic volume affect transportation in cities more and more. This problem can be solved by building more road infrastructure. But this approach is not always feasible due to multiple factors like space constraints and can even lead to an overall slowdown of traffic. This phenomenon is known as the "Braess's paradox" discovered by *Arthur Pigou* [Pig02] and refined by *Dietrich Braess* [Bra68]. It describes that adding more roads to a network in which each entity chooses its path selfishly can sometimes reduce overall performance [Bra68, Pig02].

As building new roads is not the only solution to the problem, Intelligent Transport Systems (ITS) provides an alternative way to improve traffic conditions without the need for new roads.

ITS aims to improve traffic flow by collecting and processing data using various applications like Traffic Management Systems, Transit Signal Priority, Emergency Vehicle Preemption, and much more [QA13].

One of the key aspects of Intelligent Transport Systems is the prediction of traffic factors such as speed, occupancy, etc. as these can be effectively used for the above-mentioned applications.

Traffic prediction describes the process of predicting future traffic factors based on historical data. Traffic prediction can be based on a road/network-wide- (macroscopic), lane based- (mesoscopic), or vehicle-based scale (microscopic) with each scale describing different characteristics of traffic and being used for different applications [ZYZ$^+$22].

In recent years, research in traffic prediction saw a great amount of interest and lots of different methods to optimize the task have been proposed, as highlighted by several recent surveys [BCY22] [JL22] [ZYZ$^+$22]. These algorithms range from model-based methods in the early days to current data-driven methods, specifically deep learning methods optimized for operating on graphs, as they are currently considered to be the most promising approach [ZYZ$^+$22].

These deep learning algorithms are normally trained on a selection of available real-life datasets containing data from traffic sensors with the most popular being the METR-LA and PeMS datasets [JL22]. These are only the most commonly used datasets, but other datasets from other sources are also available. Also, other data can be used to perform speed prediction like GPS data, Trip Data, weather data, and much more [JL22]. For more information on this topic refer to [JL22] as it provides a comprehensive overview of available data sources.

But as the amount of real-life data is limited, it doesn't cover all possible scenarios, and using simulations for testing models on specific scenarios like concerts or accidents from which no or only a few public datasets are available, can provide a good addition to the currently available training and testing datasets. It also has "the potential of modeling unseen graphs though, e.g., evaluating a planned road topology." [JL22].

To achieve this a framework is needed that connects a traffic simulator with deep learning libraries like PyTorch or Keras. Traffic simulators have been around for a long time with multiple simulators being available and under active development until today. But as most of them are not free, an open-source (or free) simulator is needed so that the framework can be used by everyone. Therefore, SUMO is a good choice, due to it being entirely open-source and also having extensive Python support [ALBB+18] making it easy to integrate into a Python application.
This thesis aims to further explore the possibility of using traffic simulations to train or test deep learning models for traffic prediction by using the open source, microscopic traffic simulator SUMO (Simulation of Urban MObility) [ALBB+18]. For this, a general framework for training/testing models using NumPy and SUMO will be proposed and the general approach of using simulated traffic data for this use case will be further discussed.

2 Related Works

The idea of using traffic simulations for traffic prediction is not new and has already been applied in some works. *Fukuda et al.* [FUFY20] used the microscopic traffic simulation MATES [Yos06] to evaluate the performance of graph convolutional recurrent neural networks under unusual conditions like accidents. *Song Sang et al.* [KZY+18] successfully used SUMO to test a "reinforcement learning method to optimize the route of a single vehicle in a network" [KZY+18].
SUMO has also successfully been used for deep learning use cases not directly related to traffic prediction, like by *Song et al.* [SZL+22] who used a traffic simulation with SUMO as the environment to evaluate deep reinforcement learning electric vehicle(EV) dispatching algorithms with the goal of optimizing the efficiency of EV charging stations [SZL+22].
While all these studies achieve great results, no source code of the software used to connect the models with the simulation is publicly available. And the approach of using data from a simulation for the training and testing of graph neural networks has not been thoroughly discussed.
A couple of open-source solutions that connect traffic simulations and deep learning already exist. CARLA [DRC+17] is a 3D simulator developed specifically for autonomous urban driving systems, including support for associated deep learning algorithms. A more general approach is taken by FLOW [KPW+18] which connects SUMO with deep reinforcement learning using rllab. This project however seems to be no longer under active development, as development stalled after the end of 2020. Also, rllab, which is used as the library to implement the deep learning algorithms, is officially no longer under active development. Development on the community-driven project "garage" that continued development after the official shutdown has also stalled in the last years [mai23].
As most models currently proposed for traffic prediction are implemented in either PyTorch, Keras, or TensorFlow [JL22], the framework proposed in this thesis will aim to

connect SUMO with these libraries using NumPy to allow existing models to use data generated by SUMO with only minor modifications.
According to this research, there are currently no other open-source projects that connect graph neural network models with traffic simulators using the currently most used libraries for deep learning. This is also supported by *Wang et al.* [WJJ+21] who conducted a literature survey on traffic prediction and found that "there are no open-source libraries for unifying the entire pipeline consisting of data preparation, model design and implementation, and performance evaluation"[WJJ+21] for simulated traffic data.
The proposed framework in the thesis is the first step to such a platform, enabling data collection, simulation control, and evaluation specifically designed for traffic predictions.

3 Prerequisites and Underlying Basic Technology

The framework introduced in this thesis is based on two technologies: (one) SUMO, which is the traffic simulator used to run the traffic simulation and (two) graph neural networks. After the data is extracted from SUMO it is processed to be used with graph neural networks, as they are the currently most promising approach for traffic prediction [ZYZ+22].
In order to fully understand the framework and its functionality it is therefore important to understand all these underlying technologies. This section will therefore introduce both technologies and highlight all important aspects needed for understanding the proposed framework.

3.1 SUMO

The simulation used for this thesis is the open-source traffic simulator SUMO originally developed by the DLR (Deutsches Zentrum für Luft- und Raumfahrt) in 2001 [BBEK11] and is now developed by the "openMobility Working Group at the Eclipse Foundation" [SUM]. SUMO has already been used in many projects, especially within the research fields of Vehicular Communication, Route Choice and Dynamic Navigation, and Traffic Light Algorithms [KEBB12]
This section is supposed to give an overview of the main components and workflows in SUMO and give a basic understanding of what SUMO is and how it can be used. This is done as SUMO is an integral part of this thesis and framework and a good understanding of all underlying components is necessary to fully understand all parts of this thesis. It also especially highlights and explains the features used in the proposed framework.

3.1.1 Overview

SUMO describes itself as a "highly portable, microscopic and continuous multi-modal traffic simulation package designed to handle large networks" [SUM]. It in itself however does not consist of one single application, but is a whole suite of different applications and tools supporting the core simulation [KEBB12]. These tools are used for use cases like road network generation, demand modeling, and much more as SUMO uses its own format for networks and traffic data [KEBB12]. The general workflow for creating and running a SUMO Scenario consists of the following steps:

- Creation of the road network either by hand or by importing data from other sources like other simulators, OpenStreetMap, etc. It is usually represented as a graph.
- Creation of traffic demand for the created network by either using one of several tools provided by SUMO or public data provided by traffic authorities. It is saved as a collection of individual trips for all vehicles appearing in the simulation.
- Running the simulation after the traffic network and its demand has been defined.

Of course, this describes only the most basic workflow. With more complex scenarios, the workflow complexity can increase. For more information on each step, refer to the sections below.

Each scenario can then be imported and run using SUMO. Each simulation (or scenario) has a specific runtime which is defined by the number of trips, as each trip contains a start time, a vehicle, and its path. Time is represented as *simulation steps* which usually last/represent one second each (the length of a simulation step can also be customized, but a value of one second is generally recommended). The simulation ends after the last vehicle finishes its trip.

SUMO also includes an interface for communication with the simulator using Python called "TraCi" [KZY+18], originally developed for evaluating Vehicular Ad-Hoc Networks, communicating with SUMO via a socket [WPR+08]. It allows the user to control most aspects of the simulation and modify and retrieve lots of data from the simulation. This library is used in this framework for communication between SUMO and the framework.
The following two subsections will explain some of the above-mentioned steps in more detail. For more information about creating simulations in SUMO, either refer to [ALBB+18] or the official SUMO documentation.

3.1.2 Network Generation

A SUMO network is represented by a unidirectional graph with nodes and edges representing lanes of different kinds (road lanes, bike lanes, pathways, etc.). Each edge has a variety of constant attributes assigned to it that describe numerous features about the associated lane like speed, shape, permissions, etc. Consequently, nodes represent all junctions where different lanes meet or where attributes of a lane (like the speed limit) change, as the attributes for each edge of the graph are constant. [ALBB+18]
SUMO provides the user with a tool called *NETEDIT* which is a graphical interface for creating, analyzing, and editing such networks. Since creating networks from scratch is very time-consuming, SUMO also provides the tool *NETCONVERT* which can be

used to convert data from other sources including OpenStreetMap or OpenDRIVE to scenarios usable by SUMO. [ALBB$^+$18].

3.1.3 Demand Modelling

After the successful generation of the road network, the next step is the generation of traffic demand. SUMO supports the definition of this by individual trips, flows, or as routes [ALBB$^+$18]. There are currently three ways to generate traffic demand in SUMO [UCBBC20]:

- Create routes manually by defining each vehicle's route by hand. Each route consists of a list of IDs corresponding to the edges (=streets) on the vehicle's route.
- A quick way to generate trips is using randomly generated routes using the *randomTrips.py* script provided by SUMO. The results of using this tool can however be highly unrealistic.
- If more information about the scenario is available, SUMO provides applications to generate trips from multiple different data sources, with some of the most popular being: (one) Origin/Destination Matrices. OD Matrices can be acquired from traffic authorities, and be used to generate trips for SUMO using the tool *OD2TRIPS* [ALBB$^+$18]. (two) *ACTIVITYGEN* can generate trips using the network file and a population definition. It supports the activities of school, work, and free time and bike, walking, bus, and cars as modes of transport [WBF, ALBB$^+$18]. (three) *DFROUTER* generates routes using data from traffic detectors like induction loops. This data can also sometimes be acquired from traffic authorities or by one of the above-mentioned open source data sets [SUM23a, ALBB$^+$18]. (4) *JTR-ROUTER* can generate routes from traffic volumes and turning ratios at junctions or interchanges [SUM23b, ALBB$^+$18].

For more detailed information about each application, refer to the official SUMO documentation. The documentation also includes information about other methods and tools not mentioned above. The choice of tool to be used for modeling the simulation trips is mostly dependent on the data available to the user as some methods require specific data that might not be available for the chosen network.
To assert the accuracy of the mentioned methods of demand generation, multiple studies have been conducted that compare the performance of select methods in different scenarios. *Urquiza-Aguiar et al.* compared different routers used by SUMO for trip generation using graph metrics using the financial district of Quito [UCBBC20] and the access highways to Quito [UCGBBC19] converted from OpenStreetMap. Lastly, *Ma et al.* [MHWS21] used the Jianghan Zone in Wuhan and *ACTIVITYGEN* in combination with *DUAROUTER* to evaluate the accuracy of traffic simulated by SUMO.
All the above-mentioned tools are shipped with SUMO. But over the recent years other methods to calibrate traffic simulators (not specifically focussing on SUMO) have been introduced like Cadyts [Flö09] which "calibrates the disaggregate demand in the simulation from readily available sensor data such as traffic counts" [Flö09].

3.1.4 Induction Loops

The most important feature of SUMO for speed prediction is induction loops. SUMO supports three types of traffic detectors (= induction loops): (one) Induction Loop Detectors (E1) measure data (like speed) on only one specific point on a specified lane [SUM22a]. (two) Lane area Detectors (E2) collect data along a specific area on a lane [SUM22b]. (three) Multi-Entry-Exit Detectors (E3) collect data between a set of entry and exit detectors [SUM22c]. This section aims to describe the functionality and placement of these detectors in SUMO as they form the core part of the proposed framework. Therefore a deep understanding of some of their properties is needed to understand the core parts of the framework.

All detectors are defined using a similar scheme in an *additional* XML file. The needed attributes vary by detector type, but in general contain the ID of the detector, the ID of the lane on which the detector is placed, the position of the detector on that lane, the data aggregation period, and the file in which the output data should be written.
The position is here defined as the distance between the start of the lane and the detector following the lane's shape. The aggregation period describes the *interval* in simulation steps, where data should be collected and averaged.

Detectors can either be defined by creating entries in the additional file manually or by using Netedit [ALE16]. There is currently no way to place detectors automatically. An example entry of a detector definition looks as follows:

<e1Detector id="<id>" lane="51_0" pos="42" freq="1800" file="output.xml"/>

The output data of each detector is then written into an XML file. This data again varies between the different detector types and includes data like average speed, passed vehicles, occupancy, etc.
The data of detectors can also be accessed using TraCi. TraCi allows the user to retrieve data of *E1 detectors* while the simulation is running. [Tra22] The most important values that can be retrieved using TraCi are the mean speed, occupancy, and vehicle count. This feature is used in the framework described in this thesis, which therefore *only supports E1 detectors.*

3.2 Graph Neural Networks

During the ongoing research on traffic prediction, the latest trend is making predictions using "graph neural networks". Graph neural networks extend the functionality of normal neural networks in order to improve the model's performance on tasks based on graphs.

A graph neural network (at a high level) typically contains two types of features:

- The **node features/embeddings** are the features (=representation) of each node in the graph. Node features can contain all kinds of data, similar to the input data of normal neural networks. The node features can be represented, for example, by a matrix of size $(num_{nodes}, num_{features}, features)$. In relation to graph neural networks used in traffic prediction, these features are also called the *temporal information/features*, as they describe the state of one node (=detector) at one point in time (e.g. the average speed).
- The **graph structure** is the second major component of graph neural networks and contains information about the graph's structure. It can be represented as an array of size $(num_{edges}, 2)$, containing all edges of the graph as source-target pairs. In relation to graph neural networks used in traffic prediction, this structure is also called spatial information/features.

Graph neural networks containing both spatial and temporal information are called *spatial-temporal graph neural networks* or *ST-GNNs*. There are currently many types of ST-GNNs proposed, which (according to a survey conducted by Nam Bui et al. [BCY22]) can be classified by their mathematical methods used as follows:

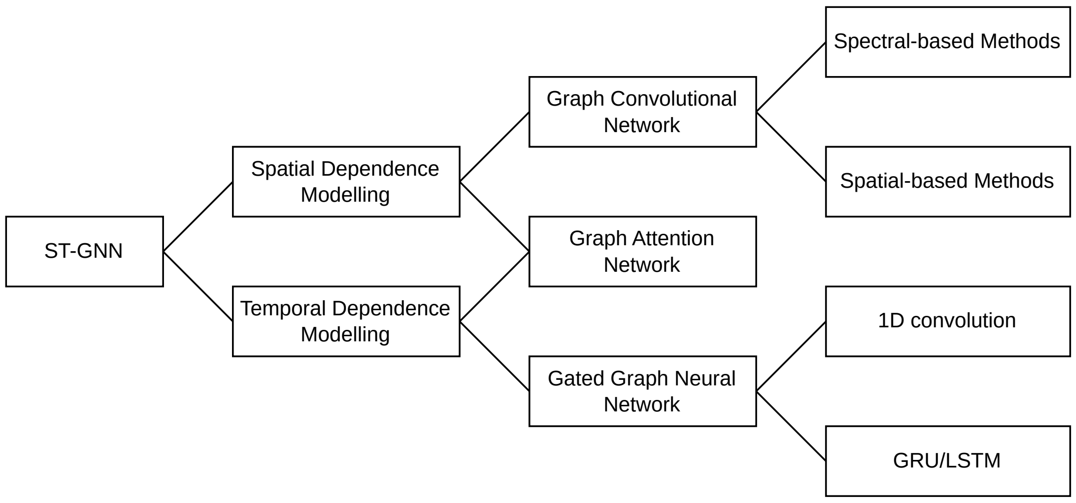

ST-GNN
Spatial Dependence Modelling
Temporal Dependence Modelling
Graph Convolutional Network
Graph Attention Network
Gated Graph Neural Network
Spectral-based Methods
Spatial-based Methods
1D convolution
GRU/LSTM

Figure 1: Overview of ST-GNN based methods [BCY22].

For more information on each individual method, refer to [BCY22, JL22].
These methods can be used for various different use cases on a node-, edge- and graph level. The basic underlying principle of each graph neural network is the so-called *message passing*, which on very high-level works as follows:
(one) Collect all node features (also called messages) of neighboring nodes for each node in the graph. (two) Aggregate all messages using an aggregation function (three) Pass new messages through an update function (usually a neural network) [SRPW21]. For more info about the fundamentals of graph neural networks refer to [SRPW21, Ham20].

This thesis will later use the so-called ST-GAT (Spatio-Temporal Graph Attention Network) for testing the proposed framework. This Model uses both spatial and temporal features and according to Figure 1 can be classified as a mixture of a "Graph Attention Network" and "GRU/LSTM" as it uses an attention mechanism for the spatial and an LSTM mechanism for the temporal features. For more detailed information about this model refer to [ZYL19]

4 Requirements

SUMO already provides a vast toolset for simulating traffic and collecting data from the simulation. But for the specific use case described in this thesis of extracting data from the simulation and using it to train neural networks, additional infrastructure is necessary. This section describes the additional requirements for the proposed framework, which can not be covered by using SUMO alone. For more information on SUMO refer to subsection 3.1 and [ALBB+18]

Continous historical data storage
SUMO itself only offers the collection of data for the last or current interval via TraCi or for the whole simulation as an XML file after the simulation finishes. A requirement of DeepSUMO is therefore to allow for continuous and historical data collection of data from SUMO during simulation runtime.
The framework should be able to:

- Store all data collected from the simulation until the program terminates
- Continuously update/add new data when it becomes available
- Collect and update data dynamically during the simulation's runtime

Real time interfacing
DeepSUMO should also provide a way to interact with the data collected by DeepSUMO while the simulation is running to provide more possibilities for the user.
The framework should be able to:

- Offer a mechanism to interact with the simulation and associated data at any point in time.
- Provide access to not only the simulation but also to the up-to-date collected data throughout the simulation.

Create detector graph
SUMO does not offer much positional information about detectors. As most up-to-date models require the relationship between detectors to be modeled as a graph it is also one of DeepSUMOs primary requirements to have the ability to create such a graph structure from the configured detectors.

The framework should be able to:

- Create a data structure containing a graph consisting of all connectors
- Offer different metrics for connecting the graph (e.g. distance, Dijkstra)
- Allow the user to create their own metrics for defining the relationship between two nodes

Easy and logical data access and integration
All the abovementioned data should be saved in a format specialized for use with deep learning models and make it easy to access the data and integrate it into (existing) deep learning models. Access to this data should also be unrestrictive.
The framework should be able to:

- Format and structure all data in an easy-to-integrate and understandable way
- Provide unrestricted access to all relevant data

Detector generation
SUMO provides no way to automatically place detectors on a road network. Placing all detectors by hand can be tedious depending on the size of the network. So it is a requirement for DeepSUMO to offer a way of automatically generating detectors on a given road network to make the scenario generation easier.
The framework should be able to:

- Allow the user to automatically place detectors on a specific road network
- Place detectors in the network based on multiple parameters like distance, street type, etc.
- Allow the user to adjust the parameters for the placement of detectors

Flow control
Another way to make the creation of scenarios easier is for DeepSUMO to have the ability to control the simulation in a way that traffic demand varies by time of day and weekday (e.g. on Monday at 8 am there are more cars on the road than on Sunday at 1 am). As modeling demand dynamically during the scenario generation can provide a challenge, as most tools only provide a static stream of vehicles, and there is currently no tool to dynamically scale the demand during simulation runtime.
The framework should be able to:

- Provide a way the control the amount of traffic in simulation based on the current weekday and hour
- Define rules for the traffic demand in an easy-to-understand format
- Allow the user to configure the rules on which the traffic is adjusted

5 DeepSUMO

This section will initially introduce the proposed framework by highlighting its data structure, context, workflow, and basic components in the subsections below. For more information on each individual component, refer to subsection 5.4.

DeepSUMO is a framework, written in Python, created to extract and process traffic detector data from SUMO in a way that it can be used with deep learning models for traffic prediction. DeepSUMO is divided into two parts, the main application, which collects/creates data from the running simulation, and an application that allows the user to automatically place detectors on a given road network, as placing them by hand can be (depending on the scenario size), very time-consuming [MHWS21].

The main application works by first creating a graph from the specified detectors by either geographical- or Dijkstra distance (with the option to create your own strategies) and then running the simulation. While the simulation is running DeepSUMO collects detector data based on the interval stated in the detector declaration (subsubsection 3.1.4). It stores and provides access to this data using NumPy in a way that makes it easy to use the collected data with existing (and new) deep learning models, as the interface is modeled after existing real-life datasets. The user is able to interact with the simulation and the collected data at any time while the simulation is running using so-called *"modules"* following the observer pattern (subsubsection 5.4.5). The client can write their own modules that are then called in fixed periods of time. The following subsections will describe these processes on a high level. For more detailed information about the components and processes, refer to subsection 5.3 and subsection 5.4.

5.1 Data model

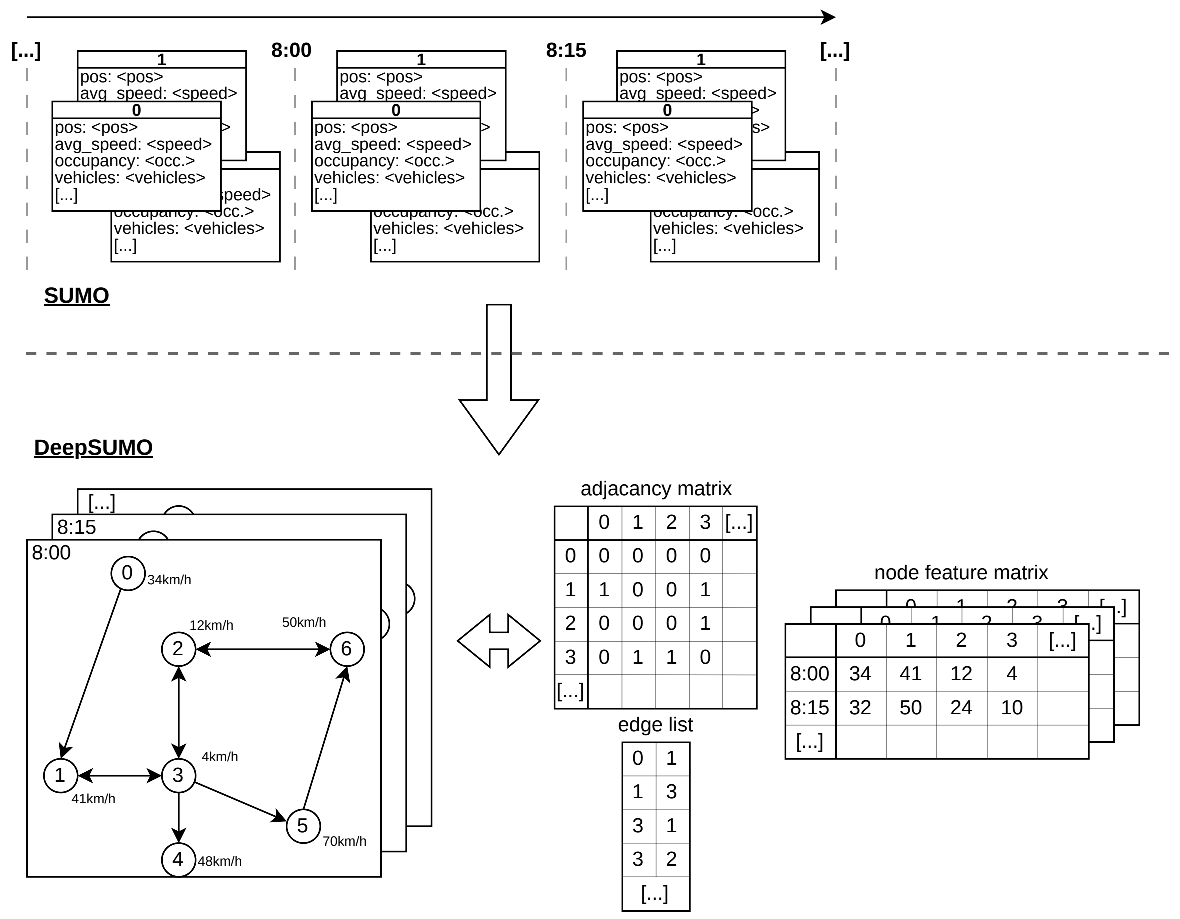


Figure 2: Data model of DeepSUMO

As DeepSUMO is a very data-centric framework it is important to have a basic understanding of what data DeepSUMO collects and how the data is structured.
Figure 2 shows a high-level overview of most of the collected data and how it is structured by DeepSUMO.
From SUMO it is possible to collect data such as configured detectors (and their attributes), information about the road network, and much more. But this data is scattered and it is one of DeepSUMO's core functionalities to collect all relevant data and structure and persist it in data structures that make it easy to use the data with deep learning methods such as graph neural networks.
As SUMO does not provide any connectivity information about detectors, DeepSUMO connects the detectors into a graph structure (for more information refer to subsubsection 5.4.1). This graph describes which detectors are relevant to each other (e.g. because they are geographically close together) and therefore might influence one another. This is done because an important part of all graph neural networks for traffic prediction is a

graph (in this case of detectors) as mentioned in subsection 3.2. The information about this graph is then saved using two main data structures: (one) in adjacency matrices and (two) in an edge list. DeepSUMO uses multiple types of these structures capturing different aspects of the graph to capture all necessary data. For more information on this refer to subsubsection 5.4.1.
The measured speeds at each detector are saved in a so-called *node feature matrix*, as seen in Figure 2. This matrix contains all detectors (=nodes) with their respective measurements. As DeepSUMO collects multiple measurements from each detector over the course of the simulation, this matrix contains data for all detectors over all their collection intervals.
DeepSUMO contains three of these matrices as three different features from SUMO are captured for each detector: speed, occupancy, and number of vehicles.

5.2 Context

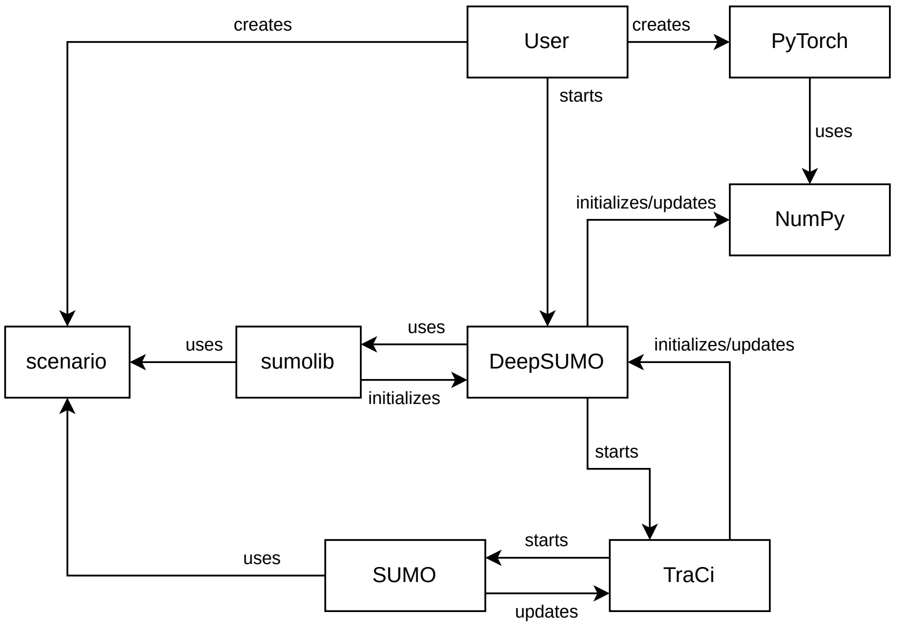


Figure 3: Initial overview of DeepSUMO - Block Diagram

Figure 3 shows the relevant technologies used in DeepSUMO and how they interact with each other on a high level. Each technology is responsible for the following:

User The user of the framework creates the initial scenario as described in subsubsection 3.1.2. He also starts DeepSUMO and creates the deep learning model.

PyTorch PyTorch includes the desired deep learning model. It interfaces with DeepSUMO using a custom dataset created from the data collected by DeepSUMO. DeepSUMO also allows the use of other libraries, as the main interface to the data collected by DeepSUMO is modeled using NumPy.

NumPy NumPy is responsible for storing the edge index and node features collected by DeepSUMO for later use with deep learning models. This data store can then be used for creating datasets for use with deep learning models. For this reason, NumPy was chosen as the technology for storing this data, as most up-to-date libraries used for deep learning provide an easy way to import NumPy arrays into their own data structures. This allows the data generated by DeepSUMO (and therefore DeepSUMO itself) to be used with a wide variety of libraries used for deep learning (e.g. PyTorch, Keras, etc.).

Scenario The Scenario contains all necessary information about the network like roads, trips, traffic detectors, and much more. It is used by SUMO to run the simulation and also accessed by DeepSUMO using sumolib to easily extract information about the road network itself (like street names, speed limits, detectors, etc.).

Sumolib Sumolib is a Python library responsible for "working with SUMO networks, simulation output, and other simulation artifacts" [SUM23c]. DeepSUMO uses it primarily to extract information about the network, but also uses it for many other applications.

SUMO SUMO is the traffic simulator used in DeepSUMO (for more information, refer to subsection 3.1). It is used to run the traffic simulation. The simulation is configured using the scenario and interfaces with DeepSUMO via TraCi

TraCi TraCi is the Python library used to connect SUMO with DeepSUMO. It allows access to most of the simulation's parameters and data and is used for simulation control and retrieval of detector data.

DeepSUMO DeepSUMO is the core application that controls the simulation and retrieves data from detectors. It does this by using all technologies listed above.

The following sections highlight the main components/classes of the framework and how they interact with each other. For this, first, a high-level overview of the order of events will be presented to understand the general flow of the program. Secondly, each component will be described in detail including important mathematical concepts used to provide a deep understanding of the functionality of each component. Lastly, the interaction between the components will be modeled on a low level to also provide a deeper understanding of the component's relationship with each other.

5.3 Basic Flow

The flow of DeepSUMO can be divided into an initialization and runtime phase. During the initialization phase, the framework creates all necessary data structures and creates and initializes all static objects needed for further processing. The framework populates the created data structures with data collected from the simulation in the second phase. The second phase starts after *the simulation* was started and lasts the entire runtime of the simulation. This phase includes processes like the collection of data, continuous control of the simulation, running of modules, and more.
Figure 4 shows the initialization process on a high level. Firstly the detector graph is created. This graph contains all preconfigured detectors (for more information refer to subsubsection 3.1.4 and subsubsection 5.4.1) as nodes and their connections as edges. The edges are created using different strategies. DeepSUMO ships with two strategies.

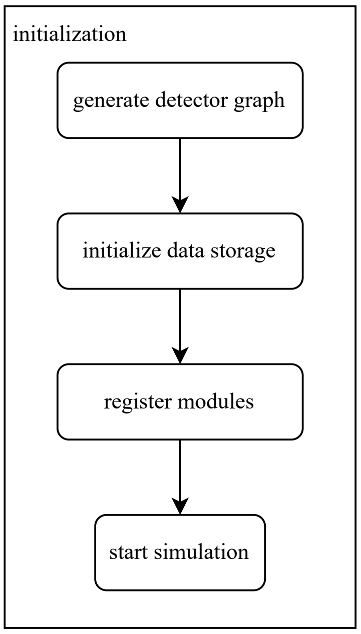


Figure 4: Basic flow of initialization phase

Nodes can either be connected by their geographical distance or by travel times between each other using Dijkstra. Neighbors are determined by calculating the distance/cost from one node to all other nodes and then applying a threshold to determine which nodes are close enough to be neighbors. The user can also create their own strategies for connecting the nodes.
Next, the data storage used to store all data collected over the course of the simulation is initialized.
Lastly, all configured modules are registered. Modules can theoretically also be registered while the simulation is running, but it is recommended to register all of them during the initialization phase.

Figure 5 shows what happens during the simulation phase on a high level. The following operations are executed on each simulation step.
Firstly SUMO simulates the next simulation step. Secondly, the data is collected from SUMO and processed/stored. The data, however, is not collected at every simulation step but rather only at

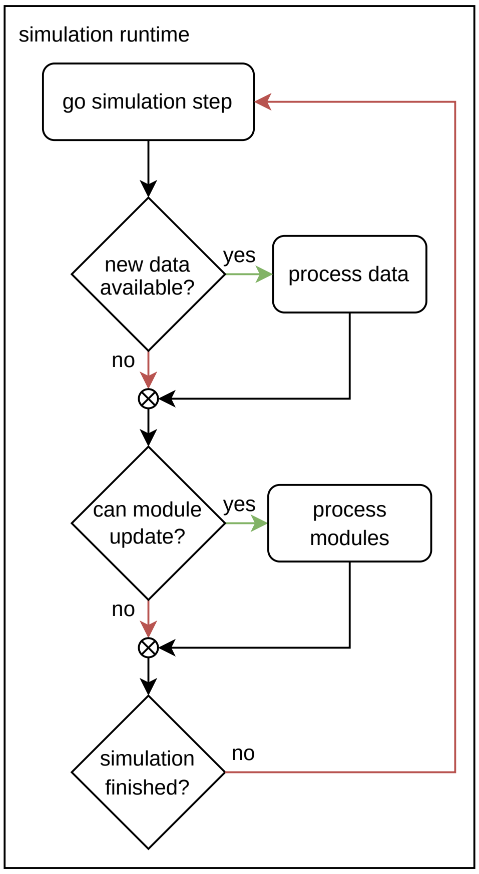


Figure 5: Basic flow of simulation phase

the steps corresponding to the end of an interval (corresponding to the detector's aggregation interval mentioned in subsubsection 3.1.4) as choosing different interval sizes leads to inconsistent data (e.g. a SUMO detector interval of 60 steps and a data collection interval in DeepSUMO of 30 steps would lead to the data of each detector interval being saved twice, as no new data is available).
After the data has been processed all configured modules update functions are called. These, however (as the data collection) do not run at every simulation step. They run according to an interval set by the user (e.g. every 60 simulation steps) as running modules (depending on their complexity) at every simulation step can slow down overall performance dramatically. For more information on this refer to subsubsection 5.4.5.
The loop exits after the simulation is finished. Then the simulation is shut down and further tasks using the collected data can be performed.

5.4 Components

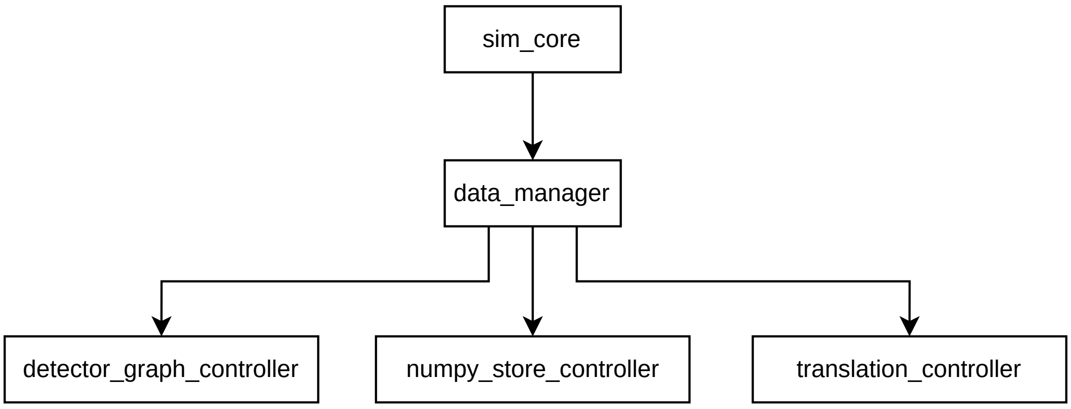


Figure 6: Most important classes of DeepSUMO

DeepSUMO consists of five core components, which are depicted in Figure 6. They control the main workflow shown in subsection 5.3. Each of these components refers to a concrete class used in DeepSUMO. Each of these components is supported by other classes to extend their functionality and provide easy maintainability. The next subsections will describe each of these components with their supporting classes in detail to provide a deep understanding of each class's functionality and basic structure.

5.4.1 Detector graph

The detector graph component is responsible for creating the graph structure, used by the deep learning models, from the configured detectors. As detectors configured for SUMO only contain information like their position relative to the lane they are on, the lane, and other things (for more information refer to subsubsection 3.1.4) they do not include any direct connectivity information. But, as most current deep learning methods used for traffic forecasting, require a graph structure, a component is needed to create such a structure containing all configured detectors connected as a graph.
The component consists of three main parts: (one) The *node_connector*. This class contains all logic necessary to connect the detectors and save the resulting connectivity information in appropriate data structures. The behavior used when connecting detectors is defined by a cost function. This function can be individually configured using the strategy pattern. (two) The *node_connector_strategy* class includes the function/strategy used when connecting the detectors. (three) The *detector_graph_controller* class is in some way a facade for the *node_connector*, as it controls access to the data structures created by the connector. It also creates a NetworkX DiGraph from the collected data to make working with the graph easier, since the raw graph data (stored by the connector) is stored in arrays making it hard to interpret.
Figure 7 shows a detailed UML-Class Diagram of these classes. Each class will now individually be explained in more detail.

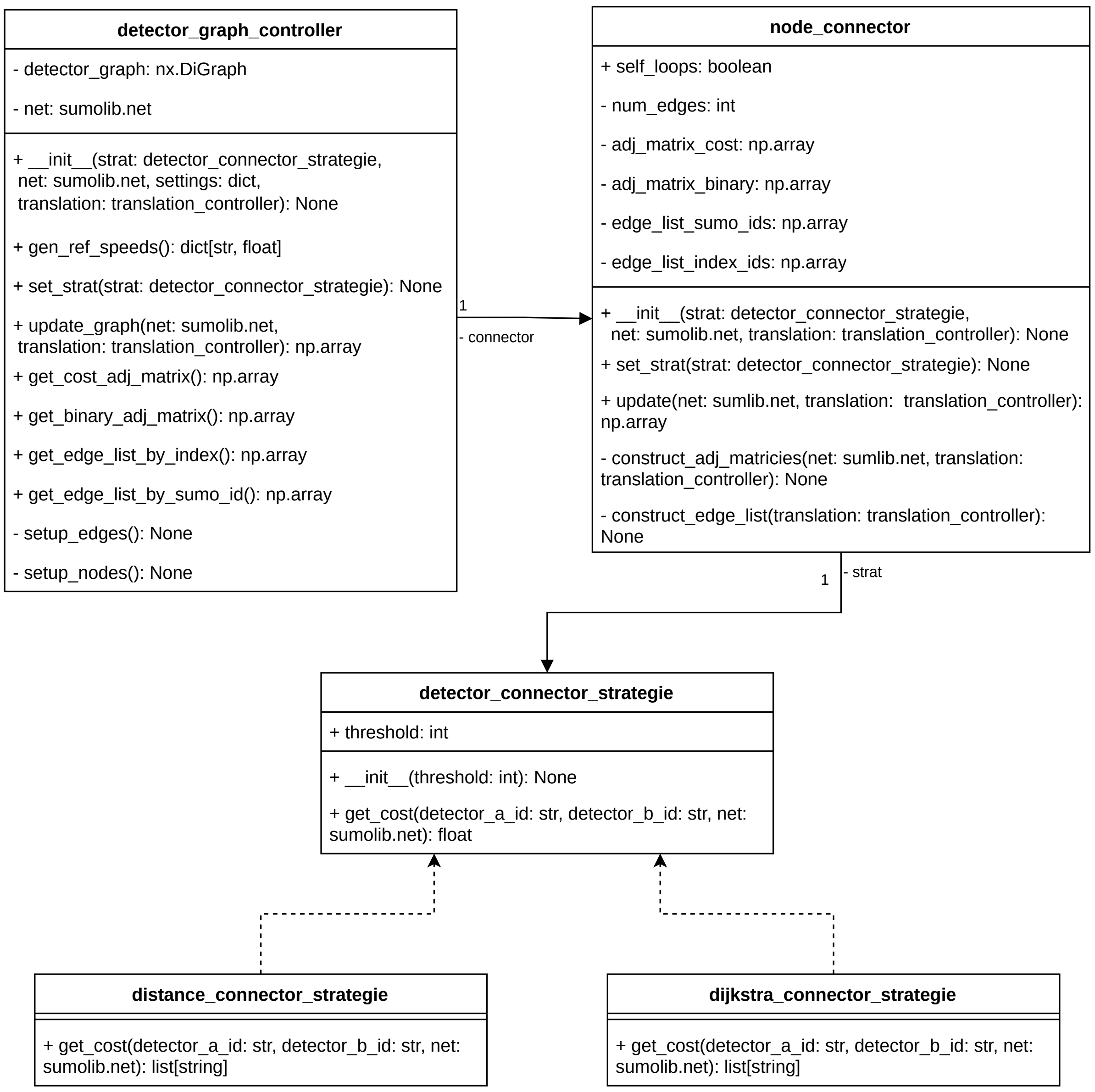

detector_graph_controller
- detector_graph: nx.DiGraph
- net: sumolib.net
+ __init__(strat: detector_connector_strategie, net: sumolib.net, settings: dict, translation: translation_controller): None
+ gen_ref_speeds(): dict[str, float]
+ set_strat(strat: detector_connector_strategie): None
+ update_graph(net: sumolib.net, translation: translation_controller): np.array
+ get_cost_adj_matrix(): np.array
+ get_binary_adj_matrix(): np.array
+ get_edge_list_by_index(): np.array
+ get_edge_list_by_sumo_id(): np.array
- setup_edges(): None
- setup_nodes(): None
1
- connector
node_connector
+ self_loops: boolean
- num_edges: int
- adj_matrix_cost: np.array
- adj_matrix_binary: np.array
- edge_list_sumo_ids: np.array
- edge_list_index_ids: np.array
+ __init__(strat: detector_connector_strategie, net: sumolib.net, translation: translation_controller): None
+ set_strat(strat: detector_connector_strategie): None
+ update(net: sumlib.net, translation: translation_controller): np.array
- construct_adj_matricies(net: sumlib.net, translation: translation_controller): None
- construct_edge_list(translation: translation_controller): None
1
- strat
detector_connector_strategie
+ threshold: int
+ __init__(threshold: int): None
+ get_cost(detector_a_id: str, detector_b_id: str, net: sumolib.net): float
distance_connector_strategie
+ get_cost(detector_a_id: str, detector_b_id: str, net: sumolib.net): list[string]
dijkstra_connector_strategie
+ get_cost(detector_a_id: str, detector_b_id: str, net: sumolib.net): list[string]

Figure 7: Class diagram of the detector graph controller component

5.4.1.1 node_connector

The node connector class is responsible for connecting all nodes(=detectors) together using a configurable cost function. It is also responsible for storing the created data in data structures that make it easy to access and understand the data.
The edges of the graph are calculated with a user-configurable cost function, which describes the relationship between two nodes. This function is implemented as a class that inherits the *detector_connector_strategie*. The user can choose which function to use and can even create their own function as this part of the class is modeled after the strategy pattern.
Each strategy/function consists of two main parts: (one) A threshold attribute, which

is used by the *node_connector* class to construct the binary adjacency matrix. This attribute is mandatory and describes the threshold that determines whether an edge between two nodes should be created (if the value of the cost function is below the threshold a new edge will be created). (two) An update method. The update method is called by the *node_connector* class when constructing the edges of the graph. It calculates the relationship between two nodes by calculating a cost value for each pair of nodes. The bigger this value is the lower is the probability of the nodes being related. To calculate this the method gets the following parameters: the two detectors (one source and one target detector, as the graph is directed), as IDs to be used with TraCi, of which the relationship should be calculated and the *net* object, which provides access to the structure of the network used by SUMO using *sumolib*. When this function is called it can be expected that SUMO (and the TraCi connection) is already initialized and ready to be used.
DeepSUMO already contains two strategies by default, which can be used for connecting the nodes.

Distance This strategy/function calculates the relationship of two nodes based on their geographical location. It returns the distance between two nodes in meters.
This distance using the coordinates of each detector is calculated in the following way:

$$\begin{aligned} dx &= x_0 - x_1 \\ dy &= y_0 - y_1 \\ distance &= |\sqrt{dx^2 + dy^2}| \end{aligned} \tag{1}$$

With *[x0, y0]* being the coordinates of the source-detector and *[x1, y1]* being the coordinates of the target-detector. Getting the position of each detector so that it can be used in this calculation presents another challenge as SUMO does not provide the coordinates of each detector. It instead only provides information about the lane on which the detector is placed and the position in terms of meters after the start of the lane (subsubsection 3.1.4). So for the calculation to work it is necessary to calculate the position of the detector as two dimensional coordinates from this information. This is possible since each lane's shape in SUMO is modeled by a polyline. The shape can be extracted using sumolib and is represented by a list of two-dimensional coordinates. This, in combination with the position attribute of the detector, allows for the calculation of the position of the detector as coordinates from these two parameters.
The line's shape can be divided into a series of vectors each defined by two coordinates $[x_{start}, y_{start}]$ and $[x_{end}, y_{end}]$.

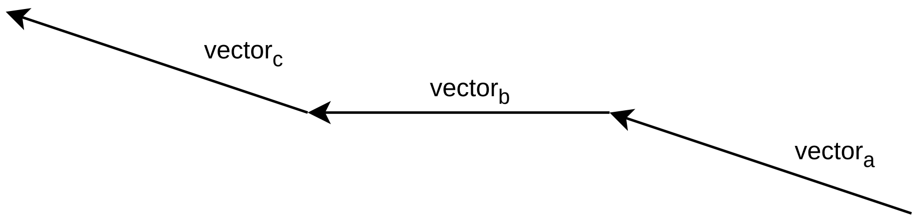


Figure 8: Example of a lanes shape

It is possible to calculate the distance of each individual vector (using the same equation as Equation 1) and add them up (starting at the start of the line) until the resulting sum (=length of the line) is bigger than the position of the detector. The detector then has to be positioned on the last added vector (for example, if in Figure 8 each vector has a length of *10* and the detector has a position of *25*, it would be placed on $vector_c$, as 10 + 10 + 10 = 30 which is bigger than 25). After the vector on which the detector is placed is known, it is possible to calculate the exact position of the detector using the vector's coordinates and the detector's position attribute.

$$\begin{aligned} d_{remaing} &= d_{vector} - (d_{total} - d_{detector}) \\ t &= d_{remaing}/d_{vector} \\ \\ x_{detector} &= (1-t)x_{start} + t * x_{end} \\ y_{detector} &= (1-t)y_{start} + t * y_{end} \end{aligned} \tag{2}$$

With $d_{remaining}$ being the remaining distance on the vector until the position is reached. d_{vector} being the length of the vector on which the detector is positioned, d_{total} being the sum of lengths (including the target vector) of all vectors until the target vector and $d_{detector}$ being the position of the detector acquired from SUMO. The position of the detector is then calculated as $[x_{detector}, y_{detector}]$.

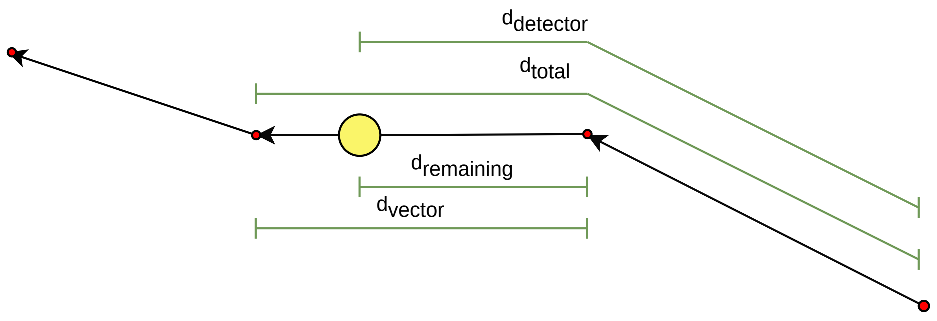


Figure 9: Explanation of parameters from Equation 2

This calculation is done for both detectors, after which the distance between them can be calculated using Equation 1. This strategy provides a quick and easy way to connect the nodes, but also has its drawbacks as it might connect two nodes that are close to one another, but have no relationship with each other (for example two streets/detectors with a river between them) [ZYZ+22].

Djikstra This strategy/function calculates the relationship of two nodes using the Dijkstra algorithm. The cost is hereby defined as the travel time between two nodes calculated using the Dijkstra algorithm. The travel time is calculated using Sumolib's built-in Dijkstra function. But since it allows only calculations from network edge (=road) to edge, and not from detector to detector the resulting travel time has to be adjusted as follows:

$$\begin{gathered} t_{dijkstra} = dijkstra(edge_a, edge_b) \\ t_{adjusted} = t_{dijkstra} - pos_a/speed_a - (length_b - pos_b))/speed_b \end{gathered} \tag{3}$$

With $t_{dijkstra}$ being the travel time calculated using Dijkstra between the two edges $edge_a, edge_b$, corresponding to detectors a, b. This corresponds to the travel time between $edge_a$ and $edge_b$. $t_{adjusted}$ being the travel time adjusted to the position of the detectors according to their position attributes (subsubsection 3.1.4). pos_a, pos_b represent the value of this attribute for detectors a, b respectively. $length_b$ describes the length of the edge (=road) that detector b is positioned on and $speed_a$ and $speed_b$ the reference speed for detector a and b.

The data collected using these functions is then stored on several data structures:

Adjacency matrix The collected data is stored in adjacency matrices of two types: (one) Cost; this matrix contains all detector-pairs cost values, as calculated by the cost function, as an adjacency matrix of size $(num_{detectors}, num_{detectors})$. (two) Binary; this matrix is composed of ones and zeroes, depending on whether the result of the cost function of two detectors is above the configured threshold or not.
All these matrices are ordered in a way so that the indices of detectors in the matrix correspond with the indices used when storing the node features, as it internally uses the translation object to construct/fill the matrices (for more information refer to subsubsection 5.4.3). This results in the ability to directly input the matrix into a deep learning model (together with a dataset created from the collected data), as the indices (representing the detectors) correlate with each other.

Edge list The collected data is also stored as an edge list of the shape $(2, num_{edges})$. This list contains all edges, each represented by a pair of IDs. DeepSUMO stores two such lists, with each containing a different representation of the detectors (one) SUMO IDs, which represent each detector using its ID used by SUMO (and TraCi), and (two) indicies, which identify each detector with respect to its index in the node features. The conversion of SUMO IDs to such indices is possible via the *translation_controller* (subsubsection 5.4.3)

5.4.1.2 detector_graph_controller

The graph controller class is mostly responsible for regulating access to the node connector class. It provides user-friendly interfaces for (one) configuring the node connector and (two) collecting data from it. It also has the ability to create reference speeds which is a dictionary containing all detectors and the maximum allowed speed of the lane on which they are placed.

5.4.2 Numpy Graph

The numpy graph component of DeepSUMO is a central piece of the software, as it is responsible for the collection and storage of data from the simulation. The data is stored as NumPy arrays, as these provide the biggest flexibility and decouple the data storage from the library used to implement the deep learning model. NumPy was deliberately chosen for this as it is widely used and libraries like PyTorch as well as Tensorflow/Keras, which currently are the most used libraries for deep learning, both provide an easy way to import data from NumPy [Ten22, PyT23].

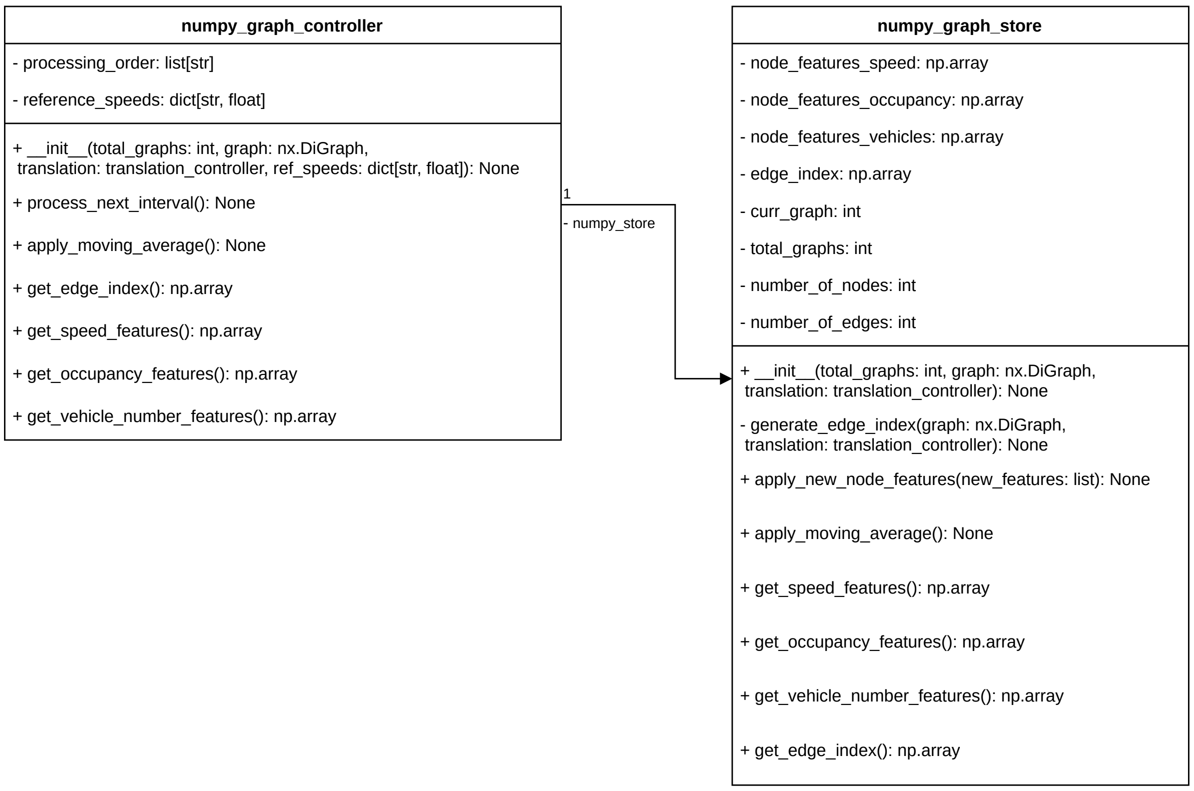

numpy_graph_controller
- processing_order: list[str]
- reference_speeds: dict[str, float]
+ __init__(total_graphs: int, graph: nx.DiGraph, translation: translation_controller, ref_speeds: dict[str, float]): None
+ process_next_interval(): None
+ apply_moving_average(): None
+ get_edge_index(): np.array
+ get_speed_features(): np.array
+ get_occupancy_features(): np.array
+ get_vehicle_number_features(): np.array
1
- numpy_store
numpy_graph_store
- node_features_speed: np.array
- node_features_occupancy: np.array
- node_features_vehicles: np.array
- edge_index: np.array
- curr_graph: int
- total_graphs: int
- number_of_nodes: int
- number_of_edges: int
+ __init__(total_graphs: int, graph: nx.DiGraph, translation: translation_controller): None
- generate_edge_index(graph: nx.DiGraph, translation: translation_controller): None
+ apply_new_node_features(new_features: list): None
+ apply_moving_average(): None
+ get_speed_features(): np.array
+ get_occupancy_features(): np.array
+ get_vehicle_number_features(): np.array
+ get_edge_index(): np.array

Figure 10: Class diagram of numpy_graph component

5.4.2.1 numpy_graph_store

The *numpy_graph_store* class is responsible for storing and organizing the data collected from SUMO. It stores the most necessary information for the training of deep neural networks. The data is stored in two main data structures: (one) the edge index. The edge index contains the adjacency information of the detector graph. The

numpy_graph_store stores this information as an edge list of size $(2, num_{nodes})$. This information however can also be accessed from the detector graph component of the framework in various other formats (subsubsection 5.4.1). (two) The node features. The node features are stored in three separate NumPy arrays. DeepSUMO collects the speed-, occupancy-, and vehicle count data of each detector, as this is the data offered by TraCi [Tra22]. Each of these arrays is of the size $(total\_graphs, num_{nodes})$. With *total_graphs* being the maximum number of graphs that are created during the simulation runtime. This number is typically calculated from the collection interval and overall simulation length (e.g. with a simulation length of 3000 steps and a collection interval of 100 steps, $3000/100 = 300$ graphs will be created during the simulation). And num_{nodes} being the number of nodes of the graph which equals the number of detectors.
These features are updated using the update method. This method takes in a list of three NumPy arrays of size (num_{nodes}) each. The list has to contain one array for each of the features mentioned above and has to be in the following order: speed, occupancy, and number of vehicles. The function does not check if these are in the correct order. The store also assumes that this method is only called *once* per interval. It does *not* keep track of the current simulation step and respective interval. Calling this function more than once in one interval will lead to unexpected behavior, errors, and duplicate data.
The store can optionally apply a moving average over the collected data to smooth it out. To achieve this it takes the unweighted mean of n measurements around a central data point for each collected data point.

5.4.2.2 numpy_graph_controller

The graph controller does similar things to the *detector_graph_controller*, in terms of it acting like a facade and regulating access to the underlying *graph_store* object. It also is responsible for properly initializing the graph store and the collection and preprocessing of data from the simulation.
Because of some characteristics of the data provided by SUMO/TraCi, the data has to be preprocessed before it is passed to the numpy_store object. The data has to be altered in two ways:

- The occupancy value has to be corrected. This is because this value is cumulative for each vehicle. This can result in values over 100% when multiple vehicles are on the detector simultaneously [SUM22a]. The occupancy value itself represents how long vehicles were present on the detector (in percent). To account for this behavior the following calculation is done to correct the given occupancy value to a value between 0 and 100%.

  $$occupancy_{corrected} = occupancy_{SUMO}/num_{vehicles} \tag{4}$$

  With $occupancy_{corrected}$ being the corrected occupancy value, $occupancy_{SUMO}$ being the occupancy value collected from SUMO, and $num_{vehicles}$ being the number of vehicles that passed the detector.

- The speed itself. If no vehicle passed the detector in an interval the average speed returned by TraCi is 0. And if only one vehicle passed the detector and was slower than the maximum allowed speed on that road TraCi will return the speed of that car which is not an accurate representation of the average speed over the span of the whole interval (e.g. if one interval is 60s long and the detector is placed on a road with a speed limit of 50km/h and one vehicle passes the detector driving 30km/h, 30km/h would not be an accurate representation of the average speed on the road since other vehicles would have been able to pass the detector at 50km/h while the car was not on the detector). As this behavior also results in irregular data (especially on roads with little traffic), the speed data is corrected in the following way:

  $$speed_{corrected} = occupancy_{corrected} * avg_{speed} + (1.0 - occupancy_{corrected}) * ref_{speed} \tag{5}$$

  With $speed_{corrected}$ being the corrected speed, $occupancy_{corrected}$ being the corrected occupancy value from Equation 4, avg_{speed} being the speed value collected from SUMO and ref_{speed} being the speed limit of the road on which the detector is placed collected using the *detector_graph_controller* class paragraph 5.4.1.2.

The controller collects all necessary data from SUMO using TraCi, applies the before-mentioned corrections, and passes them to the graph store object to be added to the store.

5.4.3 Translation

The translation component is the smallest, but also very important component of Deep-SUMO. The translation component bridges the gap between SUMO and the data used for the deep learning models. This is needed because in SUMO an individual detector is identified by an ID, which is represented by a string, while in deep learning models, each detector is identified by an index (due to the node features and edge index being saved in arrays), which is an integer. It is the translation component's responsibility to (one) provide a constant processing order containing all detectors, to make sure that detectors are always added to the node features in the same order as well as making iteration over them easier. This order is simply a list of SUMO detector IDs that internally defines the order of detectors in the node features. (two) Keep track of which SUMO detector ID corresponds to which index in the node features (and vice versa).

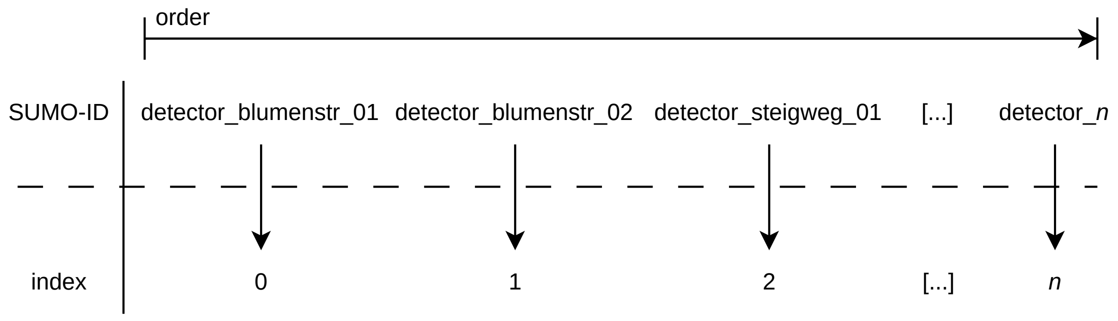


Figure 11: Explanation of translation component

Figure 11 explains the before-mentioned parameters again in a more visual way. It shows the SUMO-IDs (as strings) at the top, their corresponding index in the node features at the bottom, and the role of the *order* attribute at the very top.
A usage example of this component would be:
DeepSUMO is initialized with a runtime of 6000 steps, a collection interval of 60 steps, and 248 detectors. This would result in the creation of three arrays containing the three node features of size $(100, 248)$ each. The translation component now allows us to easily extract the node features for a specific detector, using the SUMO ID by obtaining the corresponding index of the detector by using the *get_index(sumo_id)* function and slicing the node feature array accordingly $node\_feature[:, index]$. It also works the other way around if the user, for example, needs more information (like the position, corresponding street, etc.) about a detector at a specific index, it can be collected from SUMO using TraCi and the SUMO ID acquired from calling *get_detector_id(index)*.

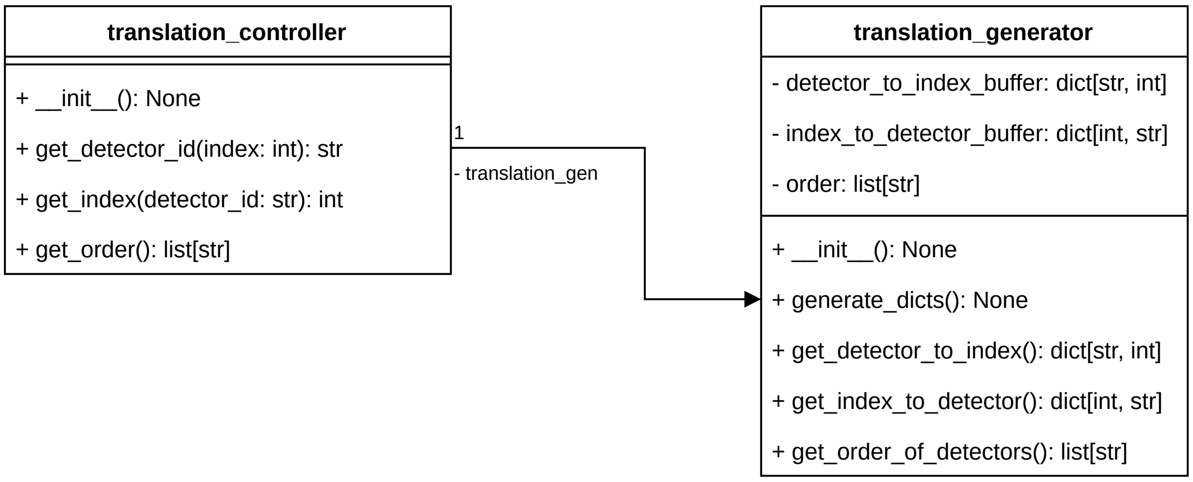

translation_controller
+ __init__(): None
+ get_detector_id(index: int): str
+ get_index(detector_id: str): int
+ get_order(): list[str]
1
- translation_gen
translation_generator
- detector_to_index_buffer: dict[str, int]
- index_to_detector_buffer: dict[int, str]
- order: list[str]
+ __init__(): None
+ generate_dicts(): None
+ get_detector_to_index(): dict[str, int]
+ get_index_to_detector(): dict[int, str]
+ get_order_of_detectors(): list[str]

Figure 12: Class diagram of translation component

translation_generator
The *translation_generator* class is responsible for generating the dictionaries used for translation as well as the processing order of detectors. It also buffers the created dictionaries and order list for later use.

translation_controller
The *translation_controller* class is a facade for the *translation_generator* class and provides/regulates access to the underlying *translation_generator* object.

5.4.4 Data manager

The data manager acts as a facade for the underlying controllers. It is the primary interface for the user to access data from DeepSUMO. Therefore it houses and provides access to a net-, *detector_graph_controller*-, *numpy_graph_controller*- and *translation_controller* object. It also controls and manages the correct initialization of all underlying controllers.

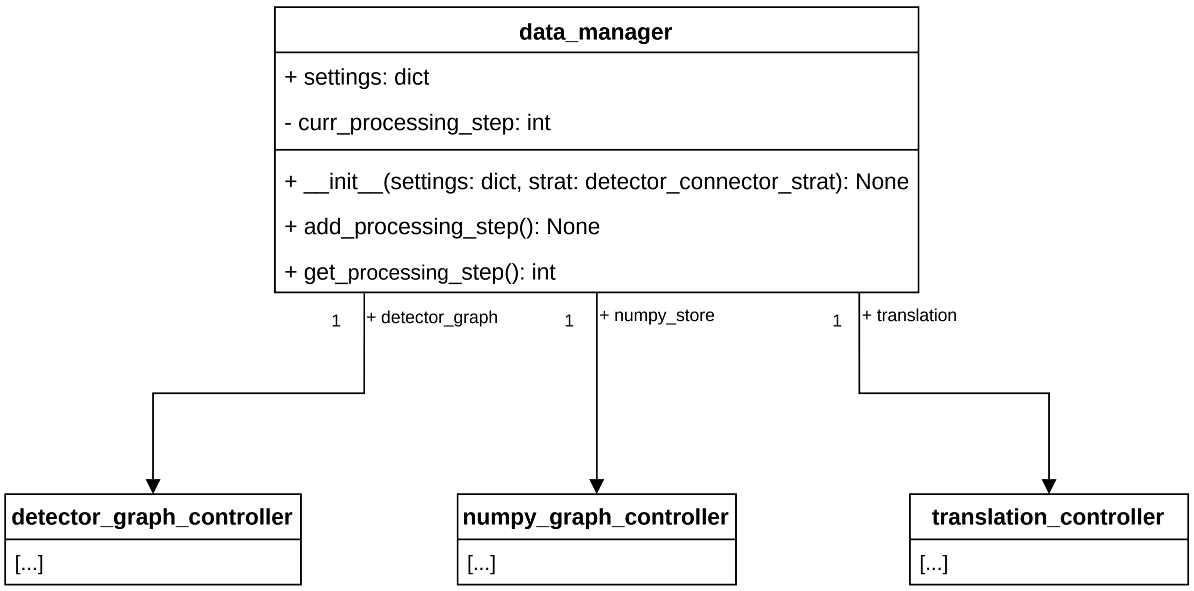

data_manager
+ settings: dict
- curr_processing_step: int
+ __init__(settings: dict, strat: detector_connector_strat): None
+ add_processing_step(): None
+ get_processing_step(): int
1
+ detector_graph
1
+ numpy_store
1
+ translation
detector_graph_controller
[...]
numpy_graph_controller
[...]
translation_controller
[...]

Figure 13: Class diagram of data manager component

5.4.4.1 data_manager

The *data_manager* class is the primary interface for user interaction with DeepSUMO. It provides access to all necessary controllers and the net object, initializes them, and also saves the current processing step. DeepSUMO provides a function to increment this value, however, it is *strongly discuraged* to use this function as they are already automatically called by DeepSUMO and require no user interaction. Calling this method can lead to unexpected behavior on modules or other applications that depend on the accuracy of the curr_processing_step variable.

5.4.5 Sim core

The simulation core is the central piece of the framework responsible for controlling the simulation and general order of events in the framework. It makes sure all necessary components are initialized and runs the simulation in a controlled manner. The *sim_core* allows the user to interact with the simulation and the framework using *simulation modules* which are implemented following the observer pattern. The component ensures that all data coming from the simulation has been fully processed before running any modules. Each of these modules contains two main parts (one) a threshold value, which is used to determine when the module is called, and (two) an update method, which provides access to DeepSUMO's data by passing a *data_manager* object and contains the main logic of the module. The runtime of each module is determined by its threshold value. Each module is run periodically with the threshold value specifying the frequency of the module in simulation steps (e.g. a threshold value of 60 means that the module's update function is called every 60 simulation steps). Running modules too often is generally not recommended, as running them (depending on their complexity) can reduce the execution speed dramatically. Inside its update method, each module has access to (one) a fully initialized TraCi connection to interface with the simulation and (two) a *data_manager* object containing all collected data up to the current simulation step including the current simulation step (for more information on this refer to subsubsection 5.4.4). DeepSUMO currently contains three modules by default that can be used as examples.

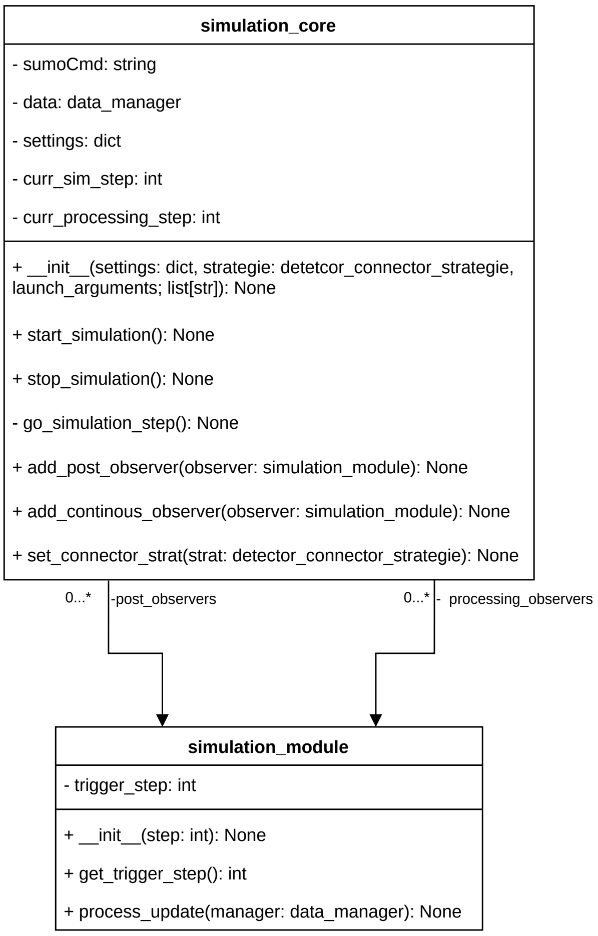

simulation_core
- sumoCmd: string
- data: data_manager
- settings: dict
- curr_sim_step: int
- curr_processing_step: int
+ __init__(settings: dict, strategie: detetcor_connector_strategie, launch_arguments; list[str]): None
+ start_simulation(): None
+ stop_simulation(): None
- go_simulation_step(): None
+ add_post_observer(observer: simulation_module): None
+ add_continous_observer(observer: simulation_module): None
+ set_connector_strat(strat: detector_connector_strategie): None
0...*
-post_observers
0...*
- processing_observers
simulation_module
- trigger_step: int
+ __init__(step: int): None
+ get_trigger_step(): int
+ process_update(manager: data_manager): None

Figure 14: Class diagram of simulation core component

progress_module This module prints the overall progress of the simulation and provides the user with additional information about the last interval like its simulation time. It also provides an estimation of the time remaining until the simulation is finished. As this module internally does not track the current simulation step, the interval of this module has to be $sim_{length}/100$, so that the module runs exactly 100 times until the simulation is finished.

print_module This module is mainly used for debugging and has the ability to print out information about a specific detector (for example its node features etc.).

simulation_flow_control_module The *simulation_flow_control_module* has the ability to control the simulations "scale" attribute depending on the current weekday and hour in the simulation. The "scale" attribute of the simulation defaults to 1.0. It controls the number of vehicles/trips in the simulation with a trip consisting of a vehicle and its designated path through the road network from a start to an endpoint. If the scale value increases the number of vehicles increases, and if it decreases, the number of vehicles in the simulation decreases. The module uses this function to allow for different amounts of traffic volumes to be simulated at different times of day without the need for the creation of specialized trip files (subsection 3.1). This module allows the user to more easily create scenarios with changing traffic densities by allowing the user to use tools like the OSMMapwizard from SUMO, which creates scenarios from OSM, to create longer simulations, as these tools, per default, do not consider the time of day.
The module acquires the current simulation time by using a start date and the current simulation step as each step (per default) is one second. The current simulation time can then, therefore, be calculated by adding the current simulation step in seconds to the start date.
The module then adjusts the simulation scale following a set of rules saved in a dictionary of the type *dict[tuple[tuple[int, int], tuple[int, int]], tuple[float, float]]*, with each entry as follows:

$$((weekday_{start}, hour_{start}), (weekday_{end}, hour_{end})) : (scale_{min}, scale_{max}) \quad (6)$$

The key consists of a tuple containing two tuples. The first is the start weekday and hour and the second is the stop weekday and hour. The weekday is represented by a number between 0 and 6 (0 being Monday and 6 being Sunday) and the hour by a number between 0 and 23 (0 being 0:00 and 23 being 23:00). These two tuples form a period of time in which the scale of the simulation will be a random value between the two values represented in the value attribute of its corresponding key. An example of a rule goes as follows:

((0, 6), (0, 9)): (2.5, 3.0)

From monday morning at 06:00 *(0, 6)* until monday morning at 09:00 *(0, 9)* the scale value will be somewhere between 2.0 and 3.0 *(2.5, 3.0)* due to the morning rush. If no rule that matches the current simulation time can be found the scale value will remain the same and will not be updated.

5.4.5.1 settings

In order to configure the behavior of DeepSUMO a couple of settings are needed. These settings are saved in a dictionary and must contain the following entries:

- *sumo_exec_path*, which is the path to the sumo executable

- *sumo_config_path*, which is the path to the config file of the desired simulation
- *sumo_net_path*, which is the path to the network file of the simulation
- *sim_length*, which is the total length of the simulation in simulation steps
- *interval_length*, which is the length of the collection interval of the detectors
- *total_graphs*, which is the total number of graphs that will be created/populated during the simulation. This value does not need to be set by hand and can be calculated as follows: *sim_length*//*interval_length*.

These settings must be passed to the *simulation_core* object in its constructor. It is also necessary to pass the desired *detector_connector_strategie* at this point.

5.4.6 Detector generator

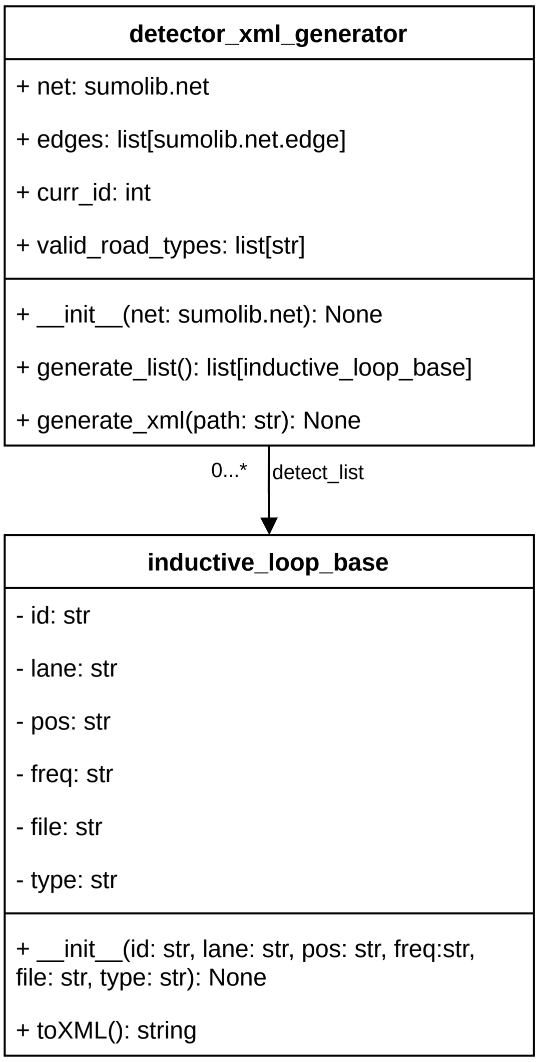
The *detector generator* is the second part of the framework and can be used for adding detectors onto an existing road network. It places detectors using pre-set spacing between them on select roads, specified by their type. It also places a detector at the beginning and end of each lane (if its corresponding road is under the allowed road types). [Ope23]. This component was specially made for the use of networks imported from OpenStreetMap, as they come with detailed type information.

5.4.6.1 inductive_loop_base

The *inductive loop base* class is a representation of a detector element contained in the detector definition file used by SUMO (subsubsection 3.1.4). It contains all necessary attributes and can create the XML-Definition of a detector as a string. This data includes the id, lane, position, frequency (=interval size), output file, and type of detector.

5.4.6.2 detector_xml_generator

The *detector XML generator* is a component that is used to create all necessary detectors and create an XML file containing their definitions to be used by SUMO.


detector_xml_generator
+ net: sumolib.net
+ edges: list[sumolib.net.edge]
+ curr_id: int
+ valid_road_types: list[str]
+ __init__(net: sumolib.net): None
+ generate_list(): list[inductive_loop_base]
+ generate_xml(path: str): None
0...*
detect_list
inductive_loop_base
- id: str
- lane: str
- pos: str
- freq: str
- file: str
- type: str
+ __init__(id: str, lane: str, pos: str, freq:str, file: str, type: str): None
+ toXML(): string
Figure 15: Class Diagram of detector generator component

The program creates the detectors by looping over all roads of the network, checking if their type is valid by checking if the road type is contained in the *valid_road_types* attribute, and (if it is a valid road type) creating detectors for each lane of the road corresponding to the set spacing. The object internally creates a list of *inductive_loop_base* objects and then creates the XML-file by using the objects *toXML()* method. The allowed road types are saved in the *valid_road_types* attributes as strings. Each string is formatted as "<key>.<value>" with the key and value values acquired from the key, value pairs in [Ope23].
The detectors are placed on a lane by first calculating the length of the lane using the first part of Equation 2. This value however has to be corrected as in SUMO each lane inherits its length (*not* shape) from its corresponding edge/road resulting in slightly inaccurate lengths. This is because, depending on the shape, the individual lane can have a different length than the corresponding road. So if the length calculated by SUMO is smaller than the length calculated by DeepSUMO the length used for the detector creation is set to the length from SUMO and if the length calculated by SUMO is bigger than the length calculated by DeepSUMO the length used for the detector creation is set to the length from DeepSUMO to avoid problems with the creation of the detector graph.
The program then creates detectors using the configured spacing along the lane until the length of the lane is reached and finally creates one additional on the lane's start and end.

5.5 Advanced Flow

After a more detailed understanding of each component/class is established, this subsection will focus on presenting the relationship between them in more detail by describing the flow and order of events in more detail than in subsection 5.3. This will then be followed by a usage code example to conclude the description of the framework.

Figure 16 shows a low-level diagram of the order of events in the initialization phase first introduced in Section 5.3. The user first sets up the settings according to the specification in section 5.4.5.1. After the user sets up the settings, he constructs the *simulation_core* object using the before-created settings and a specific connector strategy for creating the detector graph. This object then internally first initializes and starts the simulation using the parameters in the settings object. After the simulation (and TraCi) are started the *translation_controller* is initialized as it provides functionality that is needed to initialize both the *detetcor_graph* and *numpy_graph*. The translation object then internally first creates/sets the processing order of detectors (=order attribute) and then creates the dictionaries used for translation using the before-created processing order.
After the translation component has been initialized the detector graph is created to generate the NetworkX graph and adjacency information needed for the *numpy_store* component and deep learning models. This is done by first creating all necessary nodes for the NetworkX DiGraph. After which all nodes (=detectors) are connected using the specified connector and saved in various formats/matrices (subsubsection 5.4.1) for further use. In this step, the adjacency information is also added to the DiGraph.
After the detector graph has been created the *numpy_store* can be initialized.

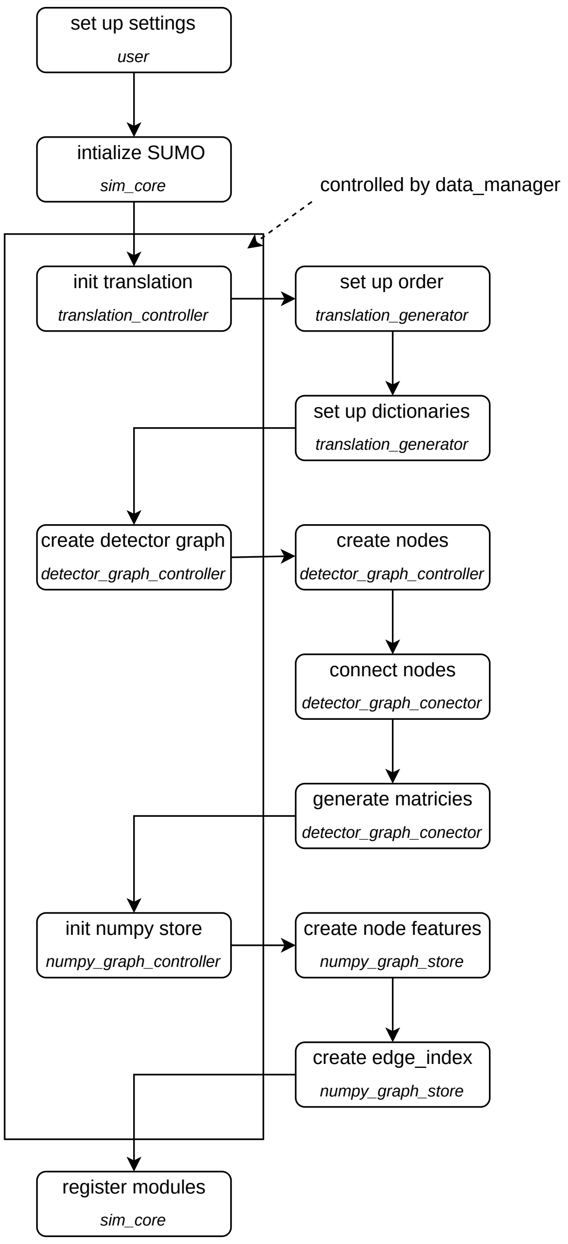


Figure 16: Detailed flow diagram of initialization phase

For this first, the node feature arrays are created corresponding to the settings and filled with zeroes. Secondly, the edge index is created and filled according to the information collected by the detector graph and translation object. After all internal components are initialized, the user can register modules (which are used for user interaction with DeepSUMO's data and the simulation) using the sim core component after which the framework is operational and the initialization phase is complete.

Figure 17 shows a more detailed flow diagram of the simulation/runtime phase of the framework. After entering the phase (after successfully completing the initialization phase) the framework enters a loop until the simulation is finished.
On each simulation step, the framework firstly checks if another collection interval has passed by comparing the current simulation step and interval size (e.g. if the collection interval is 60 steps, the framework would process new data at steps 60, 120, 180, etc.). If new data is available (one interval has passed) the framework collects the data and processes it into the *numpy_store* using the before-mentioned components.
Afterward, all registered modules are checked by comparing the current simulation step and the module's trigger value and run if the simulation step is a multiple of the trigger value by calling the modules update function.
This process is repeated until the simulation is finished. After the simulation finishes all configured post-processing modules are run once, after which the simulation and TraCi are stopped, and the runtime phase ends.

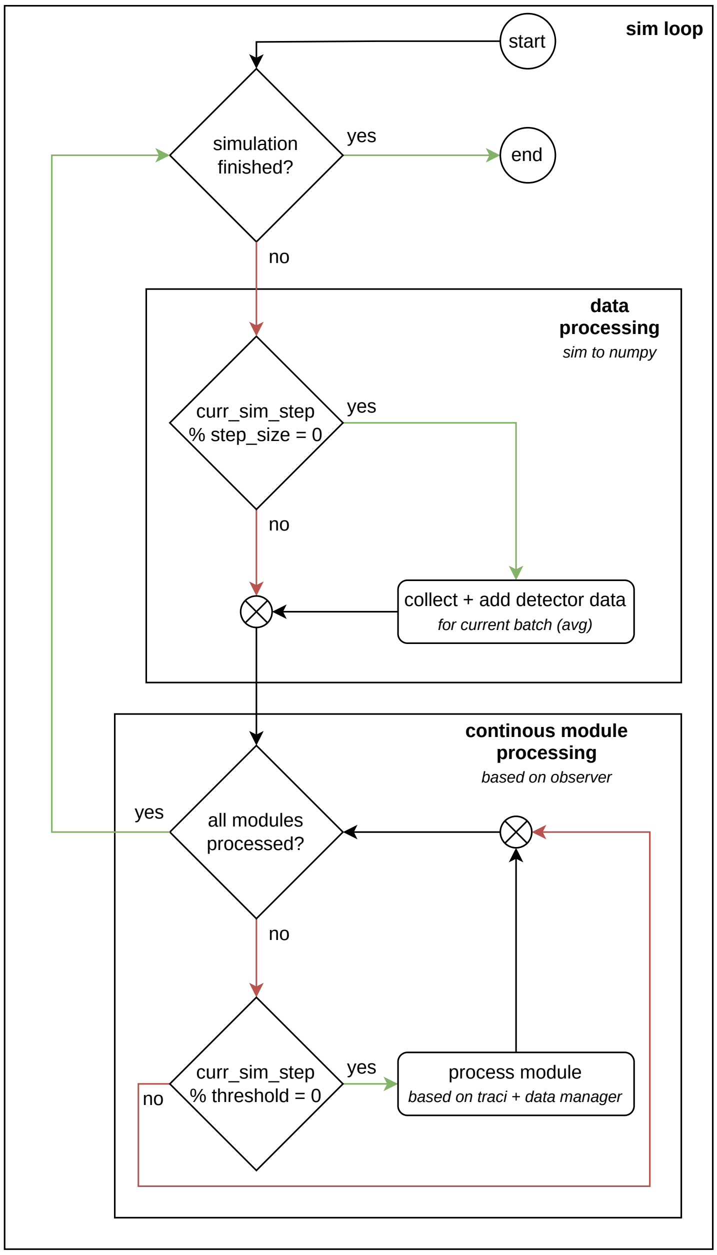


Figure 17: Detailed flow diagram of runtime phase

6 Example

After DeepSUMO has been introduced and all components and their interactions have been explained in detail this section will give an example of how to use the framework. A graph neural network model will be trained and tested using the extracted data. This section aims to show how to use DeepSUMO and its functionality in an application and ensure that DeepSUMO works properly.
For this first, the graph neural network model used in this example will be described. Second the code for generating and running a scenario using DeepSUMO will be presented and explained. Last, the created scenario will be demonstrated using DeepSUMO. The code explained and used in this section is also available in the DeepSUMO repository as the "celle_example" under *examples/celle_example*.

6.1 Pre Conditions

This section will present an example run-through of DeepSUMO from the scenario generation to the simulation, model training, and analysis. The model used in this example is the ST-GAT model, which is a "Deep Learning Approach for Traffic Forecasting" by Zhang et al. [ZYL19]. The model works by combining a *graph attention mechanism* for the spatial features with an *LSTM* (Long short-term memory) network for the temporal features. [ZYL19]. This model was chosen for its very well-documented PyTorch implementation by Julie Wang, Amelia Woodward, and Tracy Cai, which made it easy to understand and adapt the code to use data generated by DeepSUMO instead of the "PeMSD7" Dataset [WWC22, WWC23].
The original code has been slightly adapted to work best with DeepSUMO. These modifications include (one) a new dataset using data from DeepSUMO as the data source, and (two) a rework of the visualization tools simplifying them and reducing the number of needed settings parameters. The core part of the framework (= the training process and model itself) is left original and has not been modified. The code for the model and its direct infrastructure can be found in the package *src/torch_geo/**
The model uses two types of features, the spatial features (which define the structure of the graph) and the temporal features (which for example define the speeds at each node/detector of the graph).
The spatial features are input into the ST-GAT model as an edge list. This list is represented by a tensor of size $(2, num\_edges)$ and contains all edges of the graph as a pair of source and target nodes.
The node features/temporal features are saved using the so-called "Speed2Vec" representation [ZYL19] which uses a sliding window to capture the temporal features as node embeddings. This window is simply a collection of the F previous measurements of a datapoint, represented by a vector. The following example depicted in Figure 18 will highlight how the sliding window mechanism for $F = 3$ works [WWC22].

	15:00	15:15	15:30	15:45	16:00	16:15	16:30
detector a	15	1	12	7	8	15	14
detector b	50	45	49	30	39	47	43

	15:30	15:45	16:00	16:15	16:30
detector a	[15,1,12]	[1,12,7]	[12,7,8]	[7,8,15]	[8,15,14]
detector b	[50,45,49]	[45,49,30]	[49,30,39]	[30,39,47]	[39,47,43]

Figure 18: Explanation of sliding window/Speed2Vec [WWC22]

This information can then be saved in a three-dimensional feature matrix of size $(num_{timepoints}, F, num_{nodes})$ for all nodes and datapoints in the dataset [WWC22, ZYL19].

6.2 Code

The first step is creating/obtaining a SUMO scenario. This can be done in various ways, as partially described in subsection 3.1. But for the reason of simplicity, this explanation will obtain the scenario from OpenStreetMap using the import tools provided by SUMO.
To obtain a scenario from OpenStreetMap we will use the tool "OSMWebWizard". After opening the program, a webpage will open. Here we select the area that we want to convert and adjust the simulation duration to the desired runtime. As traffic prediction usually aims for longer durations it is recommended to choose long runtimes of 2419200 steps, which equals one month, or up. If desired, other parameters (like simulated vehicle types, road types, etc.) can be set up here as well before generating the scenario files. The scenario files can then be generated by clicking on the "Generate Scenario" button. Depending on the size and length of the scenario and the speed of the computer, this can take some time.
After generating the simulation, we use the created files for the next step.

It is now possible to generate detectors on the created network using the *writer* tool packaged with DeepSUMO as follows:

```
net = sumolib.net.readNet("<path to net.xml>")

gen = detector_xml_generator(net)
gen.generate_xml("<path to output file>")
```

Listing 1: Code to generate detectors on network

This will generate a file containing all detector definitions according to subsection 3.1 and paragraph 5.4.6.2. During testing, it was discovered that this method of generating detectors in combination with current deep learning models that are tuned to work on real-life datasets can lead to poor training performance. This is because most of the detectors are placed on parts of a road where not much traffic/congestion happens.

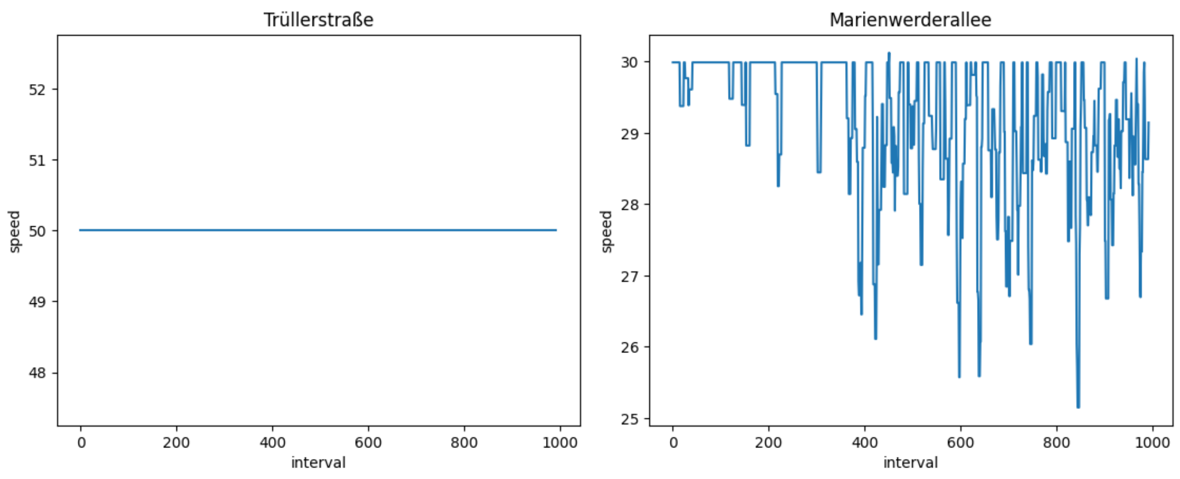


Figure 19: Comparison of observed max speeds on two roads

This phenomenon is visualized in Figure 19. Figure 19 shows the measurements of two detectors in the network over the course of 1000 measurement intervals (of five minutes each). The x-axis shows the interval and the y-axis represents the measured speed at the detector during the interval. The intervals are ordered chronologically (e.g. interval 0 being from 8:00 am to 8:05 am, 1 being from 8:05 am to 8:10 am, and so on).
The left graph represents a detector without much (if any) traffic resulting in only measuring one speed (= maximum allowed speed) for the entire length of the simulation (because no or very few vehicles pass it). The right graph shows a detector with more traffic, resulting in a much more diverse graph. It has been observed that including detectors without much traffic (like the left graph) might lead to extreme underfitting in cases where the speed actually changes (like on the right graph) with currently available graph neural networks as illustrated in Figure 20.

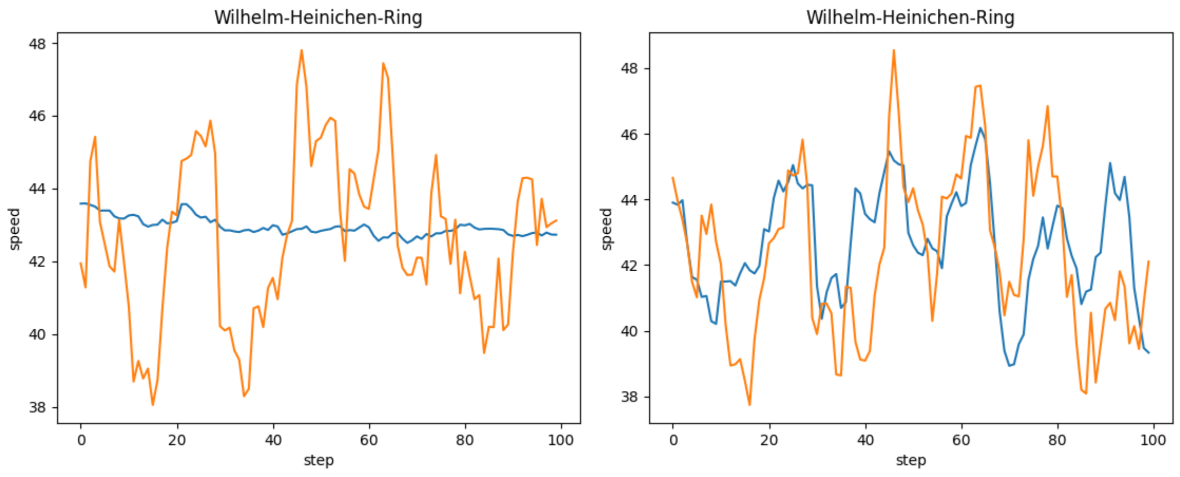


Figure 20: Comparison between predictions using a reduced dataset and full dataset

Figure 20 shows this underfitting. The graph's basic structure is the same as in Figure 19. The graphs include two curves each with the orange one being the actual measurements of the detector and the blue one being the predictions of the model. Both graphs are created with the data from the same detector using two models with the same hyper-parameters but different training datasets. The left graph gets the predictions from a model using the full dataset including all detectors. The right one gets the predictions using a model which was trained with a dataset containing only detectors having a variance greater than 1.0.
It can be seen that the predictions in the left graph are worse than in the right graph due to underfitting caused by the extra detectors in the dataset.
To overcome this issue it is recommended to only use these detectors as baselines.
One way to achieve good results is by running the simulation once (at a reduced length) and filtering out all detectors (by removing them from the definition) with a variance of less than a certain threshold (e.g. 1.0), improving the model's performance drastically as illustrated in Figure 20.

The file containing the detector definitions then has to be added to the root of the scenario files and added to the simulation by modifying the .sumocfg file by adding the file as an *additional* file as follows:

```
<input>
    <net-file value=" [...] "/>
    <route-files value=" [...] "/>
    <additional-files value=" [...], <filename/path>"/>
</input>
```

Listing 2: XML-Code to show where additional file is placed

After this, the scenario is complete and ready to be used with DeepSUMO.
When using DeepSUMO, the first step is to create a settings object containing all settings needed for running DeepSUMO and the desired deep learning model (if needed). All the framework settings *have* to be present in order for DeepSUMO to execute successfully. It is assumed that the settings object is set up correctly. Not including values or including wrong values will lead to unexpected behavior.
The model settings do not contain all settings of the original implementation of the ST-GAT model, as the testing process has been simplified resulting in some settings not being needed anymore.

```
settings : dict = {
    # ------ framework settings ------
    "sumo_exec_path": # path to SUMO executable
        "<path to sumo executable>",
    "sumo_config_path": # path to the config file of the simulation
        "<path to sumo config file >",
    "sumo_net_path": # path to the net file containing the network definition
        "<path to net.xml>",

    "sim_length": 2419200,  # total length of the simulation
    " interval_length ": 300,   # size/length of one aggregation step

    # −−−−−− model settings (hyperparameters) −−−−−−
    "N_HIST": 9, # number of preceding timesteps
    "N_PRED": 9, # number of prediction timesteps
    'WEIGHT_DECAY': 5e−5,
    'INITIAL_LR': 3e−4,
    'DROPOUT': 0.2,
    'CHECKPOINT_DIR': './runs',

    'BATCH_SIZE': 10,
    'EPOCHS': 100
}
settings ["total_graphs"] =
    int( settings ["sim_length"]) // int( settings [" interval_length "])
```

Listing 3: Code to set up settings object

Next, the simulation core has to be instantiated. The simulaton_core object needs the before-created settings object and the desired connector strategy in order to initialize all components successfully. After the simulation core has been created the user should register all desired modules using the *add_continous_observer* and *add_post_observer* methods.

```
core = sim.simulation_core( settings , distance_connector_stategieV2(50))

observer_1 = progress_module(settings["sim_length"] // 100)
observer_2 = simulationFlowControlModule(60)

core.add_continous_observer(observer_1)
core.add_continous_observer(observer_2)
```

Listing 4: Code to initialize DeepSUMO

After all modules have been added and the simulation core has successfully been initialized DeepSUMO is ready. The simulation can then be started using the following command:

```
core. start_simulation ()
```

Listing 5: Code to start DeepSUMO

DeepSUMO now runs the simulation including all modules and data collection. The function automatically finishes after the simulation and all post-processing modules. The collected data is now fully available. This example uses a slightly modified version of the original dataset used in the ST-GAT implementation to adapt it to DeepSUMO as a data source. The dataset is created and then split into training-, validation- and test data loaders as follows:

```
data = data.
    adaptive_speed2vec_dataset(core.get_data(), settings["total_graphs"])

train_threshold = int(len(data) * 0.8)
valid_threshold = int(len(data) * 0.9)
test_threshold = int(len(data) * 1.0)

train_loader = DataLoader(data[:train_threshold], shuffle=False,
    batch_size=settings["BATCH_SIZE"])
valid_loader = DataLoader(data[train_threshold:valid_threshold],
    shuffle=False, batch_size=settings["BATCH_SIZE"])
test_loader = DataLoader(data[train_threshold:test_threshold],
    shuffle=False, batch_size=settings["BATCH_SIZE"])
```

Listing 6: Code to create and split dataset

The train-, valid-, and test thresholds can be individually set as desired. The values above only serve as an example. After all data loaders have been created the training process can be run by calling:

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(f"Using {device}")

model = model_train(train_loader, valid_loader, settings, device)
```

Listing 7: Code to set device for training

After training the model it is now possible to evaluate its performance. The following code will explain how to plot the test data and model predictions side by side.
To make plotting easier the first step is to get the test data and model predictions in an easy-to-plot format. In this case, the test data and predictions are presented as arrays of size $(num_{intervals}, num_{nodes}, n_{pred})$ in chronological order. This is accomplished by the following code in Listing 8.

```
model.eval()
model.to(device)

for i, batch in enumerate(test_loader):
    # get predictions from model
    # shape -> (batch_size * num_nodes, n_pred)
    batch = batch.to(device)
    with torch.no_grad():
        pred = model(batch, device)
    truth = batch.y.view(pred.shape)

    # all values have been normalized using the z-score method,
    # so they have to be unnormalized again for evaluation
    truth = un_z_score(truth, test_loader.dataset.mean,
                       test_loader.dataset.std_dev)
    pred = un_z_score(pred, test_loader.dataset.mean,
                      test_loader.dataset.std_dev)

    truth = batch.y.view(pred.shape)
    if i == 0:
        # initialize collection variable with zeroes on first batch
        # shape -> (num_batches, batch_size * num_nodes, n_pred)
        y_pred = torch.zeros(len(test_loader),
            test_loader.batch_size * settings["N_NODE"], pred.shape[1])
        y_truth = torch.zeros(len(test_loader),
            test_loader.batch_size * settings["N_NODE"], pred.shape[1])
    # append data from current batch to data from other batches
    y_pred[i, :pred.shape[0], :] = pred
    y_truth[i, :pred.shape[0], :] = truth

# reshape into a more readable format
# (num_batches, batch_size * num_nodes, n_pred)
# -> (num_intervals, n_node, n_pred/n_hist)
y_pred = y_pred.reshape(len(test_loader) * 50,
                        settings["N_NODE"], 9)
y_truth = y_truth.reshape(len(test_loader) * 50,
                          settings["N_NODE"], 9)

# cut off the last elements if there are fewer data points in the data as
# maximum batch-size (because they are 0)
y_pred = y_pred[:len(test_loader.dataset), :, :]
y_truth = y_pred[:len(test_loader.dataset), :, :]
```

Listing 8: Code to put predictions ofdetectors in a sequence

The code in Listing 8 transforms the data by iterating over each batch of data included in the test data loader. For each batch, it does the following: first getting the predictions from the model and the real values from the current batch. The predictions and real values are both of shape $(batch\_size * num_{nodes}, n_{pred})$
If the first batch is being processed, two arrays with the shape $(num_{batches}, batch\_size * num_{nodes}, n_{pred})$ are initialized with zeroes to collectively store the data from all batches. After this, the data is unnormalized and added to the collection arrays.

After all batches have been processed, the arrays containing the collected data are reshaped into their almost final shape of $(num_{batches} * batch\_size, num_{nodes}, n_{pred})$ making the data much more readable and easier to work with.

It can sometimes happen that the collection array is not completely filled and tailed by zeroes. This happens when the number of actual data points stored does not equal the maximum number of data points that can be stored. For example: If 123 data points are included in the test data loader with a batch size of 50 the collection array is initialized with the size $(3, 50 * num_{nodes}, n_{pred})$. This allows a total of 150 data points to be stored. But, as only 123 data points are available the last 27 data points are still filled with zeroes from the initialization of the array.
These data points are removed in the last step resulting in the final shape of $(num_{intervals}, num_{nodes}, n_{pred})$.

After the data has been collected it can be easily plotted as follows:

```
s1 = []
s2 = []
# get a random node
target = random.randint(0, settings["N_NODE"] - 1)

# get name of street associated with chosen detector
sumo_id = core.get_data().translation.get_detector_id(target)
target_lane_id = traci.inductionloop.getLaneID(sumo_id)
target_lane: sumolib.net.lane.Lane = core.get_data().net.getLane(target_lane_id)
target_edge: sumolib.net.edge.Edge = target_lane.getEdge()

# get predictions and truth from the arrays and only take the first prediction
# as the model predicts a series of 9 timesteps
for i in range(len(y_pred)):
    s1.append(y_pred[i][target][0] * 3.6)
    s2.append(y_truth[i][target][0] * 3.6)

plt.title(target_edge.getName())
plt.plot(range(len(y_pred)), s1, label='pred')
plt.plot(range(len(y_pred)), s2, label='truth')
plt.xlabel("step")
plt.ylabel("speed")
plt.show()
```

Listing 9: Code to plot predictions and truth of random detector

The code in Listing 9 plots the data created in Listing 8 by doing the following: First, a random node is chosen by choosing a random number from the interval $(0, num_{nodes} - 1)$. This number is the *index* which indexes a detector at the node feature level.
Next, some information about the detector is collected. To collect data about the detector from SUMO it is necessary to first retrieve the SUMO-ID of the detector using the

translation component of DeepSUMO. After the SUMO-ID has been retrieved the lane on which the detector is placed is queried from SUMO and the name of the associated street is extracted.
After the street name has been collected two sequential lists are created by extracting the first prediction timesteps and reference values from the data created in Listing 8 for a configurable number of consecutive intervals. This works by iterating through the arrays and adding the first prediction of the chosen detector (since in this case the model predicts a total of nine timesteps) for each interval to a list.
After the lists are created they are plotted using PyPlot.

The full code can also be found as a jupyter notebook in DeepSUMO's example folder (*examples/celle_example/example.ipynb*). The next subsection will take this code, run it using an example scenario and analyze the results.

6.3 Results

The area chosen for this test scenario is the area west of the railway station of the German town "Celle" and is shown in Figure 21. The scenario has been generated using the OSMWebWizard and has not been corrected (so some junctions might be wrong).

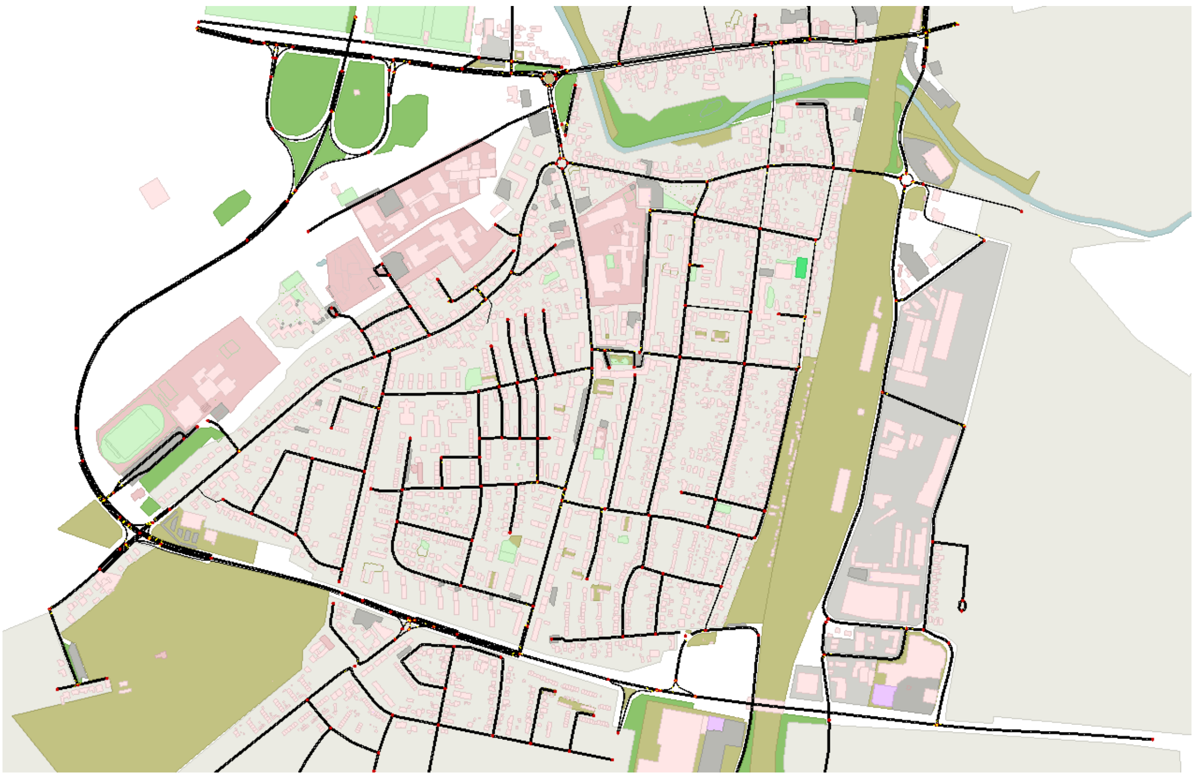


Figure 21: Screenshot of scenario

The simulation has a length of 2419200 steps, which (with a simulation step size of 1s) equals one month. The detectors have initially been generated using the tool included with DeepSUMO and have then been filtered by their variance being bigger than 1.0 over the course of one week (604800 steps) of simulation. The nodes of the graph are connected by their geographical distance to each other being smaller than 50m.
All detectors have an aggregation interval of 300 steps (= 5 minutes) and there are a total of 274 detectors in the simulation.
The demand in the simulation was initially generated using random trips but is continuously controlled by a *simulationFlowControlModule* during the simulation runtime.
The detector graph is created by connecting nodes with a geographical distance smaller than 50 meters from each other, resulting in a graph with 274 nodes and 2338 edges.
The hyperparameters of the ST-GAT model are chosen to closely resemble the original values by the creator with the only difference being increased dropout to reduce underfitting.

Parameter	**Description**	**Value**
BATCH_SIZE	Batch size of data loader	50
EPOCHS	Epochs of training	60
WEIGHT_DECAY		5e-5
INITIAL_LR		3e-4
N_NODE	How many nodes?	274
DROPOUT	Dropout	0.6
N_PRED	Forecast how many timesteps?	9
N_HIST	How many historical timesteps are available?	12

The simulation is run, after which the model is trained with the abovementioned parameters using the code described in subsection 6.2. Evaluating the model results in the following prediction:

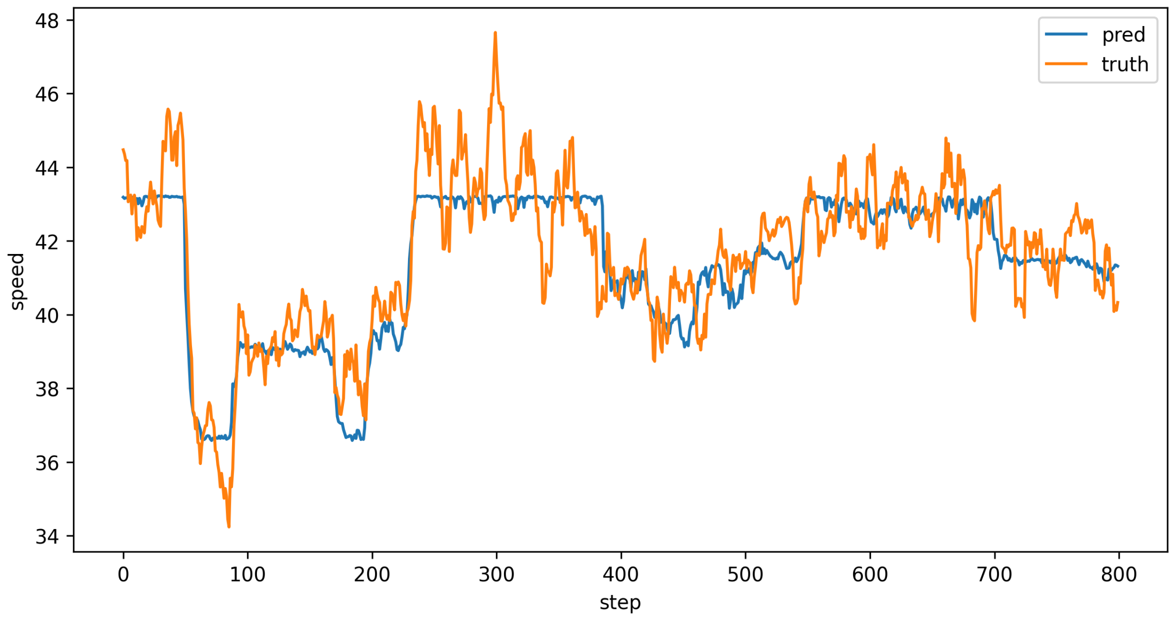


Figure 22: Results of test run using the ST-GAT model

Figure 22 shows the model's predictions (blue) and the data collected by DeepSUMO(orange). The basic structure of the graph is again the same as in Figure 19 and Figure 20. The graph covers a timespan of about three days. A Friday, a Saturday, and a Sunday. It shows some interesting aspects of the traffic captured by DeepSUMO. It shows the night before Friday (until approx. step 50) with relatively high average speeds, the day after (until approx. step 250) with much more traffic and therefore a lower average speed, and the night after with, again, higher average speeds. It shows the same pattern for the weekend as well, but with a higher average speed during the day than on a weekday, due to fewer cars being on the road.

Overall the model performed well in capturing the general direction of traffic speed, despite having difficulties with underfitting in the more detailed and extreme regions of the data (especially between steps 250 and 380 in Figure 22). The model finishes with an MAE (mean absolute error) of 0.3975, an RMSE (root mean square error) of 0.6965, and a MAPE (mean absolute percentage error) score of 3.511 for the validation dataset. It also shows that DeepSUMO was able to collect the data from the simulation correctly. The observed underfitting, in this case, can be caused by several reasons:

- Since the training data only ranges one month, the total amount of data is relatively low.
- The hyperparameters have not been adjusted correctly (due to time constraints)
- The data overall is too noisy and the model is not complex enough to capture the fluctuations. This could be fixed by implementing more advanced algorithms for smoothing the input data

But most of these issues can be resolved by further analyzing and optimizing the model and scenario to achieve a better training result. But this example shows that it is possible to train graph neural networks with data obtained from a traffic simulator successfully and that DeepSUMO is able to extract the data and feed it into a neural network.

7 Discussion

Now that a framework for working with simulated data in graph neural networks has been proposed, this section will first discuss the general approach of using simulated traffic data, in contrast to real-life datasets for the training and testing of such graph neural networks and highlight possible usage scenarios. Secondly, DeepSUMO, the created framework, will be discussed.
Since this thesis focuses on using SUMO as the main traffic simulator, this discussion will also mostly be based on SUMO as it is the used traffic simulator. However, most aspects are valid for other simulators as well. section 8 will then give some ideas for future research based on the findings of this discussion.

7.1 General approach

The possibility of using traffic simulators has already been briefly discussed in a survey for traffic predictions using graph neural networks by Jiang et al. [JL22], finding them useful for certain scenarios like unseen topology and accidents. Data obtained from a traffic simulator has also already been used to great effect in a thesis by Fukuda et al. [FUFY20] who used the *MATES* simulator to generate training and testing data for creating and evaluating a graph convolutional recurrent neural network for short-term traffic prediction under incident scenarios.
But a traffic simulation (in combination with graph neural networks) can also be used for other use cases not related to the direct training/validation process of the model itself. It can also be used for evaluating and testing applications based on traffic prediction using deep learning methods. One possibility for this would be when developing a navigation system that makes its route choices depending on the predictions of a graph neural network.
For this use case conducting large real-life tests can be expensive and hard to organize. Therefore using a traffic simulator like SUMO using a very long and large-scale simulation for testing purposes can be a good alternative to real-life tests to find out information such as (one) how often do I need to retrain my model to achieve the best accuracy? and (two) what happens when 80% of all vehicles use the software and how does it affect the predictions and route choices? and much more.
Back to the training and testing of graph neural networks, traffic simulators can also provide significant value by being able to simulate situations not available in the available datasets. One of these examples is (as already used by [FUFY20]) accidents, which are hard to get data from in real-life, but relatively easy to simulate using a traffic simulator like SUMO. Traffic simulators can also be used to test models in specific edge cases not included in the available datasets as finding real-world data of them is difficult.
Another primary use case of simulations in relation to graph neural networks is that they can be used (as mentioned in [JL22]) for evaluating models on road topology not included in the currently available datasets. For example, the publicly available datasets do not contain much data from European countries. But, since the road topology between Europe and other continents like America (and even in between cities) can be vastly different [Boe19, BGR+06], a traffic simulator can be very useful by allowing to evaluate models on road topology not represented by the available datasets, by being able to theoretically simulate an infinite number of scenarios.
Using SUMO specifically as the base traffic simulator also has some big advantages with SUMO being entirely open-source and under active development. SUMO also has a large community and is already widely used in academic research [ALBB+18]. SUMO was also specifically developed for conducting large-scale simulations, which benefits the use case of using it for training/evaluating graph neural networks on a large scale.

The biggest problem however with using a traffic simulator for testing/training neural networks is the scenario generation. Creating realistic scenarios can be (depending

on the size) very complicated and time-consuming. While SUMO allows the user to convert OpenStreetMap Data to network files understood by SUMO, the generated files often do not correctly represent reality. Especially on big junctions and roundabouts the OpenStreetMap data is most of the time not entirely correct. This leads to a lot of adjustment of lanes, traffic light placements, and -programs being needed in order to use the scenario properly [MHWS21].
Another challenge when creating a new scenario is demand modeling. Although SUMO offers various tools to help with this (like ActivityGen [CECH22], random traffic, and detector data) creating realistic demand can still provide a challenge. The best results are possible when using real-life information using data from detectors or O/D Matrices, which might be difficult to find in some cases. However, a study by Ma et al. [MHWS21] which evaluates the accuracy of ActivityGen found that in most cases it is relatively accurate with having a tendency of over smoothing traffic.
Also, the driver behavior modeled in a traffic simulation might be pretty accurate, but won't exactly represent a driver's behavior in real life. Most simulations use car following- (who does the current vehicle interact with the vehicle in front of it?), lane changing- (how does a lane change impact the adjacent lanes), and gap acceptance (what are the conditions before a lane change?) models and more to try to mimic the behavior of a real-life driver [AMAE22], which is a good abstraction, but not an exact replica of real driving behavior. This can have an impact on the simulation's accuracy.
Overall it is very hard for a simulation to capture the full complexity of traffic in the real world including the human factor. Therefore simulations will only provide a good representation and not an exact copy of the real world.
This (similar to Figure 20) might lead to over-/underfitting resulting in the model not performing well using real-world data. The nature of the models used to generate the traffic of the simulation might also bias the neural network when using only simulated data for training.
Depending on the simulation size, creating the scenario as well as running the simulation can take a long time as well as traffic simulation (in SUMO) is a single-threaded task [Erd18]. Also using a simulator adds an extra step to the workflow and requires additional knowledge on how to properly configure and use the simulator and integration framework.

7.2 DeepSUMO

Looking at DeepSUMO the goal was to develop a framework for connecting a traffic simulator in the form of SUMO with graph neural networks by extracting and processing data for use with graph neural networks. The created prototype fulfills all mentioned requirements in section 4 in an acceptable way. The prototype is able to run a traffic simulation using SUMO, extract all necessary data, and store and present it in a format specifically designed for graph neural networks. It was also modeled independently from any framework used to implement graph neural networks using NumPy. This results in the framework being usable with every currently available framework for deep learning.

It contains a good implementation for the creation of the detector graph using various strategies and allows the user to create their own strategy using the strategy pattern. After creation, DeepSUMO saves the graph data in a lot of different ways making it very easy to implement and collect the needed data.

Using the observer pattern, DeepSUMO also enables the user to interact with the simulation and collected data in a configurable and periodic way.
One drawback to this is that the module system is intended to only allow access to the data periodically (e.g. every 60 steps) and not randomly(= whenever the user wants). This is intentional and chosen because of two reasons: (one) It is easier to implement the observer pattern by calling the update functions periodically (two) Running modules at every simulation step can be *very* performance heavy, depending on the complexity of the module. Because of this, the decision was made to force the user to set a frequency for the modules, in contrast to running all modules at every step, in order to force the user to think about how often the module really has to run and adjust the frequency accordingly. This limitation can be theoretically overcome by setting the frequency to 1 or 0, but is strongly discouraged.
But all in all this method of modeling user interaction is a good way of enabling the user to interact with the simulation and collected data at the same time.

The data collection itself is modeled similarly to a module and collects data periodically at every end of an interval while the simulation is running. It does this by requesting the data from TraCi and storing it in arrays optimized for use with neural networks. This system works well and is overall a good way to capture and process the data as soon as it becomes available. DeepSUMO also offers a denoising option to smooth out the collected data. Due to time constraints, however, DeepSUMO currently only denoises the data using a moving average, which is a very basic algorithm. It would be a nice addition to add more denoising/smoothing algorithms in the future to improve the quality of the input data.
The data is also adjusted to not represent the average speed of all passed vehicles, but rather the maximum possible average speed if a vehicle would want to pass the detector (for more information refer to subsubsection 5.4.2). This is a better representation since it makes the data more robust against outliers and more representative.
A debatable decision in this section is the correction of the occupancy value. The occupancy values are divided by the number of vehicles that passed the detector. This is done because SUMO collects these values cumulatively, which means that if two vehicles are standing on the detector simultaneously values above 100% are possible. [SUM22a] This correction was chosen after conducting various tests, resulting in the corrected value overall seeming to be more representative, especially in areas with more dense traffic. This adjustment however has a negative effect on areas on areas/scenarios with little traffic. Therefore this adjustment is highly dependent on the scenario and the detector locations. Working on making this adjustment optional (maybe even on a per-detector level) would be another good addition to the framework in the future.

Another current problem with this function is a current bug in SUMO that affects DeepSUMO's functionality drastically. In the current version of SUMO (1.18.0), the data of a detector for each individual interval is different, depending on whether the data was exported to an XML-File by SUMO or captured using TraCi. As both come from the same data source this data should be identical. This results in the data captured by SUMO potentially being incorrect. This issue/bug has been reported to the SUMO developers and a fix is (as of August 7th, 2024) expected to be implemented in SUMO version 1.20, which is scheduled to release on February 20th, 2024 [BEV23].

Detectors can be automatically generated using DeepSUMO's *writer* tool. This tool works well and creates the detectors on the network as configured. One downside of this tool however was already described in subsection 6.2 with the tool generating detectors in areas without much traffic. This can result in a lot of the data being uninteresting due to the maximum possible speed always being the same. In my testing, this also has an effect on the model's training performance by introducing underfitting and may affect the model's transferability to real-life. This can be partially resolved by for example filtering out all detectors with a low variance over the course of the simulation, but DeepSUMO currently does not include any code to perform this operation due to time constraints.

During the testing in section 6 the framework performed well and successfully collected and processed all data correctly. Due to time constraints, however, the scenario chosen for this is "only" imported from OpenStreetMap and was not adjusted which leads to the lane topology and traffic lights on most (especially big) junctions being incorrect. To minimize this issue the scenario area was specifically chosen to include very few big junctions while still providing a good representation of an urban area with various different types of roads. Also, the traffic in this scenario was generated using random trips and controlled and adjusted using the *flowControlModule*. Despite the *flowControlModule* including some randomness, it still has the tendency to introduce repetitive traffic patterns due to the *flowControlModules* rules only lasting for one week. The decision to not use tools like *ActivityGen* was made due to time constraints, since creating proper scenarios can take a long time. There are also prebuild scenarios available, but they do not have sufficient length to generate enough data to train a full graph neural network. All of this results in the used scenario being more of an abstraction rather than a representation of real-life traffic.
But despite these issues, the used model still managed to predict the traffic of the simulation relatively well showing that data from a traffic simulator can be used to train a graph neural network. Also, DeepSUMO did its job perfectly by collecting all data and feeding it to the neural network, which should be the main takeaway of that section. It would be very interesting, however, to run a similar evaluation again with a better scenario and a more optimized model in the future.

8 Future research

This section will provide some questions for future research using DeepSUMO. The questions of this section are based on the result of section 7 and are divided into two sections. Firstly areas, where DeepSUMO itself can be improved, will be presented. Secondly, some research questions that can be achieved using DeepSUMO will be given.

8.1 DeepSUMO

DeepSUMO is the framework presented in the thesis to connect SUMO with graph neural networks. it works well but still has a lot of potential for future extensions.

Add more data denoising options DeepSUMO currently only offers a moving average for data denoising. Since this algorithm is very basic, a good addition would be to add more denoising options and a system to dynamically set the denoising algorithm. This can result in further improvement of the input data and prediction performance of the neural network.

Add more/dedicated support for design pipeline for neural networks As mentioned by Wang et al. [WJJ+21] "There are no open-source libraries for unifying the entire pipeline consisting of data preparation, model design and implementation, and performance evaluation". In combination with adding more visualization tools, it would be great to extend DeepSUMO's functionality to support the entire pipeline in an intuitive way since DeepSUMO currently only focuses on the data preparation and partially on the model implementation part.

Improve writer tool As described in section 7, the writer tool currently only has very limited functionality and little customizability. It also creates detectors in uninteresting locations. It would be a good addition to further develop this tool to place detectors more intelligently to avoid the creation of unnecessary detectors. Another possibility is to properly implement the filtering function, used to filter out uninteresting detectors, into DeepSUMO. The tool is also currently specialized to work with road networks obtained from OpenStreetMap. Another meaningful addition would be to extend its functionality beyond OpenStreetMap data.

Make adjustments optional As described in subsubsection 5.4.2 DeepSUMO adjusts the collected values from SUMO. As discussed in section 7 these adjustments have a big effect on the data stored by DeepSUMO. To make these adjustments more configurable by the user it would be a good addition to make these adjustments optional (maybe even on a per-detector level).

8.2 Application

As DeepSUMO is able to extract data from the simulation and store/provide access to it that makes it easy to integrate it with neural networks it opens up a lot of interesting research questions relating to the linking of traffic simulators and neural networks for traffic forecasting as described in section 7.

Test and compare DeepSUMO further/better The test conducted in subsection 6.3 was basic, using a simple scenario and basic traffic demand. It would be interesting to test DeepSUMO again using a more complex scenario with a better topology and more realistic demand. The results of this test can then also be used to interpret the transferability of neural networks trained for traffic prediction using a traffic simulator to real life and vice versa.

Create a benchmark/standard for testing of graph neural networks Similar to some real-life datasets serving as a "benchmark" for testing and most importantly comparability between graph neural networks it would be interesting to develop a set of high quality, long-lasting scenarios covering a vast amount of different types of road topology to serve as a very diverse benchmark for comparing the performance of graph neural networks for traffic forecasting.

Test applications based on graph neural networks and traffic forecasting Another application of DeepSUMO already mentioned in section 7 is using DeepSUMO to test applications that work using graph neural networks for traffic predictions. DeepSUMO can also provide a platform to test such applications by interfacing with the simulation and data using the module system. An example of this would be a navigation system, which makes routing decisions based on the predictions of such a model. DeepSUMO could provide a platform to test various aspects of the application like how often the model has to be retrained, what happens when 70% of all drivers use the navigation tool, and more.
Of course, this was only an example, since there are many more applications and ideas that can be tested and evaluated using DeepSUMO.

9 Conclusion

During this thesis, a framework for training/testing graph neural networks using data from a traffic simulator (in this case SUMO) was developed. The framework contains various functions related to this task. It was modeled after the use cases in section 4 and fulfills most of them.
It can initialize the simulation and create a graph structure from configured detectors. It can also continuously collect data from the simulation while the simulation is still running. The user can use the data from the framework using observers, which are called periodically, or after the simulation has finished. The data is also saved in various formats optimized for use/integration with graph neural networks for traffic prediction. The functionality of the framework was then examined by coding an example and testing the training, validation, and testing process of a neural network using data extracted from SUMO using DeepSUMO. This experiment went well and proved that it is possible to extract data from the simulation and train/test a graph neural network.
It can therefore be said, that the objective of developing a framework to connect a traffic simulator with graph neural networks was achieved successfully
The general aspect of using simulated traffic data for the training of graph neural networks was also discussed. The discussion came to the conclusion that simulated data can be very useful for training and especially evaluating graph neural networks on new road topology and on scenarios not present in the available real-life datasets (like accidents). It was also discovered, however, that the quality of the trained model and simulation, in general, is highly dependent on the input topology and traffic demand for the simulation. This can provide a significant challenge, as creating high-quality scenarios can be time-consuming to create, and using traffic simulators might be challenging to integrate into some workflows as using simulators requires additional knowledge of the simulation tool used.

References

[ALBB$^+$18] Pablo Alvarez Lopez, Michael Behrisch, Laura Bieker-Walz, Jakob Erdmann, Yun-Pang Flötteröd, Robert Hilbrich, Leonhard Lücken, Johannes Rummel, Peter Wagner, and Evamarie Wießner. Microscopic Traffic Simulation using SUMO. In *Proceedings of 21st International Conference on Intelligent Transportation Systems (ITSC)*, pages 2575–2582, Maui, USA, December 2018. IEEE. https://doi.org/10.1109/ITSC.2018.8569938.

[ALE16] Pablo Alvarez Lopez and Jakob Erdmann. Supporting additional elements in a simulation using Netedit. In *SUMO User Conference 2016*, Braunschweig, May 2016. https://elib.dlr.de/124399/.

[AMAE22] Taghreed Alghamdi, Sifatul Mostafi, Ghadeer Abdelkader, and Khalid Elgazzar. A Comparative Study on Traffic Modeling Techniques for Predicting and Simulating Traffic Behavior. *Future Internet*, 14(10):294, October 2022. https://www.mdpi.com/1999-5903/14/10/294.

[BBEK11] Michael Behrisch, Laura Bieker, Jakob Erdmann, and Daniel Krajzewicz. SUMO - Simulation of Urban MObility - an Overview. In *SIMUL 2011, The Third International Conference on Advances in System Simulation*, pages 55–60, Barcelona, Spain, October 2011. https://www.thinkmind.org/index.php?view=article&articleid=simul_2011_3_40_50150.

[BCY22] Khac-Hoai Nam Bui, Jiho Cho, and Hongsuk Yi. Spatial-temporal graph neural network for traffic forecasting: An overview and open research issues. *Applied Intelligence*, 52(3):2763–2774, February 2022. https://doi.org/10.1007/s10489-021-02587-w.

[BEV23] Michael Behrisch, Jakob Erdmann, and Erik Vogel. Mismatch between traci.inductionloop.getLastIntervalVehicleNumber and xml attribute nVehContrib. https://github.com/eclipse/sumo/issues/12896, July 2023.

[BGR$^+$06] J. Buhl, J. Gautrais, N. Reeves, R. V. Solé, S. Valverde, P. Kuntz, and G. Theraulaz. Topological patterns in street networks ofself-organized urban settlements. *The European Physical Journal B - Condensed Matter and Complex Systems*, 49(4):513–522, March 2006. https://doi.org/10.1140/epjb/e2006-00085-1.

[Boe19] Geoff Boeing. Urban spatial order: Street network orientation, configuration, and entropy. *Applied Network Science*, 4(1):67, August 2019. https://doi.org/10.1007/s41109-019-0189-1.

[Bra68] Dietrich Braess. Über ein Paradoxon aus der Verkehrsplanung. *Unternehmensforschung*, 12(1):258–268, December 1968. https://doi.org/10.1007/BF01918335.

[CECH22] Lara Codecà, Jakob Erdmann, Vinny Cahill, and Jérôme Härri. SAGA: An Activity-based Multi-modal Mobility ScenarioGenerator for SUMO. In *SUMO Conference Proceedings*, volume 1, pages 39–58, July 2022. https://doi.org/10.52825/scp.v1i.99.

[DRC+17] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An Open Urban Driving Simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, volume 78, pages 1–16. PMLR, October 2017. https://proceedings.mlr.press/v78/dosovitskiy17a.html.

[Erd18] Jakob Erdmann. Re: [sumo-user] The multi-threaded option in SUMO. https://www.eclipse.org/lists/sumo-user/msg01706.html, June 2018.

[Flö09] Gunnar Flötteröd. Cadyts a free calibration tool for dynamic traffic simulations. In *Proceedings of 9th Swiss Transport Research Conference*, 2009. https://infoscience.epfl.ch/record/152358.

[FUFY20] Shota Fukuda, Hideaki Uchida, Hideki Fujii, and Tomonori Yamada. Short-term prediction of traffic flow under incident conditions using graph convolutional recurrent neural network and traffic simulation. *IET Intelligent Transport Systems*, 14(8):936–946, May 2020. https://doi.org/10.1049/iet-its.2019.0778.

[Ham20] William L Hamilton. Graph Representation Learning. 14:1–159, 2020. https://doi.org/10.1007/978-3-031-01588-5.

[JL22] Weiwei Jiang and Jiayun Luo. Graph Neural Network for Traffic Forecasting: A Survey. *Expert Systems with Applications*, 207:117921, November 2022. https://doi.org/10.48550/arXiv.2101.11174.

[KEBB12] Daniel Krajzewicz, Jakob Erdmann, Michael Behrisch, and Laura Bieker. Recent Development and Applications of SUMO – Simulation of Urban MObility. 5(3):128–138, 2012. https://sumo.dlr.de/pdf/sysmea_v5_n34_2012_4.pdf.

[KPW+18] Nishant Kheterpal, Kanaad Parvate, Cathy Wu, Aboudy Kreidieh, Eugene Vinitsky, and Alexandre Bayen. Flow: Deep Reinforcement Learning for Control in SUMO. In *SUMO 2018- Simulating Autonomous and Intermodal Transport Systems*, EPiC Series in Engineering, pages 134–151, June 2018. https://doi.org/10.29007/dkzb.

[KZY+18] Song Sang Koh, Bo Zhou, Po Yang, Zaili Yang, Hui Fang, and Jianxin Feng. Reinforcement Learning for Vehicle Route Optimization in SUMO. In *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 1468–1473. IEEE, June 2018. https://doi.org/10.1109/HPCC/SmartCity/DSS.2018.00242.

[mai23] Garage maintainers. Garage. Reinforcement Learning Working Group, April 2023. https://github.com/rlworkgroup/garage.

[MHWS21] Xiaoyi Ma, Xiaowei Hu, Thomas Weber, and Dieter Schramm. Evaluation of Accuracy of Traffic Flow Generation in SUMO. *Applied Sciences*, 11(6):10, March 2021. https://doi.org/10.3390/app11062584.

[Ope23] OpenStreeMap. Key:highway - OpenStreetMap Wiki. https://wiki.openstreetmap.org/wiki/Key:highway, May 2023.

[Pig02] Arthur Cecil Pigou. Welfare and Economic Welfare. In *The Economics of Welfare*, pages 3–22. Routledge, 2002.

[PyT23] PyTorch. Tensors — PyTorch Tutorials 2.0.1+cu117 documentation. https://pytorch.org/tutorials/beginner/blitz/tensor_tutorial.html, 2023.

[QA13] Kashif Naseer Qureshi and Abdul Hanan Abdullah. A Survey on Intelligent Transportation Systems. 15(5):629–642, January 2013. https://www.researchgate.net/publication/257367335_A_Survey_on_Intelligent_Transportation_Systems.

[SRPW21] Benjamin Sanchez-Lengeling, Emily Reif, Adam Pearce, and Alexander B. Wiltschko. A Gentle Introduction to Graph Neural Networks. *Distill*, 6(9):e33, September 2021. https://doi.org/10.23915/distill.00033.

[SUM] SUMO. Eclipse SUMO - Simulation of Urban MObility. https://www.eclipse.org/sumo/.

[SUM22a] SUMO. Induction Loops Detectors (E1) - SUMO Documentation. https://sumo.dlr.de/docs/Simulation/Output/Induction_Loops_Detectors_%28E1%29.html, December 2022.

[SUM22b] SUMO. Lanearea Detectors (E2) - SUMO Documentation. https://sumo.dlr.de/docs/Simulation/Output/Lanearea_Detectors_%28E2%29.html, December 2022.

[SUM22c] SUMO. Multi-Entry-Exit Detectors (E3) - SUMO Documentation. https://sumo.dlr.de/docs/Simulation/Output/Multi-Entry-Exit_Detectors_%28E3%29.html, November 2022.

[SUM23a] SUMO. Dfrouter - SUMO Documentation. https://sumo.dlr.de/docs/dfrouter.html, April 2023.

[SUM23b] SUMO. Jtrrouter - SUMO Documentation. https://sumo.dlr.de/docs/jtrrouter.html, January 2023.

[SUM23c] SUMO. Sumolib - SUMO Documentation. https://sumo.dlr.de/docs/Tools/Sumolib.html, March 2023.

[SZL+22] Yaofeng Song, Han Zhao, Ruikang Luo, Liping Huang, Yicheng Zhang, and Rong Su. A SUMO Framework for Deep Reinforcement Learning Experiments Solving Electric Vehicle Charging Dispatching Problem. https://doi.org/10.48550/arXiv.2209.02921, September 2022.

[Ten22] TensorFlow. Load NumPy data | TensorFlow Core. https://www.tensorflow.org/tutorials/load_data/numpy, December 2022.

[Tra22] TraCi. Induction Loop Value Retrieval - TraCi Documentation. https://sumo.dlr.de/docs/TraCI/Induction_Loop_Value_Retrieval.html, December 2022.

[UCBBC20] Luis Urquiza-Aguiar, William Coloma-Gómez, Pablo Barbecho Bautista, and Xavier Calderón-Hinojosa. Comparison of SUMO's vehicular demand generators in vehicular communications via graph-theory metrics. *Ad Hoc Networks*, 106:102217, September 2020. https://www.sciencedirect.com/science/article/pii/S1570870520301773.

[UCGBBC19] Luis F. Urquiza-Aguiar, William Coloma Gómez, Pablo Barbecho Bautista, and Xavier Calderón. Comparison of Traffic Demand Generation Tools in SUMO: Case Study: Access Highways to Quito. In *Proceedings of the 16th ACM International Symposium on Performance Evaluation of Wireless Ad Hoc, Sensor, & Ubiquitous Networks*, PE-WASUN '19, pages 15–22, New York, NY, USA, November 2019. Association for Computing Machinery. https://doi.org/10.1145/3345860.3361521.

[WBF] Piotr Woznica, Walter Bamberger, and Matthew Fullerton. Activity-based Demand Generation - SUMO Documentation. https://sumo.dlr.de/docs/Demand/Activity-based_Demand_Generation.html.

[WJJ+21] Jingyuan Wang, Jiawei Jiang, Wenjun Jiang, Chao Li, and Wayne Xin Zhao. LibCity: An Open Library for Traffic Prediction. In *Proceedings of the 29th International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '21, pages 145–148, New York, NY, USA, November 2021. Association for Computing Machinery. https://doi.org/10.1145/3474717.3483923.

[WPR+08] Axel Wegener, Michał Piórkowski, Maxim Raya, Horst Hellbrück, Stefan Fischer, and Jean-Pierre Hubaux. TraCI: An interface for coupling road traffic and network simulators. In *Proceedings of the 11th Communications and Networking Simulation Symposium*, CNS '08, pages 155–163, New York, NY, USA, April 2008. Association for Computing Machinery. https://doi.org/10.1145/1400713.1400740.

[WWC22] Amelia Woodward, Julie Wang, and Tracy Cai. Predicting Los Angeles Traffic with Graph Neural Networks. https://medium.com/stanford-cs224w/predicting-los-angeles-traffic-with-graph-neural-networks-52652bc643b1, January 2022.

[WWC23] Julie Wang, Amelia Woodward, Tracy Cai, and Haoteng . Spatial-Temporal Graph Attention Networks:A Deep Learning Approach for Traffic Forecasting - Repository. https://github.com/jswang/stgat_traffic_prediction, April 2023.

[Yos06] Shinobu Yoshimura. MATES: Multi-Agent based Traffic and Environment Simulator - Theory, implementation and practical application. *CMES - Computer Modeling in Engineering and Sciences*, 11(1):17–25, January 2006. https://www.researchgate.net/publication/285016110_MATES_Multi-Agent_based_Traffic_and_Environment_Simulator_-_Theory_implementation_and_practical_application.

[ZYL19] Chenhan Zhang, James J. Q. Yu, and Yi Liu. Spatial-Temporal Graph Attention Networks: A Deep Learning Approach for Traffic Forecasting. *IEEE Access*, 7:166246–166256, November 2019. https://doi.org/10.1109/ACCESS.2019.2953888.

[ZYZ+22] Zewei Zhou, Ziru Yang, Yuanjian Zhang, Yanjun Huang, Hong Chen, and Zhuoping Yu. A comprehensive study of speed prediction in transportation system: From vehicle to traffic. *iScience*, 25(3):103909, February 2022. https://doi.org/10.1016/j.isci.2022.103909.

List of Figures

Listings