Linköpings universitet | Institutionen för datavetenskap
Examensarbete på avancerad nivå, 30hp | Datateknik
2023 | LIU-IDA/LITH-EX-A--23/107--SE

Trafikruttplanering med hjälp av grafiska neurala nätverk

Method for using graph neural networks to predict future traffic for traffic route planning
– Traffic route planning using graph neural networks

Lukas Olsson

Handledare : Johan Andersson
Examinator : Nikolaos Pappas

Linköpings universitet
SE–581 83 Linköping
013-28 10 00 , www.liu.se

Sammanfattning

With urbanization and cities becoming more congested, the need for effective traffic flow management is increasing. With real-time traffic sensors, the current traffic flow can be measured and more effective transportation routes in cities can be made using this information. Using the data from traffic sensors, a graph neural network (GNN) can be trained to predict future traffic flow. GNNs are neural networks specifically made to handle graph-structured data, which road networks can be represented by. Synthetic traffic flow data at different intersections were used to train the model. The data was collected using the *simulation software Simulation of Urban Mobility*, which is widely used in academic literature. The data is structured in one-minute intervals, where the model was trained to predict one minute into the future. Test scenarios were constructed where a vehicle had to find a route between two nodes, inside the simulated road network. This was done to test the effectiveness of the sensors and model. The tests were carried out by seven digital twins, one baseline that used the fastest path, three that used the real-time sensor data, and the last three that used the model to predict future traffic flow. It was found that the digital twins that used both sensor data and the trained GNN model to find the route performed on average better than the baseline in the metrics: time traveled, Co2 emission, time stopped, and average speed.

Författarens tack

I like to thank Combitech AB for the opportunity to do my master's thesis with them. I want to thank Johan Andersson for being my supervisor at Combitech and allowing me to work on this thesis. I would also like to thank Nikolaos Pappas at Linköping University for being such an active and supportive examiner.

Innehåll

Figurer

Tabeller

1 Introduction

1.1 Motivation

Today around four billion people live in cities, and it is predicted to increase to seven billion by 2050 [18], This presents a growing need for more efficient traffic flow management. To solve this problem, new solutions are needed to move people around with more efficiency.

This master's thesis is done at Combitech and is about using digital twins where digital twins are a digital representation of a vehicle on the road and AI methods such as deep learning to improve the efficiency and effectiveness of traffic flow control. Traffic flow management is a critical aspect of modern transportation systems and aims to optimize road infrastructure use and ensure road user's safety. Effective traffic flow management reduces travel time and emissions.

This problem is crucial as it is an area that affects a lot of people. With better traffic flow, road users can get from point A to B more quickly and easily. This saves time for the individual and contributes to a higher quality of life as the person does not need to spend the same amount of time in the car. In addition to the savings made in time, there could also be a reduction in fuel usage and a reduction in the wear and tear on vehicle components.

1.2 Problem description

The main problem that this thesis aims to solve is to first get a clear image of the current traffic situation at different parts of the road network. Then, with this information suggest different routes that aim to minimize the travel time for drivers.

In this thesis, digital twins as mentioned are a digital representation of a vehicle on the road. The digital twin will be used to simulate different actions the vehicle can take, such as how to navigate to its destination. To do this as effectively as possible, the digital twin needs access to real-time sensor data that can give additional route information. The information from the sensor can be fed to a deep learning algorithm that tries to find a faster route to the destination by, predicting what the traffic will look like along different points of the journey. The information that is received could be how congested different intersections are, or if there has been an accident or some other event that has occurred that interrupts traffic flow. With this information, the model can suggest the most efficient route to the driver's destination.

Many methods exist to find the shortest path between nodes, such as Dijkstra's algorithm [9] or A*. However, more recent papers on the subject of route planning suggest the use of machine learning methods. These methods include reinforcement learning [17] and supervised learning [12].

To determine what effects the digital twin will have, a simulator will be used. The simulator in this project will need to accurately simulate a large number of vehicles realistically to make sure that the results received can be trusted. In this project, the simulator Simulation of Urban Mobility (SUMO)[13] will be used. SUMO is a free and open-source traffic simulation software that allows the modeling of different traffic systems. There are several reasons why SUMO was chosen. One of the main reasons is that SUMO can simulate large environments, which in this project is needed to evaluate meaningfully different routes the vehicles can take to reach their destination. SUMO is also used in academic literature, making the results found trustworthy. The simulation is deterministic meaning that doing the same test twice will give the same result making the experiments reproducible.

1.3 Aim

This thesis aims to investigate if traffic flow prediction with deep learning can be used to improve traffic route creation. To determine what effect this has, the routes generated will be tested using digital twins in the traffic simulator SUMO. The metrics that will be investigated are: travel time, distance, emissions, and time stopped. This will be compared to a baseline route where the fastest path is used without traffic flow prediction. To predict the traffic situation, data from sensors that measure traffic and are located around the map is used to train the neural networks. Specifically will, graph neural networks be used as they show promising results in the traffic prediction domain. Additionally, this thesis will also investigate the effect the number of sensors will have on the model's ability to predict future traffic. Lastly, a literature study will investigate if using sensors already deployed in today's cities could improve the performance metrics.

1.4 Research questions

This thesis aims to answer the following research questions:

- How the digital twin be implemented in the real world and what sensors will be needed?
- Are routes generated with traffic flow prediction more performant than their respective baseline in regards to the metrics travel time, distance, emissions, and time stopped?
- How does the amount of sensors affect the model's ability to predict future traffic?

1.5 Method

The thesis will be carried out in the following stages. First, research in traffic prediction will be done to identify promising models for use. Second, maps for the simulation will be obtained, and traffic scenarios will be created. Third, sensors are placed on the map, and traffic is simulated to gather data. Fourth, the models are trained on the sensor data. Fifth, routes will be generated and tested in the simulator and evaluated. Lastly, an investigation into how current traffic sensors could be used to improve traffic prediction and route generation.

1.6 Delimitations

While it would be interesting to investigate how the digital twin would perform in real-life, this thesis will be limited to simulated environments. Also, Synthetic data will be generated, which means that the traffic scenarios and models trained could not be entirely accurate to real-life. The hardware used for running the simulation and training the graph neural networks was equipped with a single Nvidia RTX 2070 super.

1.7 Structure

This thesis is structured as follows. Chapter 2 will describe the relevant theory and related work required to understand this report. Chapter 3 describes the method that will be used to answer the research questions. Chapter 4 will present the results and findings. Chapter 5 will discuss the results of the experiments carried out. Chapter 6 will discuss the findings regarding implementing the digital twin in real-life, this is followed by Chapter 7 which will conclude the findings.

2 Theory

This chapter focuses on the theory needed for the following chapters. The chapter is divided into two parts, background and related work. Background cover concepts, while related work focuses on how these concepts can be applied.

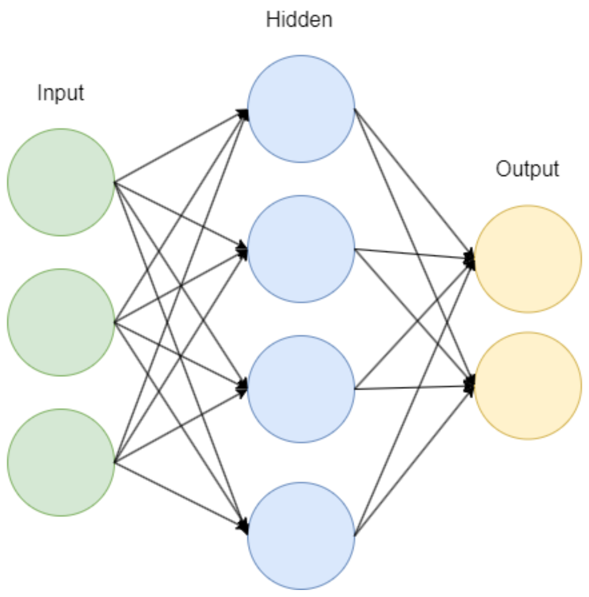
2.1 Background

2.1.1 Neural networks

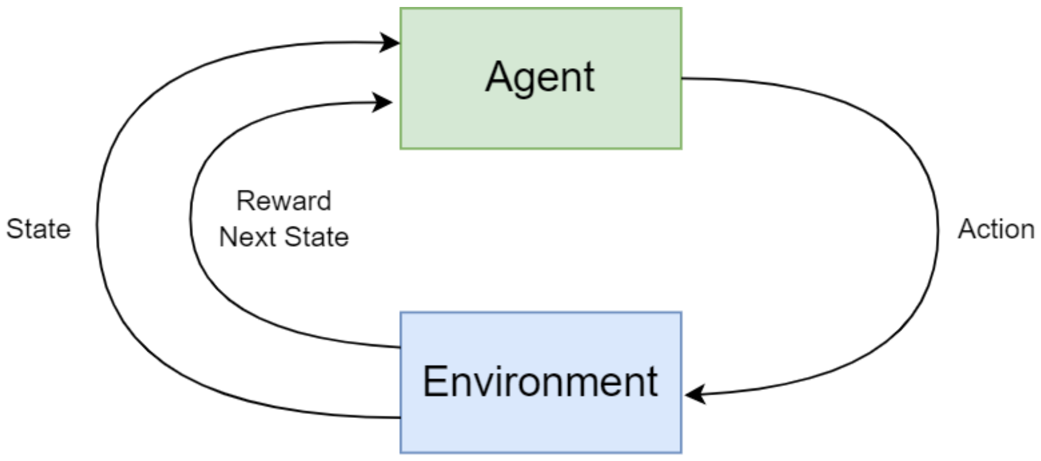
A neural network (NN)[19] is a type of machine learning technique that uses artificial neurons that loosely mimic the human brain. The NN learns to perform different tasks, such as image classification, by training on various labeled examples. NNs consist of layers of neurons that are interconnected with some or all neurons in the next layer. The connected neurons that are connected and performer mathematical operations which aims to learn complex functions. NNs are divided into three different types of layers: the input layer, the hidden layer, and the output layer. The input layer receives the data fed into the network (In image classification, this would each pixel value in an image). The hidden layer's neurons receive the inputs from the previous layer and calculate a weighted sum of those inputs. These sums are then passed through an activation function and are used to introduce non-linearity which is needed to learn complex functions. The resulting output of the hidden layer is then passed as the input to the next hidden layer or output layer. The output layer of the network is the last layer of neurons and produces the network's output. Depending on the task, the output layer can be different. However in the case of classification, the output is a probability that a class is true, where each class is a neuron in the output layer.

The way NNs learn is through the back-propagation algorithm. Back-propagation aims to minimize the error between the predicted output and the true output. The algorithm propagates the error from the output layer back through the network to the input layer. The gradient of the loss function is calculated concerning the weights and biases of the neuron; this is done using the chain rule. The gradient is used to update the weights and biases of each neuron, and the process repeats until the input layer is reached.

Figure 2.1 shows a simple neural network. The network contains three input neurons that are connected to four hidden neurons that are connected to two output neurons. All neuron are connected to all neurons in the next layer. This is called a fully connected network.



Figur 2.1: A Basic neural network with three input neurons, four hidden neurons, and two output neurons.



Figur 2.2: Interaction between agent and environment

2.1.2 Reinforcement learning

Reinforcement learning (RL)[22] is a type of machine learning in which an agent learns to make decisions in a given environment by maximize a cumulative reward. The agent interacts with the environment by taking actions and receiving a reward or punishment based on the action taken; see Figure 2.2.

The goal for the agent is to learn which actions maximize the total reward it can receive. This is also called finding the optimal policy for the environment. A reinforcement learning system include the following components:

1. Agent: The learner which takes actions in the environment.
2. Environment: The environment in which the agent operates and interacts.
3. Actions: The choices that the agent can make.
4. States: The the observations of the environment made by the agent at a given time.

5. Rewards: The feedback that the agent receives after taking action.

The RL process can be described in three steps. The first step is for the agent to observe the current state. The second step is to take action based on the information received. In the third step, the agent receives a reward based on the quality of its action. The interactions between an agent and its environment can be formulated using the Markov decision process.

2.1.3 Markov decision process

Markov decision processes (MDP)[22] is a mathematical framework for modeling decision-making problems in a stochastic environment where the outcomes of the actions are undetermined. MDPs are defined by the following components:

- S: A set of states.
- A: A set of actions.
- R: Reward function specifies the immediate reward received by moving from one state to another with a specific action.
- P: Transition function which specifies the probability of moving from one state to another when an action is taken
- γ: Discount factor

The transition and reward functions depend solely on the current state from which the action is taken, without considering the history of past states and actions. This is called the Markov property.

MDPs aim to find a policy that maximizes the expended cumulative reward. A policy is a function that works by mapping each state to an action. The optimal policy is the policy that maximizes the expected cumulative reward.

2.1.4 Q-learning

Q-learning is an RL algorithm used to learn the optimal policy in an MDP system. Q-learning is a model-free algorithm, meaning it does not require knowledge of the transition probabilities and rewards from the MDP. Instead, it learns Q-values, representing the expected cumulative reward of taking an action in a given state. The Q-values are updated using the Bellman equation, which expresses the value of a state in terms of the importance of its successor states. The Q-learning algorithm initializes the Q-values for all state-action pairs to a random value. The agent will then interact with the environment, and at each time step, it selects an action using an exploration-exploitation strategy. The strategy must balance between choosing the best-known action and exploring new actions [24].

When taking an action, the agent receives a reward and observes the next state. The Q-value for the current state-action pair is updated, see Equation: 2.1.

$$Q(s,a) = Q(s,a) + \alpha[r + \gamma max(Q(s',a') - Q(s,a))] \tag{2.1}$$

In the equation s is the current state, a is the action taken, r is the reward received, s′ is the next state, and α and γ are the hyperparameters that control the learning rate and the discount factor, respectively. The Q-value update rule uses the difference between the current estimate of the Q-value and the target estimate, which is the sum of the reward received and the maximum Q-value over all possible actions in the next state.

The agent continues interacting with the environment, updating the Q-values for each state-action pair based on the observed rewards and transitions. Over time, the Q-values converge to their optimal values, and the agent can use them to select the best action in each state. The final policy is derived from the Q-values by choosing the activity with the highest Q-value in each state.

2.1.5 Deep Q-learning

Deep Q-learning is a type of reinforcement learning that combines Q-learning with deep neural networks to learn. In normal Q-learning, the agent learns a policy maximizing the expected cumulative rewards for the state-action pairs. Deep Q-learning takes this concept further by replacing the table of Q-values with a deep neural network that takes the states as input and outputs a vector of Q-values for each possible action. The neural network will thus be an approximation of the table of Q-values using wight parameters θ, see Equation 2.2.

$$Q(s,a;\theta) \approx Q*(s,a) \tag{2.2}$$

The advantage of this approach is that Deep Q-learning can handle a larger amount of state-action pairs, in normal Q-learning as the number of state-action pairs grows, the table of Q-values grows exponentially. In turn, this will cause the training time to go out of proportion, limiting the size of states and actions. Deep Q-learning approximates the Q-table, allowing it to learn optimal policies in larger and more complex environments.

During learning, a Deep Q-learning network uses two networks, the online network and the target network. The networks have the same architecture but use different weights. The reason for this is that the target is a variable and non-stationary in the loss function; see Equation 2.3 [3]. This is different from regular deep learning, where the target does not change. The online network is responsible for adjusting the parameters of the network, and the target network is a copy of the online network that is periodically updated with the weights of the online network.

$$target = r + \gamma \max_{\alpha'} Q(s',a') \tag{2.3}$$

2.1.6 Recurrent neural network

A Recurrent Neural Network (RNN) is a type of neural network designed to handle sequential data. It can process a sequence of inputs, one at a time, while maintaining an internal state, or memory that captures information about the sequence that it has seen so far. This memory allows the RNN to consider the context of each input when processing subsequent inputs in the sequence [21].

The key feature of an RNN is its ability to maintain and update an internal state, which is passed from one step of the sequence to the next. The internal state is represented as a fixed-size vector, which is updated by combining the current input with the previous state using a set of learned parameters. The network output at each step can be computed based on the updated internal state.

2.1.7 Long short-term memory

Long Short-Term Memory (LSTM) is a type of recurrent neural network (RNN) architecture that is made to solve the vanishing and exploding gradient problem in traditional RNNs. The main difference between LSTM and RNN is the internal structure of LSTM that allows it to remember or forget information from previous steps selectively. The core idea behind LSTM is that it introduces a memory cell that retains information over long periods, and three gates that control the flow of data in and out of the cell[5]:

1. The input gate: Decides how much new information to let into the memory cell at each time step.

2. The forget gate: Decides how much information to *ease* from the memory cell at each time step
3. The output gate: Decides how much information to *output* from the memory cell at each time step.

A sigmoid function controls the gates, which outputs values between 0 and 1. The values are computed based on the current input, the previous hidden state, and the last cell memory state.

The output from the LSTM at each time step is a combination of the output from the memory cell and the output from the output gate. This allows the LSTM to selectively pass information forward in time while discarding irrelevant or redundant information.

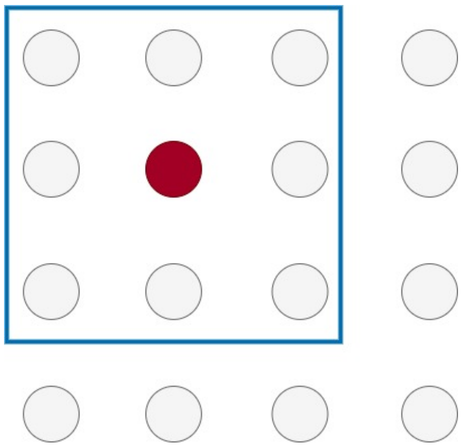
2.1.8 Graph Neural Networks

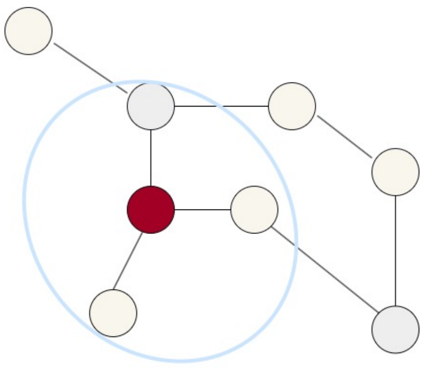
Graph neural networks (GNN) are neural networks designed for structured data like graph data and where graph data is data based on nodes and the connected edges. The information about the graph can be represented in either an adjacency matrix or a list. In an adjacency matrix, the matrix is an NxN binary matrix where N is the number of nodes in the graph. The value can be either a 0 or 1, depending on whether the nodes are connected. In an adjacency list however, each node connection is stored as a pair in a list. An adjacency list is more common because it is more space efficient than adjacency matrices. As with the adjacency matrix in GNNs there exist both node and edge feature vectors that hold some information about the node or edge, which will be used in learning. GNN aims to learn a function that maps the input feature vectors to the output feature vectors [20].

GNNs can perform different tasks. three common uses of GNNs are Node-level predictions, Edge-level predictions, and graph-level predictions [25]. In node-level predictions, the aim is to predict certain attributes of the node or classify them, an example of this could be predicting the likelihood that a user engages with a post on social media. Edge-level predictions are used to predict if there is a connection between two nodes. An example of a Edge-level predictions can be to predict connections between people and items they could be interested in buying. Graph-level predictions are used to classify or predict attributes of an entire graph. Using the information from the graph such as node features and their connections, The GNN will output new embeddings for the nodes. These embeddings contain structural and feature information about the other nodes. This means that each node in the graph has some information about the other nodes and its connection to those nodes, which are later used to make predictions. Keep in mind that the size of the embeddings can be different from the initial node input size.

The way that the GNN creates the embeddings is with the use of message-passing layers inside the neural network. The idea behind the message-passing layers is to gather the information from neighboring nodes and combine that information with new embeddings and then update the node with this new information. This process is often called graph convolution. This is similar to the convolutional layer in convolution neural networks (CNN). In CNNs, the convolutional operation uses a matrix called kernel or filter to slide across different parts of usually image data, performing convolution operation at each point. This results in a new image called a feature map [4]. The difference between the two types of convolution is shown in Figure 2.3.

More concretely, the message-passing layer has three important steps: message generation, message aggregation, and updating. The message generation step generates messages by combining the features of the node itself and its neighboring nodes, which takes the form of a transformed version of the combined features. The generated messages are aggregated for each node for the message aggregation step. This step involves combining the messages from neighboring nodes to summarize the information from the neighborhood. In the update step, the aggregated messages are then used to update the node representations, where the


(a) Convolution in a CNN.


(b) Convolution in a GNN.

Figur 2.3: The two figures demonstrate the difference between the convolution operation in a CNN compared to a GNN. In the convolution, in the CNN all neighboring pixels are used in the red pixel, and in the GNN, the connected nodes are used.

updates are applied to the current node [20] [2]. The function for updating the value of a node will be as follows in Equation 2.4[2]:

$$h_u^{(k+1)} = UPDATE^{(k)}\left(h_u^{(k)}, AGGREGATE^{(k)}\left(h_v^{(k)}, \forall v \in N(u)\right)\right) \tag{2.4}$$

where $h_u^{(k+1)}$ is the updated node, UPDATE is the update step and AGGREGATE is the aggregation step, which is performed on the neighboring nodes $h_v^{(k)}$. Depending on what type of GNN is implemented the update and aggregation step may look different. The update functions typically found include: mean, max, neural networks, or recurrent NNs. For the aggregate function, there are functions that include: mean, max, normalized sum, or NNs. Some of the common GNN models are Graph convolution networks (GCNs), GraphSAGE, Graph Attention Networks (GAT), and Graph Isomorphism Networks (GINs).

2.1.9 Graph convolution networks

Graph convolution networks (GCNs) are a type of graph neural network. As mentioned in the previous section, the difference between the different GNNs is the aggregation and update functions used. Similar to the previous Equation 2.4 the function for GCN is as follows in Equation 2.5[10]:

$$H^{(l+1)} = \sigma(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}) \tag{2.5}$$

where in this equation, the normalized adjacency matrix is $\tilde{A}$ = A + I_N. I_N is the identity matrix. $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$ and $W^{(l)}$ is a layer-specific trainable weight matrix. The σ symbol denoted an activation function. $H^{(l)} \in \mathbb{R}^{NxD}$ is the matrix of activations in the layer l. Note that $H^{(0)}$ is the same as the input feature vector [10].

2.2 Related work

2.2.1 Urban Multiple Route Planning Model Using Dynamic Programming in Reinforcement Learning

This paper, written by N. Peng et al.[17] proposed a method for urban route planning using dynamic programming and reinforcement learning. This method was used to solve urban path planning problems using traffic flow information.

The authors mention different solutions to this problem, such as Dijkstra's algorithm and A* as classic path-finding algorithms that could be used. They mention that for a fully observed environment, Dijkstra's algorithm will find the optimal solution. The problem is that it cannot consider dynamic factors like traffic flow. Similarly, A* could find the optimal solution but the problem is that it is difficult to design a working heuristic function for the task. They also mention the ant colony optimization algorithm a nature-inspired meta-heuristic method and a model-free RL-based method deep Q network. They found that both methods struggle with large and complex urban environments. The method the authors propose aims to solve these issues, with both being able to handle both large and complex environments but also dynamic ones. The method that they made is a model-based RL and dynamic programming method.

They used an undirected graph to represent their road network, where the nodes in the network represented the intersections and the edges are representing the road segments. The reward function was made so that paths taken that got closer to the destination were rewarded, and the paths that got further away got penalized, to guide the agent to its destination. The reward function has a second part called the distance contribution index (DCI) which measures the distance reduction possible at the next intersection.

For their experiments, they used the road network from Shenzhen and they used travel time to evaluate the planned routes. They chose 20 origin-destination pairs as test scenarios. They compared their method to the Dijkstra algorithm, which they used as optimal results to verify the correctness of their method.

The results of the authors found that their method is consistent with the Dijkstra results and has therefore potential to find the optimal solution. However, as their methods give three routes as output their method can mitigate the congestion drift problem that is caused by only giving one result.

In the discussion, the authors highlight three areas that their method provides. The first part is the DCI, which ensures that unnecessary detours are avoided. The second part is that their method not only gives an optimal route but also two close alternatives that mitigate the congestion drift problem. Lastly, they have shown that their model can be applied to city-scale networks efficiently. They also highlighted a limitation with their method, that the method required modeling of the environment which makes it harder to implement in practice.

2.2.2 A Traffic Prediction Enabled Double Rewarded Value Iteration Network for Route Planning

This paper by J Li et al.[12] has a different approach to route planning than the last paper. Here a value iteration network (VIN) is used for route planning, LSTM network is used for predicting future traffic. Combined, these methods are used to give a route that is faster than simply taking the shortest path.

The LSTM they use is trained using the experience of taxi drivers, this is because they have in-depth knowledge of traffic trends and usually drive near-optimal routes. The architecture of their model has four parts. The first part is a pre-processing step. The data is fed into two pipelines: one is used to create a traverse time map (TTM), and the other is used to create a decision sequence. The second part is the prediction module, where the TTMs are fed to the LSTM model. In the third part, the result from the first and second parts are fed into two separate VINs, one uses the result from the prediction, and the other uses the TTM and decision sequence. The VIN with the predicted data contributes more to long-term planning, while the other one is used for short-term planning. The last part is the decision module which generates the route itself.

For the experiments, they used taxi trajectory data, requiring the data to conform to a grid map. The trajectories are from real taxi drivers from Beijing. The data used is from times between 8:00 to 22:00 on weekdays. These trajectories are then simplified to a grid-based route, where the sequence of grids is tagged with arrival and leaving time. The grid itself had the size of 20x20 and the time step used was 20 minutes.

With the model trained on the trajectories, the model's generalization ability is tested. The measurements taken are top-1 and top-2 accuracy, success rate and save time rate. They found that the model had not that great top-1 accuracy but good top-2 accuracy. This indicates that the model had learned some of the driving patterns. The success rate is the number of successful routes generated compared to the total amount.

The authors conclude that the prediction-enabled double reward values iteration network proposed can achieve human-like performance. This on the data from the taxi driver's trajectories from driving in Beijing.

3 Method

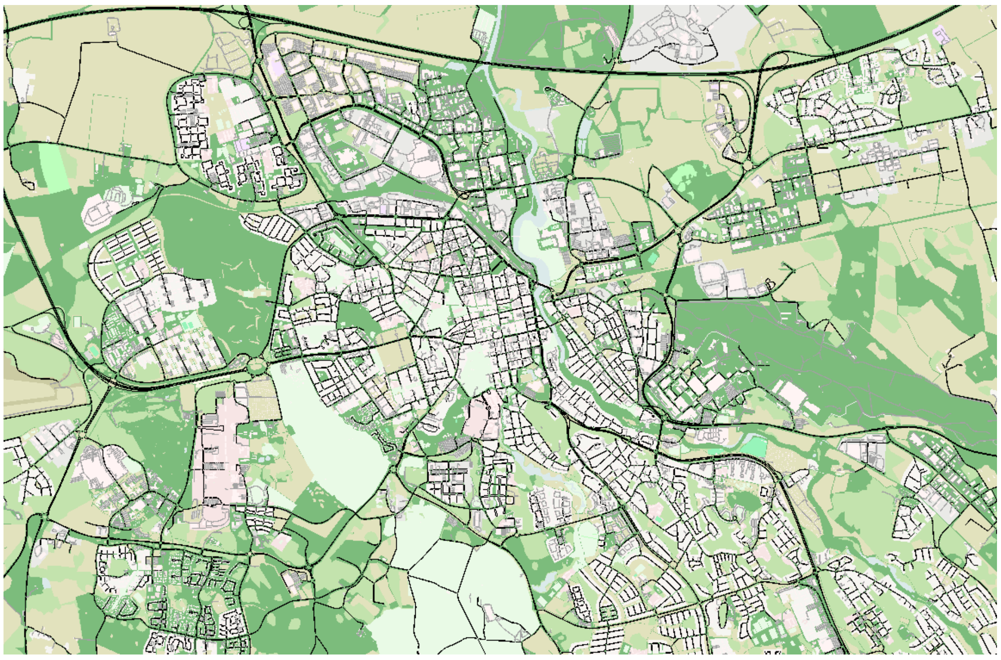
This chapter will describe the method used to answer the research questions.

3.1 Environment

In this thesis, an urban environment will simulate the digital twin's abilities. There are two ways to get an environment, one way is to create one from scratch, and the other way is to import a map from *OpenStreatMap*[16] (OSM). OSM is an open platform where a user can download sections of a real-world map. For this thesis, a map will be imported to save time in constructing the urban environment. When downloading a map from OSM, not only do roads get included, but also railways and bicycle paths. This is not of value for this thesis and will be needed to be removed to simplify the simulation. One way to do this is to use the tool OSM web wizard. This allows the user to select which type of road they want to download such as highway, pedestrian, railway and more. The user can also select what area they want to extract from the map, for example, selecting an area of a city. There are also maps that other users have already created that can be used. For this thesis, the map used will be of Linköping Sweden, because of the author's familiarity with the city and can therefore judge the accuracy of the map easier, see Figure 3.1. These maps will allow vehicles to choose different paths which are needed to give opportunity for the agent to make different choices. The files downloaded will be in the XML format for later use these maps need to be converted to work with the simulator if directly downloaded from OSM, if the wizard is used this is not needed.

3.1.1 SUMO

As mentioned in the introduction, this thesis will use SUMO to simulate the traffic environment. The maps retrieved from OSM will be used in SUMO as where the simulated vehicles will traverse. To run the simulation, a route file is required, this is where the scenario itself is specified. The scenario contains information on which time step the vehicle should start driving, and what route the vehicle should take. This is either specified as a sequence of connected edges the vehicle should go to or a flow with a start and an end node. SUMO will use a standard path-finding algorithm such as Dijkstra's algorithm or A* to find a route for the



Figur 3.1: Map of Linköping in SUMO environment.

flow during run time. Running the simulation, each time step represents one second in real life. The simulation will run until all vehicle in the scenario has arrived at their destination.

3.1.2 Generating traffic

The generation of traffic is a necessary step in simulating vehicular movement on maps lacking pre-existing traffic information. While manual creation of route files can be a viable option for simpler maps, larger maps with hundreds or thousands of simulated vehicles demand a more practical solution. SUMO software provides an automated traffic generation capability using the *randomTrips.py* script. The script offers adjustable parameters, such as the start and end time intervals (specified with the -b and -e flags and are by default 0 and 3600, respectively), where the trips are uniformly distributed. The repetition rate (-p flag and is by default 1) is also configurable, which sets the number of trips to be generated. To generate traffic for a specified network the following command can be used:

```
> python randomTrips.py -n <net-file> -e 2000 -p 0.5
```

This trip will generate 2 trips every step for 2000 steps for the specified network file. However, it is not guaranteed that a generated trip will be viable, in which case the route will be discarded. If all trips must be possible, the *–validate* flag can be used.

By default, the vehicles generated by the script are randomly and uniformly distributed across the entire network. Alternatively, the *fringe-factor* parameter can be adjusted to increase the probability that the start and end nodes of a trip will be located at the outer edges of the network.

To get a more accurate representation of the traffic, real traffic data collected by real sensors in Linköping will be compared to the generated traffic. For example, when generating traffic at a certain time on a weekday, the data from the real sensors can be used to get an expectation of the traffic amount. This can then be used to configure the generated traffic to have similar average traffic amounts.

3.1.3 Sensors

To get information about the traffic situation, sensors are used. In SUMO, there are three types of traffic sensors that can be used. These are Induction loops, Lane-area detectors, and Multi-entry-exit detectors.

3.1.3.1 Induction loop

In real life, this sensor consists of a loop of wire that is buried under the road surface, typically at intersections or other points on the road where traffic data is needed. Induction loops work by detecting changes in the magnetic field that is generated by passing vehicles. When a vehicle passes over the loop, it causes a change in the magnetic field, which is detected by the loop's sensors. There are two main types of induction loops: presence loops and count loops. Presence loops are used to detect the presence of a vehicle, while count loops are used to measure the volume of traffic. Count loops are often installed in pairs, with one loop used to measure traffic going in one direction, and the other loop used to measure traffic going in the opposite direction. The code for defining an induction loop is shown in Listing 3.1:

```
<additional>
   <inductionLoop id="<ID>" lane="<LANE_ID>" pos="<POSITION_ON_LANE>" period="<
    AGGREGATION_TIME>" file="<OUTPUT_FILE>" friendlyPos="true"/>
</additional>
```

Listing 3.1: How to define induction loop

3.1.3.2 Lane-area detector

This sensor is used to capture traffic in an area on one or more lanes. The real-life counterpart to this is a vehicle tracking camera. The difference to the induction loop in SUMO is that the lane-area detector has a length attribute. This makes this sensor more suited to measure queues of standing vehicles and is able to keep track of all vehicles currently inside the area. The code for defining a lane-area detector is shown in Listing 3.2

```
<additional>
   <laneAreaDetector id="<ID>" lanes="<LANE_ID1> <LANE_ID2> ... <LANE_IDN>" pos="<
    START_POSITION_ON_FIRST_LANE>" endPos="<END_POSITION_ON_LAST_LANE>" friendlyPos=
    "<BOOL>" period="<AGGREGATION_TIME>" file="<OUTPUT_FILE>"  timeThreshold="<FLOAT
    >" speedThreshold="<FLOAT>" jamThreshold="<FLOAT>"tl="<TRAFFIC_LIGHT_ID>"  to="<
    LANE_ID>"/>
</additional>
```

Listing 3.2: How to define Lane-area detector

3.1.3.3 Multi-entry-exit detector

Multi-entry-exit detectors are used to detect the passage of vehicles through multiple specific entry and exit points. This detector will is mainly used to measure traffic in a specific area such as intersections. Compared to the lane-area detector, the area of the multi-entry-exit detectors covers more than just one road or lane. The code for defining a multi-entry-exit detector is shown in Listing 3.3

```
<additional>
   <entryExitDetector id="<ID>" period="<AGGREGATION_TIME>" file="<OUTPUT_XMLFILE>"
   timeThreshold="<FLOAT>" speedThreshold="<FLOAT>">
      <detEntry lane="<LANE_ID1>" pos="<POSITION_ON_LANE>" friendlyPos="<BOOL>"/>
      <detEntry lane="<LANE_ID2>" pos="<POSITION_ON_LANE>" friendlyPos="<BOOL>"/>
      <detExit lane="<LANE_ID1>" pos="<POSITION_ON_LANE>" friendlyPos="<BOOL>"/>
      <detExit lane="<LANE_ID3>" pos="<POSITION_ON_LANE>" friendlyPos="<BOOL>"/>

      ... further entries ...
```

```

  </entryExitDetector>
</additional>
```

Listing 3.3: How to define Multi-entry-exit detector

3.1.3.4 Usage

Using sensors in SUMO is done firstly by defining them, this is done by defining the sensors in an additional file, which is separate from the main map file. To define a sensor in SUMO, the user can either manually create the sensor file as seen in listings: 3.1 3.2 3.3, or use the NETEDIT GUI provided by SUMO. In the NETEDIT GUI, the user can specify the sensor type, location, and parameters. Once the sensors have been placed on the map, they are saved to a separate file from the main map file. To use the sensors when running the simulation, the sensor file needs to be included in the SUMO configuration file. Once the configuration file is set up, the user can run the simulation and collect data from the sensors. When creating the sensors the user can also specify if the sensor data should be saved in a data file after the simulation. For each of the sensors defined, a unique file will be created to store the information that is captured.

3.1.4 Communication with SUMO

Communication with SUMO from code is done through the Traffic Control Interface (TraCI). TraCI is an interface in SUMO that allows external programs to interact with a running SUMO simulation in real-time. TraCI provides a standardized communication protocol between SUMO and external programs, which enables the exchange of data such as vehicle positions, speeds, routes, and traffic light states. This is how the digital twin will communicate with 'the real world' and see how different actions actually affect the performance.

The code below shows a basic setup for TraCI using Python 3.4. The code imports necessary libraries and checks for the *SUMO_HOME* environment variable. It defines a function to parse command-line options and a function to run the simulation using the TraCI library. If the *–nogui* option is specified, it runs the simulation without a GUI. Otherwise, it runs with a GUI. Finally, the code starts the simulation using the sumoBinary and runs the TraCI control loop. The control loop is where information from the simulation can be gathered, and modification to it can be made. The following parts of TraCI are of importance for this thesis:

- Vehicle value retrieval: From this information such as time traveled, average speed and more can be gotten from each vehicle in the simulation.
- Sensor value retrieval: From this information from sensors can be collected in real time.
- Router: This is used to create new routes that vehicles in the simulation can use.

```
import os
import sys
import optparse
import traci
from sumolib import checkBinary

if 'SUMO_HOME' in os.environ:
    tools = os.path.join(os.environ['SUMO_HOME'], 'tools')
    sys.path.append(tools)
else:
    sys.exit("please declare environment variable 'SUMO_HOME'")


def get_options():
    opt_parser = optparse.OptionParser()
```

```python
    opt_parser.add_option("--nogui", action="store_true", default=False, help="run
    the commandline version of sumo")
    options, args = opt_parser.parse_args()
    return options


# contains the traci control loop
def run():
    while traci.simulation.getMinExpectedNumber() > 0:
        traci.simulationStep()
        # Get simulation information using traci from here

    traci.close()
    sys.stdout.flush()


if __name__ == "__main__":
    options = get_options()

    if options.nogui:
        sumoBinary = checkBinary("sumo")
    else:
        sumoBinary = checkBinary("sumo-gui")

    # traci starts sumo as a subprocess and then this script connects and runs
    traci.start([sumoBinary, "-c", "config.sumocfg", "--tripinfo-output", "tripinfo.
    xml"])
    run()
```

Listing 3.4: Basic traci setup using Python

3.2 Data

In this thesis, data is used to make predictions on traffic. To get this data, the SUMO simulator is used to simulate different traffic scenarios. For making predictions, sensor data needs to be gathered to create a picture of how the traffic situation looks at different parts of the map and how those parts will look in the future. This data is the traffic flow at different junctions which will later be used to train a neural network.

3.2.1 Graph representation

Representing road networks as graphs is common in traffic prediction [7, 17]. The nodes and edges in the graph of the road network can be defined as follows:

- Nodes: Each node in the graph represents an intersection or endpoint of a road segment. For example, if there is an intersection of three roads, then there will be a node at that intersection. Nodes can also represent the start and end points of a road segment, even if there are no intersections.

- Edges: Each edge in the graph represents a road segment connecting two nodes. The weight of the edge can represent the distance between the two nodes or the travel time required to traverse that segment. The edges can be directed or undirected, depending on whether the roads are one-way or two-way.

The nodes and edges also need to have information on which edges are connected to which nodes and vice-versa. This can be done using an adjacency matrix, but this matrix will become very large with the number of nodes and the sparsity as most nodes are only connected to a handful of other nodes. Instead, an adjacency list will be used.

Tabell 3.1: Example of how sensor data matrix could look like.

sensor	t=x	t=x+1	t=x+2	t=x+3	...
1:	0	1	2	3	...
2:	1	1	0	1	...
3:	0	2	3	4	...
4:	3	1	2	0	...
5:	2	4	3	2	...
...	...	...	...	...	...

3.2.2 Creating graph from SUMO map

From the graph, representation discussed, a graph based on the SUMO map needs to be created for use both in path-fining and for training of neural networks. The map itself has all the information that is needed, all that is needed is to convert it to a more manageable format. Reading the map file requires a sumo function to convert it to a Python object. From the map object, all the information for building the graph is found on the edges. The important parts that are used are the from and to node ids, this is used to build the adjacency list. The edge also has useful information, such as its own id and length. The id of the edge is important as routes in SUMO are built from edge to edge and not node to node. With this information, a graph representation can be made. In a Python dictionary, the node id can be used as keys and contain a list of all connected edges to that node.

3.2.3 Gather sensor data

The sensors that are added also need to be a part of the graph, this is so that the traffic observed by the sensor can be pinpointed to the correct place in the graph. When the location of the sensor in the graph is clear, data can be gathered. As mentioned before, sensor data is saved to a file, and for each sensor, there is a file with data. This data can also be retrieved in real-time from TraCI, using the id for each sensor. Translating each set of sensor to a corresponding node in the graph is needed to be able to change the node costs. The nodes with sensors on them can be saved into a list that contains the amount of traffic going through them. Saving information every time step will not create useful data, instead, data needs to be collected in time intervals. The time interval used for this thesis is 60-time steps as it's a good balance between collecting data not too frequently and being able to get a sizeable data set in a reasonable amount of time (To collect 10 000 data points, 600 000 simulation steps are needed which translates to around 167 hours of simulation). On the data, a discretization step is also performed to reduce the state space. This is from the number of cars past through the node to a scale of different traffic volumes in that node. The scale goes in five steps from low, low medium, medium, high, and very high represented as numbers from 0 to 4. The data saved will be a matrix where the number of rows will be the number of sensors and the rows will be the number of data points collected as seen in Table 3.1.

3.3 Traffic prediction

Improving the routes generated, information on how traffic in different parts of the map will look in the future could improve the routes generated. As we only know what the traffic situation looks like at the moment, we train neural networks to predict future traffic situations. The previous section described how data is collected, in this section that data is used for training and testing of the neural network. The model that is to be created is a GNN model. To implement these models the framework PyTorch and PyTorch Geometric is used respectively.

3.3.1 GNN

This section will describe how the graph neural network traffic prediction was implemented, how the data set was constructed, how the model is structured, and lastly how the model is trained.

3.3.1.1 Data set

The raw data from the sensors need to be converted to a data set and processed so that it can be used to train the model. To make it easier to handle, the data is converted from a matrix to a data frame. The data frame has four columns:

- Node: What node it is.
- Time: What time step the data was collected
- Traffic: What the traffic situation looks like right now at the node
- Future: What the traffic will look like in the future at the node, our gold standard values

As the memory size of the data itself is not too large, an in-memory data set can be used. The data need to be made to fit into a Data object, and this object requires x, y, and edge indices which correspond to the node features, target values, and edge connections. The node features and target values are created by going over each data group, where the data is grouped by time. Going through each group, the node features are made by extracting the traffic features with the same time value. These are then converted into a tensor. The target values are the gold standard values and are also put into a tensor. Lastly, the edge indices are the adjacency list for the nodes in the graph, which are also put into a tensor. A torch data object is created and put into a list for each group. The last step when each group has been converted is to collate the data and save the data set into to disk.

3.3.1.2 Model

Implementing the GNN model is done using the Python library torch and torch geometric. The type of GNN that is implemented in this thesis is a Graph convolution network (GCN). GCNs use GCN layers which are equivalent to the message-passing operation. The initial convolutional layer has in-channels equaling the number of features and the out-channel has equaling the embedding size. The embedding size is not strictly defined and is chosen based on the implementation and the data set used, but 64 can be used as a starting point. To increase the depth of the network multiple convolutional layers are stacked after the initial convolutional layer, as many as needed, the only difference now is that the in-channel also has the size of embedding size. The results from each convolutional layer are also passed through a ReLU activation function. Before the output layer, the results are passed through a global pooling layer. The final part of the model is to pass the result from the pooling layer through a linear classifier. The results from the classifier is the prediction made. The code below shows an implementation of the described GCN model is found in the Listing 3.5.

```python
import torch
from torch.nn import Linear
from torch_geometric.nn import GCNConv
from torch_geometric.nn import global_mean_pool as gap, global_max_pool as gmp
from GNN.data import SensorDataset
embedding_size = 64

dataset = SensorDataset(path)

class GCN(torch.nn.Module):
    def __init__(self):
        # Init parent
        super(GCN, self).__init__()

        # GCN layers
        self.initial_conv = GCNConv(dataset.num_features, embedding_size)
        self.conv1 = GCNConv(embedding_size, embedding_size)
        self.conv2 = GCNConv(embedding_size, embedding_size)
        self.conv3 = GCNConv(embedding_size, embedding_size)

        # Output layer
        self.out = Linear(embedding_size * 2, dataset.num_classes)

    def forward(self, x, edge_index, batch_index):
        # First Conv layer
        hidden = self.initial_conv(x, edge_index)
        hidden = torch.relu(hidden)

        # Other Conv layers
        hidden = self.conv1(hidden, edge_index)
        hidden = torch.relu(hidden)
        hidden = self.conv2(hidden, edge_index)
        hidden = torch.relu(hidden)
        hidden = self.conv3(hidden, edge_index)
        hidden = torch.relu(hidden)

        # Global Pooling (stack different aggregations)
        hidden = torch.cat([gmp(hidden, batch_index),
                            gap(hidden, batch_index)], dim=1)

        # Apply a final (linear) classifier.
        out = self.out(hidden)

        return out, hidden
```

Listing 3.5: Code for a graph convolution network implementation

3.3.1.3 Training

With the data set created, the model can now be trained. First, the data is split into training and testing data. The data is put into two different data loader objects, which will shuffle the data and put the data into mini-batches. The Adam optimizer is used with an appropriate learning rate to train the model. Using the data loader,w each batch can be iterated through. Each iteration will pass the node features and the connections to the model. With the prediction received from the model, the loss can be calculated. The loss is calculated using the mean square error (MSE). Finally, the model is updated using the gradients.

The training is done in several epochs. The number of epochs is determined by early stopping. Early stopping is performed by calculating the MSE on the test data. This is done after a set amount of epochs with a constant interval. The model is determined to have finished training when the MSE is no longer decreasing on the test data.

Lastly, after the model has finished training, the weights and biases of the model are saved to disk for later use.

3.4 Digital twin

A digital twin is used to test the effects of the sensors and the model. To do this, three different types of digital twins will be compared. A baseline digital twin, a twin that will have access to sensor information about the current traffic situation, and a twin that will also use the GNN model to predict future traffic.

3.4.1 Baseline

The baseline digital twin has no information regarding the current or future traffic. The only information it has is the graph, which is used where Dijkstra's algorithm is used to find the quickest path from the origin to the destination. The edge cost between nodes is calculated using the road's length divided by the speed limit to get the fastest path.

3.4.2 Digital twin with sensors

Unlike the baseline, this digital twin will also have information about the current traffic situation. This will change the edge cost function. When calculating the cost of the edge, check if one or both of the nodes has a sensor in them; if they do, check the traffic situation in the node. From that information, calculate a multiplier for the cost; if the traffic is significant, then the multiplier should increase, and if it's low, the multiplier should decrease. The multiplier was calculated according to the Equation: 3.1. Where d is the discretization function, and v is the traffic flow volume.

$$m = (d(v) + 1)/2 \tag{3.1}$$

3.4.3 Digital twin with prediction

This version of the digital twin is similar to the previous one, with a difference in multiplier calculations. Before that, the GNN model is imported, and the saved weights and biases are applied as the model requires three inputs, the node data, the adjacency list, and the batch indices. Both the node data and adjacency list use the same pre-processing steps as when constructing the dataset, though there are no y values this time. The batch index is simply a list of zeroes with the same length as the number of sensors used. Putting this into the model will return the traffic prediction for the next minute. To get further predictions, use the output of the model as input. Repeat for how many minutes in the future are wanted to predict. As the output is discretized, that step can be skipped in further iterations. The predictions are saved into a list where the index corresponds to how many minutes the prediction is for in the future. Index zero will be the sensor values and the other ones will be predictions of those values.

Calculating the edge cost now has some differences. When calculating the multiplier, the traffic volume is dependent on time. As the cost function measures the expected time to get to a node, that time can be used to index the prediction list. This is by dividing the expected time by 60 and rounding it to the nearest whole number. This can then be used to index the correct sensor values. The updated equation is as follows: 3.2.

$$m = (d(v[t]) + 1)/2 \tag{3.2}$$

3.5 Testing and evaluation of the digital twins

This section will describe how the testing and evaluation of the digital twin will be performed.

3.5.1 Test scenarios

A testing scenario will be created to test the digital twin's ability to find routes for the vehicle. This scenario is created the same way as in the generating traffic section. The scenario created will be used to evaluate the digital twins described in the previous section. To do this, 20 different origin-destination node pairs will be randomly generated for the map. These origin-destination pairs will then be sent to the different digital twins to create a route for them. The simulator will use the routes created to drive these routes.

The test scenarios will also include testing different configurations of sensor amounts. These include three different sensor amounts. Each digital twin that uses sensors is tried for three different configurations. This means that the 20 other origin-destination node pairs are tested seven times. One for the baseline, three for the configuration with sensor data, and three for the prediction configuration.

3.5.2 Evaluation

While running the test scenario, multiple metrics of the vehicle's performance will be captured using TraCI. The following metrics of the vehicle:

- Time to reach the destination
- Distance traveled
- Co2 emission
- Time stopped
- Average speed

To measure the time taken for the vehicle to arrive at its destination, save the time step that the vehicle started and arrived, then subtract the start time from the arrival time. Time is measured in seconds. For the metrics Distance, Co2 emission, and time stopped TraCI is used to get the information from the vehicle. This is because each vehicle in the simulation saves some information about itself. The total distance traveled and Co2 released are updated each time step. Though the Co2 released is only for the last time step, the total needs to be kept track of. The total distance traveled is measured in meters, and Co2 is measured in grams. The car itself does not save how much time it spends not moving. To get this information, manually check if the speed is zero each time step, and count how many times steps this was true. The time stopped is measured in seconds. Lastly, the average speed is calculated by dividing distance by time and is measured in meters per second.

4 Results

This chapter will present the results of the experiments described in the methods chapter. This chapter is structured in the following way; first, the Data gathered from sensors is given, then the models trained, and finally, the performance of the different digital twins is shown.

4.1 Environment and Data

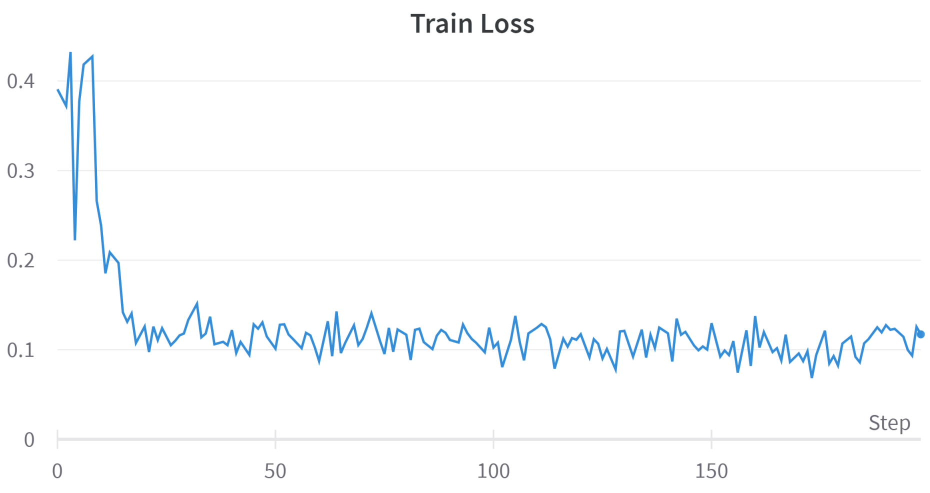
The resulting environment used was a map of Linköping, as shown in the theory chapter. Three sensor graphs were produced; the node and edge count are in table 4.1. Using these sensor setups, data from the simulation was gathered in 10 unique epochs of 24-hour simulations, each with a unique traffic scenario generated. This gave 720 000 sensor data values at 14 400 different time steps. Graph 1 used all the values to create a dataset from these values, while Graphs 2 and 3 used a sub-set of these values. With these data values, three data sets were constructed.

4.2 Models

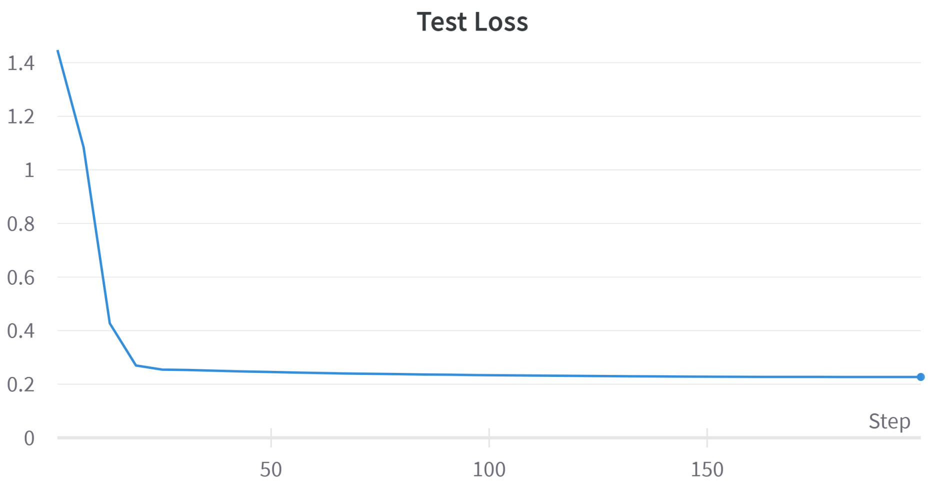
The three models were trained on the created datasets. The graph's models used the same count of convolutional layers at four and the same embedding size of 128. In training the first GNN, a learning rate of 0.000005 was used, and training took 198 epochs with a final training loss of 0.1172 and a test loss of 0.227 before early stopping, graphs showing training and test loss are found at Table 4.1 and 4.2. In the second GNN, the same learning rate was used; this model had a final training loss of 0.065 found in Table 4.3 and a test loss of 0.206 found in Table 4.4, and the training took 24 epochs. For the last GNN, a learning rate of 0.000001 was

Tabell 4.1: The different node and edge count of the graphs

Graphs	1	2	3
Node count	50	25	10
Edge count	77	38	15



Figur 4.1: Graph showing the train loss over time.



Figur 4.2: Graph showing the test loss over time.

used. This GNN model had a final training loss of 0.206 graph found in Table 4.5 and a test loss of 0.423 found in Table 4.6 and took 109 epochs to train.

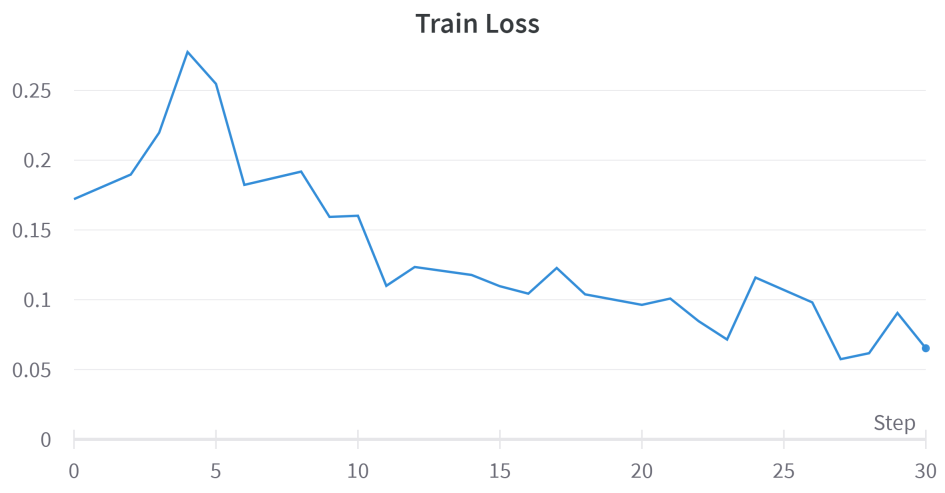
4.3 Digital twin

In this section, the results of the seven different digital twins are presented, and their results are compared to the baseline digital twin. The values in the graphs are on a scale of percentages where a value of 100% represents a value equal to the baseline. The graphs are segmented into 20 different tests; the tests represent the 20 origin-destination pairs used.

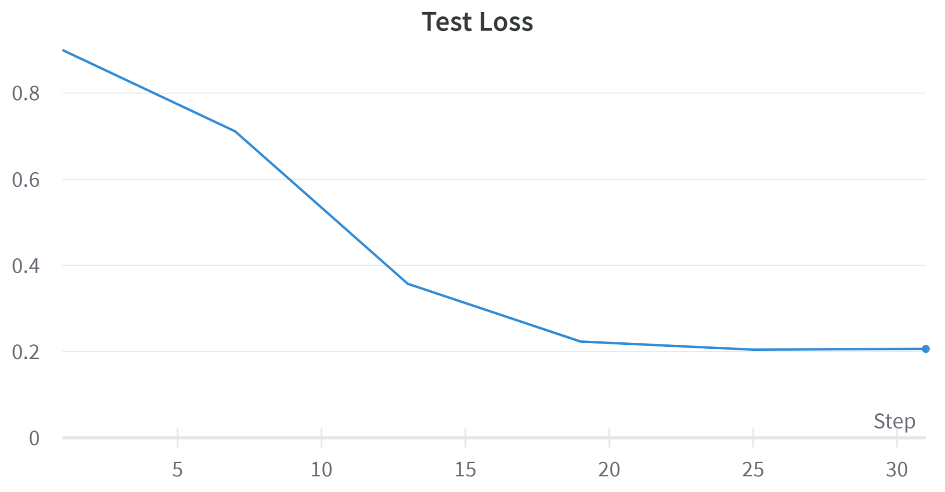
4.3.1 Time

4.3.1.1 No prediction

The graph in Figure 4.7a shows the result of the time taken for the 20 tests for the three models without prediction compared to the baseline. With the model that used 50 sensors,



Figur 4.3: Graph showing the train loss over time.

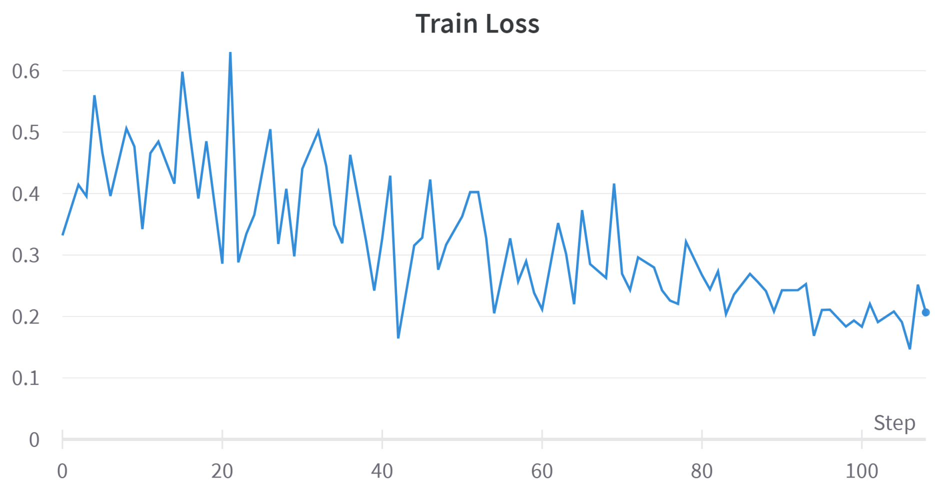


Figur 4.4: Graph showing the test loss over time.

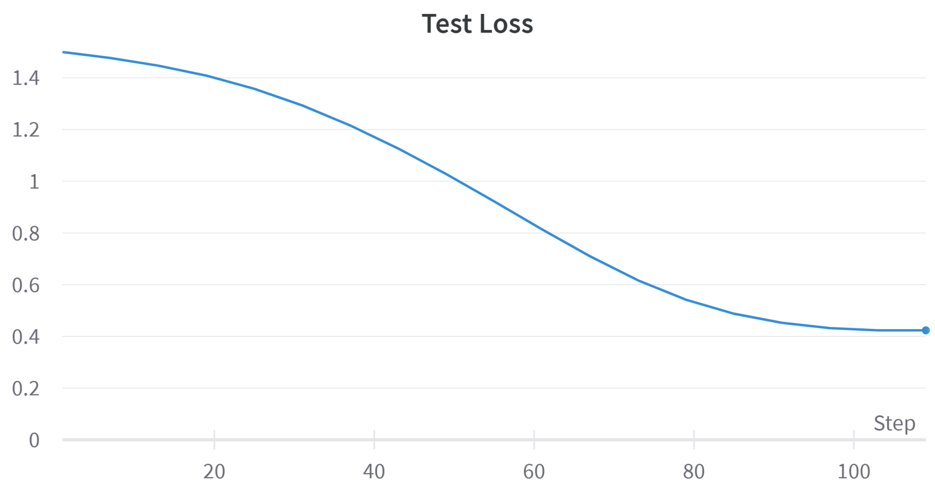
the average time to arrive at the destination increased by 0.81%, with Tests 1,6,7,9,16,17,18, and 19 showing a worse time than the baseline. While tests 5,10,11,13,14, and 20 showed a better time than the baseline. The remaining tests shows a performance equal to the baseline. For the model with 25 sensors, the average time to arrive at the destination was slightly worse at 0.84% longer to reach the goal. The tests that performed worse were 1,7,9,16,17,18, and 19. The tests that performed better than the baseline were 10,11,13 and 20. On average, the model with ten sensors reached its destination 0.6% faster than the baseline. These tests were 11 and 13 and no tests performed worse than the baseline, only better.

4.3.1.2 With prediction

In figure 4.7b, the results of the time taken for the three models using prediction are shown. For the digital twin with 50 sensors, The average time taken was 0.65% longer than compared to the baseline. The tests that performed worse were 1,3,4,6,7,9,16,17 and 20. The tests that performed better were 5,10,11,13,14 and 19. The remaining tests performed were equal to the baseline. For the digital twin with 25 sensors, the time to reach the destination was 0.41%



Figur 4.5: Graph showing the train loss over time.



Figur 4.6: Graph showing the test loss over time.

longer than the baseline. Tests 1,3,4,6,9,16,17, and 20 performed worse, while tests 10,11,13 and 19 performed better. Lastly, for the digital twin with ten sensors, the average time to reach the destination was 1.08% faster compared to the baseline. Tests 3 and 17 performed worse; tests 7,10,11, and 13 performed better.

4.3.2 Co2

4.3.2.1 No prediction

Figure 4.8a displays the Co2 emission during the tests of three digital twins without prediction compared to the baseline. In the digital twin with 50 sensors, the average emission increased by 1.23% compared to the baseline. The tests that saw increased emissions were 1,6,8,16,17,18,19, and 20. The tests that saw reduced emissions were: 5,7,9,10,11, and 14. With 25 sensors, the average co2 emissions increased by 0.34%. The tests that performed worse than the baseline were: 1,13,16,17,18,19, and 20. Tests 7,9,10, and 11 showed a reduction in Co2 emissions. The digital twin using ten sensors saw a decline in emissions by 0.69% compared to the baseline. Test 13 performed worse, while test 11 performed better. The remaining tests performed precisely the same as the baseline.

4.3.2.2 With prediction

Figure 4.8b shows the Co2 emissions from the digital twins using prediction. Using 50 sensors, the average Co2 emission increased by 1.94% compared to the baseline. The tests that had worse performance were: 1,4,6,7,13,16,17, and 20. Tests 3,5,9,10,11,14, and 19 had lower emissions than the baseline. The digital twin, with 25 sensors, saw an increase in emissions by 1.65% compared to the baseline. The tests with higher emissions than the baseline were: 1,4,6,13,16,17, and 20. Tests with lower emissions were: 3,9,10,11, and 19. The digital twin with ten sensors had 1.00% lower Co2 emissions on average compared to the baseline. This was the tests 13 and 17 had higher emissions, and tests 3,7,10, and 11 had lower emissions. The remaining tests saw equal Co2 emissions compared to the baseline.

4.3.3 Distance

4.3.3.1 No prediction

In figure 4.9a, the distance of the generated routes is compared to the routes generated by the baseline without prediction. On average, the digital twin with 50 sensors had a distance 2.80% longer than the baseline. The routes that were longer came from tests 1,6,9,10,13,16,17,19, and 20. Tests 5,7,11, and 14 had shorter routes and the digital twin with 25 sensors had, on average 1.20% longer routes. The tests that had longer routes were 1,9,10,13,16,17,19, and 20. Tests with shorter routes were 7,11, and 18. Lastly, using ten sensors gave an average distance of 0.40% longer. The test with a longer route was test 13, and test 11 had a shorter route. All other routes were of the same length as the baseline.

4.3.3.2 With prediction

Figure 4.9b shows the distance of the generated routes with the digital twins that used prediction. The average length of the routes generated for the digital twin with 50 sensors was 3.78% longer. The tests with longer routes were 1,4,6,7,9,10,13,16,17, and 20. Tests with shorter routes were 3,5,11,14, and 19. Using 25 sensors, the average length of the routes increased by 3.61%. The tests with longer routes than the baseline were 1,4,6,9,10,13,16,17, and 20. The tests that had shorter routes were 3,11, and 19. On average, the digital twin with ten sensors had 1.05% longer than the baseline. Tests 10,13, and 17 had longer routes, and tests 3,7, and 11 had shorter ones.

4.3.4 Time stopped

4.3.4.1 No prediction

Figure 4.10a shows the time the vehicle spent stationary during the route compared to the baseline. For the digital twin using 50 sensors, the average time stopped increased by 49.12% with the tests 1,17, and 18 seeing an increase in time waited. Tests 5,6,7,8,10,11,13,16,19, and 20 saw a decrease. The digital twin with 25 sensors saw an average time stopped rise of 55.67%. The tests that saw an increase in wait time for 50 sensors also increased for 25. Tests that saw a decrease in time stopped were 7,10,11,13,16,19, and 20. For the digital twin with ten sensors, the average time stopped decreased by 4.66%. This, with no tests having an increased wait time, tests 11 and 13 saw a decrease in wait time.

4.3.4.2 With prediction

Figure 4.10b shows the average time stopped compared to the baseline using prediction. The digital twin with 50 sensors had an average time stopped the increase of 38.40%. Tests that saw a higher wait time than the baseline were 16 and 17. The tests that saw a decrease were 3,4,5,6,10,11,13,19, and 20. For the digital twin with 25 sensors, the average time stopped increased by 41.67%, and tests 16 and 17 had an increase in wait time, while tests 3,4,6,10,11,13,19, and 20 saw a decrease. With ten sensors, the digital twin saw a rise in wait times by 41.42% on average. Test 17 was the only test that saw an increase in time stopped. Tests 3,7,10,11, and 13 had a decrease.

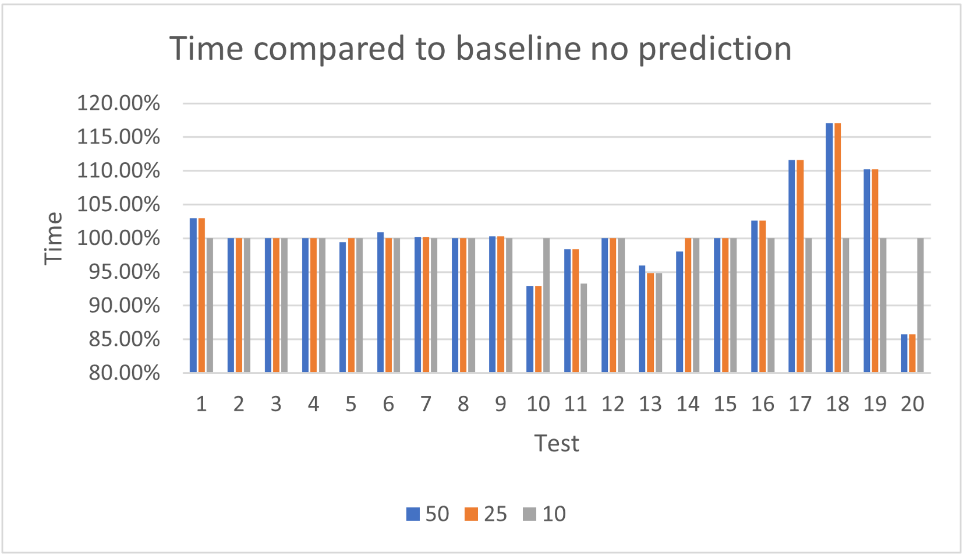
4.3.5 Average speed

4.3.5.1 No prediction

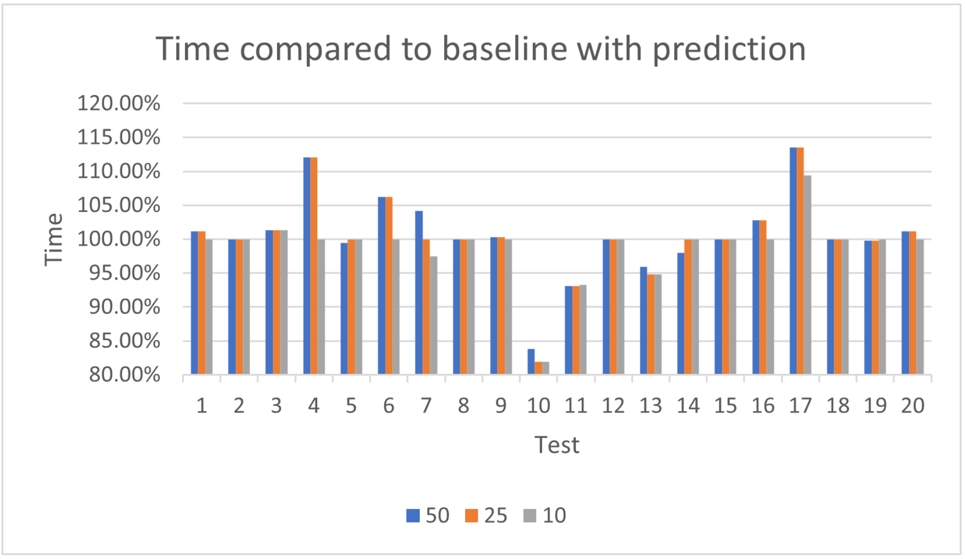
Figure 4.11a shows the average speed compared to the baseline. The digital twin with 50 sensors had an average speed of 2.51% faster. The tests with higher average speeds were 6,10,13,14,16, and 20. The tests that with slower speeds were 1,7,9,11,17,18, and 19. With 25 sensors, there was an increase of 0.91% on average. The tests with higher speeds were 10,13, and 20, and the tests with slower speeds were the same as those for 50 sensors. For the digital twin with ten sensors, the average speed increased by 1.04% compared to the baseline. Test 13 had a higher average speed, and test 11 was slower.

4.3.5.2 With prediction

Figure 4.11b shows the average speed of the digital twin using prediction compared to the baseline. The digital twin with 50 sensors had an average speed of 3.51% faster. The tests with higher speeds were 5,6,10,13,14,19, and 20. Tests 1,3,4,7,9,11,16, and 17 were, on average slower than the baseline. Using 25 sensors, the average speed increased by 3.66% with tests 6,10,13,19, and 20 faster than baseline, and tests 1,3,4,9,11,16 were slower. Lastly, the average speed for ten sensors increased by 2.62% compared to the baseline. The tests with higher average speeds were 10 and 13, with tests 3,7,11, and 17 slower.

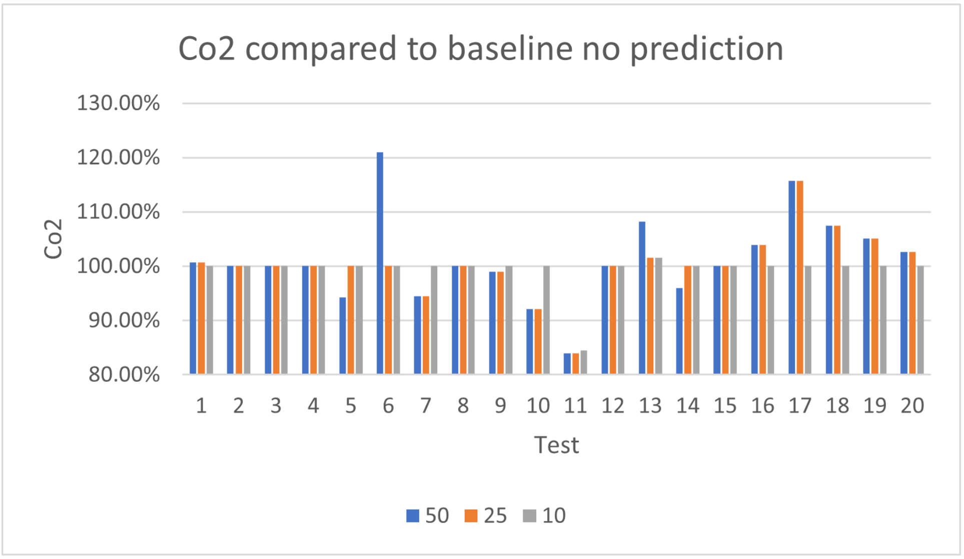


(a) Graph showing the time taken to drive from the start position to its destination compared with the baseline.

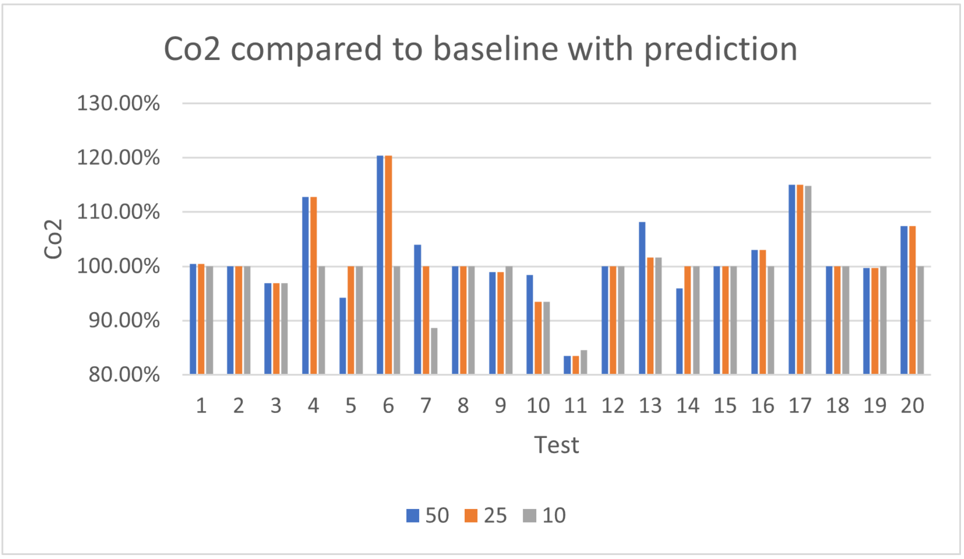


(b) Graph showing the time taken to drive from the start position to its destination compared with the baseline.

Figur 4.7: Shows the graphs for the time taken to reach the destination for each test.

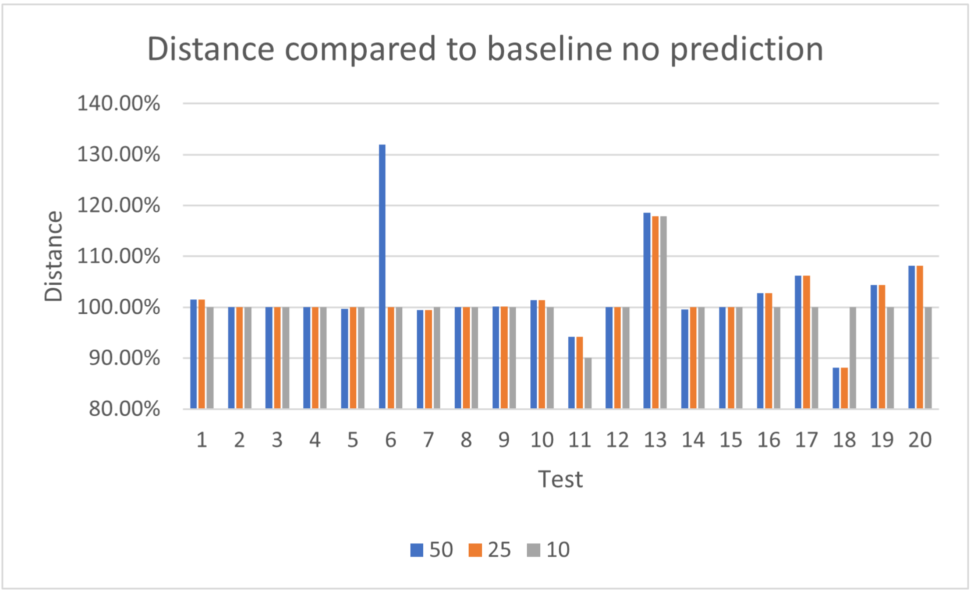


(a) Graph showing the Co2 emissions from each test scenario compared with the baseline.

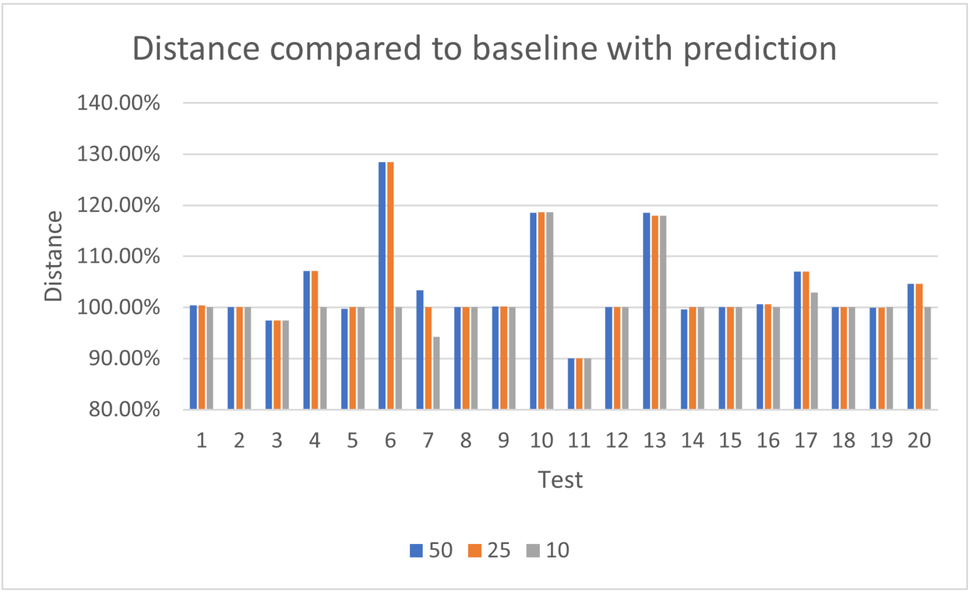


(b) Graph showing the Co2 emissions from each test scenario compared with the baseline.

Figur 4.8: Shows the graphs for the Co2 emissions for each test.

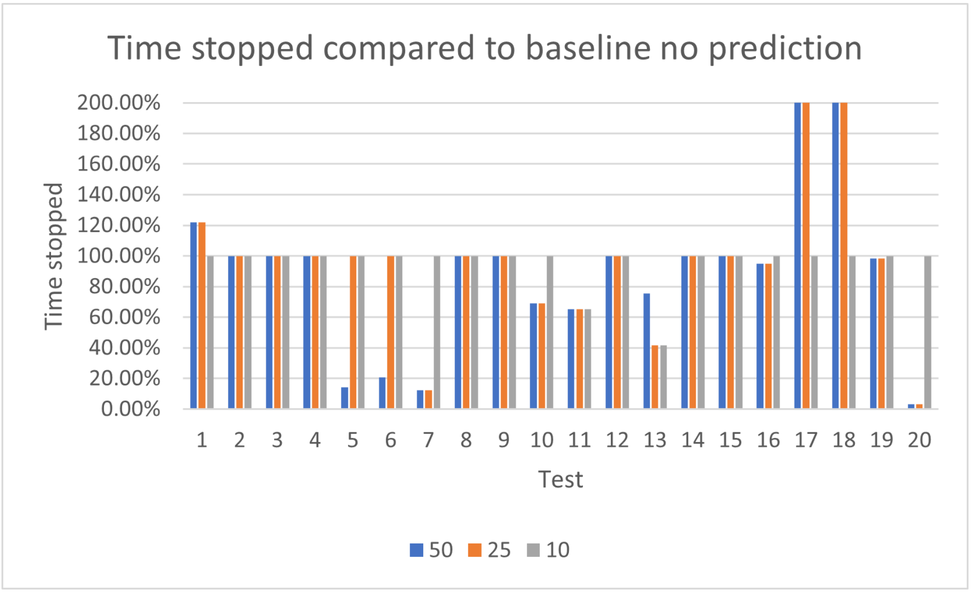


(a) Graph showing the distance traveled compared with the baseline.

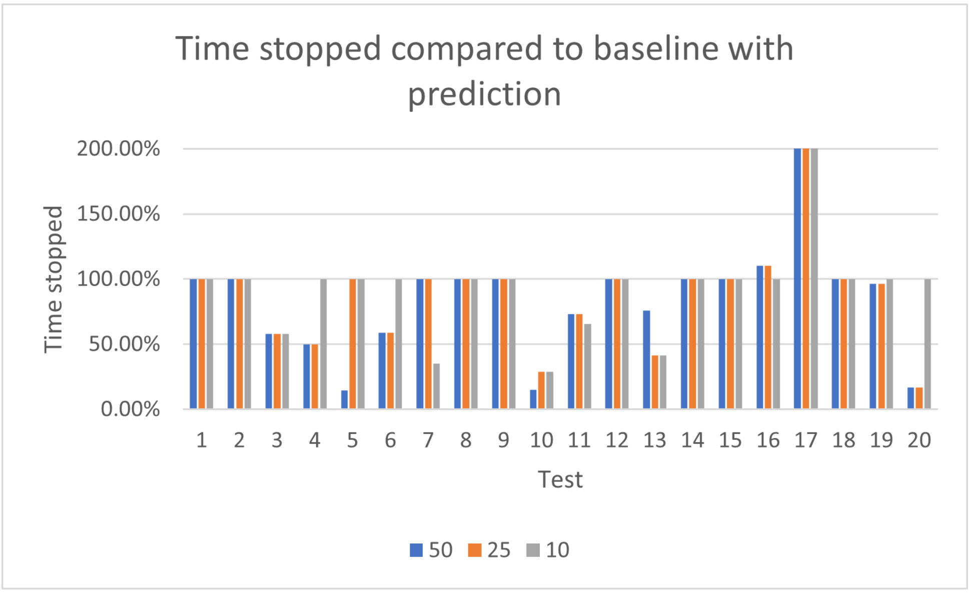


(b) Graph showing the distance traveled compared with the baseline.

Figur 4.9: Shows the graphs for the distance traveled for each test.

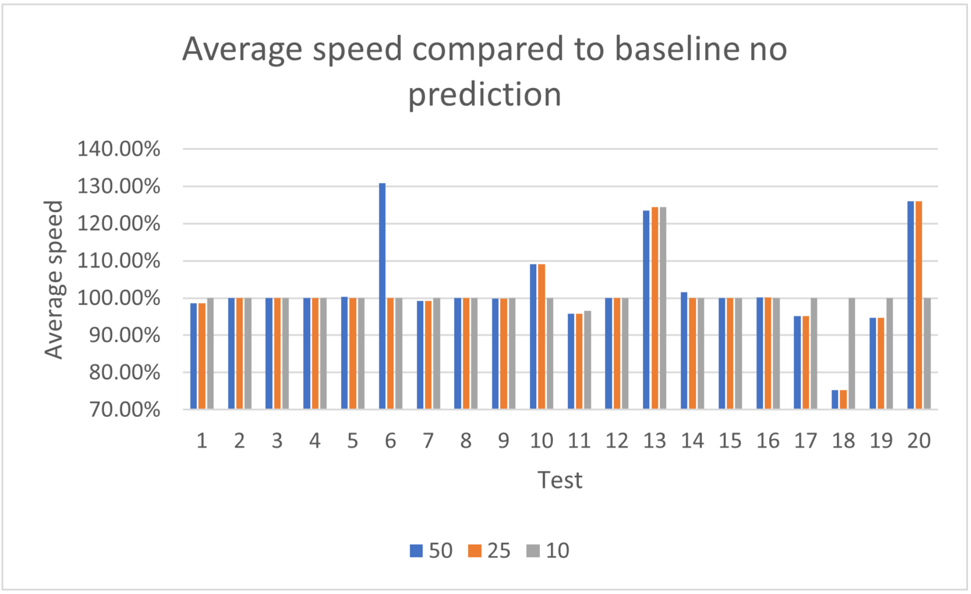


(a) Graph showing the time stopped compared with the baseline.

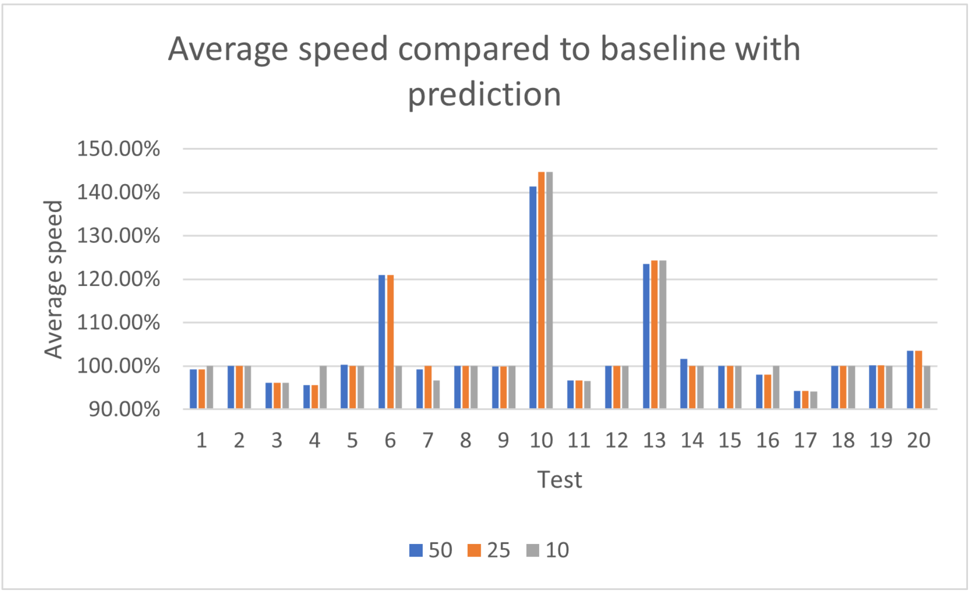


(b) Graph showing the time stopped compared with the baseline.

Figur 4.10: Shows the graphs for the time stopped for each test.



(a) Graph showing the average speed during the test scenario compared with the baseline.



(b) Graph showing the average speed during the test scenario compared with the baseline.

Figur 4.11: Shows the graphs for the average speed for each test.

5 Discussion

5.1 Results

The results will be discussed in the following way. First, the results of the digital twins will be analyzed from the metrics point of view, and then Specific test scenarios will be looked at.

5.1.1 Metrics

From the results of the time taken to reach the destination, only the digital twins that used ten sensors saw a performance improvement when comparing each test against the other. However, this does not tell the entire story. For example, the average time to reach its destination for the baseline was 594.85s. Comparing this to the average time for the other digital twins found in Table 5.1 and 5.2. In the first table, similar to before, the digital twins with 50 and 25 sensors perform worse, and the one with ten performs better. The exciting part is found in table 5.2. Here its shows that all digital twins outperform the baseline when comparing the average time. Though the digital twin with ten sensors still shows the most significant time saved, it's surprising that the other twins also outperform the baseline when looking at the overall time taken but not when comparing each test. This could be because when the digital twin creates a worse route, the route is usually shorter, and when it is predicted well, the path is more prolonged. While all the digital twins using prediction were found to be better, it was not by much. With 50, 25, and ten sensors, the average time saved was 1.45s, 3.6s, and 11.9s, respectively. In the best-case scenario, using the digital twin with prediction should save an average time of around 1-2% per trip. This is not that much for each trip, but this could add up to much time saved over many trips.

Looking at the metric Co2 emissions, a similar observation can be made regarding the performance of the different digital twins. The twins using 50 and 25 sensors have, on average,

Tabell 5.1: The average time in seconds for the different digital twins that did not use prediction to reach their destination.

Digital twins	50 sensors	25 sensors	10 sensors
Average time	597.2	597.1	590.75

Tabell 5.2: The average time in seconds for the different digital twins that used prediction to reach their destination.

Digital twins	50 sensors	25 sensors	10 sensors
Average time	593.4	591.25	582.95

Tabell 5.3: The average Co2 emission in grams for the different digital twins that did not use prediction to reach their destination.

Digital twins	50 sensors	25 sensors	10 sensors
Average Co2 emission	1617.75	1605.27	1599.33

Tabell 5.4: The average Co2 emission in grams for the different digital twins that used prediction to reach their destination.

Digital twins	50 sensors	25 sensors	10 sensors
Average Co2 emission	1633.54	1621.15	1585.19

a bit higher emissions than the baseline, and the twins using ten sensors saw a small improvement. This is true regardless if the twin used prediction or not, though, the twins that did not perform prediction did a small amount better than their predicting counterparts. Again looking at the overall Co2 emission, it's not entirely the same. The average Co2 emission for the baseline was 1609.74 grams per trip. The tables 5.3 and 5.4, the average Co2 emission for the other digital twins is shown. For the twins not using prediction, only the one using 50 sensors performed worse than the baseline, and the others had fewer emissions. For the ones using prediction, the one with 50 and the one with 25 sensors still had higher emissions than the baseline. This means that the time saved comes at the cost of higher Co2 emissions for those digital twins. For the one using ten sensors, the digital twin, on average, saves time on the trip and at an emission reduction.

For all digital twins, the length of the routes was longer than the baseline, when comparing the routes individually, as seen in the results. Similarly, looking at the averages of the distance traveled for each digital twin seen in Tables 5.5 and 5.6, The average distance for the baseline was 5882 meters per test scenario, lower than any of the other average distances found for the other digital twins. One of the reasons for this outcome could be the use of the cost function. The cost function of the Dijkstra method included distance as one of its metrics, and it also had speed. For two roads with equal speed, the first road checked should also be the shortest, in combination with similar speed limits inside the city might lead the baseline to be more likely to choose shorter paths than the other digital twins. But as seen from 4.9a and 4.9b, the distance was not always shorter.

When looking at the metric time stopped from the results, most digital twins performed poorly compared to the baseline. The one that, from the results, performed better was the one with ten sensors and no prediction, which had to wait nearly 5% less than the baseline. The other ones were from 40 to 50% worse comparatively. When looking at tests, two outliers skewed the results. Mainly test 17 but also test 18 for some of the twins. For test 17, most of

Tabell 5.5: The average distance traveled in meters for the different digital twins that did not use prediction to reach their destination.

Digital twins	50 sensors	25 sensors	10 sensors
Average distance	6025	5962	5929

Tabell 5.6: The average distance traveled in meters for the different digital twins that used prediction to reach their destination.

Digital twins	50 sensors	25 sensors	10 sensors
Average distance	6133	6122	6003

Tabell 5.7: The average time stopped in seconds for the different digital twins that did not use prediction to reach their destination.

Digital twins	50 sensors	25 sensors	10 sensors
Average time stopped	32.15	33.1	35.1

Tabell 5.8: The average time stopped in seconds for the different digital twins that used prediction to reach their destination.

Digital twins	50 sensors	25 sensors	10 sensors
Average time stopped	28.05	28.5	29.2

the digital twins were 1300% worse than the baseline, and if only looking at the result like this, a very bad outcome. But, looking at the test, the baseline only had a time stop of one second, while the other tests were 13 seconds, and for a trip that took 458 seconds for the baseline and around 500 seconds for the digital twins, most of the time increase did not come from the time stopped.

The average time stopped for the baseline was 36.75 seconds per trip. The average time for the other digital twins can be found by looking at the tables 5.7 and5.8. Unlike the results, which showed an increase in time stopped, the real-time waited per trip is lower across the board. An interesting behavior shown here is a reverse of what has been seen before, where the fewer sensors showed more promising results. Here the best results are from the digital twins using 50 and 25 sensors and not ten. Then a takeaway from using the current setup is that avoiding stopping is more prioritized than getting to the destination faster. This could result from the high amount of traffic at intersections being too penalized compared to the actual benefit of avoiding that intersection.

From the results of the average speed, All of the digital twins had a higher average speed compared to the baseline. The average speed for the baseline was 9.84 meters per second. Tables 5.9 and 5.10 show the average overall speed for the other digital twins. This is not all that surprising considering the previous results, where it was shown that both the average overall time was lower but also, the average distance of the routes was longer. This should imply that also the average speed should be higher, which is what we see.

5.1.2 Test scenarios

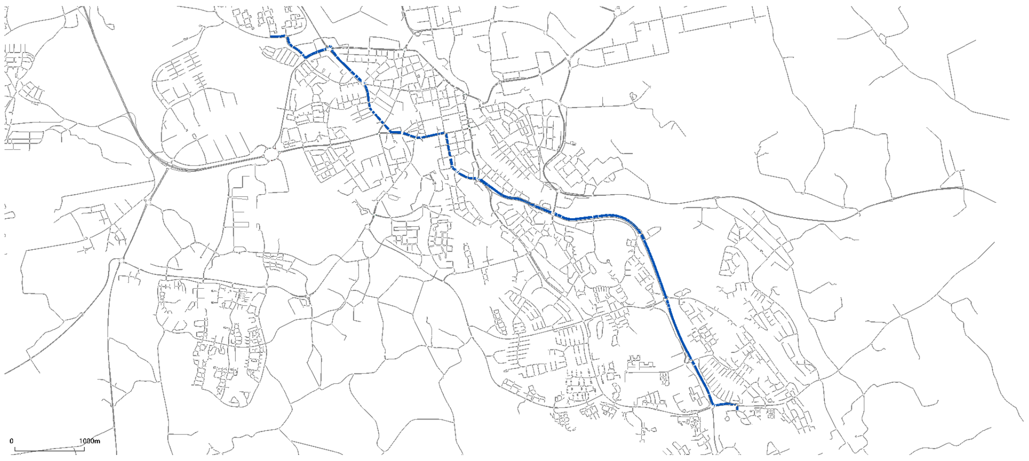
The results from running the tests can broadly be put into three categories, the tests that showed improvement, those that showed a decline, and those that gave results equal to the baseline. This section will discuss these results and try to find out why the digital twins ma-

Tabell 5.9: The average speed in meters per second for the different digital twins that did not use prediction to reach their destination.

Digital twins	50 sensors	25 sensors	10 sensors
Average speed	10.0	9.9	9.95

Tabell 5.10: The average speed in meters per second for the different digital twins that used prediction to reach their destination.

Digital twins	50 sensors	25 sensors	10 sensors
Average speed	10.17	10.18	10.1



Figur 5.1: Image of the route generated for test scenario 10 using the baseline

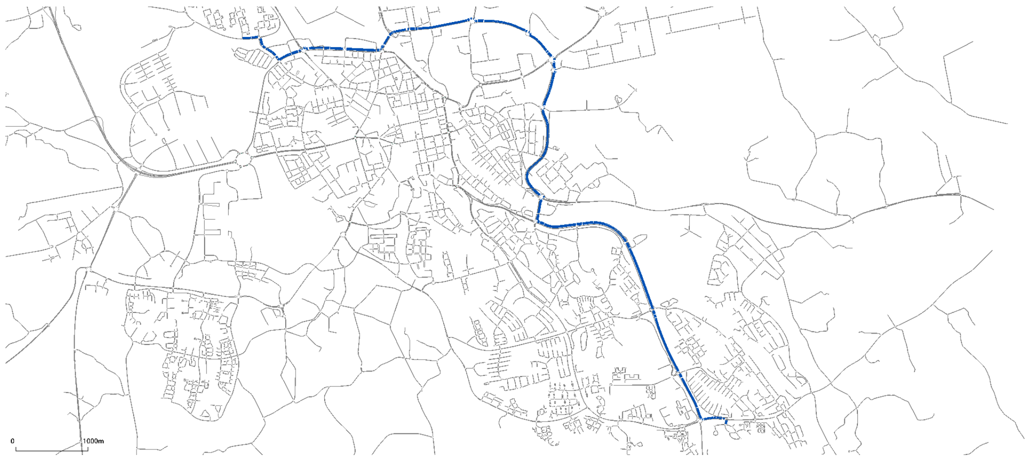
de certain decisions. This is done by comparing the routes generated for the different test scenarios.

Tests 2,8,12, and 15 had the same results given the different digital twins. This would indicate that the route found is the fastest with and without traffic, and it's doubtful that a quicker way can be found. While it's not impossible, given that the test was conducted with seven different digital twins, some differences should have been observed if there was a candidate route that was a least similar to the baseline as shown in the other tests, where some tests performed very close to the baseline, but not precisely.

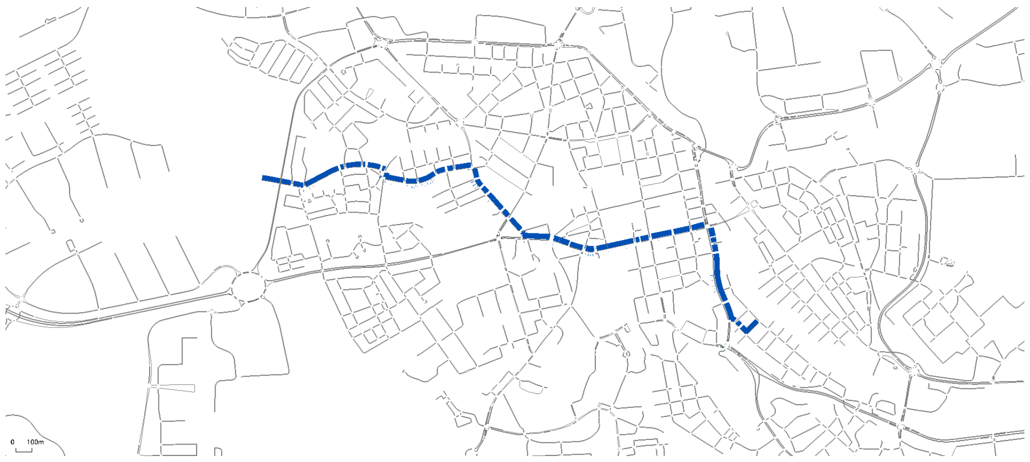
The test that showed the largest improvement for all tests was number 10, specifically for the digital twins using prediction with 25 and ten sensors; here, it was 18.04% faster than the baseline. The time saved was 191 seconds for those two digital twins. Looking at the other metrics, significant time savings came from the reduction in the time stopped; here, at the baseline, the vehicle stood still for 122 seconds, whereas the other two twins only stopped for 35 seconds. The rest of the time was saved, because of how much faster it was, in terms of average speed. Here the digital twins drove 44.68% faster than the baseline even though the route was 18.59% longer; the average speed more than made up for the difference. Also, the Co2 emission was lowered by taking this route and this was not entirely expected. Driving faster should, in most cases, cause higher emissions, but not in this case. This could mean that in the baseline, there was more start and stop traffic. Having the accelerate and decelerate often might have led to the higher emissions seen in the baseline.

In figure 5.1 and 5.2, the routes are drawn for both baseline and the digital twin using ten sensors and prediction. From those images, the difference in the route taken is clear, as the baseline decided that a route through the city would be faster, and the other digital twin took a route around the city. The routes later meet a point further down and continue to the destination. The difference in the route taken happens early on in this test scenario and has a significant effect on the metrics, as discussed above.

A test scenario that the digital twins performed poorly on overall was 17. Here, only one of the twins could match the baseline. The rest were 9 to 13% worse in time. The time for the baseline to reach its destination was 458 seconds, and its route is shown in Figure 5.3. Comparing the baseline to using 50 and ten sensors using prediction, is found in Figures 5.4

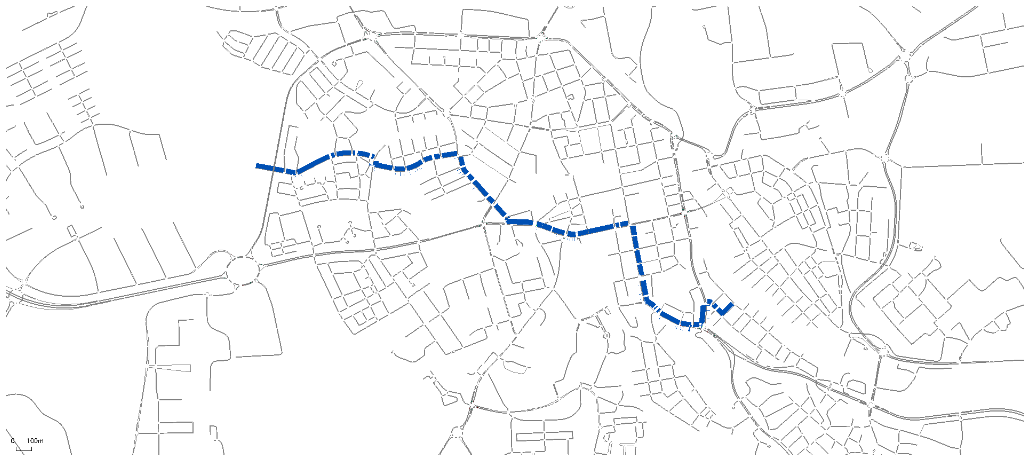


Figur 5.2: Image of the route generated for test scenario 10 using the digital twin with ten sensors and prediction

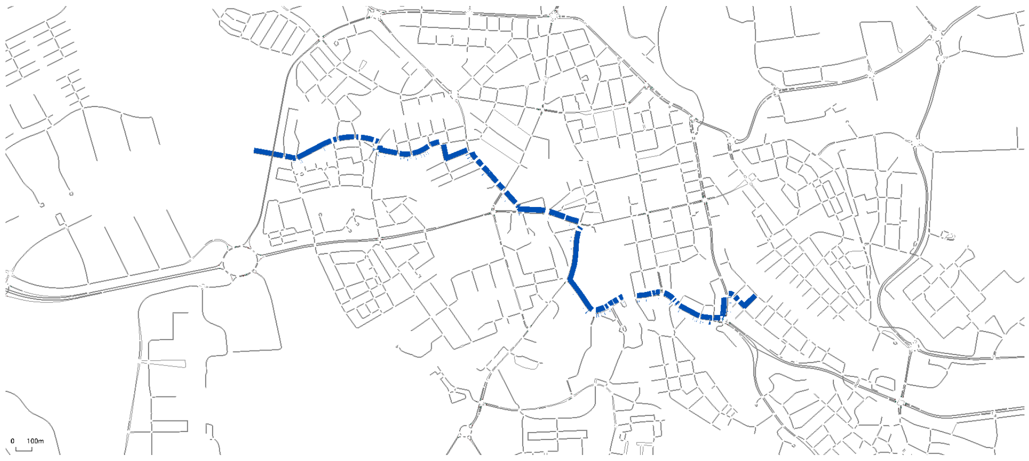


Figur 5.3: Image of the route generated for test scenario 17 using the baseline

and 5.5. The baseline route can be argued to be more straightforward compared to the others. The difference between baseline and using ten sensors is not that major. The difference seen when using 50, however, is larger. The route looks to take more of a path on side roads instead of the main road. Viewing the average speed of the different routes, the baseline was the fastest, and the other two had similarly slow speeds. And since the route using 50 sensors was longer, it was also the slowest in time. One more thing to consider when inspecting the routes for the baseline and ten sensors, the routes are similar, yet the one generated by the ten sensors is 43 seconds slower, yet it's only 116 meters longer. It can't be explained by time stopped alone, as it's only a difference of 13 seconds. Can it be that when the route generated by the baseline was driven, it was lucky that it did not have to wait at junctions as much or at all? If so, one could wonder what difference this could make for other tests, what tests were the baseline lucky or unlucky. This, however, should apply less to the other digital twins as the point of the sensors and prediction is to have foresight into what the traffic should look like at different times of the route. This is compared to the baseline, which has no such advantages. One way to alleviate this issue is to run the same tests for more than one traffic scenario and use the average of many tests to get a more concrete answer. This, however, is outside the scope of this thesis.



Figur 5.4: Image of the route generated for test scenario 17 using the digital twin with ten sensors and prediction



Figur 5.5: Image of the route generated for test scenario 17 using the digital twin with 50 sensors and prediction

5.2 Method

5.2.1 Environment

The choice to use the SUMO simulator came down to what was available and had the necessary features to answer the research questions. SUMO is both free and open source, which is suitable for availability. Then the question became if the feature set of the simulator was enough. Reviewing the literature on the subject made it clear that SUMO has been used extensively in research, in the area of traffic prediction and other types of traffic simulations. This was confirmation of the SUMO's credibility in the use of answering the research questions for this thesis. This will also make this thesis more comparable with other research literature using SUMO to get their results.

In this thesis, SUMO version 1.16 was used. As this software is constantly updated, it's not guaranteed that the behavior of the cars will stay constant for future updates. This can affect the results found in this thesis and should be considered when trying to replicate this study. The exact feature set for previous versions is unknown to the author, and we can not guarantee that everything can be replicated for those older versions. The only way to be sure that future studies have the same behavior is to use the same version.

The map used in this thesis represents the city of Linköping. To get this map, OSM was used. This was done to get a realistic road network for the digital twin to traverse. This makes the result closer to want it might look like in reality. The map was captured at the beginning of 2023. Changes to the road network could come in the future, as the map is updated from changes to the physical location or other corrections are made.

The traffic sensors used, measure the volume of traffic at different intersections. The measurements were confirmed to be accurate, by comparing the values the sensors gave and counting the number of vehicles manually. The placement of sensors was done manually. This introduced a lot of personal biases as to where each of the sensors would be placed and can be seen as the largest weakness in terms of replicability in the thesis. To minimize the bias added by this step, some of the chosen locations were real-life sensors. All sensors were not done this way, because many of the locations of real-life sensors were in uninteresting locations that were out of the way and not on major intersections. Replicating the experiments would require the placement of sensors in the same intersections as done here. But seeing as the amount of personal bias in the choices, a more robust solution of where to place the sensors should be investigated.

5.2.2 Data

To get the data for training the GNN, first traffic was generated, and the simulation was run to get sensor data. The traffic generated was random but was made to mimic the traffic volume seen at different times of the day. The start and end destinations are chosen at random with a uniform distribution. The generation also uses seeds. For the traffic generated for the dataset, seeds were chosen to start at one and go to 10 for all ten days of traffic generated. These seeds should not have too large of an impact on the traffic overall. What's important is the choice of repetition rate. As mentioned, it was selected to mimic how traffic looks in reality. This was done by getting data from the real-life traffic sensors and selecting repetition rates for each hour of the day to be as close to real-life values as possible. This was done to increase the validity of the data.

The data itself was of the volume of traffic at the junction over a time of a minute. This was done using the multi-entry-exit detector. In the case of counting vehicles in a junction, this was the easiest way, because they could cover each junction's entries and exits. This could also have been done using induction loops, but implementation would have taken extra time. Each induction loop must be paired with a node in the graph instead of just one multi-entry-exit detector.

Another option for measuring traffic would have been to measure individual roads. Using this approach, the traffic volume could be directly used to change the cost of the edge, instead of looking to adjacent edges to a junction and going from there. Then the lane-area detector would have been a great option.

5.2.3 Traffic prediction

For traffic prediction, a graph neural network was implemented. This was because of the graph-like structure of road networks, making it a good match. The type of prediction used was the type node prediction. This is where predictions on the nodes are made; this, in turn, was then used to predict future traffic, given the current status of the graph. The data gathered from the sensors was used to create a data set for the model to train. The data was the volume of traffic at the junction over one minute. This was done so that predictions of future traffic could be made in iterations of one minute. There are both positives and negatives to this approach. Firstly, getting frequent updates will, in theory, give the most up-to-date information on what the traffic should look like, giving the digital twin the highest chance of making good decisions when creating the routes. One of the downsides of this is that traffic in a junction can change a lot from minute to minute, which can be challenging for the model to learn. Looking at other research literature, 15 minutes was frequently used as a prediction interval. This, however, is not very useful for the test scenarios that were performed in this thesis, as the average route took around ten minutes to drive. There could, however, be a case made for other prediction intervals than just one minute. For example, there could be a two, three, or five-minute interval. This would be easy to extend the current implementation to use these intervals. This is because the one-minute data intervals could easily be added together to create whichever other larger interval wanted. The only interval that would not work as easily would be a shorter interval, such as a 30-second interval. This would require new data to be collected.

5.2.4 Digital twin

All the digital twins used similar cost functions for pathfinding. They were all based on finding the fastest path. The fastest path, in this case, means the route that takes the shortest amount of time. Another cost function that could be interesting to investigate is the shortest path. This, in combination with using sensor data and prediction, could give exciting results.

Regarding the function for the multiplier, it would be interesting to have tried more configurations of the formula. For example, instead of a multiplier, a constant cost could have been tried. A cost that still increases based on the amount of traffic, but instead of multiplying the edge cost, add value could have been added. This is interesting because of how the edges and nodes are connected, but also how the length of edges matters. If, for example, the cost for an edge near a junction is calculated, here, the edge is short but connected to an edge behind it which is longer. As the short edge is connected to the junction, its cost is multiplied by the amount of traffic, but the longer edge is not. In this example, if the traffic is high, the total cost of the route for taking that specific edge would not be that drastic. It would be larger if the cost were additive, then the length of the edge would matter less. An alternative to this is to cascade the cost increase to more than just the adjacent edges to the junction but also to edges connected to the adjacent edges. The cost could propagate through the sumo graph, making it almost like a heat map, where junctions are the heat sources. Having a heat map of the traffic would not only help in traffic prediction but could also be used by city planners to find weaknesses in the road network.

5.2.5 Testing and evaluation

The testing was done by creating 20 different test scenarios. This is by choosing 20 pairs of origin and destination points randomly. This was based on what was done in a similar scientific paper. The amount chosen also had a connection to the reliability of the experiments. With this amount, there is a high likelihood that others performing the same experiment would receive similar results. Although, as discussed in the results, outliers can still skew the results, and if more time was had, more test scenarios should have been made.

The tests were done in a traffic scenario with a good amount of traffic. Other configurations, such as with even more traffic, would also have been interesting to see how the different digital twins would perform. This, however, is a quick way to double or triple the number of tests needing to be run and would require a lot more time to do the extra tests.

The evaluation process relies on the information the simulator gives to be accurate. The metric time to reach the destination is the easiest to confirm, as the simulation ends when the test vehicle reaches its destination. When the simulation is finished, the current time step can be subtracted from the start time. For the experiments, it was found to be always accurate. The other metrics are a bit more difficult to determine if they are accurate. However, as long as it is internally consistent with itself, there should not be any issue. As the SUMO simulator has been used many times in scientific literature, there is little doubt that the numbers given by the simulator should be inaccurate, and the results can be regarded as valid. The vehicle model used in the simulator was the standard car implementation. This means that if it is changed, the values received might not be precisely what was found in this thesis.

5.2.6 Source criticism

Regarding the sources critics, The main sources used in the theory chapter were from trusted publications and are trusted to have accurate information. Not all sources used were from these publications. Some of the sources for the GNN were not from published journals, and they could be seen as less trustworthy, though, Google researchers wrote these sources used. Some books, and lectures from other universities were also used as sources. As I'm not an expert in the field, it is difficult to say if the sources were biased in some way.

5.3 The work in a wider context

The work presented is about providing better routes for drivers in vehicles. This could be seen as a societal benefit as the goal is to shorten their travel time. From another aspect, this work focuses on drivers in cars and does not take into account people without them. From an ethical aspect, this could exclude those people. As the work focuses on improving cars, no consideration was made for other types of travel that these changes suggested might impact. This is something that should be investigated. From the results, it was found that the Co2 emissions were sometimes reduced. This can only be seen as a societal benefit.

6 Implementation in real life

Implementing a version of the digital twin in real life, there are some technical challenges to overcome. This chapter aims to present the challenges and possible solutions. The first section will investigate the data collection process, the second section will look at the data processing, and the final section will investigate how the information from the sensors and models can be delivered to the drivers.

6.1 Sensors and data collection

The Graph neural network requires information about the current traffic situations at different junctions, then uses that information to predict future traffic. In the Simulator, there were three ways to measure traffic: Induction loops, Lane-area detectors, and Multi-entry-exit detectors. These sensors are also used in real-life or have real-life equivalents. For Lane-area detectors and Multi-entry-exit detectors, the real-life traffic equivalents would be traffic-surveillance cameras. Another way to measure traffic motioned by Tomar et al. [23] is using global position system (GPS) sensors inside vehicles. These sensors will be evaluated in the following ways: cost, reliability, and applicability. Cost measure the investment required to install and operate the given sensor. Reliability will investigate how given sensor data can be trusted to be accurate. Lastly, applicability determines how a given sensor setup can be used.

6.1.1 Induction loop

The induction loop shows promising results in previous research [1]. In terms of cost, the induction loop is an excellent option, as it's very cheap to operate and maintain. Also, induction loops are popular to install at junctions as they are used for traffic signal control. The only actual extra cost for implementing this system is to get the information from the sensors in real-time to a central location, if not already implemented. Induction loops have also been found to be reliable, because of their resistance to environmental hazards as it's buried in the ground. The applicability aspect is also straightforward, as the location of the individual induction loops would be known. Measuring the traffic at a specific junction would be just adding the different values together.

6.1.2 Traffic cameras

Monitoring traffic with cameras is a heavily researched topic [8, 14]. Using cameras has both advantages and disadvantages. With cameras, more information can be captured in the junction compared with induction loops, information such as the type of vehicles, speeds, and other movement patterns. Other benefits of using cameras are that they can be used for other applications such as capturing traffic violations and identifying vehicles by their license plates [11]. There are, however, some disadvantages to using traffic cameras in terms of cost and reliability. Firstly installing cameras comes with cost and infrastructure requirements. Some costs that need to keep in mind are the installation of the cameras, network infrastructure, and data storage. There is also a cost in terms of computing power, were using cameras as traffic monitor tool, require the use of computer vision to analyze the images. Regarding reliability, cameras have some issues; firstly, cameras may have limitations regarding coverage, image quality, and field of view. Cameras might not perform as well during challenging weather conditions or at night. Regarding applicability, as mentioned, cameras need to use computer vision to find and identify different vehicles, however, the exact method for this can be debated. Still, two promising methods for detecting vehicles are Region-Based Convolutional Neural Networks (R-CNN), and You Only Look Once (YOLO) [14]. A Challenge that will need to be solved when using these methods is to keep track of what vehicles are new and those who have already been counted which could otherwise lead to overcounting. Another way mentioned to measure traffic by Mandal et al. [14] is to use the area covered by cars as in the image to estimate the traffic. However, with this approach, changes to other parts of the digital twin need to be made. As we now have a percentage of cars covering an image instead of a quantity, a translation from this percentage to a representation of how much traffic there is at that moment. With that addition, the rest of the code could remain unchanged.

6.1.3 GPS

The use of GPS systems in traffic for traffic monitoring tasks has shown promising results in previous research [15, 6]. With smart devices or built GPS systems in vehicles, accurate information of a vehicle's position and speed allows for monitoring traffic conditions at junctions. With GPS monitoring, a larger area can be covered, compared to the other sensors, where vehicles can be tracked through multiple junctions and provide a more comprehensive view of traffic patterns. In terms of cost, GPS-based traffic monitoring does not require extensive infrastructure to be built. Instead, using already available GPS devices in vehicles or through smartphone applications, thus making it a very cost-effective solution. The reliability of the system can be varied. GPS can provide accurate location data, which can be used in various ways, such as speed, travel distance, and time. However, there might be limited accuracy in areas where the GPS signal is poor, for example, near tall buildings or inside tunnels. This can reduce the accuracy and reliability of the system. Another issue with the use of GPS is the dependency on external devices. Having a system entirely reliable on external might cause issues if too few users participate. If, for example, a user is not willing to share their location or a given vehicle is old a does not have GPS capabilities. In terms of applicability, Martín et al. [15] suggested the use of virtual induction loops with the GPS. This could be used to count the number of vehicles going through a junction, making this solution very close to regular induction loops. The only difference is that no infrastructure needs to be built.

6.1.4 Sensor comparison

The three suggested induction loops, traffic cameras, and GPS have strengths and weaknesses. In terms of cost, induction loop and GPS were inexpensive solutions, while traffic cameras would require more investment. In terms of reliably, all sensors would work. However, induction loops have little to no weaknesses, while the other two might run into issues in unfavorable conditions. In terms of applicability, again, induction loops are close to what was used in the experiments and would not require significant architectural changes. Similarly, GPS would not require changes, and that leaves the use of traffic cameras, where some changes or extensions would need to be made. So for implementing this solution in real-life, induction loops seem like the best choice. However, If, for example, a city planner wants more information than induction loops can provide, then traffic cameras or GPS could be of value.

6.2 Processing of data

The previous section has covered some data processing but, not in its entirety. There is a two-step process for processing most of the data from the sensors. The first step is the initial processing that can be done at the location. Then that data needs to travel to a central location where data from multiple locations can be put together. For induction loops, as mentioned, there can be simple processing at the location where the results are concatenated, or each sensor can send in its results, and the associated sensors can have their results put together.

For the traffic cameras, one could debate how many cameras are required for each junction, As this depends on the camera used and its properties, such as resolution or field of view. And as mentioned in the applicability of traffic cameras, multiple ways of processing the data could be used. Depending on the frame rate and resolution of the camera, more processing power is needed. If the requirements are great, then processing the images at a central location might be required. However, streaming video feeds from many cameras will need a lot of bandwidth. This will become a balancing game, where a choice on cost and reliability needs to be made.

The processing required for the GPS sensor would be minimal in comparison. As there are no physical sensors on location, the data is sent directly from either the vehicle or smart device. The data would be sent in intervals, and comparisons could be made between where the vehicle was and where it's now. This information makes it easy to see if it passed through a junction. This, however, assumes that the updates are frequent enough.

After receiving the sensor data and after it has been processed, the data can be saved for future training of the graph neural network and used to make future predictions on an already-trained network. This is similar to how it was done in the experiments.

6.3 Information delivery

Getting information about current and future traffic conditions to the drivers has many solutions. First, can one consider a solution using digital signage. The digital signs could be at intersections that suggest different routes for driving to specific popular destinations in the city. However, there is only so much information that could fit into the sign. And the suggested routes would be limited to what information they give to drivers. But if, for example, there is road maintenance, an accident has occurred, or other reasons for a road to be closed, that information could be displayed on the signs and suggest alternative routes.

A more direct approach would be to deliver the information directly to the driver. This could be done using an application in the car or an app on the phone. The application would be similar to any other navigation app, though this one would have detailed information about current and future traffic conditions. This would require each driver to install and run an extra application, reducing the benefits for those who chose not to use the application. Another approach is to make the real-time information open and easy to access. Then the other navigation applications could use the traffic information in their route generation. This would be beneficial, so drivers could continue to use their preferred application. However, it's not a guarantee that the makers of these applications will start using the provided traffic data.

7 Conclusion

7.1 Conclusions

This thesis investigates if the machine learning method graph neural network could improve traffic route creation. The routes were tested using digital twins inside the traffic simulator SUMO. Five metrics were used to measure the effect this would have: travel time, Co2 emission, distance traveled, time stopped, and average speed. Seven digital twins were tested, a baseline, three sets of digital twins that used sensor data to avoid traffic, and the last three used sensor data and machine learning to predict future traffic. The six digital twins that used sensor data were further divided into three groups that used different amounts of sensors to measure the effect more sensors would have. It also investigated what effect the number of sensors had on the ability of the machine learning method to predict future traffic.

In terms of time traveled, four of the six digital twins performed better on average than the baseline. For the Co2 emission, only two of the digital twins performed better. For distance traveled, it was found that all digital twins took, on average, a longer route than the baseline. It was the opposite, for time stopped, and all digital twins waited, on average, a shorter amount of time than the baseline. Lastly, for average speed, it was found that all digital twins had a higher average speed than the baseline. It was found that the digital twins that used fewer sensors were, on average better travel time, Co2 emission, and distance traveled. And the other twins performed better in the other metrics.

Regarding the graph neural network learning to predict future traffic, it was found that the model trained using 50 and 25 sensors was close to equally good when looking at the validation data. When using ten sensors, it performed less than the other two.

7.2 Future work

As for future work, a few areas would be interesting to investigate. Firstly, It was found in this thesis that a lower amount of sensors generally performed better. For future work, it would be of interest to investigate the optimal amount of sensors for a given city. And as a follow-up question on that one, what is the optimal placement of these sensors with the given number of sensors?

When predicting future traffic, a time step of 60 seconds was used. One could also investigate how other intervals could change the model's ability to predict future traffic.

During this thesis, only graph neural networks were investigated for predicting future traffic. LSTMs were also found in the literature to be popular in this domain. Then, How do LSTMs compare to GNNs in predicting future traffic? But also, how do the generated routes compare, given the original metrics, when using LSTM and GNN?

Litteratur

[1] Lala Bhaskar, Ananya Sahai, Deepti Sinha, Garima Varshney och Tripti Jain. ”Intelligent traffic light controller using inductive loops for vehicle detection”. I: *2015 1st International Conference on Next Generation Computing Technologies (NGCT)*. 2015, s. 518–522. DOI: `10.1109/NGCT.2015.7375173`.

[2] Ameya Daigavane, Balaraman Ravindran och Gaurav Aggarwal. ”Understanding Convolutions on Graphs”. I: *Distill* (2021). https://distill.pub/2021/understanding-gnns. DOI: `10.23915/distill.00032`.

[3] Lex Fridman. *Deep Reinforcement Learning*. `https://deeplearning.mit.edu/`. [Online; accessed 27-Mars-2023]. 2018.

[4] Jiuxiang Gu, Zhenhua Wang, Jason Kuen, Lianyang Ma, Amir Shahroudy, Bing Shuai, Ting Liu, Xingxing Wang, Gang Wang, Jianfei Cai m. fl. ”Recent advances in convolutional neural networks”. I: *Pattern recognition* 77 (2018), s. 354–377.

[5] Sepp Hochreiter och Jürgen Schmidhuber. ”Long short-term memory”. I: *Neural computation* 9.8 (1997), s. 1735–1780.

[6] Qiuyang Huang, Yongjian Yang, Yuanbo Xu, Funing Yang, Zhilu Yuan och Yongxiong Sun. ”Citywide road-network traffic monitoring using large-scale mobile signaling data”. I: *Neurocomputing* 444 (2021), s. 136–146.

[7] Rongzhou Huang, Chuyin Huang, Yubao Liu, Genan Dai och Weiyang Kong. ”LSGCN: Long Short-Term Traffic Prediction with Graph Convolutional Networks”. I: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*. Utg. av Christian Bessiere. Main track. International Joint Conferences on Artificial Intelligence Organization, juli 2020, s. 2355–2361. DOI: `10.24963/ijcai.2020/326`. URL: `https://doi.org/10.24963/ijcai.2020/326`.

[8] Neeraj Kumar Jain, RK Saini och Preeti Mittal. ”A review on traffic monitoring system techniques”. I: *Soft Computing: Theories and Applications: Proceedings of SoCTA 2017* (2019), s. 569–577.

[9] Donald B Johnson. ”A note on Dijkstra’s shortest path algorithm”. I: *Journal of the ACM (JACM)* 20.3 (1973), s. 385–388.

[10] Thomas N Kipf och Max Welling. ”Semi-supervised classification with graph convolutional networks”. I: *arXiv preprint arXiv:1609.02907* (2016).

[11] Venkata Satya Rahul Kosuru, Ashwin Kavasseri Venkitaraman, Vijay Dattatray Chaudhari, Neha Garg, Aln Rao och A Deepak. ”Automatic Identification of Vehicles in Traffic using Smart Cameras”. I: *2022 5th International Conference on Contemporary Computing and Informatics (IC3I)*. IEEE. 2022, s. 1009–1014.

[12] Jinglin Li, Dawei Fu, Quan Yuan, Haohan Zhang, Kaihui Chen, Shu Yang och Fangchun Yang. ”A Traffic Prediction Enabled Double Rewarded Value Iteration Network for Route Planning”. I: *IEEE Transactions on Vehicular Technology* 68.5 (2019), s. 4170–4181. DOI: `10.1109/TVT.2019.2893173`.

[13] Pablo Alvarez Lopez, Michael Behrisch, Laura Bieker-Walz, Jakob Erdmann, Yun-Pang Flötteröd, Robert Hilbrich, Leonhard Lücken, Johannes Rummel, Peter Wagner och Evamarie Wießner. ”Microscopic Traffic Simulation using SUMO”. I: *The 21st IEEE International Conference on Intelligent Transportation Systems*. IEEE, 2018. URL: `https://elib.dlr.de/124092/`.

[14] Vishal Mandal, Abdul Rashid Mussah, Peng Jin och Yaw Adu-Gyamfi. ”Artificial intelligence-enabled traffic monitoring system”. I: *Sustainability* 12.21 (2020), s. 9177.

[15] Juan Martın, Emil J Khatib, Pedro Lázaro och Raquel Barco. ”Traffic monitoring via mobile device location”. I: *Sensors* 19.20 (2019), s. 4505.

[16] OpenStreetMap contributors. *Planet dump retrieved from https://planet.osm.org*. `https://www.openstreetmap.org`. 2017.

[17] Ningyezi Peng, Yuliang Xi, Jinmeng Rao, Xiangyuan Ma och Fu Ren. ”Urban Multiple Route Planning Model Using Dynamic Programming in Reinforcement Learning”. I: *IEEE Transactions on Intelligent Transportation Systems* 23.7 (2022), s. 8037–8047. DOI: `10.1109/TITS.2021.3075221`.

[18] Hannah Ritchie och Max Roser. ”Urbanization”. I: *Our World in Data* (2018). https://ourworldindata.org/urbanization.

[19] Stuart Russell och Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3. utg. Prentice Hall, 2010.

[20] Benjamin Sanchez-Lengeling, Emily Reif, Adam Pearce och Alexander B. Wiltschko. ”A Gentle Introduction to Graph Neural Networks”. I: *Distill* (2021). https://distill.pub/2021/gnn-intro. DOI: `10.23915/distill.00033`.

[21] Mike Schuster och Kuldip K Paliwal. ”Bidirectional recurrent neural networks”. I: *IEEE transactions on Signal Processing* 45.11 (1997), s. 2673–2681.

[22] Richard S. Sutton och Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: `http://incompleteideas.net/book/the-book-2nd.html`.

[23] Ishu Tomar, Indu Sreedevi och Neeta Pandey. ”State-of-Art review of traffic light synchronization for intelligent vehicles: current status, challenges, and emerging trends”. I: *Electronics* 11.3 (2022), s. 465.

[24] Christopher JCH Watkins och Peter Dayan. ”Q-learning”. I: *Machine learning* 8 (1992), s. 279–292.

[25] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang och S Yu Philip. ”A comprehensive survey on graph neural networks”. I: *IEEE transactions on neural networks and learning systems* 32.1 (2020), s. 4–24.