Industrial engineering and management

Decision Support for Traffic Management using Reinforcement Learning

Author:
A. Heijnen

Supervisors:
M.R.K Mes
W.J.A. van Heeswijk
P.W.M. van Bakel

Case owners:
Technolution

Date:
February 12, 2025

Management Summary

This research investigates the application of Deep Reinforcement Learning (DRL) as a decision-support tool with the objective to reduce urban traffic congestion across multiple intersections. As the currently used solution method in traffic management, rule-based decision-making, is time-consuming to implement, lacks scaling, and is difficult to design well, an alternative is required. DRL is chosen as the solution method due to the complex nature of traffic and the long-term effects of actions. The strengths of neural networks and Reinforcement Learning (RL) are combined to learn the long-term effects of actions and find the best solution given a traffic state. The main research question is "**How suitable is deep reinforcement learning for providing real-time decision support to traffic engineers for reducing congestion across multiple intersections in an urban traffic network?**"

As the aim is to provide decision support to traffic engineers, the outcome of the DRL algorithm has two requirements: interpretability and infrequent adjustments. This research adopts a macro-level approach that adjusts traffic flows over 15-minute intervals, unlike micro-level optimizations that alter traffic light phases every few seconds. The selected method, perimeter control, regulates traffic that enters and exits a defined city center region by modifying traffic light phase durations at its perimeter, being the outside edges of the region. This strategy reduces congestion in the city center, maintaining fluent traffic flow with reduced average travel times.

However, real-world traffic is rarely uniform, resulting in potential inefficiencies such as wasted green time at certain traffic lights. To address this, a secondary layer of control is introduced, max pressure. This micro-level technique balances traffic queues at individual intersections by considering incoming traffic flow and the available space at destinations. Together, these two layers form an integrated control system: Perimeter control for strategic adjustments combined with max pressure for micro-level control for localized optimization.

First, this study simulates a 25-intersection grid network using the SUMO traffic simulator, representing a real city scenario. Supervised learning is then used to train a neural network capable of predicting average traffic speed over 7.5-minute intervals. The results show a prediction accuracy with a mean absolute percentage error (MAPE) of 20–25%.

Second, iterative supervised learning is used to choose the best action given a traffic state. The newly generated data is then used to fine-tune the neural network with the newfound results. After five iterations of using the trained neural network to choose the traffic light policy, the average speed is increased by 5-9% with respect to the initial performance of static random policies. When compared to a better standard static policy the mean speed is increased by only 0-3%. Although the mean speed is increased by this approach, the results are bad since the policy it is compared to involves no intelligence and should thus be significantly outperformed. On the positive side, there is still potential for further improvements as more iterations are done. The process of iterative supervised learning is also computationally intense making it unrealistic to perform a large additional amount of iterations. Fine-tuning the original neural network does improves the prediction accuracy, with a newfound MAPE of 8–10%.

Lastly, DRL is used to find the best action given a traffic state. This is first done without max pressure and thus no local traffic light optimization. All traffic lights are changed with the same amount according to the action found by the DRL algorithm. As more iterations are done, learning can be observed, as the reward increases. Despite this, the increase in performance plateaus slightly below the mean speed of the standard traffic light settings. This means that the best-found solution of the DRL algorithm performs worse than making no changes, which is also a possible action for the DRL algorithm to select. This leads to the belief that the DRL algorithm is not able to fully understand the impact of an action on the environment or the long-term effects. Supervised learning shows that predicting future traffic is achievable. Nonetheless, choosing actions to improve the average speeds leads to relatively small improvements.

Reflecting on the research question: In theory, DRL should be able to provide decision support to traffic engineers and in consequently reduce congestion in the city. However, the implementation of DRL is challenging and unsuccessful during this research. Alternatively, supervised learning using neural networks is found to be effective in predicting traffic. This could be used as a form of decision support for traffic engineers.

Key findings and recommendations:

- Supervised learning using neural networks is a promising tool for predicting complex data such as traffic. With additional fine-tuning of the neural network a MAPE of 8-10% is achieved. Supervised learning can also be used to make actions increasing the average speed by 0-3% compared to a standard static policy.
- Deep reinforcement learning (DRL) should in theory be a useful tool for traffic management but implementation is difficult due to the black-box nature of neural networks. In this study, DRL is not able to outperform the base traffic light policy. If DRL is the desired solution method for a problem, sufficient time and expertise is required for successful application.
- For gathering data, both supervised learning and DRL are computationally intense. Once both techniques are trained, the time required to find a good solution is short. Hence, both techniques are still applicable in cases that need fast response times.
- The most useful artificial intelligence tool identified during this research is neural networks, which are used to predict future traffic. This can assist in providing decision support to traffic engineers and could be a promising service for Technolution to provide.
- To investigate techniques to reduce computational time future research is recommended by the author. As well as exploring alternative reward structures, state formulations, and actions to improve the DRL approach.

Acknowledgements

I am grateful for the opportunity that Technolution has presented with this research. I always enjoy coming to the office and talking to helpful, friendly colleagues. Completing this research would not have been possible without the help of my supervisor from Technolution, Tijs van Bakel. I offer my sincere appreciation for all the help and useful feedback.

A special thanks to my first- and second supervisor, Martijn Mes and Wouter van Heeswijk, for giving guidance, and thorough feedback during my research. Your input has significantly helped me during this project.

Lastly, I would like to thank my girlfriend, family, and friends for the review, encouragement, and support that helped me in completing this research.

Contents

List of Figures

Glossary

Ant Colony Optimization	A bio-inspired algorithm that mimics the foraging behavior of ants. Ants leave pheromones on paths while searching for food, and the algorithm simulates this by favoring paths with stronger pheromone concentrations. Over time, the algorithm converges to optimal or near-optimal solutions.
Deep Loosely Coupled Q-network (DLCQN)	Type reinforcement learning which allows agents to act independently or cooperatively depending on circumstances to prevent policy bias.
Deep Reinforcement Learning (DRL)	Technique of training software to make decisions using a neural network to get the optimal result.
Deep Repeated Update Q-network (DRUQN)	Type of reinforcement learning which updates action values depending on the probability of selecting an action to avoid policy bias.
Markov Decision Process (MDP)	Mathematical framework for modeling decision making where outcomes are partially random and controlled by the decision maker.
Reinforcement Learning (RL)	Technique of training software to make decisions to get the optimal result.

Note: In this thesis, the terms "Reinforcement Learning (RL)" and "Deep Reinforcement Learning (DRL)" will be used interchangeably, as DRL is just RL using neural networks.

1 Introduction

This chapter gives background information about the company, a description of the problem, as well as an approach to solving this problem with the corresponding research questions and subquestions. To conclude, an overview of the research design will be provided.

1.1 Company description

Technolution is a Dutch technology company founded in 1987. Technolution specializes in developing innovative, high-tech solutions across various industries, including mobility, energy, and security. The mobility sector focuses on aiding traffic engineers, finding solutions to traffic problems and automatization of processes. Their software MobiMaestro is the most used platform for traffic management in The Netherlands. MobiMaestro provides users with insight, overview, and control regarding environment, traffic, and city management (Technolution Move, 2024).

1.2 Research plan

The following subsection will cover the problem description as well as the objective of this research based on the assignment provided by Technolution.

1.2.1 Problem description

Technolution would like to have a new system where the users (traffic engineers) can implement their conditions by assigning priorities to the road network. The new system will provide decision support to traffic engineers, making it easier for them to configure cities based on their prioritized objectives. This decision support system would tell the traffic engineer which changes to make to the traffic control system to get the preferred result of the traffic engineer, making it a prescriptive decision support system. Nonetheless, determining the optimal traffic control system settings for all systems presents a complex algorithmic challenge. In large cities, there are hundreds or even thousands of traffic lights. The different traffic situations that are encountered are also very high, making the mathematical model and thus algorithm complex to define and model. As it is not feasible to mathematically compute an optimal solution with so many traffic scenarios, a degree of uncertainty, and long-term effects, an alternative approach is required that can address these challenges without relying on exhaustive calculations of every possible scenario (Inoue et al., 2024).

1.2.2 Objective of the research

As Technolution wants to explore the possibility of giving decision support to traffic engineers, a solution method must be interpretable and require infrequent adjustments. In addition, the solution method must address the challenges of managing long-term effects and navigating an extremely large number of scenarios that are computationally infeasible to solve through traditional mathematical iteration.

The primary objective of this research is therefore to design a way to give decision support to traffic engineers that fulfills the aforementioned requirements. A solution method that Technolution is interested in is Reinforcement Learning (RL) for its ability to make sequential decisions under uncertainty while accounting for long-term effects. By combining RL with deep learning as Deep Reinforcement Learning (DRL), it can use a neural network to generalize and efficiently predict the long-term effects given a state (Li, 2018). This research aims to use the strengths of DRL to provide traffic engineers with interpretable recommendations for managing traffic efficiently.

1.3 Research questions and approach

This study investigates the ability to use the theoretical advantages of DRL for traffic optimization while overcoming challenges such as interpretability and scalability. In accordance with the objective of the research, the following main research question will be answered:

> ***How suitable is deep reinforcement learning for providing real-time decision support to traffic engineers for reducing congestion across multiple intersections in an urban traffic network?***

To answer the main research question, additional questions and subquestions are formulated. These are as follows:

- How is traffic management used to improve the desired flow of traffic?
  - What are commonly used techniques in city traffic management?
  - What are the challenges in traffic management?
- How has artificial intelligence been applied in traffic management according to the literature?
  - How is supervised learning used for traffic management?
  - What makes RL a suitable solution for city traffic management?
  - What are the main challenges of RL in city traffic management?

First, a literature study will be conducted to gain a better understanding of traffic management and to study the application of artificial intelligence in traffic management. This literature review covers the subquestions of questions one and two. The literature study ends with a short conclusion to answer questions one and two: ”How is traffic management used to improve the desired flow of traffic?” & ”How has artificial intelligence been applied in traffic management according to the literature?”.

- How can the solution method be designed to reduce traffic congestion while being able to provide decision support to traffic engineers?
  - What are the key elements of the traffic congestion problem?
  - How can the problem be formulated as a MDP?
  - How can the traffic congestion problem be solved?
  - How will this solution be given as advice to traffic engineers?
  - What RL method is most suitable for solving this city traffic management problem?

Second, the findings of the literature will be used to model the problem and find a fitting solution. This chapter will include a model of the problem in terms of a Markov decision process (MDP), as well as a solution method that can be applied to solve the problem and the tools needed for this. The chapter on the modeling approach concludes with an answer to question three ”How can the solution method be designed to reduce traffic congestion while being able to provide decision support to traffic engineers?”.

- How should the RL algorithm be applied to optimize traffic flow?
  - What modeling choices and assumptions need to be made?
  - How will the RL algorithm be verified?
  - How can computational time be kept at a minimum?

Third, after modeling the problem and the solution method, the solution method will be applied. The traffic simulator Simulation of Urban MObility (SUMO) will be used to first simulate traffic and second to understand how traffic policies influence flow under various circumstances. Using the traffic simulator, many phase durations and traffic intensities are simulated to gather data to train a neural network. This trained neural network is then used for iterative supervised learning where the neural network decides the best policy given a traffic state. The newly generated data is also used to fine-tune the existing neural network. After this approach, DRL will be applied to find the best action given a traffic state. The chapter on applying the reinforcement learning algorithm concludes by answering question four: ”How should the RL algorithm be applied to optimize traffic flow?”.

- How well do supervised learning and the RL algorithm perform?
  - How well can a neural network predict future traffic congestion?
  - How well can a neural network be applied to modify traffic light configurations to improve the mean speed?
  - How does the performance of the RL algorithm compare to the original traffic performance in terms of mean speed?
  - What are the limiting factors for implementation?

Fourth, the performance of the neural network as well as the RL algorithm are analyzed. To do this, the prediction accuracy of the neural network is determined. In addition to evaluating the performance, in terms of mean speed, when the neural network is applied to change the traffic light configurations. The performance of the RL algorithm will be compared to a benchmark policy for validation. These results will answer the subquestions of question five. The chapter on experimental results provides an answer to question five: ”How well do supervised learning and the RL algorithm perform?”.

Lastly, the conclusion presents the answer to the main research question: ”How suitable is deep reinforcement learning for providing real-time decision support to traffic engineers for reducing congestion across multiple intersections in an urban traffic network?”, as a final result. This will be done by reflecting on the results of the previous chapters and their research questions and subquestions.

1.4 Outline of the report

Chapter 2, Literature study, gives insights into the challenges and solution methods within traffic management. As well as how supervised learning and DRL are applied in traffic management. The chapter will conclude by answering the first two research questions. Chapter 3, Approach for modeling the problem and the solution method, includes a model of the problem as well as an in-depth explanation of the approach used to solve this problem. The chapter concludes with an answer to the third research question. Chapter 4, Experimental settings, covers how the solution method is implemented. Including all the steps taken to get towards the final solution. The chapter concludes with an answer to research question four. Chapter 5, Experimental results, covers the results found during this research. This covers the training of the neural network, the iterative supervised learning approach as well as the DRL approach. The chapter will conclude with an answer to the final research question. Chapter 6, Discussion, will discuss the findings of the results including limitations and challenges. Chapter 7, Conclusion, will summarize the research and the found results, rounding off by answering the main research question. The research finishes off with Chapter 8, Recommendations and future research, in which the author outlines potential directions for future research as well as recommendations for Technolution.

2 Literature study

This chapter aims to gain a better understanding of the challenges within traffic management and to gain familiarity with solution methods for traffic management problems. Additionally, the chapter is used to gain insight into how supervised learning and reinforcement learning are used in traffic management. The chapter rounds off by answering subquestion 1 and 2: ”How is traffic management used to improve the desired flow of traffic?” & ”How has artificial intelligence been applied in traffic management according to the literature?”.

2.1 Challenges and solutions for traffic management

The first step towards solving a traffic management problem is to understand the challenges encountered in traffic management. The major challenge in traffic management is that the number of vehicles increases faster than the infrastructure in place supporting it. This leads to road congestion which has various consequences: Poor road user experience, loss of time, a decrease in productivity, and an increase in pollution (Ouallane et al., 2022). Ouallane et al mention two main principles to combat traffic congestion: Traffic guidance and traffic light control. Traffic guidance includes different methods for finding alternative routes for vehicles with the aim of preventing congestion. Traffic light control includes several methods to improve traffic flow at intersections and thus also aims at the reduction of congestion. Further, Ouallane et al show that researchers tend to propose traffic light control solutions for isolated intersections. However, for the solutions to be efficient, coordination between intersections is important to have an understanding of local fluent flow but also a global fluent flow. Ouallane et al conclude their work by recommending the combination of modern sensor technology to gather data and artificial intelligence to optimize efficient road traffic management.

In the Dutch booklet Handboek verkeersmanagement – Module Regelaanpak (CROW, n.d.) they mention four possible actions a traffic manager can take to solve congestion. These actions are: Redirecting traffic, reducing inflow, promoting outflow, and informing road users. All four options can cause follow-up problems as traffic is changed. Two possible follow-up problems can be overloaded alternative routes and creating new bottlenecks. Consequently, this challenge will have to be addressed as well. Some solutions for redirecting traffic may be a good solution but are not allowed in The Netherlands (CROW, n.d.). An example is that traffic is not allowed to be redirected in a way that disturbs the cycling traffic of kids going to school in urban areas. An additional challenge that traffic engineers have to deal with is that although road users can be informed of upcoming traffic situations, they might not see the sign or simply choose to ignore it. The combination of the above-named challenges suggests the complexity of solving traffic problems and that solutions can require multiple steps.

Ravish and Swamy (2021) did research into intelligence-based algorithms, and computational methods that mimic human or natural intelligence, using artificial intelligence, to solve tasks. Ravish and Swamy found that intelligence-based algorithms, used to mitigate congestion, require a large time to compute. The computational time for Ant Colony Optimization for a network of 142 links is found to be almost 14 hours (Cong et al., 2013). Ravish and Swamy found that genetic algorithms solve some issues such as tuning parameters and handling of objectives and constraints. However, this leads to even higher computational costs than ant colony optimization. Swarm optimization is another intelligence-based algorithm that is used for traffic optimization. Shi et al. (2021) applied swarm optimization to optimize a single large intersection. They were able to reduce the waiting time by 20%. However, the computational times are again long, ranging from 5 to 14 hours depending on the traffic intensity that is simulated.

A relatively simple technique for optimizing local traffic lights is the use of the max pressure (MP) algorithm (Varaiya, 2013). The algorithm is used for deciding the duration of green time per phase. A phase is defined as a possible combination of green and red lights at an intersection. A commonly used approach to determine phases for traffic lights is the ring-barrier approach. This approach gives eight possible phases as can be seen in Figure 1. These phases can transition into each other. The implementation of the ring barrier approach works as follows: The traffic light cycle starts with phases 1 and 5 concurrently. When phase 1 ends, phase 2 can start, while phase 6 waits for phase 5 to finish. The dotted lines shown in Figure 1 indicate permissive phases. In contrast to protected phases which always have priority, permissive phases are allowed to go if there is a gap but must give priority to conflicting phases. Returning to the working of max pressure, the choice of the green duration of a lane is based on the queue of that lane, which is interpreted as pressure, at the intersection as well as the queues of the links around the intersection (Tsitsokas et al., 2021). This is shown in Figure 2. The lane which has the most pressure is then given green.

Varaiya (2013) mentions three advantages of MP. First, since MP is for local intersection control it only requires the queue length of the lanes at the intersection and the queue lengths of the links. As a result, the communication requirements between intersections are low. There is also no need to change other intersections as a direct result of the action at one intersection. Second, MP adjusts phase durations to a shift in demand without the need to predict or estimate demands. Third, normal MP requires knowledge of turn rations and saturation flow rates. However, by implementing the adaptive MP method, those parameters can be estimated, further simplifying the implementation of max pressure.

For a larger scale of traffic management, perimeter control can be implemented (Tsitsoka et al., 2023). This method ”guards” an area enclosed by a perimeter. The guarding behavior can be explained as a perimeter line that moderates the incoming traffic such that a desired traffic outcome in the area is achieved. This can be a limited amount of congestion or a minimum speed. An example of this application is in the Maas tunnel in Rotterdam. Here multiple layers of perimeter control are implemented to prevent the average tunnel speed from dropping below 15 km/h (Technolution Move, 2023). This is achieved by moderating inflow at various intersections surrounding the tunnel. Here the goal is not to improve overall traffic flow but rather to fulfill the must-have safety requirements for driving in the tunnel such that cars are not slower than the smoke or toxic fumes in the tunnel in case of a calamity. The reason for this is that because the tunnel is not allowed to have visible ventilation at the entrance as it is a national monument, it uses longitudinal ventilation. This will push the smoke at a speed of 15 km/h. If traffic were to move more slowly than this, cars could become stuck in the smoke, creating a safety hazard. The PC approach used for the Maastunnel can also be applied to cities. By reducing the in-flowing traffic on the outskirts of the city, the congestion inside the city can be reduced.

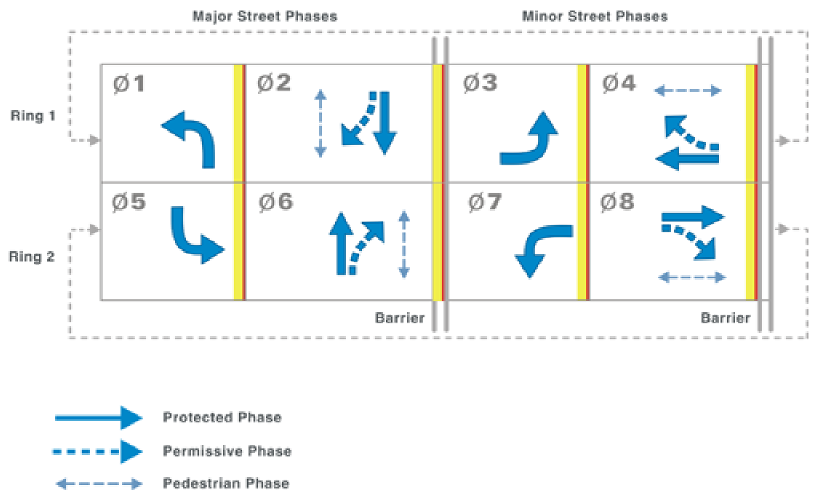


Figure 1: 8 phases in the ring barrier traffic signal control (`https://ops.fhwa.dot.gov/publications/fhwahop08024/chapter4.htm`).

2.2 Supervised learning in traffic management

When trying to solve a problem that changes over time it is important to know how the problem will evolve. This is also the case for traffic management as Tampubolon and Hsiung (2018) state: Traffic management systems rely on the ability to predict traffic flows. Tampubolon and Hsiung trained a neural network to predict traffic for a day in the future. They use the past 3 days of traffic as well as the historical traffic of that day. They found good performance on training data but the neural network struggled to generalize and would instead overfit on training data. The result is that the validation performance was rather poor. They predict that the small size of the training data is the cause of this. To prevent the neural network from overfitting they added dropout regularization. This technique will randomly choose to ignore a percentage of the neurons. As a neural network trains the neurons can specialize, developing specific roles in the neural network. This can cause neighboring neurons to depend too heavily on this specialization. The result is that complex co-adaptations are formed which causes the model to be overly specific for the training data. By randomly removing neurons this effect can be prevented and instead, neurons will be more independent preventing overfitting (Sanagapati, 2019). After applying the dropout technique the neural network was able to generalize better. Tampubolon and Hsiung found that the availability of data was a great limitation and more data should improve the performance. They also state that the long time required for training a deep neural network on a large amount of data is time-consuming.

A review of the applications of deep learning in traffic management by Patil (2022) mentions that deep learning applications are seeing an increase in uses in traffic management. The applied areas are traffic prediction, object detection and recognition, autonomous vehicles, traffic flow optimization, and parking management. Traffic prediction is the most used application, enabling traffic engineers to anticipate congestion. Despite positive results, Patil states that traffic engineers can struggle to interpret why deep learning models are making certain predictions. In addition, traffic managers experience difficulties adjusting the parameters of the deep learning models to

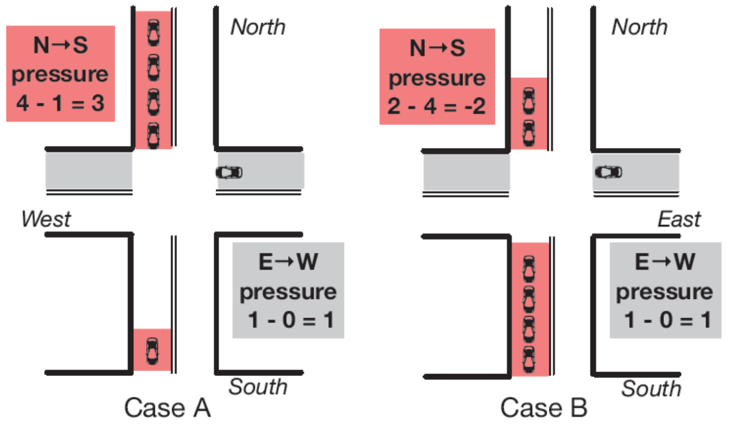


Figure 2: Illustration of max pressure control.
Illustration of max pressure control. In Case A, the green signal is set in the north-to-south direction. In Case B, the green signal is set in the east-to-west direction (Wei et al. 2019a).

increase performance. This can limit the effective use of deep learning applications. In accordance with results found by Tampubolon and Hsiung, Patil also mentions that the need for a large amount of data is a limitation of deep learning. A solution given by Patil is to continuously train the model on new data to increase accuracy.

2.3 RL in traffic management

After being able to predict traffic using a neural network the next step is to perform actions taking into account the long-term effects that are predicted, this is the field of Reinforcement Learning (RL). RL is useful in cases where there is a stochastic environment with sequential decision-making. This is very fitting to traffic as traffic is rapidly changing and decisions will have a long-term effect (Inoue et al., 2024). The Markov property is also respected as the future state (congestion levels, vehicle positions, and traffic flows) depends only on the current state and the decisions made. With these criteria being met, RL should be a fitting solution method to solve traffic congestion problems.

There are multiple cases found that apply a form of RL to optimize urban traffic control. The results are promising as waiting time is reduced (Hegde et al., 2024). Lin et al. (2018) show that a DRL algorithm can outperform traditional rule-based approaches. They suggest further research to be done regarding methods to transform traffic networks into state representations that a neural network can effectively process.

McCluskey et al. (2016) mention the potential for RL in traffic signal control. These traffic signal control cases have been researched and worked out various times applying forms of DRL that yield positive results (Li et al., 2021; Genders & Razavi, 2016; Zheng et al., 2019). The results found the delays to be reduced by 14 to 87% as well as an average queue length reduction of 17 to 73% (Rasheed et al., 2020).

A challenge when applying RL to traffic management is transforming the traffic conditions into a state space. This is difficult because the state space can become too vast to learn (Lin et al., 2018). The data collection for the agent to train on for an online learning approach in itself is also considered a challenge. A limited amount of data is available whereas a very large amount of data is required to train a DRL algorithm effectively. For offline learning via a traffic simulator, there are concerns about how accurately the reality is simulated (Han et al., 2023). Lin et al. (2018) also mention the accuracy of traffic models to be a limiting factor for the implementation of RL. However, there is active research being done on developing traffic simulation tools specifically to train DRL algorithms to optimize traffic (Wu et al., 2017; Zhang et al., 2019).

Inoue et al. (2024) mention that artificial intelligence methods using neural networks and reinforcement learning perform well for rapidly changing traffic patterns. However, the large amount of control variables and model parameters makes it computationally intense when applied to a network of more than a few traffic lights. A way to solve this is by reducing the amount of control variables. Instead of trying to optimize every local traffic light that is computationally intense, optimize a simpler control parameter such as modifying flow in a certain direction. This should also require less micro-scale information such as queues for each lane of each traffic light but rather macro-scale parameters that describe the congestion of a network. By limiting the dependent variables and simplifying the action it should be easier to learn for a reinforcement learning algorithm which in turn reduces computational time.

2.4 Formulation of RL approaches

Most RL approaches in the literature include a Markov decision process (MDP) of the problem. The components of the MDP are as follows: State, action, reward, and transition. The transition component is not covered here as all approaches use a traffic simulator for the state transitions. These findings are respectively shown below in Table 1, 2, and 3. Note that these formulations are for general traffic light optimization where many studies focus on a single traffic light.

State formulations	
State	Source
Number of vehicles waiting at each incoming lane except the right lane and the phase of the traffic light.	X. Huang et al., 2021
Current and previous speed limit as well as average speed and density of highway.	E. Walraven et al., 2016
Vehicle density, speed of vehicles, direction of vehicle movement, distance between intersection and vehicle, number of edges (segments) in a lane, and last traffic signal in a lane.	Paul & Mitra, 2020
Queue of each lane at a point in time at an intersection with k incoming lanes. As well as the phase of the traffic light.	J. Gu et al., 2021
Queue length, phase of the traffic light, time of day, and day of week.	M. Muresan et al., 2021
Incoming traffic in a given phase. As well as the number of incoming vehicles per lane.	R. Zhu et al., 2023
Incoming and waiting traffic per lane.	Y. Wang et al., 2022

Table 1: Overview of state formulations.

It can be observed from Table 1 that the states are generally described as the number of vehicles waiting at the traffic light as well as the current phase of the traffic light. Some sources also add additional information to the state such as the speed of the vehicle or a time index. This describes the state more accurately but also increases the dimension of the state space.

Action formulations	
Action	Source
Alternation and duration of traffic lights on each incoming approach. As well as the phase selection for the next period.	X. Huang et al., 2021
Speed limits. *Note that this is applied for highways.	E. Walraven et al., 2016
Turn a traffic light on or off.	Paul & Mitra, 2020
Keep the current phase or move to the next phase.	J. Gu et al., 2021
Keep the current phase or move to the next phase.	M. Muresan et al., 2021
Change the times of the phase cycle.	R. Zhu et al., 2023
Switching to the next phase or predefined set of phases.	Y. Wang et al., 2022

Table 2: Overview of action formulations.

It can be observed from Table 2 that most approaches have the changing of traffic light phases as action while respecting the constraint of a minimal green time. In contrast to most cases in the literature that use frequent actions for one traffic light, this research instead aims for infrequent actions over a network of traffic lights.

Reward formulations	
Reward	Source
Negative reward for queue length per lane	X. Huang et al., 2021
Number of vehicles per hour. Punishment if the number of vehicles per hour decreases. *Used for highways.	E. Walraven et al., 2016
Positive/negative reward for increasing/decreasing waiting time	Paul & Mitra, 2020
Number of passed vehicles divided by the number of stopped vehicles	J. Gu et al., 2021
Vehicles discharged * a reward x – vehicles waiting * a reward y.	M. Muresan et al., 2021
Individual reward: Number of vehicles waiting per lane per stage. Global reward: Average vehicles waiting on all lanes per phase. *Multi-agent.	R. Zhu et al., 2023
Negative reward for all the combined queue lengths	Y. Wang et al., 2022

Table 3: Overview of reward formulations.

It can be observed from Table 3 that most sources choose queue length as the reward. Some have tried to formulate this differently by giving a reward to a ratio of vehicles that drive through the traffic light or use weights for passing and waiting vehicles. For the weights approach, the reasoning was to improve learning speed by giving an additional negative reward when vehicles wait.

2.5 Conclusion literature study

Traffic increases at a faster rate than the infrastructure supporting it which causes congestion. Traffic management aims to reduce congestion as well as optimize the use of road infrastructure. Techniques such as traffic light control and perimeter control help manage traffic by either guiding vehicles to alternative routes or regulating inflows to prevent congestion. Techniques such as max pressure control enable localized decisions, based on real-time data to improve traffic flow at intersections. A challenge for traffic management is the stochastic nature of traffic. First, the future traffic state must be predicted. Second, given the prediction, a solution to the traffic problem must still be found which takes long-term effects into account. A solution approach to the first challenge, predicting traffic, is supervised learning using neural networks. Neural networks allow for the generalization of data which in turn enables predicting the future state of traffic. The results are promising but the need for large amounts of data remains a challenge. A solution approach to the second challenge, finding a traffic management solution that takes long-term effect into account, is Deep Reinforcement Learning (DRL). DRL can use the strengths of neural networks and Reinforcement Learning (RL) to make sequential decisions under uncertainty with long-term effects. DRL has promising applications in traffic management as it has been shown to reduce waiting times, delays, and queue lengths. However, challenges such as scaling in large traffic networks, the high computational costs, and the accuracy of simulated training environments must be addressed for widespread implementation.

3 Modeling the problem and the solution method

This chapter will give additional insights into the problem and the requirements for the solution. Followed by explaining which tools are used to solve the problem. Rounded off by a solution method and the steps required to work towards that solution. This chapter will conclude by answering the following research question: ”How can the solution method be designed to reduce traffic congestion while being able to provide decision support to traffic engineers”.

3.1 Breakdown of the problem

In simple terms, the problem is that there is too much traffic within cities, which causes congestion. Traffic engineers in essence have two options: First, do their best to solve this problem reactively by diverting traffic or moderating in/out flows. A second alternative is to program a rule-based scenario that consists of many if statements. This rule-based approach will then apply preset solutions if certain traffic conditions are met. Correctly applying this to a whole city is cognitively hard, a lot of work, and difficult to oversee and maintain. A solution method for optimization problems is to mathematically compute an optimal solution for each possible scenario. However, this is not computationally feasible for traffic management due to the stochastic nature of traffic and the sequential decision-making with long-term effects. Consequently, mathematical optimization is not possible. Therefore, a different approach must be used to predict and prevent congestion in a prescriptive manner.

3.2 Markov decision process

As already seen in the literature, RL problem formulations often include a Markov decision process (MDP). This sub-section will go over the MDP that is used, the result can be seen below. Note that multiple variations of state space formulations, actions, and rewards have been considered in this research.

- **State:** The state contains information to represent the current state of the environment. This is a vector containing: Average lane occupancy, total vehicles, total vehicles inside the perimeter, total vehicles outside the perimeter, average speed, total queue length, average queue length, the queue outside the perimeter, the queue inside the perimeter, total travel time, average travel time, inflowing vehicles and outflowing vehicles.

The state space can be mathematically represented as follows:

State (S): The state $s \in S$ is a vector that contains the relevant traffic information in the network at time t:

$$s_t = \begin{bmatrix} \text{average lane occupancy,} \\ \text{total vehicles,} \\ \text{total vehicles inside perimeter,} \\ \text{total vehicles outside perimeter,} \\ \text{average speed,} \\ \text{total queue length,} \\ \text{average queue length,} \\ \text{queue outside perimeter,} \\ \text{queue inside perimeter,} \\ \text{total travel time,} \\ \text{average travel time,} \\ \text{inflowing vehicles,} \\ \text{out flowing vehicles} \end{bmatrix}$$

Each element in s_t is a scalar that represents a specific traffic feature at time t. This should provide the required information for the neural network to understand the environment.

Throughout this research addition state space formulations are attempted to find the best formulation. The additional state space formulations that are considered include adding more information such as queues for each lane as well as less information by using only a part of the aforementioned state space. The state space seen above is most the successful. Additional details on state space experimentation can be found in Section 4.3.

- **Action:** The action is a vector containing scalars with which the respective phase duration will be multiplied. For example, if the first element is equal to 0.75 the original green time of the first phase (28 seconds) will now become 28*0.75 = 21 seconds.

This can be mathematically formulated as follows:

Action (A): The action $a \in A$ is a vector that adjusts the green phase durations for traffic lights in the network. Let $a = [a_1, a_2, a_3, a_4]$, for each a_i, with $i = [1, 2, 3, 4]$, be a multiplier for the duration of the original phase d, with $d = [28, 11, 28, 11]$. The new phase duration c is then computed as follows:

$$c = d \times a = [d_1 \times a_1, d_2 \times a_2, d_3 \times a_3, d_4 \times a_4]$$

Similar to the state formulation, multiple action formulations are attempted throughout this research. Simpler versions such as only adjusting the main phases' duration or choosing a ratio between the main phases have been tried. The action formulation above is the most intuitive formulation and yields similar results to the other formulations.

- **Reward:** The reward is the mean speed over the whole network.

The reward can be mathematically formulated as follows:

Reward (R): The reward $R(s, a)$ for a state s and action a is the mean speed of all vehicles in the network, which can be represented as:

$$R(s_t, a_t) = \frac{1}{N} \sum_{i=1}^{N} v_i$$

Where N is the total number of vehicles in the network at time t, and v_i is the speed of vehicle i.

This research also experimented with various reward formulations, using the mean waiting time, trip completion rate, and weighted combinations of the average speed, mean waiting time, and trip completion rate. The reward above is the best-performing reward and also the most intuitive.

- **State transitions:** The state transitions will be handled by SUMO (Simulation of Urban Mobility), a traffic simulator. SUMO will perform the steps in the simulation which moves all the cars to the next location as well as controlling the traffic lights. After many steps, the next state is forwarded to Python. Here the DRL algorithm chooses an action based on the state. This action is then sent to SUMO, which in turn will adjust the phase durations of the traffic lights on the perimeter accordingly.

The state transitions can be mathematically formulated as follows:

State Transitions ($P(s'|s, a)$): The state transitions are managed by SUMO, which updates the environment. Formally, the transition probability $P(s'|s, a)$ represents the probability of moving to a new state s' given a current state s and action a. Since the exact transition probability is complex and done by SUMO, it is described as:

$$s_{t+1} = f(s_t, a_t)$$

Where f is a function (simulated by SUMO) that takes the current state s_t and action a_t and outputs the next state s_{t+1} based on vehicle movements, traffic light changes, and other dynamics in the simulation.

- **Stages:** The duration between actions is kept at 90 seconds, the time of a traffic light cycle.

The goal of this research is to provide decision support at a larger time scale of 15 minutes, however, this is not feasible due to computational reasons. Only making a decision every 15 minutes would reduce the samples per iteration by $(15 * 60)/90 = 10$. As a result, training the reinforcement learning algorithm will take roughly 10 times as long. So for simplicity, 90 seconds is taken, if this approach works it is worth experimenting with a stage of 900 seconds.

3.3 Requirements of the solution method

- Technolution puts significant emphasis on wanting to give decision support, thereby augmenting the intelligence of the tool's user. This means that the output solution must be easy to interpret for human users.

- The solution should not have to be applied every couple of seconds, but rather over a larger time interval. The results from the literature research in Section 2.4 show that most approaches focus on optimizing a single traffic light. The results would then output a phase selection every 5 seconds. Although this approach is proven to be effective the results might not be easy to grasp. In addition, an output is given in a very short time interval. This makes it not suitable for decision support as a traffic engineer will not input direct traffic phase durations. Therefore, a macro-scale approach is chosen to optimize a network, instead of looking at the micro-scale optimization of a single traffic light. The goal here is to output an intuitive solution over a longer time horizon.

- The solution method must also be able to deal with the complex nature of traffic management. As well as finding an alternative to the mathematical approach of iterating the solution for every possible traffic situation. Since this is not feasible with the size and stochastic nature of the problem.

In summary, the solution method should be easy to interpret, over a long time interval and find an alternative to mathematical computation for every situation.

3.4 Solution method to solve the problem

This subsection will first discuss the tools that are used, followed by the solution method to solve the two challenges of traffic management found in the literature. The first challenge is to predict traffic due to its stochastic nature. The second is finding actions, which solve the traffic problem that take into account long-term effects. Neural networks are chosen to solve the first challenge, predicting traffic, for the following abilities: To learn complex problems, being able to handle large state spaces resembling traffic, adaptability to real-time data, and the ability to make fast predictions once trained. Deep Reinforcement Learning (DRL) is chosen as the solution method to the second challenge, finding actions that solve traffic problems that take into account long-term effects. DRL is chosen for the following abilities: Scalable to large networks using neural networks to approximate a policy as well as the ability to make decisions under uncertainty while taking long-term effects into account. In addition to the ability to continuously train and improve on new data and the ease of using a simulation tool to train in. The steps in which these tools are used are described in the next paragraph.

The idea is to use DRL to find the best action to prevent future congestion. The output must be intuitive to understand and be applied over a larger time period, given the requirements stated in Section 3.3. As previously mentioned, this eliminates single-traffic light optimization. A good solution method for a larger area is perimeter control. Limiting the inflow to an area so that the traffic remains below a certain threshold, can be seen in Figure 3. The red line marks the protected region on which perimeter control will take place. Along the red line, the flow into and out of the city will be moderated. Given that in this case a neural network can predict the future traffic inside a city with sufficient reliability, a DRL algorithm can find the optimal action for a traffic state to achieve the highest reward. The reward that is chosen to be optimized is the mean speed across the entire traffic network, which will be maximized. The DRL algorithm should then give as output an inflow/outflow moderation. This will be a vector that has an element for each phase (excluding yellow). The elements in this vector represent a percentage increase or decrease per phase. This is intuitive to understand as it simply tells a traffic manager per direction if the durations should be increased/decreased. In addition, the action of modifying phase durations would not need to be applied every 5 seconds like local traffic light optimization. This is because the inflow/outflow of vehicles over 5 seconds will not significantly impact the average flow of the whole city network. The approach of perimeter control thus covers all requirements as the output is intuitive and over a longer time period.

The problem that does arise is that local optimization of traffic lights is still required. Simply moderating the duration of the inflowing phase for every single traffic light on the perimeter by a fixed amount is not optimal. In reality, traffic is not uniformly distributed at every entry point of the network. The result of simply increasing green time uniformly for all traffic lights in a direction would be wasted green time. To solve this problem and prevent increasing green time unnecessarily, a second control layer is added: Max pressure (MP). MP will regulate the traffic lights at a local level to prevent a waste of green time at local traffic lights. Meanwhile, perimeter control will regulate the congestion within the city on a global level by adjusting in and out flows at the perimeter. While perimeter control prevents congestion by increasing or decreasing global flows, MP makes sure that these flow modifications are being used in the most efficient way possible at a local level per traffic light. These methods are then combined to have both local and global optimization of traffic. See Section 2.1 for more detailed information on the working of max pressure and perimeter control. This approach will give an understandable output to the traffic manager every 15 minutes.

To summarize the main challenges and requirements with the respective design choices for the solution method:

- Since traffic is complex, stochastic, and changes have long-term effects, the solution method must be able to deal with this challenge. DRL is chosen as the tool to solve the problem as it combines the strengths of neural networks and reinforcement learning to deal with sequential decision-making under uncertainty in complex state spaces.
- Since the output is used for decision support, it must be easy to understand. To achieve this, the percentage adjustment per phase is provided as an action, which should be intuitive and straightforward for traffic engineers.
- The output should also be over a longer period of time to prevent the traffic engineer from having to make constant changes. To achieve this, a macro optimization approach is chosen due to the longer time periods between actions. The specific method is perimeter control, adjusting the flow from and to the city for the traffic lights on the perimeter.
- Since perimeter control uniformly adjusts traffic at a global level and traffic is not uniform, a form of local optimization is required to prevent wasted green time. This will be done by max pressure to ensure that the adjusted in and outflows are well distributed.

- The goal in traffic is to have everything moving as fast as safely possible. A good measure for this is the mean average speed in the whole network. The mean speed is also measurable in the real world. For these reasons, this has been chosen as the optimization parameter.

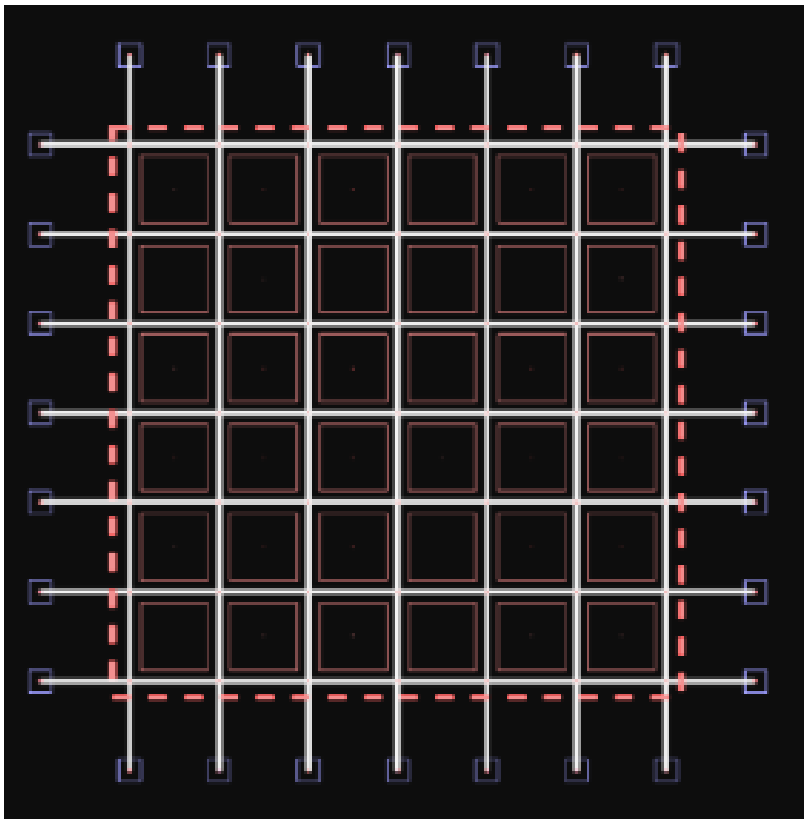


Figure 3: Illustration perimeter control.
Perimeter control and thus adjustment of traffic light phase durations will take place at traffic lights on the red line. Note that the grid network used in this research is 5x5, not 7x7 as seen in the figure (Kampitakis & Vlahogianni, 2024).

3.5 Breakdown of the solution steps

Before implementing the final solution method of DRL with perimeter control and max pressure intermediate steps should be taken. The steps taken to work towards the problem are as follows:

First, supervised learning can be used to predict traffic congestion. In this case, a neural network can be trained to predict a label (future traffic condition) given an input (current traffic condition). In contrast to a mathematical approach, a neural network does not require data for every single traffic possibility. Instead, a neural network is designed to generalize the data by recognizing patterns. This is done for various states to determine the optimal state formulation. Successfully training the neural network to make predictions covers the first part of the problem: Predicting upcoming congestion.

Second, after it is possible to predict congestion, a preventive measure for future congestion should be found. Optimizing sequential actions based on predicted outcomes from a neural network falls under the domain of Deep Reinforcement Learning (DRL). The goal of RL is to find the best action given a certain state by maximizing the reward. DRL in turn is a form of RL that uses a neural network to predict the best action, given a state. This replaces the need for a traditional tabular approach or function approximators. Now, also the second part of the problem is covered: Finding solutions to prevent upcoming congestion.

Third, to verify that everything works as intended a simple action for the DRL algorithm is chosen to begin with. This approach will naively modify the duration of the traffic light phases uniformly for all traffic lights. This action is the action that is presented in Section 3.2. This solution method is not expected to perform very well due to the lack of local optimization but should outperform the static baseline policy.

Fourth, the DRL algorithm will use perimeter control in combination with local optimization. Note that this local optimization is not yet done by the final local optimizer, max pressure, but instead, a different integer linear programming formulation that has been proven to work in combination with DRL (Kampitakis & Vlahogianni, 2024). This is done to again verify that everything is working as intended while working towards the final goal. In this approach, the action made by the DRL algorithm is the amount of cars that are allowed to enter the perimeter during the next time step. The amount of cars is then divided among the intersections on the perimeter depending on the queue present at the intersection. This local optimization is done by solving a set of linear equations.

Lastly, perimeter control is combined with max pressure to optimize traffic at a global and local scale. An in-depth explanation of these two techniques is given in the following two sections as well as explaining how and why both techniques should be combined.

3.6 Max pressure

This section covers how max pressure (MP) can optimize local traffic lights using an optimizer. A complete mathematical notation and an integer linear problem formulation are presented. The max pressure approach uses pressure to describe how much green time a phase requires. Pressure is calculated as follows: The difference between the queue length of the incoming lane and the outgoing lane. A more in-depth explanation can be found in Section 2.1.

3.6.1 Notations

An overview of the notations, adapted from Tsitsokas et al. (2022), can be found below:
N = The set of intersections, elements n
Z = The set of links, elements z
I = The set of incoming links, elements i
O = The set of outgoing links, elements w

D = The set of directions, elements d
J = Phase of the traffic light
C_n = Cycle time
C_z/C_w = Capacity
S = Saturation flow
B = Turning movement rates
G = Green time
k = Control cycle
L = Fixed total loss time

3.6.2 Max pressure formulation

This section covers how MP can be described mathematically and implemented. A mathematical description of the max pressure policy along with the constraints is shown below. These formulas are adapted from Tsitsokas et al. (2022). The constraints are as follows:

$$\sum_{j \in \mathcal{F}_n} g_{n,j}(k_n) + L_n = C_n \tag{1}$$

$$g_{n,j}(k_n) \leq g_{n,j,min} \quad j \in F_n \tag{2}$$

Formula 1 prevents the sum of green durations + lost time (yellow) from exceeding the cycle time. Formula 2 makes sure that the green duration chosen is larger than the minimum green time required. This minimum green time is mandatory for all traffic lights, see (Handboek Verkeerslichtenregelingen 2022, 2022). The amount of pressure $p_z(k)$ of every incoming link $z \in I_n$ of node n for control cycle k is calculated as follows:

$$p_{z,d}(k) = \left[\frac{x_{z,d}(k)}{x_{z,d,max}} - \sum_{d \in \mathcal{D}} \frac{x_{w,d}(k)}{x_{d,max}} \right] S_z, \quad z \in I_n \tag{3}$$

Formula 3 calculates the pressure as the queue length of the current link divided by the link's capacity - the sum of the pressure for the lanes connected to the link. It is important to note that here a lane is a part of the road, not the complete road. Each lane has its own traffic light and pressure. Since negative pressure is meaningless an additional constraint, Constraint 4 is added:

$$p_{z,d}(k) \geq 0 \tag{4}$$

3.6.3 Phase selection

For phase selection two approaches can be used: freedom of phase selection or a predefined order of phases. The first option can be applied as follows: A phase gives green to a certain direction at the traffic light. Some phases can be active at the same time. To choose which phases to activate, the total pressure of the combined possible phases is determined. For example: phases 1 and 5 can have green simultaneously. The total pressure would be the pressure of the south lane in direction left + the pressure of the north lane in direction left. An overview of the visual representation of the phases can be found in Figure 1. The pressure is calculated for all possible combinations of phases. The second option is simpler. It is only necessary to calculate the total time per phase, based on the modeling assumption that the phases follow a fixed order. This is combined with the assumption that every phase will be activated every cycle with a minimal green time. This simplified approach will be considered in this study.

3.6.4 Determining the total cycle length

Determining the green time duration can be done in two ways. The first approach is to set a fixed cycle length where the time is filled up by having each phase active for a certain amount of time. A second possibility is to apply a cycle-free approach where there is no set time in which a cycle of phases must be completed. This creates a lot of freedom but also difficulties. The problem with a cycle-free approach is how to determine the duration of the phase. Keeping a phase for longer allows more traffic to pass but will have other lanes wait more as a direct consequence. Frequent switching of phases, due to a short cycle length, leads to more lost time due to the yellow time. This is a difficult trade-off to optimize. Instead, the simpler approach of a fixed cycle is chosen. Now the challenge is to determine the total cycle time. The method to determine the total cycle time is the Webster method adapted from Yadav et al. (2022). The following notations are used:
N = The set of phases, elements n
C_0 = Total cycle time
L = Lost time per cycle
R = All red time
y = Flow ratio
q = Normal flow
s = Saturation flow rate

This method determines the total cycle time as follows:

$$C_0 = \frac{(1.5L + 5)}{(1 - Y)} \tag{5}$$

With

$$L = 2n + R \tag{6}$$

$$Y = y_1 + y_2 + ... + y_n \tag{7}$$

$$y_n = \frac{q_n}{s_n} \quad \forall n \in N \tag{8}$$

The saturation flow rates are taken from Yadav et al. (2022). These have the following values per width of the lane:

The value for q can be obtained via data analysis. Substituting the values for q and s in Equation 8 and adding these for all phases gives the flow ratio. After filling in Equation 6 to get the lost cycle time, all variables for Equation 5 are known. Filling this Equation in returns the total cycle time.

Width in meters	3.0	3.5	4.0	4.5	5.0	5.5
Saturation Flow (PCU/hr)	1850	1890	1950	2250	2550	2990

Table 4: Standard values for saturation flow, S, for widths 3 to 5.5 meters according to Webster.

3.6.5 Distribution of green time

The distribution of green time can be determined by solving an optimization problem based on a set of equations describing the goal and the constraints. These steps and equations are adapted from Varaiya, P. (2013).

First the green time according to MP is determined. The total green time is the cycle time minus the lost time. This can be described as follows:

$$G_n = C_n - L_n \tag{9}$$

Second, the green time per phase is defined as follows:

$$\tilde{g}_{n,j}(k) = \frac{P_j(k)}{\sum_{i \in F_n} P_i(k)} G_n, \quad j \in F_n \tag{10}$$

Here the pressure per phase is divided by the total pressure to get the proportion of pressure as a result. This is multiplied by the total green time to get the total green time per phase. To ensure that the green time per phase found in Equation 10 also complies with the traffic light guidelines, an additional optimization problem is formulated:

$$\begin{aligned} \underset{G_{n,j}(k)}{\text{minimize}} \quad & \sum_{j \in F_n} \left(\tilde{g}_{n,j} - G_{n,j}\right)^2 \\ \text{subject to} \quad & \sum_{j \in F_n} G_{n,j} + L_n = C_n, \\ & G_{n,j} \geq g_{n,j,\min}, \quad \forall j \in F_n, \\ & |G_{n,j} - g^p_{n,j}| \leq g^R_{n,j}, \\ & G_{n,j} \in \mathbb{Z}^+. \end{aligned} \tag{11}$$

The aim is to have the allowed green time $G_{n,j}$ be as close as possible to the green time per phase found by MP $\tilde{g}_{n,j}$. The constraints are as follows: First, total green time + lost time is equal to total cycle time. This is the same as Formula 1. Second, the green time must be larger than the minimal green time. Third, there is a maximum change of green time $g^R_{n,j}$ allowed between the current cycle $G_{n,j}$ and the previous cycle $g^p_{n,j}$. This optimization problem can be solved by for example Optuna optimizer.

Despite the requirement to give decision support, it is not possible to have a large time step for micro-level traffic control, in contrast to macro-level control. On a micro-level, the time step chosen will be 5 seconds. This means the selected phase will be revised every 5 seconds. During this revision, the pressure for all lanes will be calculated again and a phase will be selected accordingly. The part that will be used for decision support is perimeter control which will be used on a macro scale with a significantly larger time step of 15 minutes. This is covered in the next subsection.

3.7 Perimeter control

The goal of perimeter control is to reduce congestion in the perimeter by changing the green times of the traffic lights on the perimeter. This is done by adjusting the weights for the pressure that the MP method uses to give green durations, the weights are given in Equation 12. Here the weights modify the amount of pressure a lane has, which then indirectly modifies the in and out flow of that lane. To facilitate the addition of weights, the formula for pressure 3 is modified to include these weights w_d:

$$p_{z,d}(k) = w_d \left[\frac{x_{z,d}(k)}{x_{z,d,max}} - \sum_{d \in \mathcal{D}} \frac{x_{w,d}(k)}{x_{d,max}} \right] S_z, \quad z \in I_n \tag{12}$$

When analyzing Equation 12 it can be observed that the pressure is just the difference between upstream and downstream occupancy of the links. By adding a weight to this pressure, it is possible to adjust the interpretation of the pressure by the controller. If a factor smaller than one is used the perceived pressure is less and a shorter green time duration will be provided. This way the traffic into the perimeter is moderated. The opposite holds for weights greater than one. Intersections not on the perimeter will have a weight of one which turns Equation 12 back into Equation 3.

3.8 Supervised learning using neural networks

Supervised learning is a subgroup of machine learning where models are trained on labeled data to learn the relationship between the input features and the target outputs. In this research, it will be applied to predict the future traffic situation based on the current traffic situation. The labeled data is the current traffic situation and the target output is the future traffic situation, in this case, the mean speed. Different state formulations will be tried to find the best state formulation to describe the traffic situation. By using neural networks it is possible to predict the future traffic situation based on the input of the current traffic situation.

3.9 Iterative supervised learning

As described previously, supervised learning can predict a target or label, given a set of input features, which can be used to make decisions. If the traffic light configuration is part of the state vector that describes the current traffic situation, it is possible to predict the best traffic light configurations based on the current traffic situation. Applying this decision-making sequentially to a traffic simulation should improve the mean speed. Doing this process iteratively will continuously improve the mean speed as well as the accuracy of the predictions when using the newfound data to fine-tune the existing neural network. After multiple iterations, the mean speed improvements should decrease and eventually converge. The flowchart of this process can be seen in Figure 4. In conclusion, repeatedly using a neural network to make decisions based on the current state of traffic can be a way of improving the mean speed and thus reducing congestion.

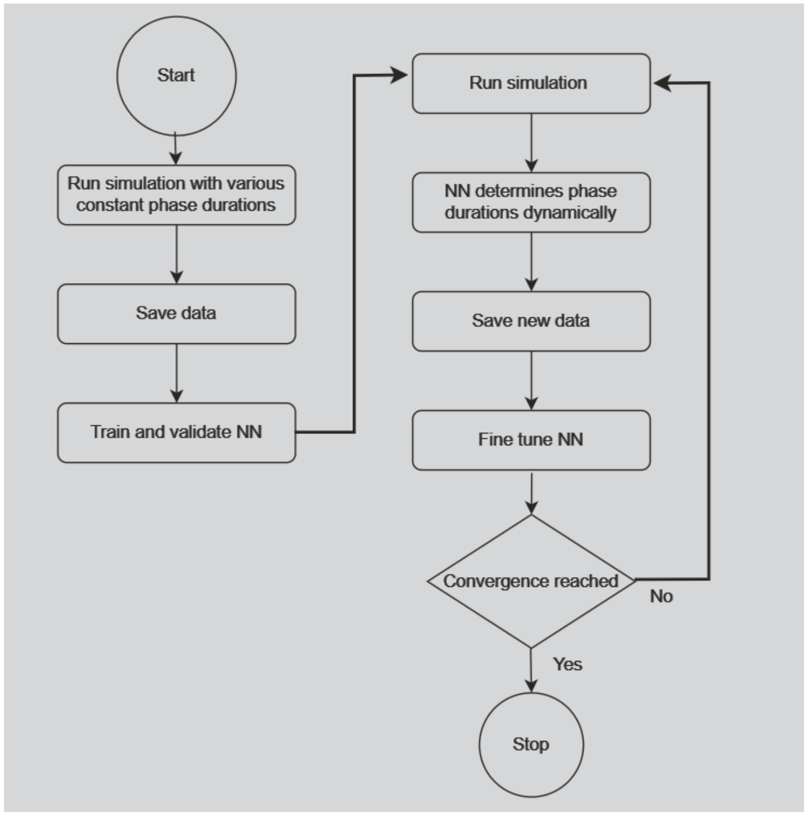


Figure 4: Flowchart for iterative supervised learning approach.

3.10 Combination of perimeter control and max pressure using DRL

After making decisions using a neural network, the next step is to use Deep Reinforcement Learning (DRL) to make sequential decisions under the uncertainty of traffic. The DRL algorithm can be used to output weights that modify the pressure of a lane and in turn the in and out flows. As described earlier, a second layer of control is required for local optimization. This will be done by solving Equation 12 for all lanes at each traffic light. The result of using DRL to formulate weights for the pressures followed by max pressure optimization should lead to global and local optimization of traffic lights.

3.11 Choice of DRL algorithm

The action space of the problem should be a continuous range of values to modify the in/out flow and not be limited to a fixed set of options. As the action space of the problem is continuous, discrete action space RL algorithms are eliminated as a possibility. One of the most popular continuous action space DRL algorithms is proximal policy optimization (PPO) (Schulman et al., 2017). The reason for this popularity is due to the ease of implementation whilst having relatively high performance (Van Heeswijk, 2023). Due to the ease of implementation and allowing a continuous action space PPO is chosen as the DRL algorithm. In addition, twin-delayed deep deterministic policy gradient(TD3) is applied as well. The PPO and TD3 implementations are taken from the StableBaselines3 library (Raffin et al., 2019).

3.12 Concluding approach for modeling the problem and the solution method

The difficulties that are found in solving traffic congestion problems is the ability to effectively predict and solve them. A solution for this is using a neural network to predict the future traffic state and in turn congestion. By using deep reinforcement learning (DRL) the best action can then be found given a traffic state. The DRL problem formulation can be described as follows. The current traffic conditions in the network are used as an input state. The in- and outflow moderation as action. The mean speed throughout the network as the reward. The traffic simulator SUMO will be used for the state transitions. The stage duration will be 90 seconds.

To make sure that the solution provided by the DRL algorithm is over a larger time period, global optimization of the city network is done, as global optimization uses larger time intervals between actions. The global optimization method used is perimeter control. To prevent wasting green time on the boundary of the perimeter, perimeter control is combined with max pressure. See Figure 5 for a flow chart showcasing how the phase durations are determined and Figure 6 for the flowchart showing how the process is done during simulation. Max pressure will optimize local traffic lights while perimeter control optimizes global traffic lights. The DRL algorithm will regulate in- and outflow of the perimeter. This solution is both intuitive and operates over a longer time scale, aligning well with the solution requirements.

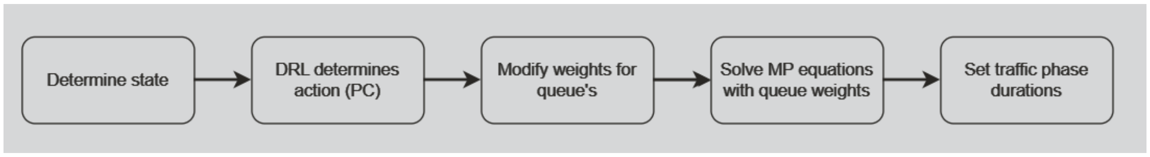

Determine state
DRL determines action (PC)
Modify weights for queue's
Solve MP equations with queue weights
Set traffic phase durations

Figure 5: Flow chart displaying process for determining phase durations.
The DRL algorithm does PC by modifying the weight of queues which will change how heavily each lane contributes to the phase duration in the calculation for MP, see Equation 12.

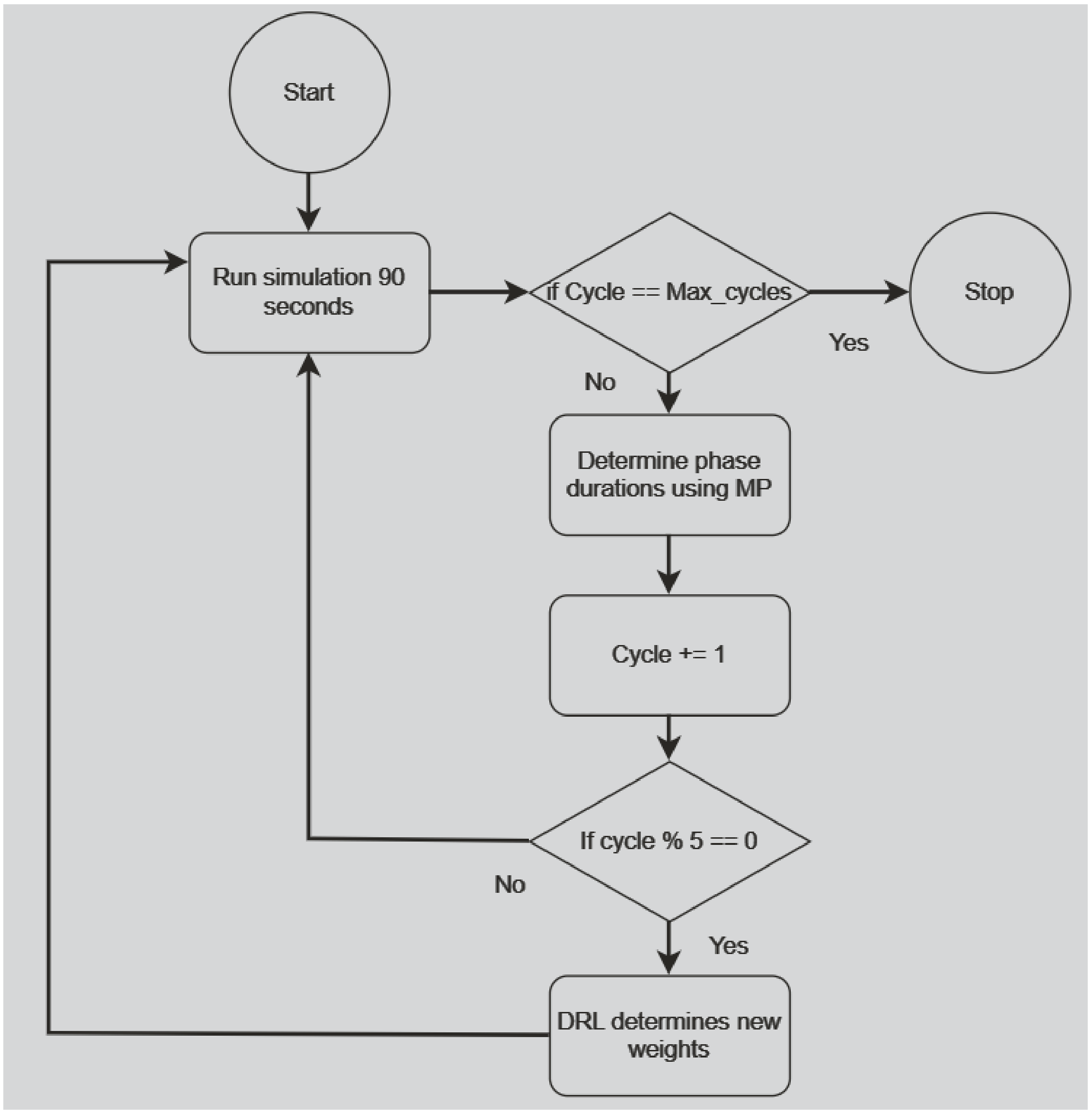

Start
Run simulation 90 seconds
if Cycle == Max_cycles
Yes
Stop
No
Determine phase durations using MP
Cycle += 1
If cycle % 5 == 0
No
Yes
DRL determines new weights

Figure 6: Flow chart displaying the decision-making during the simulation. Showcasing the constant use of MP at the local level and the occasional modification of queue weights by PC determined by the DRL algorithm.

4 Experimental settings

This chapter covers the experimental settings and what choices are made. The development of the solution method is done step-wise and starts with training a neural network on traffic data to predict future traffic. This neural network is then used to choose the best phase durations during a simulation to represent DRL in a simple way. Afterward, DRL is used with a form of naive perimeter control. Here the phase durations of the traffic lights on the perimeter are all changed equally without the use of local optimization. The before last step is to recreate the perimeter control approach from Kampitakis & Vlahogianni (2024) with a local optimizer, different from max pressure. The final step is to combine perimeter control with max pressure. These intermediate steps are done to slowly increase complexity while maintaining insights into the changes and possible problems. The chapter concludes by answering the following research questions: ”How should the RL algorithm be applied to optimize traffic traffic flow?”.

4.1 Modeling choices and assumptions

For simplicity, the traffic network is modeled as a square grid network. A grid network might not be as common in Europe but these are commonly found in America and thus represent a real-world traffic network. The network consists of 25 traffic lights connected by 200-meter double-lane two-way roads. These network settings are chosen to match the network used by Kampitakis and Vlahogianni (2024), except a bit smaller using a 5x5 network instead of a 7x7. The network is created in Simulation of Urban Mobility (SUMO), a commonly used traffic simulator. The network can be seen in Figure 7. Python will be used to interact with SUMO.

Traffic will be created to arrive and depart throughout the entire network to represent a real city. Traffic will follow a thousand randomly generated routes. However, this process is not entirely random. To prevent certain roads from being disproportionately included in a significantly higher number of routes, and thereby experiencing increased traffic, the following solution is implemented: Giving probability weights that decrease the probability of selecting frequently used roads in the random route-generating process. The result is that the likelihood of roads being disproportionally included is significantly reduced. This makes the traffic a bit more uniform throughout the city without being exactly evenly distributed. Roads are on average included in 30-40 routes. This will make sure that all parts of the network are used. The majority of the traffic will be from the outside of the perimeter to the inside of the perimeter. This is chosen to represent a morning rush where people from outside the city go to work in the city. The arrival process along these created routes is a Poisson arrival process. When the simulation is started a warm-up state is loaded, this is the same every single time. This is computationally faster than running the traffic simulation for a couple of minutes for the network to fill up and also makes the starting states more realistic. To prevent overfitting, the routes are randomly generated for each iteration. Combined with the stochastic arrival rate this should make the traffic sufficiently random.

The intensity of the created traffic is between the following three conditions: Low traffic, almost no waiting times, and everything flowing well. Medium traffic, congestion starts to occur and queue times become longer. High traffic, the network is becoming fully congested and queues become very long. Note that traffic not only consists of these 3 types but also many in-between types of traffic.

For configuring the traffic light settings the following assumptions are made. To make the problem realistic there is a minimal green time. This is implemented as a constraint, preventing the algorithm from turning the light off within 8 seconds. There also is a minimal time between two conflicting directions switching from red to green. This is covered by the in-between yellow phase of 3 seconds. These are common parameters in the real world. The optimization of local traffic lights using max pressure will be done according to the formulas provided in Section 3.6. Pyomo, an open-source optimization package, will solve the set of linear equations.

The states are chosen in a way that these can be applied in a real-world scenario and thus must also be measurable. Because of this for example traffic density is excluded as this is only something that can be accurately measured in a simulation. According to Edwin Mein (personal communication, July 25, 2024), an expert in the field of traffic management, a more realistic approach is to use average traffic speed. This data is already available and would not require any new infrastructure to be built. This also gives a good indication of the congestion in a city as speed and congestion are directly related. Alongside average speed, additional state features are chosen, and an in-depth explanation of the state formulation is provided in the next section.

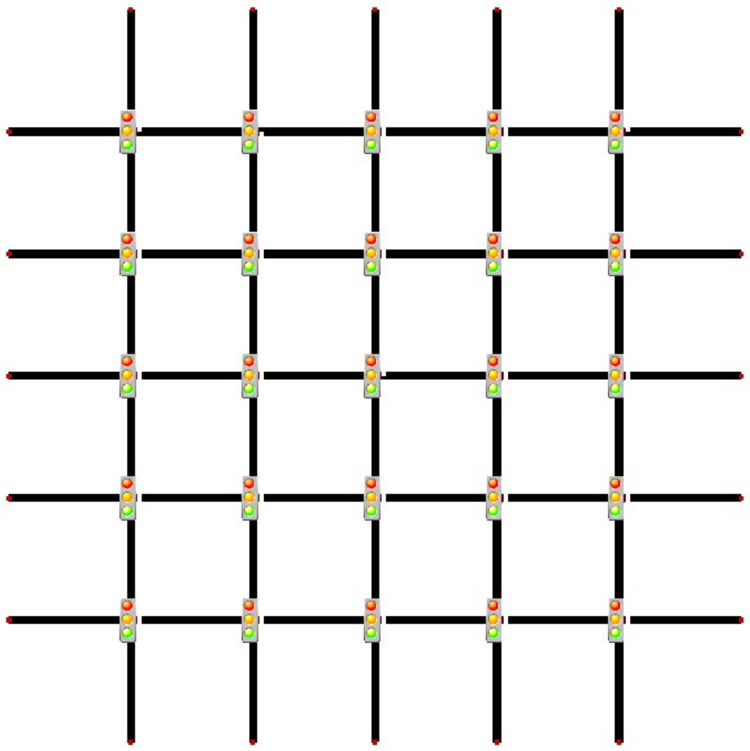


Figure 7: SUMO traffic network.

4.2 Data collection for neural network training

To generate data for training the neural network many simulations are done. Each simulation changes either the traffic intensity or the traffic light settings. By choosing a wide range of traffic intensities and traffic light settings a variety of data is collected. A data point is generated each traffic light cycle (90 seconds). The collected data is then saved in a text file containing the state of the traffic, the traffic intensity, traffic light settings, and the cumulative mean speed over the next five cycles. As mentioned before, for each simulation run a new set of random routes is created to prevent overfitting.

The training data does not cover the traffic intensities and the traffic light settings of the validation data. Some traffic light settings and traffic intensities for the validation data are also out of the range of the training data. With these differences in the data, it is assumed that the two data sets are significantly different and the neural network would indeed need to predict unknown scenarios. The reason for this two-step data generation approach where different settings are used for validation and training is that otherwise predicting a random part of the training data in the validation process is too easy as the data is sequential and very similar per step. If a random point separated from the training data would be selected for validation the neural network would know the state and reward just before that and just after that data point making the prediction a simple linear regression instead of testing whether the neural network understands the data and can predict new situations.

To train the neural network 12000 samples are taken. Ideally, more samples would be used but creating 12000 samples takes more than 24 hours. The validation data contains 1,500 samples. 10% of the training data is randomly selected and separated to use as validation during the training process to monitor performance. This is not to be confused with the final validation data which is kept separate. But rather the data used to update the weights of the neural network during training. The results found using the best hyperparameter and neural network architecture for different state spaces can be found in Section 5.2.

4.3 Training a neural network to predict future traffic situations

The first step that is taken before trying to apply a form of DRL is to check whether a neural network can predict the relations between input data (traffic state, traffic intensity, and phase durations) and labels (mean speed, reward) with sufficient accuracy. The traffic state contains information about the traffic in the network, the traffic intensity is the expected value for the Poisson arrival rate, and the phase duration is a vector of length four, showing the green time per phase. The label (output) is the sum of the mean speed over the next five cycles. To train a neural network to predict future traffic the following steps are done: First, many different traffic intensities are simulated using a wide variety of traffic light settings. Second, the data is then normalized to ensure all state features are between values of 1 and -1. The reward is also normalized to be between 1 and -1 using a different normalization vector. The normalization vector is saved to later be able to transform the data back to their real values for visualization. Third, the normalized data is then used to train a neural network. Lastly, the trained neural network is compared to the validation data to analyze the performance of the trained neural network. For the neural network training the scikit-learn library (Pedregosa et al., 2011) is used.

To find the best hyperparameter settings and neural network architecture the GridSearchCv and RandomizedSearchCV functions from scikit-learn are used. These functions try a variety of hyperparameter and neural network architecture combinations. Judging based on the mean square error (MSE) between the target and expected label, the best-performing combination is used. The hyperparameters can be seen in Table 5.

Table 5: Hyper parameters for Neural Network Training

Hyper parameter	**Value**
Learning Rate	0.001
Batch Size	64
Optimizer	Adam
Hidden layers	128, 64, 32, 16
Activation Function	ReLU
Learning rate	Constant

To determine what formulation of state space is most effective, a variety of state space formulations are used. The different state formulations that are used are shown below. Note that the current phase durations and traffic intensity are included in all states and for simplicity will not be shown in the overview of different state formulations in Table 7.

Feature	**State 1**	**State 2**	**State 3**	**State 4**	**State 5**	**State 6**
Total lane occupancy	✓	✓	✓			
Average lane occupancy	✓		✓	✓		
Total vehicles	✓	✓	✓	✓		
Total vehicles inside perimeter	✓	✓	✓	✓		✓
Total vehicles outside perimeter	✓	✓	✓	✓		✓
Average speed	✓	✓	✓	✓	✓	✓
Total queue length	✓	✓	✓			
Average queue length	✓		✓	✓		
Queue outside perimeter	✓	✓	✓	✓	✓	
Queue inside perimeter	✓	✓	✓	✓	✓	
Total travel time	✓	✓	✓			
Average travel time	✓		✓	✓		
Inflowing vehicles	✓	✓	✓	✓		
Outflowing vehicles	✓	✓	✓	✓		
Queue length for every traffic light			✓			

Table 6: Comparison of state formulations and their features.

4.4 Iterative supervised learning

Before applying reinforcement learning, the trained neural network will be iteratively fine-tuned. A simulation of traffic is done as usual but now every five cycles (450 seconds) the neural network predicts the best traffic light phase durations for the upcoming five cycles. This is done as follows: First, the current traffic situation data from the simulation is gathered and put into a state of the same form that the neural network has been trained on. Second, for a large list of phase durations the label (mean speed over the upcoming five cycles) is predicted. Third, the phase durations that give the highest mean speed are selected and applied in the traffic simulation for the next five cycles. The process is then repeated until the simulation ends. This is done for a large amount of traffic intensities and various starting traffic light configurations. The data created from these simulations is then used to fine-tune the existing neural network. While the training data is sufficient to show the increase in mean speed over iterations, an additional validation run is required to show the new found prediction accuracy of the fine tuned neural network. To do this the same process as for training is repeated to create validation data but then with different, mostly unseen, traffic intensities. A flowchart of the iterative supervised learning process can be seen in Figure 4. Performing iterations of supervised learning should increase accuracy in predicting future traffic. Additionally, the mean speed of all simulations should be higher for the data where the neural network makes decisions. The mean speed should improve per iteration but the degree of improvement should decay and eventually converge. This approach should yield a similar result to reinforcement learning but is easier to understand. For this reason, the iterative supervised learning approach is done before DRL. The results can be found in Section 5.3.

4.5 Naive perimeter control

To work towards the final solution, individual components are first created and tested, as it is incredibly difficult to debug the final model of combining max pressure and perimeter control. First, perimeter control is used on the outer traffic lights excluding the corner points. All traffic lights in the network have standard phase durations and a fixed cycle time. The RL algorithm adjusts the phase durations of the traffic lights on the perimeter. The phases of the traffic light are shown in Figure 8. These adjustments still need to adhere to the fixed cycle time. This is done by dividing the sum of the chosen durations by the cycle time. Afterwards, the action is multiplied by this number. The action is once again continuous, therefore PPO and TD3 are both applied here. The Markov decision process for this approach is discussed in Section 3.2.

A slightly modified version of the above-described method for naive perimeter control is also attempted. This modified version is created to get a better understanding of how a different action method would affect the performance. The previously mentioned version of naive perimeter control uses a vector action to modify the duration of all phases. The alternative version instead uses a scaler as action, which determines the ratio of the main phases that are parallel and perpendicular to the perimeter while keeping the secondary phases constant. Figure 8 shows all phases, here the phase that is green, starting at roads 0,1 & 6,7, is the main phase. The left turn on roads 2 and 8 are secondary phases. This approach would thus determine the ratio of green time between the main phases, roads 0,1 & 6,7 and roads 3,4 & 9,10. This is done by taking the cycle time, removing yellow time, and secondary phase durations. This gives the remaining phase duration for the two main phases. The scalar is then used to divide this time between the two main phases. The state and

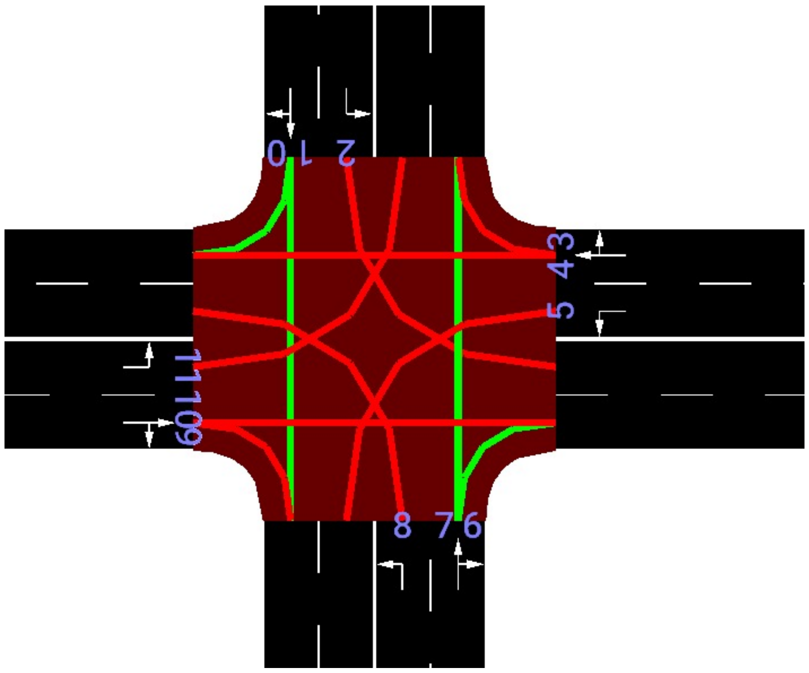


Figure 8: Traffic light phases.

reward are kept the same as in the regular naive perimeter control.

4.6 Combining perimeter control with an optimizer

During this research, another research was being conducted with a similar idea. Kampitakis & Vlahogianni (2024) also attempt to implement perimeter control to reduce congestion inside the city. The difference in their solution is the solving of wasted green time along the perimeter as well as a different action. The action that the DRL algorithm does is determine the number of cars that are allowed to enter the network. The local optimization is then done by distributing these cars among the traffic lights on the perimeter by assigning green durations. The result is that the overall idea of combining local and global optimization is similar but the execution is slightly different. As this approach is similar, a slightly simplified version of the approach is applied as an intermediate step towards the final solution which uses DRL to combine perimeter control and max pressure. This will help to check if it is correctly implemented as well as give additional insights.

4.7 Final version, perimeter control with max pressure

This is the final step that is worked towards. Here a DRL algorithm uses perimeter control to moderate in/out flows which are applied to local traffic lights on the perimeter using max pressure. An in-depth explanation of this approach is given in Section 3.6 & 3.7.

4.8 Model verification

Verification is checking whether a model performs as it is intended to perform. In the case of this research, it is if the simulation is running as intended, are the traffic lights using the correct durations, if the traffic is generated and flowing as intended and are the state features calculated correctly.

The environment is verified using the graphical user interface (GUI) option from SUMO to observe how the traffic flows, as well as the working of the traffic lights. The code is tested using various print statements to verify that the state, action, and reward are correctly calculated and applied. The DRL algorithm is also run with an action space consisting of one action, the static standard traffic light setting, to see if it gives the same results as the simulation without the DRL algorithm. All of these tests were successful. The progress of the DRL algorithm is analyzed using Tensorboard. Furthermore, it is assumed that the DRL algorithm functions correctly and robustly unless proven otherwise since they are taken from the widely used Python library StableBaselines3.

For supervised learning, the scikit-learn library is used to train the neural network. This is a commonly used package in machine learning and is assumed to work as intended. When applying the iterative approach the action selection method is thoroughly tested to make sure that it performs as intended. This also passed all the verification tests.

4.9 Computational time

Training an RL agent requires a lot of data, thus Computational time is a huge bottleneck for this current research. For the DRL approach to achieve decent performance, the simulation must run for 12+ hours. The reason for this is that there are many cars, as the network is large and the traffic should represent a busy city. Simulations take longer as the cumulative amount of cars in the simulation increases. The program needs to keep track of the speed and location of all these cars causing the simulation to start taking very long as the network fills up. A simulation of an hour of traffic will take multiple minutes. This is a huge challenge faced during this research.

There are two main factors for the computational time: First, the most significant factor, simulating a congested city. As there are so many cars in the simulation the simulation speed becomes very slow. Second, for the DRL algorithm to function a lot of information must be extracted from the simulation. As the number of cars increases, so does the data that needs to be extracted, which makes it slow. The decision-making of the DRL algorithm is not the bottleneck but the simulation itself.

To account for the aforementioned challenge of computation time, computationally faster coding methods were implemented as well as running four environments in parallel to train one neural network. This allowed for faster data generation and collection. A remote desktop was also used to double the computational resources available. However, this was still not within the desired time and thereby remains a large issue.

4.10 Conclusion of the solution approach

In this chapter, multiple small steps are presented in working towards a final solution. These steps include first using supervised learning to predict future traffic. Second, using iterative supervised learning to gain insights into good policies under various traffic situations. Third, using the 5x5 grid network with a simple action using DRL. Fourth, creating an approach that is proven to work using perimeter control. Lastly, implementing the final model combining perimeter control and max pressure. Applying these steps ensures gradual build-up and limits mistakes. The DRL algorithms of choice are PPO and TD3 for their continuous action space and ease of implementation. For verification, the SUMO graphical user interface will be used to see if the traffic lights work as intended. To limit computational time code optimization is done as well as parallel processing. By using parallel processing, 4 simulations were able to be run simultaneously to train one neural network.

5 Experimental results

This chapter covers the validation as well as the results of the approaches described in Chapter 4. This includes the following: First, all approaches are validated. Second, the performance of the neural network in terms of predicting the state of future traffic. Third, the reflection on the iterative supervised learning approach and how this converges to an optimal policy. Fourth, the findings of naive perimeter control, perimeter control with an optimizer, and finally perimeter control with max pressure. The chapter concludes with a summary of the findings.

5.1 Model validation

Model validation is making sure that the simulation model resembles the real world. As it is assumed that the simulation model is accurate the validation is done by comparing the performance to a benchmark policy. A standard static traffic light policy is chosen as a benchmark. Iterative supervised learning can outperform the benchmark policy in terms of mean speed. However, the DRL algorithm is not able to outperform the benchmark. For the DRL algorithm, every attempted combination of state formulation, action, hyperparameters, and traffic resulted in performance just below the benchmark. As the DRL algorithm can choose the same policy as the benchmark policy there seems to be something wrong. Thorough testing is done but the exact cause is not found. It is expected that the problem is difficult to understand for the DRL algorithm which combined with a weak, delayed reward signal leads to poor performance.

5.2 Results neural network performance

This subsection covers the result of the neural network training. Here multiple state definitions are used and their performance is evaluated with the goal of finding the best state definition. For each unique state, the neural network architecture and hyperparameters are optimized for a fair comparison. This optimization however did not yield significantly different results. All state definitions performed best using almost identical parameters. One exception was state 5 which was better with an alpha of 0.01 instead of 0.001. Although the change in performance is insignificant. For ease of reading a recap of the states from Section 4.3 are given below:

Feature	State 1	State 2	State 3	State 4	State 5	State 6
Total lane occupancy	✓	✓	✓			
Average lane occupancy	✓		✓	✓		
Total vehicles	✓	✓	✓	✓		
Total vehicles inside perimeter	✓	✓	✓	✓		✓
Total vehicles outside perimeter	✓	✓	✓	✓		✓
Average speed	✓	✓	✓	✓	✓	✓
Total queue length	✓	✓	✓			
Average queue length	✓		✓	✓		
Queue outside perimeter	✓	✓	✓	✓	✓	
Queue inside perimeter	✓	✓	✓	✓	✓	
Total travel time	✓	✓	✓			
Average travel time	✓		✓	✓		
Inflowing vehicles	✓	✓	✓	✓		
Outflowing vehicles	✓	✓	✓	✓		
Queue length for every traffic light			✓			

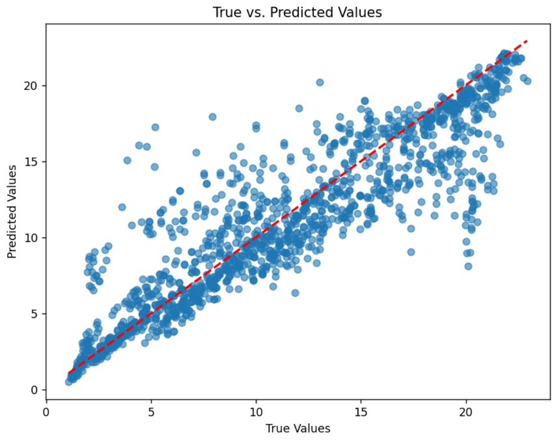
Table 7: Comparison of state formulations and their features.

State formulations 2 and 4 consist mostly of state 1 but the average values and total values were removed respectively if the alternative (total and average respectively) is present. State formulation 3 is an extension of state formulation 1 where the queue for each traffic light is added as a vector. States 5 and 6 are both parts small parts of state 1. After optimizing the neural network architecture and hyperparameters the following results were found. Note that the data is created using the same traffic.

Table 8: MAE, RMSE, and MAPE results for different state formulations.

Model	MAE	RMSE	MAPE %
State 1	1.97	2.99	25.6
State 2	1.84	2.66	22.4
State 3	2.21	3.12	25.2
State 4	1.79	2.62	21.1
State 5	2.31	2.96	24.9
State 6	2.08	3.09	25.2

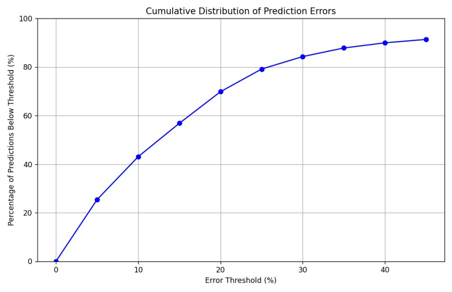
Table 8 shows the mean absolute error (MAE), root mean square error (MRSE), and mean absolute percentage error (MAPE). The results show that state formulations 2 and 4 seem to be the best performing out of all variations tried. These two states are both similar to number one except for not having the combination of both ″average of x″ and ″total of x″ for the values that include both. It seems that having both these values in the state is redundant and decreases performance. It can also be observed that adding a lot of additional information such as the queue per traffic light decreases performance. Smaller state spaces with limited information such as states 5 and 6 give too little information. Because the performance of state space 4 is the best it has been chosen as the general state space for supervised learning and deep reinforcement learning for this problem. This is the same state that is given in Section 3.2.



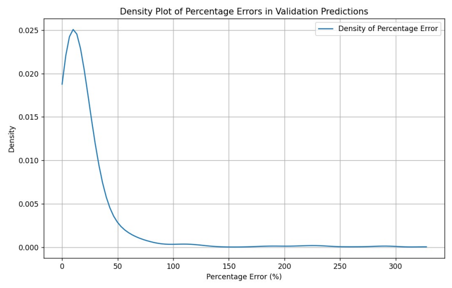
Pictures

Figure 9: Parity plot of the validation data using state 4.

Figure 9 shows how accurately the neural network can predict the labels. Most predicted values are close to the true values. However, there are a decent amount of outliers.



(a) Cumulative Error Distribution Plot using state 4.



(b) Density Plot of Prediction Errors using state 4.

Figure 10: Additional performance evaluation of the trained neural network using state 4.

Figure 10 shows how likely errors are expected to occur and the degree of the error as an additional measure of performance. Figure 10a shows that 70% of the values have an error margin below 20% and 90% of the data has an error margin below 40%.

The performance of the neural network might not seem great. This is in part due to the nature of the validation data, as the validation data is significantly different from the training data. The general relationship between the input state and the label seems to be understood by the neural network. When using similar traffic light settings and traffic intensities the parity plot has a much tighter fit and the error also reduces significantly.

5.3 Results iterative supervised learning

This section covers the results of applying a neural network to assign phase durations during the traffic simulation. After a lot of simulations are done the results can be found in Table 9. Here it can be seen that the error decreases over all performance parameters. This means that the iterative approach helps in improving the initial mediocre performance. This is done for both state 1 and state 4. This also shows that state formulation 4 remains superior to state formulation 1, even when fine-tuning is done to increase performance.

Iteration	**State 1**			**State 4**		
	MAE	RMSE	MAPE (%)	MAE	RMSE	MAPE (%)
1	1.97	2.99	25.6	1.79	2.62	21.1
2	1.18	1.94	15.4	0.88	1.28	8.4
3	0.87	1.37	10.0	0.97	1.52	9.0
4	0.86	1.44	8.24	1.04	1.62	10.6
5	0.91	1.56	9.20	0.79	1.17	6.3

Table 9: MAE, RMSE, and MAPE values for state 1 & 4 per iteration of supervised learning.

In addition to the performance of predicting labels accurately, the performance of choosing good phase durations is also shown. Here the average speed over all data points is shown per iteration for both state 1 and state 4. This can be seen below in Figure 11. Note that iteration one is using a constant policy and is thus the same for both state 1 and state 4. Iteration two and onward use the model that was trained based on the data collected in the previous iteration.

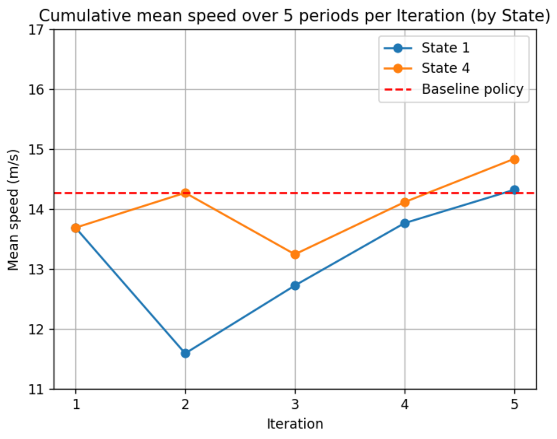
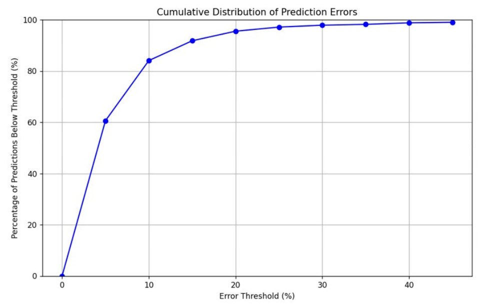


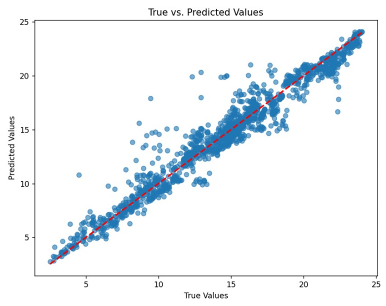
Figure 11: Average speed per iteration of neural network updates for state 1 and state 4.

The results from Figure 11 show that the average speed over the simulation can be improved using the iterative supervised learning approach. Using the trained neural network to determine the policy yields a better average speed over many simulations. At the 5th iteration, the cumulative mean speed over five periods of the simulation is increased by more than 1 m/s with respect to the starting random static policy. This is a percentage improvement of 5-8% which is not a lot. Compared to the standard static baseline policy the improvements are even smaller with the average speed being 0.5 m/s faster for the best-performing state. This is a percentage increase of 0-3% which is bad considering the base policy involves no intelligence. Noticeable in this graph is the decrease in performance for one iteration. This happens when a prediction of the neural network is significantly off and as a result, makes a really bad move. Consequently, this bad move causes severe congestion to the point that the overall traffic becomes stuck. In this case, it is no longer possible to fix the traffic. The result is an extremely low mean speed for that simulation run.

Except for the one iteration which decreases performance the other iterations all slowly increase performance. This performance is still on an upward trend at the 5th iteration, showing that more iterations and fine-tuning of the neural network can continue to improve the mean speed. More iterations have not been done due to the significant amount of time required to compute as creating the training and validation data takes a day per state.



(a) Cumulative Error Distribution Plot using state 4 at the 5th iteration.



(b) Parity plot of validation data using state 4 at the 5th iteration.

Figure 12: Performance of state 4 at the 5th iteration.

At the 5th iteration of fine-tuning for state 4 the error distribution has shifted as seen in Figure 12a. Now 85% of the data has an error of 10% or less. Originally this was 70% of the data having an error of 20% or less. Additionally, the parity curve is now much tighter around the identity line as seen in Figure 12b.

The difference between the performance of each state is also still prevalent. State 4 consistently outperforms state 1. However, despite state 1 being one of the worst states initially in terms of MAE, RMSE, and MAPE it still manages to improve over iterations. The errors are also reduced significantly as seen in Table 9. After fine-tuning the errors for both state 1 and state 4 have similar error values.

With the iterative supervised learning approach able to improve the mean cumulative speed over 5 cycles it should also be possible to apply DRL. These results will be discussed in the next section.

5.4 Results deep reinforcement learning

For the DRl model, a large combination of states, actions, and hyperparameters are used. The process to get this working took a lot of effort showing the complexity of correctly applying a DRL solution. The results seen in Figure 13 are the best results that are found using the naive DRL approach. Some learning can be observed as the reward increases compared to the starting point. However, this increase is not a lot and the mean speed of the best-performing iteration is still slightly below the standard static phase durations. It is assumed that the problem is too complicated to grasp. The state representation is very broad and the reward signal is delayed due to the nature of traffic. The combination of these factors is most likely the cause of the poor performance.

Changing from PPO to TD3 does not provide significant changes to the performance. Due to the similar results of PPO and TD3, it is assumed that the DRL algorithm itself is not the problem. What stands out in Figure 13 is that the line for PPO continues much longer. This is done to see if further improvements would be found if the simulation would be running for a very long time. This is not done for TD3 as it takes over a day to compute. The Figure also shows that both PPO and TD3 initially have a significant decrease in performance as they update their policy. This is a common occurance in DRL for multiple reasons such as exploration leading to poor results, value function instability where the learned value function is initially inacurate, and a poor formulation of the reward function which might encourage short term sub-optimal decisions. In this research the reason is most likely the initial inaccuracy in the value function. This is also shown in the iterative supervised learning approach that the initial prediction accuracy of a state-action pair is low. This low accuracy could lead to bad moves and thus a drop in reward. This change in reward is much smaller for PPO than for TD3, showing the effect of the clipping mechanism of PPO to prevent large sudden policy updates. Overall, the DRL learns over time but plateaus at a lower mean speed than the baseline policy which means that the performance is very poor.

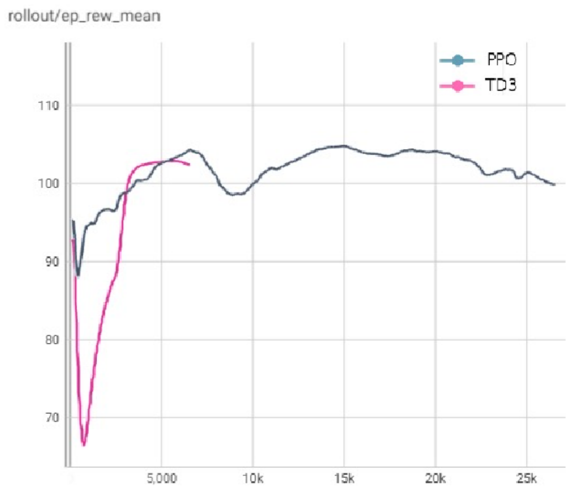


Figure 13: Episode reward for TD3 and PPO using the naive action.

As the naive DRL approach shows such poor performance the focus of this research shifts more towards improving the naive method as well as the supervised learning approaches instead of increasing the complexity of a solution method that is not working. Because of this the final solution method is not worked out and thus no results can be shown. The final solution method should in theory work well and fits all the criteria for the problem but unfortunately due to lack of time, it was not worked out.

5.5 Conclusion of the results

The neural network trained on regular traffic data performs fairly well on traffic states that are not shown before with 70% of the values having less than 20% error. The prediction error is significantly reduced by fine-tuning with 85% of the data having an error smaller than 10%. Iterative learning can be used to choose the phase durations and by doing so will improve the average speed by 5-8% compared to a static random policy or 0-3% compared to a standard static policy. This process takes a lot of time but the bottleneck is the simulation speed, not the neural network choosing the phase durations. The DRL approach is very challenging to implement correctly. After a lot of effort, the best results are still worse than the standard base traffic phase durations by a small margin.

6 Discussion

The results of this research show that it is possible to predict future congestion. However, the accuracy is limited with a MAPE ranging from 21-25% depending on the state formulation. The importance of providing a neural network with enough but also not too much information is shown by experimenting with different state formulations. After additional fine-tuning of the original neural network, the performance does increase significantly with a newly found MAPE between 8-10%. Now that the error is reduced significantly the neural network can be applied in the real world. As the computational bottleneck is collecting data and not generating outputs neural networks can be used in real time. This could be an interesting application for Technolution to provide.

It would be interesting to perform more iterations of updating the neural network to see how much it can still be improved. The error metrics seem to not improve much anymore but the mean cumulative speed is still increasing per iteration. It is assumed that eventually, the improvements will plateau but it is unknown whether that is after one or two more iterations or only after another ten or more iterations. This takes a lot of computational time but depending on the result it might be worthwhile if it offers a significant increase in the mean speed.

The iterative supervised learning approach also struggles to significantly outperform the fixed phase duration approach despite being able to predict future traffic relatively well. As the neural network can predict traffic, the problem might be in understanding the effects of the action. However, changing the action did not improve performance. Perhaps the problem is not the formulation of the action but rather the delayed reward signal as it takes a while for a traffic policy to have a significant effect on a network. This does showcase the challenges of working with neural networks, it is difficult to understand why they fail.

In contrast to the positive results achieved in predicting traffic states, the performance of the DRL model falls short of expectations. It consistently underperforms when compared to a standard scenario with fixed phase durations. Despite extensive testing and adjustments, pinpointing the exact reasons for this underperformance remains challenging. Potential causes may include issues with the reward structure or state representation, although numerous iterations and refinements have been attempted leading to the belief that the problem may be too complex for the DRL algorithm to learn effectively. This is further evidenced by the policy and value losses observed in TensorBoard, which remain exceedingly high and fail to decrease over time, which means that the algorithm struggles to accurately interpret the environment and understand the impact of its actions.

A possible way to improve the DRL performance would be to try different reward structures in addition to the ones tried before. Kampitakis & Vlahogianni (2024) for example used a scaled combination of trip completion rate, mean speed, and punishment for each simulation step taken to push the algorithm to empty the network as fast as possible as they had a finite amount of traffic. The trip completion rate is a great performance metric but is impossible to determine in reality. This is also an issue for the state representation, determining all values of the state accurately in reality is quite a challenge. Many traffic detection systems could be placed in the city, which will be costly. Even then, measuring for example how many cars are in the city as well as the amount of cars approaching the city is difficult. Similarly, determining the mean travel time for cars is nearly impossible.

For the DRL approach, the focus is on policy-based DRL where the algorithm aims to find the policy that maximizes the reward directly. Alternatively, a value-based DRL approach could be used, which instead estimates a value function to predict the reward for a given state-action pair. Traditional value-based DRL methods such as Q-learning would not be suitable given the dimension of the action space. However, Deep Q-learning (DQN) could be effective in using the strengths of neural networks to learn the value function. Although, to use DQN the problem would have to be reformulated to include a discrete action space limiting the possible traffic light settings. The approach might also struggle with the complex state space of this problem which was already seen in the policy-based approach implemented in this research. Nonetheless, it could be interesting to try given that the value-based DRL approach is more sample efficient than the policy-based approach meaning that it could require less computational time which is a big issue as will be discussed in the next section.

One of the challenges encountered when trying to improve performance is that it takes a long time to see the result of a change. This is a significant limiting factor in exploring possible solutions to the performance issue. A large network that represents a crowded city is simply very slow to simulate. The network used in this research is not even that large containing only 25 intersections. If a large city would have to be simulated the computational times would be extremely long. A possible way to reduce the computational problem would be pre-training to reduce the iterations required for the DRL to reach convergence.

7 Conclusion

This section summarizes the research and finishes with an answer to the main research question. The literature highlights the complexity and stochastic nature of traffic, making it challenging to optimize and leading to time-intensive training for preventative solution methods. Reinforcement learning has emerged as a promising approach in traffic management due to its ability to make sequential decisions under uncertainty. However, most applications focus on single traffic light optimization, which, while effective, is not suitable for decision support given effective decision support requires infrequent actions and interpretable solutions.

To provide decision support an alternative to micro-level optimization is needed, in this case, global optimization over a larger time scale. A suitable method for this is perimeter control, which modifies the inflow and outflow of a city to reduce congestion. This technique requires infrequent actions and is intuitive to understand, making it excellent for decision support. The problem with only using perimeter control is that it modifies all traffic lights equally, which only works if traffic is uniform. Since in reality traffic is not uniform, an additional control layer is required, being max pressure. Max pressure ensures that green time is not wasted at traffic lights, thus providing local optimization. Deep reinforcement learning can be used to make the decision for perimeter control, determining the in and out flow modifications. In the mean time, a solver solves a set of linear equations to implement max pressure. This combined approach of global and local optimization should be able to provide decision support to help traffic engineers reduce congestion.

To work towards the final DRL model a few intermediate steps are taken to work towards the final DRL model: First, train a neural network to determine the effectiveness in predicting traffic congestion. Second, using the trained neural network to make decisions, to see if the trained neural network can not only learn the state but also find a good action in a given state. Lastly, creating a DRL model that implements perimeter control.

The results show that neural networks can be used for predicting traffic congestion, but unfortunately, not yet very accurate. Without any fine-tuning the neural network can predict traffic with a MAPE of 21-25%. With such a high margin of error at this point, it is not yet very practical. Alternatively, when using the neural network to make decisions, by adjusting the phase durations, the results show that the neural network can improve the mean speed. After multiple iterations, the average speed is improved by 5-8% with respect to the original random static policy and 0-3% with respect to the standard static policy. It is a computationally intense approach as the neural network improves per iteration, which takes a long time to simulate. Nevertheless, the newfound data can be used to fine-tune the original neural network, which improves the accuracy significantly resulting in a MAPE of 6-8% after fine-tuning the original neural network. This fine-tuned neural network can be effectively used as the results are fas and now also accurate.

The results of the DRL approach are not promising. The performance remains lower than the static base traffic light configuration despite experimenting with different rewards, state formulations, DRL algorithms, and hyperparameters. This approach is also computationally intense since many episodes are required, which takes a long time to simulate. In theory, the DRL approach should work well but sadly did not in this research. The problem assumed by the author is likely due to the DRL algorithm having difficulty understanding the effects of the actions on the state. A reason for the author's assumption is the consistently high-value loss and policy loss.

To reflect on the research question "**How suitable is deep reinforcement learning for providing real-time decision support to traffic engineers for reducing congestion across multiple intersections in an urban traffic network?**", DRL is in theory very promising for sequential decision-making under uncertainty and has been shown to improve traffic in various cases. However, this research experienced a lot of difficulty with successfully applying DRL to optimize traffic. The performance is worse than a standard static policy making it unfit to be applied in the real world unless more time is invested. In addition, training the DRL algorithm itself is computationally intense. Although once trained, the DRL algorithm can return actions quickly. An alternative solution found to provide decision support is using neural networks to predict traffic. After fine-tuning the neural network can predict future congestion with good accuracy. Traffic engineers can input the state of traffic and the neural network can then return a prediction on what would happen in the future, giving valuable insights to traffic engineers.

8 Recommendations and future research

For future work, it would be recommended to use methods to reduce computational times as this was one of the main challenges. For running the simulation the following actions could be done to reduce computational time: Use smaller networks, less traffic, additional code optimization, or a faster traffic simulation software. To reduce the iterations required for training a DRL algorithm pre-training can be done. Alternatively, using advanced architectures like SimBa (Simplicity Bias) can be applied to enhance sample efficiency and reduce the episodes required for convergence (Lee et al. 2024). Kampitakis & Vlahogianni (2024) used pre-training based on behavior cloning to reduce the number of iterations required to reach convergence of the DRL algorithm. The idea is to learn a policy directly from a set of expert demonstrations. The goal is to teach the agent to replicate the expert's behavior. This can then be used as a strong starting policy. Another approach is to use better hardware that supports more parallel environments or GPU acceleration.

A difficult problem encountered during this research was understanding why the DRL decides on specific traffic light configurations. In the future, it could be possible to use a real-world network with traffic based on real data. Ideally, this network will already be optimized by a traffic expert. The DRL algorithm would then have to work towards the best solution which is assumed to be the one provided by the traffic expert. This could help understand the learning steps and could also provide insights to traffic experts on how artificial intelligence approaches this problem.

All in all, it is recommended that Technolution continues research into DRL as it is promising in theory. However, if they want to implement DRL they need to be aware that significant time and expertise is required. An alternative solution is to train neural networks to predict traffic congestion, which traffic engineers can use to gain insights. Although this is not the original goal it could be a useful tool that is much simpler to provide to traffic engineers.

9 References

References

[1] Technolution Move. (2024). MobiMaestro - Technolution Move. Retrieved November 20, 2024, from https://www.technolution.com/move/mobimaestro/

[2] Inoue, D., Yamashita, H., Aihara, K., & Yoshida, H. (2024). Traffic signal optimization in large-scale urban road networks: an adaptive-predictive controller using Ising models. arXiv.org. https://arxiv.org/abs/2406.03690

[3] Li, Y. (2018) Deep Reinforcement Learning https://arxiv.org/abs/1810.06339

[4] Wyner, A. (2023). Sustainable safety: The dutch approach to safe road design https://www.ippi.org.il/sustainable-safety-the-dutch-approach-to-safe-road-design/

[5] Naeem, M., Rizvi, S. T. H., & Coronato, A. (2020). A Gentle Introduction to Reinforcement Learning and its Application in Different Fields. IEEE Access, 8, 209320–209344, https://doi.org/10.1109/access.2020.3038605

[6] Allied Market Research, (2023). Reinforcement Learning Market Size, share, Competitive landscape and Trend analysis Report by deployment mode, by enterprise size, by end user: Global Opportunity Analysis and Industry Forecast, 2023-2032. Allied Market Research. https://www.alliedmarketresearch.com/reinforcement-learning-market-A229407#:~:text=Reinforcement%20Learning%20Market%20Statistics%3A%202032,41.5%25%20from%202023%20to%202032

[7] Wu, C., Kreidieh, A., Parvate, K., Vinitsky, E., & Bayen, A. M. (2017). Flow: Architecture and benchmarking for reinforcement learning in traffic Control. arXiv (Cornell University). https://arxiv.org/pdf/1710.05465.pdf

[8] Zhang, H., Feng, S., Liu, C., Ding, Y., Zhu, Y., Zhou, Z., Zhang, W., Yu, Y., Jin, H., & Li, Z. (2019). CityFlow: A Multi-Agent Reinforcement learning environment for large scale city traffic scenario. In Proceedings of the 2019 World Wide Web Conference (WWW '19), May 13–17, 2019, San Francisco, CA, USA. https://doi.org/10.1145/3308558.3314139

[9] Qiang, W., & Zhongli, Z. (2011). Reinforcement Learning Model,Algorithms and Its Application. International Conference on Mechatronic Science, Electric Engineering and Computer, Jilin, China, pp. 1143-1146, https://doi.org/10.1109/MEC.2011.6025669

[10] Creswell, J. W. (2023). Research design: Qualitative, Quantitative and Mixed Methods approaches. Los Angeles: SAGE Publications Inc. From http://www.revistacomunicacion.org/pdf/n3/resenas/research_design_qualitative_quantitative_and_mixed_methods_approaches.pdf

[11] McCluskey, T., Kotsialos, A., Mller, J. P., Klgl, F., Rana, O. F., & Schumann, R. (2016). Autonomic Road transport support systems. In Springer eBooks. https://doi.org/10.1007/978-3-319-25808-9

[12] Varaiya, P. (2013). Max pressure control of a network of signalized intersections. Transportation Research Part C Emerging Technologies, 36, 177–195. https://doi.org/10.1016/j.trc.2013.08.014

[13] Technolution Move. (2023). AFM - Technolution Move. https://www.technolution.com/move/cases/afm/

[14] Rasheed, F., Yau, K. A., Noor, R. M., Wu, C., & Low, Y. C. (2020). Deep Reinforcement Learning for Traffic Signal Control: a review. IEEE Access, 8, 208016–208044. https://doi.org/10.1109/access.2020.3034141

[15] Abdulhai, B., Pringle, R., & Karakoulas, G. J. (2003). Reinforcement learning for true adaptive traffic signal control. Journal of Transportation Engineering, 129(3), 278–285. https://doi.org/10.1061/(asce)0733-947x(2003)129:3(278

[16] Genders, W., & Razavi, S. (2016). Using a deep reinforcement learning agent for traffic signal control. arXiv (Cornell University). http://export.arxiv.org/pdf/1611.01142

[17] Zheng, G., Zang, X., Xu, N., Wei, H., Yu, Z., Gayah, V. V., Xu, K., & Li, Z. (2019). Diagnosing reinforcement learning for traffic signal control. arXiv (Cornell University). https://arxiv.org/pdf/1905.04716.pdf

[18] Li, Z., Yu, H., Zhang, G., Dong, S., & Xu, C. (2021). Network-wide traffic signal control optimization using a multi-agent deep reinforcement learning. Transportation Research Part C Emerging Technologies, 125, 103059. https://doi.org/10.1016/j.trc.2021.103059

[19] Ouallane, A. A., Bahnasse, A., Bakali, A., & Talea, M. (2022). Overview of Road Traffic Management Solutions based on IoT and AI. Procedia Computer Science, 198, 518–523. https://doi.org/10.1016/j.procs.2021.12.279

[20] Ravish, R., & Swamy, S. R. (2021). Intelligent Traffic Management: A review of challenges, solutions, and future perspectives. Transport and Telecommunication, 22(2), 163–182. https://doi.org/10.2478/ttj-2021-0013

[21] Cong, Z., De Schutter, B., & Babuška, R. (2013) Ant colony routing algorithm for freeway networks. Transportation Research Part C: Emerging Technologies, 37, 1-19. https://www.sciencedirect.com/science/article/pii/S0968090X13001885

[22] Shi, Y., Qi, Y., Lv, L., & Liang, D. (2021). A Particle Swarm Optimisation with Linearly Decreasing Weight for Real-Time Traffic Signal Control. Machines, 9(11), 280. https://doi.org/10.3390/machines9110280

[23] Tsitsokas, D., Kouvelas, A., & Geroliminis, N. (2023). Two-layer adaptive signal control framework for large-scale dynamically-congested networks: Combining efficient Max Pressure with Perimeter Control. Transportation Research. Part C, Emerging Technologies, 152, 104128. https://doi.org/10.1016/j.trc.2023.104128

[24] Tsitsokas, D., Kouvelas, A., & Geroliminis, N. (2021). Modeling and optimization of dedicated bus lanes space allocation in large networks with dynamic congestion. Transportation Research Part C Emerging Technologies, 127, 103082. https://doi.org/10.1016/j.trc.2021.103082

[25] Tampubolon, H., & Hsiung, P. (2018). Supervised Deep Learning Based for Traffic Flow Prediction. 2018 International Conference on Smart Green Technology in Electrical and Information Systems (ICSGTEIS), Bali, Indonesia, pp. 95-100, doi: 10.1109/ICSGTEIS.2018.8709102

[26] Sanagapati, P. (2019). What is Dropout Regularization? Find out:). Kaggle. Retrieved November 12, 2024, from https://www.kaggle.com/code/pavansanagapati/what-is-dropout-regularization-find-out

[27] Patil , P. (2022). Applications of Deep Learning in Traffic Management: A Review. International Journal of Business Intelligence and Big Data Analytics, 5(1), 16–23. Retrieved November 12, 2024, from https://research.tensorgate.org/index.php/IJBIBDA/article/view/26

[28] Wei, H., Chen, C., Wu, K., Xu, K., Gayah, V., & Li, Z. (2019). Presslight: Learning max pressure control to coordinate traffic signals in arterial network. In Proceedings of the 25th ACMSIGKDD International Conference on Knowledge Discovery & Data Mining. https://ojs.aaai.org/index.php/AAAI/article/view/5744/5600

[29] Hegde, S., Premasudha, B., Hooli, A., Akshay, M. (2024). A Review on Smart Traffic Management with Reinforcement Learning. In: Yang, XS., Sherratt, S., Dey, N., Joshi, A. (eds) Proceedings of Ninth International Congress on Information and Communication Technology. ICICT 2024 2024. Lecture Notes in Networks and Systems, vol 1004. Springer, Singapore. https://doi.org/10.1007/978-981-97-3305-7$_3$7

[30] Cong, Z., De Schutter, B., & Babuška, R. (2013). Ant Colony Routing algorithm for freeway networks. Transportation Research Part C Emerging Technologies, 37, 1–19. https://doi.org/10.1016/j.trc.2013.09.008

[31] Han, Y., Wang, M., & Leclercq, L. (2023). Leveraging reinforcement learning for dynamic traffic control: A survey and challenges for field implementation. Communications in Transportation Research, 3, 100104. https://doi.org/10.1016/j.commtr.2023.100104

[32] Lin, Y., Dai, X., Li, L., & Wang, F. (2018). An efficient deep reinforcement learning model for urban traffic control. arXiv (Cornell University). http://export.arxiv.org/pdf/1808.01876

[33] Han, Y., Wang, M., & Leclercq, L. (2023). Leveraging reinforcement learning for dynamic traffic control: A survey and challenges for field implementation. Communications in Transportation Research, 3, 100104. https://doi.org/10.1016/j.commtr.2023.100104

[34] Li, Z., Xu, C., Zhang, G. (2021). A Deep Reinforcement Learning Approach for Traffic Signal Control Optimization https://arxiv.org/abs/2107.06115

[35] , L., Babuska, R., & De Schutter, B. (2010). Multi-agent reinforcement learning: An overview. In D. Srinivasan, & L. C. Jain (Eds.), Innovations in multi-agent systems and applications - 1 (pp. 183-221). Springer.

[36] Li, Y. (2018).Deep Reinforcement Learning. urlhttps://arxiv.org/abs/1810.06339

[37] Wang, X., Wang, S., Liang, X., Zhao, D., Huang, J., Xu, X., Dai, B., & Miao, Q. (2024). Deep Reinforcement Learning: a survey. IEEE Transactions on Neural Networks and Learning Systems, 1–15. https://doi.org/10.1109/tnnls.2022.3207346

[38] Walraven, E., Spaan, M. T. J., & Bakker, B. (2016). Traffic flow optimization: A reinforcement learning approach. Engineering Applications of Artificial Intelligence, 52, 203–212. https://doi.org/10.1016/j.engappai.2016.01.001

[39] Paul, A., Mitra, S. (2020). Deep Reinforcement Learning based Traffic Signal optimization for Multiple Intersections in ITS, IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS), New Delhi, India, 2020, pp. 1-6. https://doi.org/10.1109/ANTS50601.2020.9342819

[40] Gu, J., Lee, M., Jun, C., Han, Y., Kim, Y., Kim, J. (2021). Traffic Signal Optimization for Multiple Intersections Based on Reinforcement Learning. Appl. Sci. 2021, 11, 10688. https://doi.org/10.3390/app112210688

[41] Muresan, M., Pan, G., & Fu, L. (2021). Multi-Intersection Control with Deep Reinforcement Learning and Ring-and-Barrier Controllers. Transportation Research Record, 2675(4), 308-319. https://doi.org/10.1177/0361198120980321

[42] Zhu, R., Ding, W., Wu, S., Li, L., Lv, P., & Xu, M. (2023). Auto-learning communication reinforcement learning for multi-intersection traffic light control. Knowledge-based Systems, 275, 110696. https://doi.org/10.1016/j.knosys.2023.110696

[43] Wang, Y., Xu, T., Niu, X., Tan, C., Chen, E., & Xiong, H. (2022). STMARL: A Spatio-Temporal Multi-Agent Reinforcement Learning Approach for Cooperative Traffic Light Control. IEEE Transactions on Mobile Computing, 21(6), 2228–2242. https://doi.org/10.1109/tmc.2020.3033782

[44] Tsitsokas, D., Kouvelas, A., & Geroliminis, N. (2022). Efficient Max-Pressure traffic management for Large-Scale congested urban networks. Infoscience. https://infoscience.epfl.ch/record/301369

[45] Handboek verkeerslichtenregelingen 2022. (2022). Kennisbank.crow.nl. Retrieved June 4, 2024, from https://kennisbank.crow.nl/public/VERMGT/Handboek_Verkeerslichtenregelingen_2022/Minimale_groentijd_en_groenknippertijd_voor_voetgangers/114372

[46] Yadav, N., Mathur, P., Kumar, P., & Prasad, N. (2022). Road Traffic Signal Design By Using Webster Method. International Journal of Scientific Engineering and Applied Science (IJSEAS) – Volume-8, Issue-2. https://ijseas.com/volume8/v8i2/IJSEAS202202102.pdf

[47] Van Heeswijk, W., (2023). Proximal Policy Optimization (PPO) explained - towards data science. Medium. https://towardsdatascience.com/proximal-policy-optimization-ppo-explained-abed1952457b

[48] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal Policy Optimization Algorithms. https://arxiv.org/abs/1707.06347

[49] Raffin, A., Hill, A., Ernestus, M., Gleave, A., Kanervisto, A., & Dormann, N. (2019). Stable-Baselines3: Reliable Reinforcement Learning Implementations. GitHub repository. https://github.com/DLR-RM/stable-baselines3

[50] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12, 2825–2830. http://scikit-learn.org/stable

[51] Kampitakis, E., & Vlahogianni, E. (2024). Unified cooperative management system: a framework to integrate the control strategies. Manuscript submitted for publication.

[52] Lee, H., Hwang, D., Kim, D., Kim, H., Tai, J., Subramanian, K., Wurman, P., Choo, J., Stone, P., Seno, T. (2024). SimBa: Simplicity Bias for Scaling Up Parameters in Deep Reinforcement Learning. https://arxiv.org/abs/2410.09754

A Appendix

A.1 Additional figures

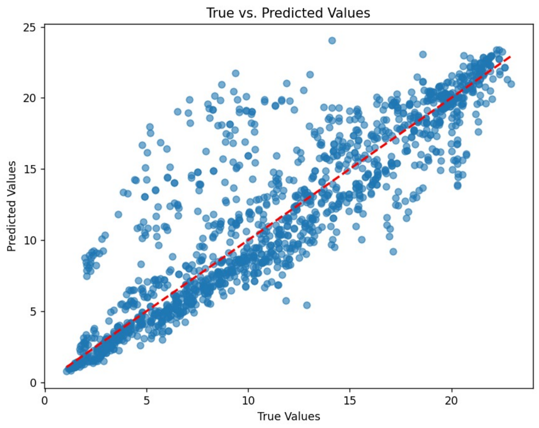
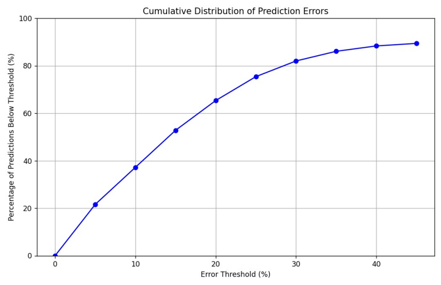
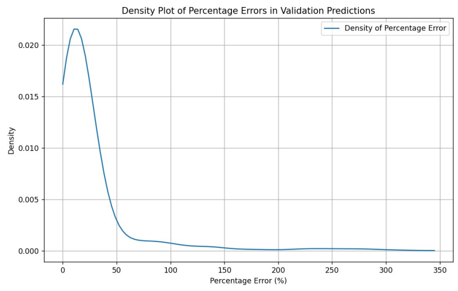


Figure A.1: Parity plot of the validation data using state 1.



(a) Cumulative Error Distribution Plot using state 1.



(b) Density Plot of Prediction Errors using state 1.

Figure A.2: Additional performance evaluation of the trained neural network using state 1.

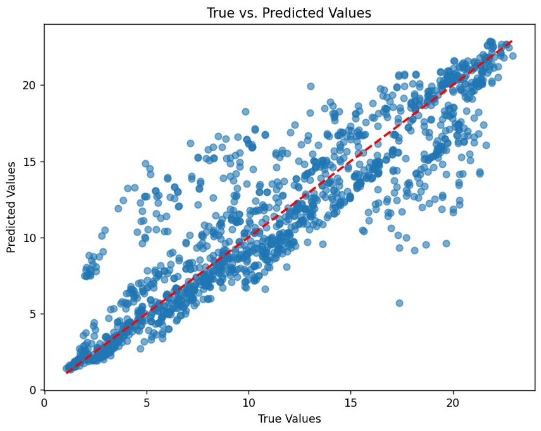
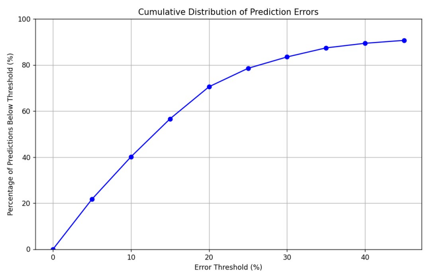
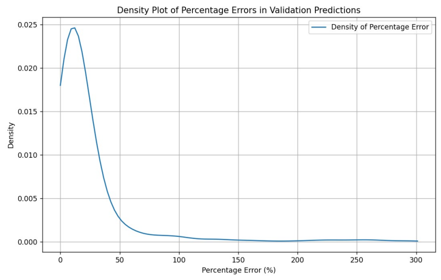


Figure A.3: Parity plot of the validation data using state 2.



(a) Cumulative Error Distribution Plot using state 2.



(b) Density Plot of Prediction Errors using state 2.

Figure A.4: Additional performance evaluation of the trained neural network using state 2.

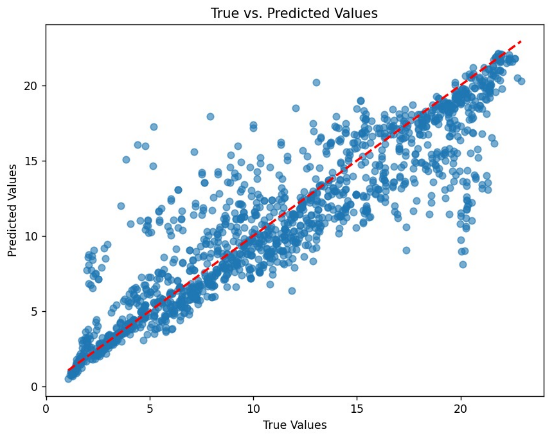
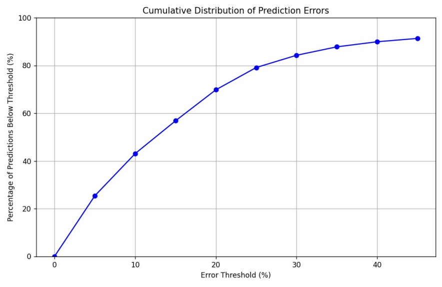
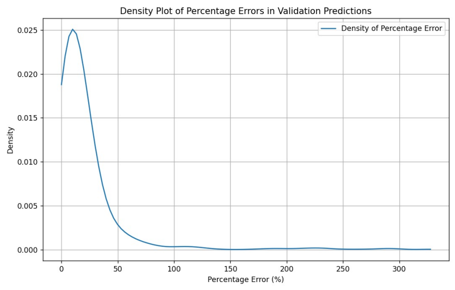


Figure A.5: Parity plot of the validation data using state 3.



(a) Cumulative Error Distribution Plot using state 3.



(b) Density Plot of Prediction Errors using state 3.

Figure A.6: Additional performance evaluation of the trained neural network using state 3.

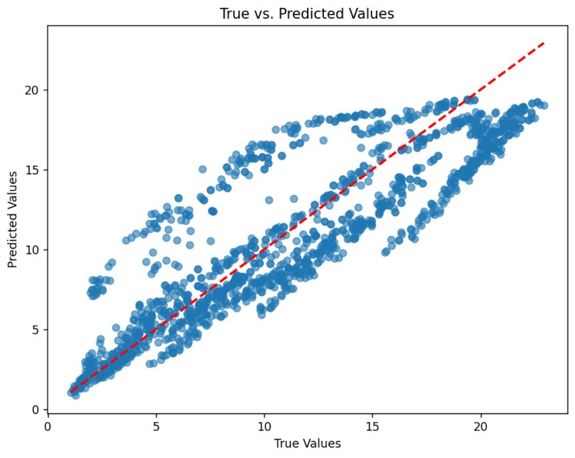
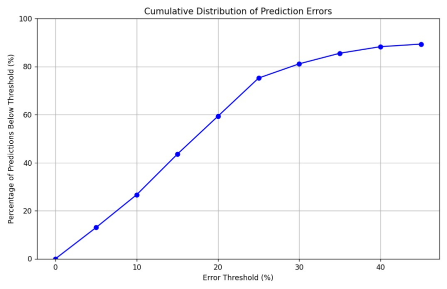
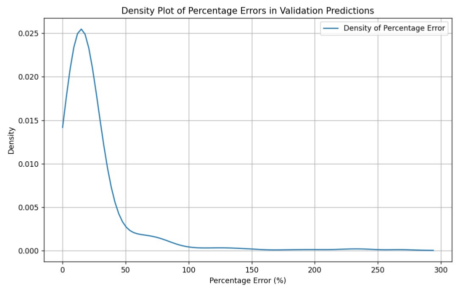


Figure A.7: Parity plot of the validation data using state 5.



(a) Cumulative Error Distribution Plot using state 5.



(b) Density Plot of Prediction Errors using state 5.

Figure A.8: Additional performance evaluation of the trained neural network using state 5.

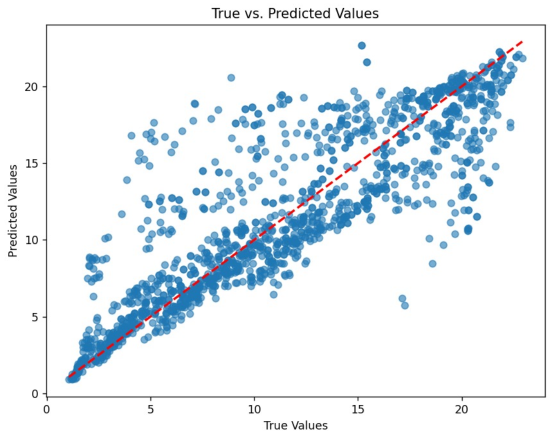
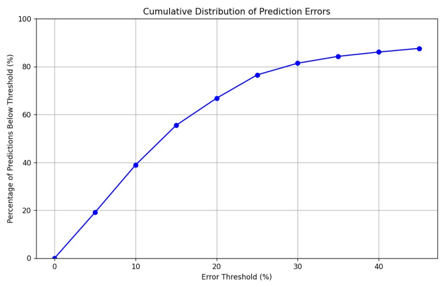
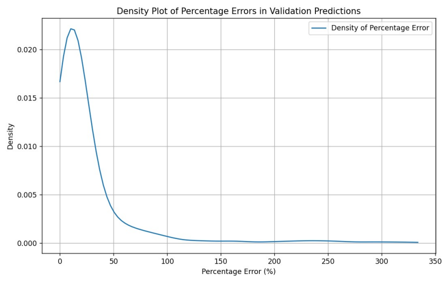


Figure A.9: Parity plot of the validation data using state 6.



(a) Cumulative Error Distribution Plot using state 6.



(b) Density Plot of Prediction Errors using state 6.

Figure A.10: Additional performance evaluation of the trained neural network using state 6.

A.2 Python code

```python
import traci
import sumolib
# this generates the actual trips used in the warm up simulation
from generate_trips_NN import run_full_pipeline

""""This file will create the warm up state for the city simulations.
    This will create a .xml file which can be loaded in as starting point for other simulati

def save_traffic_state(sumo_cmd, state_file, pre_sim_duration=600):
    """
    Runs a pre-simulation phase and saves the traffic state to a file.

    Parameters:
    - sumo_cmd: SUMO command for starting the simulation.
    - state_file: Path to save the state file.
    - pre_sim_duration: Duration of the pre-simulation phase (in seconds).
    """
    traci.start(sumo_cmd)
    for step in range(pre_sim_duration):
        traci.simulationStep()
    traci.simulation.saveState(state_file)
    traci.close()
    print(f"Successfully saved the state to {state_file}")


    input_file = state_file

    # Read the original file, filtering out lines that start with '<flowState id="'
    with open(input_file, "r") as file:
        lines = file.readlines()

    # Write back only lines that do not start with '<flowState id="'
    with open(input_file, "w") as file:
        for line in lines:
            if not line.strip().startswith('<flowState id="'):
                file.write(line)

if __name__ == "__main__":
    network_file = "sumo_rl/nets/PC-Alex/city_network.net.xml"
    output_file = "sumo_rl/nets/PC-Alex/warm_up_routing.rou.xml"
    fringe_factor = 1  # Adjust fringe factor as needed
    begin_time = 0
    end_time = 600
    trip_count = 2200

    # step 1: generate routing
    # Run the full pipeline with the given variables to create traffic
    run_full_pipeline(network_file, output_file, fringe_factor, begin_time, end_time, trip_c

    # Step 2: Save the initial traffic state
    sumoBinary = sumolib.checkBinary('sumo-gui')
    sumo_cmd = [
    sumoBinary,
    "-n", "sumo_rl/nets/PC-Alex/City_network.net.xml",
    "-r", output_file ,
    "--fcd-output", "fcd.xml",
    "--vehroute-output", "vehroutes.xml",
    "--default.speeddev", "0",
    "--no-step-log",
    "--summary-output", "summary_output.xml"]
    state_file = "City_warm_up.xml"
    save_traffic_state(sumo_cmd, state_file, end_time)
```

```
import subprocess
import os
import xml.etree.ElementTree as ET
import math
import random
import numpy as np
import time

" this file will generate trips for the simulation creating a route file"

def generate_center_focused_trips(network_file, output_file, center_nodes, outside_nodes_sta
    """
    Generates random trips focusing more on nodes near the center of the network.

    Parameters:
    - network_file: Path to the SUMO network file (.net.xml).
    - output_file: Path to save the generated routes file (.rou.xml).
    - center_nodes: List of nodes or edges near the center of the network to favor for start
    - fringe_factor: The fringe factor to influence trip start and end locations.
    - begin_time: The start time for generating trips.
    - end_time: The end time for generating trips.
    - trip_count: The number of trips to generate.
    """
    command = [
        "python",
        os.path.join(os.environ["SUMO_HOME"], "tools", "randomTrips.py"),
        "-n", network_file,
        "-r", output_file,
        "--fringe-factor", str(fringe_factor),
        "-b", str(begin_time),
        "-e", str(end_time),
        "-p", str((end_time - begin_time) / trip_count),
        "--trip-attributes", "",
        "--validate",
    ]

    # Generate trips with more focus on the center nodes
    try:
        # Create a temporary trips file with all trips and adjust start/end points
        temp_trip_file = output_file.replace('.xml', '_temp.xml')
        command.extend(["-o", temp_trip_file])

        # Run the command to generate trips
        subprocess.run(command, check=True)

        # Parse the generated trips and adjust their start/end points
        # this modifies the style and will only include a start and end point
        adjust_trips_to_center(temp_trip_file, output_file, center_nodes, outside_nodes_star

        print(f"Successfully generated center-focused trips and saved to {output_file}")
    except subprocess.CalledProcessError as e:
        print(f"Error generating trips: {e}")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")

def adjust_trips_to_center(temp_trip_file, output_file, center_nodes, outside_nodes_starting
    """
    Adjusts trips in the temporary trip file to favor starting or ending near center nodes.

    Parameters:
    - temp_trip_file: The temporary trips file to adjust.
    - output_file: Path to save the adjusted routes file (.rou.xml).
    - center_nodes: List of nodes to favor for trip start/end points.
    """

    # Initialize dictionaries to track usage counts for each node
    center_usage = {node: 0 for node in center_nodes}
    starting_usage = {node: 0 for node in outside_nodes_starting}
    exit_usage = {node: 0 for node in outside_nodes_exit}

    def weighted_choice(node_usage):
        """Select a node with a probability inversely proportional to its usage count, with
        # Adjust the weight factor to prioritize nodes with lower usage more strongly
        weight_factor = 3
        total = sum((1 / (count + 1)) ** weight_factor for count in node_usage.values())
        weights = [((1 / (count + 1)) ** weight_factor) / total for count in node_usage.valu
        chosen_node = random.choices(list(node_usage.keys()), weights=weights, k=1)[0]
        node_usage[chosen_node] += 1  # Update usage count for the chosen node
        return chosen_node

    tree = ET.parse(temp_trip_file)
    root = tree.getroot()

    # modify this to change the ratio of traffic going into the city
    percentage_of_trips_into_city = 0.7
    for trip in root.findall('trip'):
        if random.random() < percentage_of_trips_into_city:
            # Trip towards the center from the outside
            to_node = weighted_choice(center_usage)
            from_node = weighted_choice(starting_usage)
        else:
            # Trip from the center to the outside
            from_node = weighted_choice(center_usage)
            to_node = weighted_choice(exit_usage)

        # Set the 'from' and 'to' for each trip
        trip.set('from', from_node)
        trip.set('to', to_node)

    # Write adjusted trips to the final output file
    tree.write(output_file)
    os.remove(temp_trip_file)  # Clean up temporary file


def run_full_pipeline(network_file, output_file, fringe_factor=1, begin_time=600, end_time=3
    """
    Runs the full pipeline: generates trips and extracts unique routes.
    Arg: network_file: the .xml file which contains the network
    arg: output_file: file to write the assigned trips to
    arg: fringe_factor: degree of traffic spawning on edges compared to center
    arg: begin_time: start time for traffic
    arg: trip_count: how many trips will be created
    arg: traffic_dist: the type of traffic that will be created
    """
    print(f"Generating trips with output file: {output_file}")
    # all the edges that are not on or outside the perimeter
    center_nodes = ["X2Y4.5L", "X2Y4.5R", "X3Y4.5L", "X3Y4.5R", "X4Y4.5L", "X4Y4.5R", "X1.5Y
                    "X2Y3.5L", "X2Y3.5R", "X3Y3.5L", "X3Y3.5R", "X4Y3.5L", "X4Y3.5R", "X1.5Y
                    "X2Y2.5L", "X2Y2.5R", "X3Y2.5L", "X3Y2.5R", "X4Y2.5L", "X4Y2.5R", "X1.5Y
                    "X2Y1.5L", "X2Y1.5R", "X3Y1.5L", "X3Y1.5R", "X4Y1.5L", "X4Y1.5R",
                    ]

    # all outside edges that go into the perim (valid starting point)
    outside_nodes_starting = ["X1Y5.5L", "X2Y5.5L", "X3Y5.5L", "X4Y5.5L", "X5Y5.5L",
                     "X0.5Y5D", "X0.5Y4D", "X0.5Y3D", "X0.5Y2D", "X0.5Y1D",
                     "X1Y0.5R", "X2Y0.5R", "X3Y0.5R", "X4Y0.5R", "X5Y0.5R",
                     "X5.5Y5U", "X5.5Y4U", "X5.5Y3U", "X5.5Y2U", "X5.5Y1U",
                    ]

    outside_nodes_exit = ["X1Y5.5R", "X2Y5.5R", "X3Y5.5R", "X4Y5.5R", "X5Y5.5R",
                     "X0.5Y5U", "X0.5Y4U", "X0.5Y3U", "X0.5Y2U", "X0.5Y1U",
                     "X1Y0.5L", "X2Y0.5L", "X3Y0.5L", "X4Y0.5L", "X5Y0.5L",
                     "X5.5Y5D", "X5.5Y4D", "X5.5Y3D", "X5.5Y2D", "X5.5Y1D",
                    ]

    # Step 1: Generate random trips
    # in this function adjust_trips_to_center is called wich modifies the trip file to have
    generate_center_focused_trips(network_file, output_file, center_nodes, outside_nodes_sta

    # step 2: The new center focussed trips include a start and end point but require a rout
    generate_complete_routes(output_file, end_time, traffic_dist)

    # Step 3: Extract unique routes from the generated trip file
    input_route_file = output_file # input is the output file of generate_center_focused_tri
    output_route_file = output_file  #only adjust the content, not the name


def generate_complete_routes(input_route_file, end_time=3600, traffic_dist='exp(0.0043)'):
    """
    Uses duarouter to generate complete routes with intermediate edges and writes them to a

    Parameters:
    - input_route_file: Path to the input trip file (.xml).
    - output_route_file: Path to save the complete routes file (.rou.xml).
    - temp_trip_file: Path for a temporary .trips.xml file for duarouter processing.
    - end_time: End time for the flow generation.
    - traffic_dist: Defines the traffic distribution type (poisson, uniform, etc.)
    """

    # Step 1: Create a temporary .trips.xml file with <trip> elements

    # temp trip file to save the trips to
    temp_trip_file = input_route_file.replace('.xml', '_temp_input.xml')
    # this file will then be renamed to the input_route_file as this is the desired output f
    # the original input_route_file will be deleted after it's information is used
    temp_output_route_file = input_route_file.replace('.xml', '_temp_output.xml')

    tree = ET.parse(input_route_file)
    root = tree.getroot()

    # Write temporary trips file
    with open(temp_trip_file, 'w') as f:
        f.write('<?xml version="1.0" encoding="UTF-8"?>\n<trips>\n')
        for trip in root.findall('trip'):
            trip_id = trip.get('id')
            depart = trip.get('depart')
            from_edge = trip.get('from')
            to_edge = trip.get('to')
            f.write(f'    <trip id="{trip_id}" depart="{depart}" from="{from_edge}" to="{to_
        f.write('</trips>')

    # Step 2: Run duarouter to compute routes with all intermediate edges
    try:
        subprocess.run([
            "duarouter",
            "-n", "sumo_rl/nets/PC-Alex/city_network.net.xml",
            "-t", temp_trip_file,
            "-o", temp_output_route_file,
            "--ignore-errors",  # This prevents duarouter from stopping due to minor connect
        ], check=True)
        print(f"Routes successfully generated in {temp_output_route_file}")
    except subprocess.CalledProcessError as e:
        print(f"Error generating routes with duarouter: {e}")
        return

    # Step 3: Open the generated .rou.xml file and add flows if necessary
    tree = ET.parse(temp_output_route_file)
    root = tree.getroot()
    with open(temp_output_route_file, 'w') as f:
        f.write('<?xml version="1.0" encoding="UTF-8"?>\n<routes>\n')

        # Define vehicle types outside of the routes
        f.write('<vType id="CarA" accel="2.0" decel="4.5" sigma="0.5" length="5.0" maxSpeed=

        begin_time = 600 # end time of the warm up period
        # Write each unique route and associated flows
        for i, vehicle in enumerate(root.findall('vehicle')):
            route_id = vehicle.get('id')
            edges = vehicle.find('route').get('edges')
            f.write(f'    <route id="{route_id}" edges="{edges}"/>\n')

           # Adding flow with custom period based on traffic_dist
           # if period is equal to x it means there are x seconds between each vehicle creat
            if traffic_dist == 'step':
                # i loops over the vehicles, as more vehicles are created, the period betwee
                if 0 <= i < 1000:
                    period = 5 #light traffic
                elif 1000 <= i < 2000:
                    period = 3 # moderate traffic
                else:
                    period = 1  # heavy traffic
            elif traffic_dist == 'sinusoidal':
                # this creates a period with base x +/- 2 as math.sin(i / 1000.0) oscilliate
                period = 5 + 3 * math.sin(i / 10)
            elif traffic_dist.startswith("poisson"):
                # Extract the values inside the parentheses
                # slices up the input "poisson(x)", and takes the value between poisson( and
                avg_rate = float(traffic_dist[len("poisson("):-1])
                period = np.random.exponential(1 / avg_rate)
            else:
                period = traffic_dist

            if i % 50 ==0:
                begin_time += 200

            # after a while the simulation starts getting too full, this reduces the input
            if begin_time > 1000:
                avg_rate = float(traffic_dist[len("exp("):-1])
                avg_rate = 0.85*avg_rate
                period = f"exp({avg_rate})"
            if begin_time > 1500:
                avg_rate = float(traffic_dist[len("exp("):-1])
                avg_rate = 0.75*avg_rate
                period = f"exp({avg_rate})"
            if begin_time > 2000:
                avg_rate = float(traffic_dist[len("exp("):-1])
                avg_rate = 0.7*avg_rate
                period = f"exp({avg_rate})"

            f.write(f'    <flow id="flow{i+1}" route="{route_id}" type="CarA" beg="{begin_ti
        f.write('</routes>')




    # Clean up the temporary trip file
    os.remove(temp_trip_file)

    def safe_remove(file_path, retries=5, delay=2):
        """Attempts to remove a file with retries in case of permission issues."""
        for attempt in range(retries):
            try:
                os.remove(file_path)
                print(f"Successfully removed {file_path}")
                break
            except PermissionError:
                print(f"PermissionError on attempt {attempt + 1} to remove {file_path}. Retr
                time.sleep(delay)
        else:
            print(f"Failed to remove {file_path} after {retries} attempts.")
            raise PermissionError(f"Could not remove {file_path} after multiple attempts")
    safe_remove(input_route_file)

    time.sleep(0.1)
    # Rename `output_route_file` to `output_file`
    os.rename(temp_output_route_file, input_route_file)

if __name__ == "__main__":
    network_file = "sumo_rl/nets/PC-Alex/city_network.net.xml"
    output_file = "sumo_rl/nets/PC-Alex/unique_routes_NN.trips.xml"
    fringe_factor = 1
    begin_time = 600
    end_time = 3600
    trip_count = 4000

    # Run the full pipeline with the given variables
    run_full_pipeline(network_file, output_file, fringe_factor, begin_time, end_time, trip_c
    #generate_center_focused_trips(network_file, output_file, center_nodes, fringe_factor, b

    # Usage
    input_route_file = "sumo_rl/nets/PC-Alex/unique_routes_NN.trips.xml"  # Path to your ini
    output_route_file = "sumo_rl/nets/PC-Alex/unique_routes_NN.rou.xml"  # Path to save the
    temp_trip_file = "sumo_rl/nets/PC-Alex/temp.trips.xml"  # Temporary trip file for duarou

    generate_complete_routes(input_route_file, output_route_file, temp_trip_file)
```

```
import traci
import sumolib
# this generates the actual trips used in the war up simulation
from generate_trips_NN import run_full_pipeline

""""This file will create the warm up state for the city simulations.
    This will create a .xml file which can be loaded in as starting point for other simulati

def save_traffic_state(sumo_cmd, state_file, pre_sim_duration=600):
    """
    Runs a pre-simulation phase and saves the traffic state to a file.

    Parameters:
    - sumo_cmd: SUMO command for starting the simulation.
    - state_file: Path to save the state file.
    - pre_sim_duration: Duration of the pre-simulation phase (in seconds).
    """
    traci.start(sumo_cmd)
    for step in range(pre_sim_duration):
        traci.simulationStep()
    traci.simulation.saveState(state_file)
    traci.close()
    print(f"Successfully saved the state to {state_file}")


    input_file = state_file

    # Read the original file, filtering out lines that start with '<flowState id="'
    with open(input_file, "r") as file:
        lines = file.readlines()

    # Write back only lines that do not start with '<flowState id="'
    with open(input_file, "w") as file:
        for line in lines:
            if not line.strip().startswith('<flowState id="'):
                file.write(line)
```

```

if __name__ == "__main__":
    network_file = "sumo_rl/nets/PC-Alex/city_network.net.xml"
    output_file = "sumo_rl/nets/PC-Alex/warm_up_routing.rou.xml"
    fringe_factor = 1  # Adjust fringe factor as needed
    begin_time = 0
    end_time = 600
    trip_count = 2200

    # step 1: generate routing
    # Run the full pipeline with the given variables to create traffic
    run_full_pipeline(network_file, output_file, fringe_factor, begin_time, end_time, trip_c

    # Step 2: Save the initial traffic state
    sumoBinary = sumolib.checkBinary('sumo-gui')
    sumo_cmd = [
    sumoBinary,
    "-n", "sumo_rl/nets/PC-Alex/City_network.net.xml",
    "-r", output_file ,
    "--fcd-output", "fcd.xml",
    "--vehroute-output", "vehroutes.xml",
    "--default.speeddev", "0",
    "--no-step-log",
    "--summary-output", "summary_output.xml"]
    state_file = "City_warm_up.xml"
    save_traffic_state(sumo_cmd, state_file, end_time)
```

```python
import traci
import libsumo
import numpy as np
import pandas as pd
import xml.etree.ElementTree as ET
import collections
import sumolib
from shapely.geometry import LineString, Point
import generate_trips_NN
import time
from joblib import load
import math

"file used to collect data when the NN is used for decision making (iterative supervised lea

class SumoDataCollector:
    """Class to collect data for supervised learning"""

    def __init__(self, sumo_cmd, use_libsumo=False):
        self.sumo_cmd = sumo_cmd
        self.data = []
        self.avg_speeds_per_step = []
        self.cycle_mean_speed = 0
        self.total_arrived_vehicles = 0
        self.perim_box = 0, -200, 800, 600 #modify this if the permiter changes
        self.min_green_time = 8
        self.controlled_tl_lights = []
        self.vehicle_ids = []
        self.traffic_light_data = {}
        # Import the correct module based on `use_libsumo var`
        if use_libsumo:
            import libsumo as sumo_interface
            print(' using libsumo')
        else:
            import traci as sumo_interface
            print(' using traci')
        self.sumo_interface = sumo_interface

        # Parse the .net.xml file
        tree = ET.parse('sumo_rl/nets/PC-Alex/City_network.net.xml')
        root = tree.getroot()

        # Get all lanes
        self.all_lanes = []
        for edge in root.findall('edge'):
            for lane in edge.findall('lane'):
                self.all_lanes.append(lane.get('id'))

    def create_traffic(self, traffic_dist, trip_count):
        """"
        Generates the traffic flows for the simulation. This is called so that the NN can ha
        param: traffic_dist: intensity of traffic on the created routes as poisson arrival p
        param: trip_count: how many trips are present during the simulation (how busy the ne
        """
        network_file = "sumo_rl/nets/PC-Alex/city_network.net.xml"
        output_file = "sumo_rl/nets/PC-Alex/custom_route_NN.rou.xml"
        fringe_factor = 10
        begin_time = 0
        end_time = 3600

        # Call the function from the other script
        generate_trips_NN.run_full_pipeline(network_file, output_file, fringe_factor, begin_

    def run_simulation(self, traffic_dist, filename):
        """
            Run the simulation and collect (state, reward) tuples
            Param: traffic_dist: intensity of traffic that is used
            Param: filename: name of the save file
        """

        cycle_count = 0
        cycle_length = 90
        # for efficiency have the number of steps be a multiple of the cycle length so that
        # every 5 cycle's a data point is taken. Per simulation total_cycles/5 = data points
        total_cycles = 30
        look_ahead_cycles = 5 # over which horizon the rewards will be calculated
        num_steps = total_cycles*cycle_length

        # this var keeps track of the rewards over the last 5 cycles
        # needs to be reset every simulation, hence not initialized in init
        state_buffer = []
        reward_buffer = collections.defaultdict(list)

        # run the simulation for a amount of steps
        for step in range(num_steps):
            self.sumo_interface.simulationStep()  # Advances the simulation by one step
            self.vehicle_ids = traci.vehicle.getIDList()
            # Accumulate the total speed of all vehicles and count the vehicles
            speed_all_veh = np.mean([self.sumo_interface.vehicle.getSpeed(vid) for vid in se
            arrived_vehicles = self.sumo_interface.simulation.getArrivedIDList()
            self.total_arrived_vehicles += len(arrived_vehicles)
            self.avg_speeds_per_step.append(np.mean(speed_all_veh))

            # keep track of how many cycles are run given a policy
            # Every 5 cycles a data point is taken
            if step % cycle_length == 0 and step != 0:
                cycle_count += 1
                self.cycle_mean_speed = np.mean(self.avg_speeds_per_step)
                # empty list for next cycle
                self.avg_speeds_per_step = []

                # retrieve the current state
                current_state = self.get_state()

                # retrieve the best tl setting given the state and traffic dist
                action = self.get_best_tl_settings(current_state, traffic_dist)

                # apply the best found tl setting
                self.apply_action(action)

                state_buffer.append(current_state)

                # Get the reward for this cycle
                current_reward = self.calculate_reward()

                # Add the current reward to all states in the buffer
                for state_idx in range(len(state_buffer)):
                    reward_buffer[state_idx].append(current_reward)

                first_key = next(iter(reward_buffer))

                # Check if enough cycles have passed to link a state to its future rewards a
                if len(reward_buffer[first_key]) == look_ahead_cycles or step + look_ahead_c
                    # Link the oldest state to the cumulative reward of the past x cycles
                    # Initialize sums for each element
                    sum_0, sum_1, sum_2 = 0, 0, 0
                    for x in range(look_ahead_cycles):
                        sum_0 += reward_buffer[first_key][x][0]
                        sum_1 += reward_buffer[first_key][x][1]
                        sum_2 += reward_buffer[first_key][x][2]

                    # Results
                    cumulative_reward = [sum_0, sum_1, sum_2]
                    #print('state buffer pre pop', state_buffer)
                    past_state = state_buffer.pop(first_key) #returns and removes the oldest
                    reward_buffer.pop(first_key)
                    #print('reward_buffer post pop', reward_buffer)
                    self.data.append((past_state, action, cumulative_reward, self.traffic_di
                    self.save_data(filename)  # Save data after each cycle

    def apply_action(self, action):

        """
        Modifies the action order to allign correctly for tl on the perimeter according to t
        Then forwards the action to self.set_red_green_phases to implement the new tl config

        First action value is always inflow/outflow direction of PN forward+right turn
        Second action value is always inflow/outflow direction of PN left turn
        Third action is perpendicular forward+right turn
        Fourth action is perpendicular left turn
        Since the action is given to all tl lights on the perim and the phases always start
        the order must be changed depending on the position of the tl on the perim

        :param action: action given by rl algo
        """
        # Change traffic light configurations based on action
        traffic_lights = self.sumo_interface.trafficlight.getIDList()

        for tl in traffic_lights:
            #checks if the tl is on perimeter, else no modification, SUMO preset phases are
            # line counter is used to determine which line passed the " True" value for the
            on_line, line_counter = self.is_traffic_light_on_perimeter(tl)
            # only modify tl which are on perim line
            if on_line is True:
                #west side tl
                if line_counter == 0:
                    # Create a copy of the action array for the modified action
                    modified_action = action.copy()
                    # Swap the first and third values
                    modified_action[0], modified_action[2] = action[2], action[0]
                    # Swap the second and fourth values
                    modified_action[1], modified_action[3] = action[3], action[1]
                    # print('modified action west side tl', modified_action)
                    self.set_red_green_phases(modified_action, tl)
                # north side tl
                if line_counter ==1:
                    # already correct order
                    self.set_red_green_phases(action, tl)
                # east side tl
                if line_counter ==2:
                    # Create a copy of the action array for the modified action
                    modified_action = action.copy()
                    # Swap the first and third values
                    modified_action[0], modified_action[2] = action[2], action[0]
                    # Swap the second and fourth values
                    modified_action[1], modified_action[3] = action[3], action[1]
                    self.set_red_green_phases(modified_action, tl)
                # south side tl
                if line_counter ==3:
                    self.set_red_green_phases(action, tl)

    def get_best_tl_settings(self, state, traffic_dist):
        """"
        This function will use the NN to predict the reward given a TL setting and a state
        From these the TL setting that gives the best reward will be selected as action

        return: tl setting
        """
        # Load the model from the file
        # Modify the name of the correct model and scaler that is used based on iteration nu
        model = load('iter4_nn_model_rew2.joblib')
        scaler = load('x_scaler_rew2_iter4.joblib')
        best_reward = 0
        best_tl_setting = []

        def evaluate_expression(traffic_dist):
            """Extract the numerical part from expressions like 'exp(0.0041)', the return wi
            return float(traffic_dist[len("exp("):-1])

        def predict_reward(state, policy, traffic_dist):
            """
            Predicts the reward for a given state, policy, and trip count using the trained

            Parameters:
            state (list): The state features as a list of numerical values.
            policy (list): The policy features as a list of numerical values.
            trip_count (int): The number of trips as an integer.

            Returns:
            float: The predicted reward.
            """

            # modify the format of the traffic dist from string to float
            traffic_dist = evaluate_expression(traffic_dist)

            # Combine the state, policy, and trip_count into a single feature vector
            new_input = np.hstack((state, policy, traffic_dist))

            # Standardize the input using the loaded scaler
            new_input_scaled = scaler.transform([new_input])

            # Predict the reward using the trained model
            predicted_reward = model.predict(new_input_scaled)

            # Return the predicted reward
            return predicted_reward

        # List of tl settings for the NN to choose from
        tl_settings = [
            [27, 12, 27, 12],
            [26, 13, 26, 13],
            [25, 14, 25, 14],
            [24, 15, 24, 15],
            [28, 11, 28, 11],
            [29, 10, 29, 10],
            [30, 9, 30, 9],
            [31, 8, 31, 8],
            [32, 9, 28, 9],
            [34, 10, 24, 10],
            [36, 11, 20, 11],
            [37, 12, 17, 12],
            [33, 11, 23, 11],
            [28, 9, 32, 9],
            [24, 10, 34, 10],
            [20, 11, 36, 11],
            [17, 12, 37, 12],
            [23, 11, 33, 11],
            [9, 28, 9, 28],
            [15, 24, 15, 24],
            [18, 21, 18, 21],
            [21, 18, 21, 18],
            [19.5, 19.5, 19.5, 19.5],
        ]

        list_of_predicted_rewards = []
        #loop over all tl_settings to find the best one
        for tl_setting in tl_settings:
            # Predict the reward
            predicted_reward = predict_reward(state, tl_setting, traffic_dist)
            predicted_reward = predicted_reward.item()
            list_of_predicted_rewards.append(predicted_reward)
            # keep track of the best reward and the associated tl_setting
            if predicted_reward >= best_reward:
                best_tl_setting = tl_setting
                best_reward = predicted_reward
        return best_tl_setting

    def set_simulation_settings(self, policy_settings, traffic_dist_array, route_count_array
        """
        Will define the tl policy and traffic and run simulation
        :param: policy_settings: list of multiple different inputs for the traffic light set
        :param: traffic_dist_array: list of the type of traffic dist
        :param: route_count_array: amount of trips that are created
        """
        # policy will change the traffic light settings
        for policy in policy_settings:
            # traffic will adjust the kind of traffic in the simulation
            for traffic_dist in traffic_dist_array:
                # trip count determines how many cars will be in the simulation
                for trip_count in route_count_array:
                    self.traffic_dist = traffic_dist
                    # keep track of the duration of one simulation
                    start_time = time.time()
                    # create the desired traffic for this simulation
                    self.create_traffic(traffic_dist, trip_count)
                    self.total_trips = trip_count # keep track of this to use for TCR
                    # run warm up for each simulation as load-state is used in sumo_cmd
                    self.sumo_interface.start(self.sumo_cmd)
                    # adjust the traffic light policy to the used policy after warm up
                    self.set_traffic_light(policy)
                    # run the actual simulation
                    self.run_simulation(traffic_dist, file_name)
                    self.sumo_interface.close()
                    print(' *!*!*!*!*!*!*!!*!*!*!*!*!*!*!*!*')
                    print('------------one simulation duration is', time.time() - start_time


    def get_state(self):
        """
        Extract the current state
        Returns: state (array)
        """

        total_vehicles_inside = 0
        outside_queue_count = 0
        inside_queue_count = 0

        # assumed we look at the complete network
        # elsewise check for controlled tl's -> controlled lanes -> filter for unique
        travel_times_per_veh = []
        for vehicle_id in self.vehicle_ids:
            travel_times_per_veh.append(self.sumo_interface.vehicle.getAccumulatedWaitingTim
        total_travel_time = np.sum(travel_times_per_veh)
        avg_travel_time = np.mean(travel_times_per_veh)

        lane_occupancy_per_lane = [self.sumo_interface.lane.getLastStepOccupancy(lane) for l
        total_lane_occupancy = sum(lane_occupancy_per_lane)
        avg_lane_occupancy = np.mean(lane_occupancy_per_lane)
        vehicle_ids = self.sumo_interface.vehicle.getIDList()
        total_vehicles = len(vehicle_ids)
        queu_per_lane = [self.sumo_interface.lane.getLastStepHaltingNumber(lane) for lane in
        total_queue_length = sum(queu_per_lane)
        avg_queue_length = np.mean(queu_per_lane)
        avg_speed = self.cycle_mean_speed
        inflowing_vehicles, outflowing_vehicles = self.get_inflowing_outflowing_vehicles()

        # Get queue length for each traffic light-controlled lane
        tl_queue_lengths = []
        traffic_lights = self.sumo_interface.trafficlight.getIDList()
        for tl_id in traffic_lights:
            controlled_lanes = self.sumo_interface.trafficlight.getControlledLanes(tl_id)
            tl_queue_length = sum(self.sumo_interface.lane.getLastStepHaltingNumber(lane) fo
            tl_queue_lengths.append(tl_queue_length)


        # Calculate the total speed and number of vehicles inside and outside perimeter
        for vehicle_id in self.vehicle_ids:
            if self.is_vehicle_inside_perimeter(vehicle_id):
                total_vehicles_inside +=1
                if traci.vehicle.getSpeed(vehicle_id) < 0.1:  # Define queue as vehicle movi
                    inside_queue_count += 1
            else: #not inside so vehicles outside perimeter
                if traci.vehicle.getSpeed(vehicle_id) < 0.1:  # Define queue as vehicle movi
                    outside_queue_count += 1
        total_vehicles_outside = total_vehicles-total_vehicles_inside


        # Combine all state elements including traffic light queue lengths
        # state = np.array([
        #     total_lane_occupancy, avg_lane_occupancy, total_vehicles, total_vehicles_insid
        #     total_vehicles_outside, avg_speed, total_queue_length, avg_queue_length,
        #     outside_queue_count, inside_queue_count, total_travel_time, avg_travel_time,
        #     inflowing_vehicles, outflowing_vehicles
        # ] + tl_queue_lengths)

        state = np.array([total_lane_occupancy, avg_lane_occupancy, total_vehicles, total_ve
avg_speed,
                             # 6                    7                    8
9               10                      11
                             total_queue_length, avg_queue_length, outside_queue_count, inside_q
                             # 12                   13
                             inflowing_vehicles, outflowing_vehicles])
        return state

    def set_traffic_light(self, policy):

            """
            Modifies the policy order to allign correctly for tl on the perimeter according
            Then forwards the policy to self.set_red_green_phases to implement the new tl co

            First policy value is always inflow/outflow direction of PN forward+right turn
            Second policy value is always inflow/outflow direction of PN left turn
            Third policy is perpendicular forward+right turn
            Fourth policy is perpendicular left turn
            Since the policy is given to all tl lights on the perim and the phases always st
            the order must be changed depending on the position of the tl on the perim

            :param policy: policy given by rl algo
            """
            # Change traffic light configurations based on policy
            traffic_lights = self.sumo_interface.trafficlight.getIDList()

            for tl in traffic_lights:
                #checks if the tl is on perimeter, else no modification, SUMO preset phases
                # line counter is used to determine which line passed the " True" value for
                on_line, line_counter = self.is_traffic_light_on_perimeter(tl)
                # only modify tl which are on perim line
                if on_line is True:
                    #west side tl
                    if line_counter == 0:
                        # Create a copy of the policy array for the modified policy
                        modified_policy = policy.copy()
                        # Swap the first and third values
                        modified_policy[0], modified_policy[2] = policy[2], policy[0]
                        # Swap the second and fourth values
                        modified_policy[1], modified_policy[3] = policy[3], policy[1]
                        # print('modified policy west side tl', modified_policy)
                        self.set_red_green_phases(modified_policy, tl)
                    # north side tl
                    if line_counter ==1:
                        # already correct order
                        self.set_red_green_phases(policy, tl)
                    # east side tl
                    if line_counter ==2:
                        # Create a copy of the policy array for the modified policy
                        modified_policy = policy.copy()
                        # Swap the first and third values
                        modified_policy[0], modified_policy[2] = policy[2], policy[0]
                        # Swap the second and fourth values
                        modified_policy[1], modified_policy[3] = policy[3], policy[1]
                        self.set_red_green_phases(modified_policy, tl)
                    # south side tl
                    if line_counter ==3:
                        self.set_red_green_phases(policy, tl)

    def set_red_green_phases(self, action, tl_id):
        """
        Applies the action given by the rl algo for all tl on the perimeter (this check is a

        First action value is always inflow/outflow direction of protected network (PN) forw
        Second action value is always inflow/outflow direction of PN left turn
        Third action is perpendicular forward+right turn
        Fourth action is perpendicular left turn
        Since the action is given to all tl lights on the perim and the phases always start
        the order must be changed depending on the position of the tl on the perim

        :param: action: action given by rl algo
        :param: tl_id: traffic light which should be modified
        """

        # keep track of the original program which will be modified so that we prevent modif
        if not hasattr(self, 'original_program'):
            # Get the current (original) program logic
            current_program_tuple = self.sumo_interface.trafficlight.getAllProgramLogics(tl_

            current_program = current_program_tuple[0]

            # Correct the phase states
            corrected_phases = self.init_correct_phase_states(current_program.phases)

            # Create a new traffic light program with the corrected phases
            corrected_program = self.sumo_interface.trafficlight.Logic(
                current_program.programID,
                current_program.type,
                current_program.subParameter,
                corrected_phases
            )

            # Store the modified program
            # modified is removing the conflicting green phases, action is NOT yet applied
            self.original_program = corrected_program
        else:
            # Use the stored original program for modification
            current_program = self.original_program

        action_counter = 0
        modified_phases = []
        for phase in current_program.phases:
            # If the phase contains yellow lights ('y'), skip modifying it
            if 'y' in phase.state or 'Y' in phase.state:
                modified_phases.append(phase)  # Keep the yellow phase unchanged
                continue
            # Modify the duration of the non yellow phases
            if action_counter < len(action):
                new_duration = action[action_counter]
                # prevent durations being smaller than the min green time (should not be pos
                if new_duration <= self.min_green_time:
                    new_duration = self.min_green_time
                action_counter += 1
            else:
                new_duration = phase.duration
            # Create a new phase with the modified duration and the same state
            modified_phases.append(self.sumo_interface.trafficlight.Phase(new_duration, phas


        # Create a new traffic light program with the modified phases
        program = self.sumo_interface.trafficlight.Logic("0", 0, 0, modified_phases)

        # Set the new traffic light program
        self.sumo_interface.trafficlight.setProgramLogic(tl_id, program)

    def calculate_reward(self):
        """
        Calculate the reward for the current state-policy pair
        return: reward (list of 3 floats)
        """
        mean_waiting_time_network = 0
        # Calculate the total speed and number of vehicles inside and outside perimeter
        vehicle_ids = self.sumo_interface.vehicle.getIDList()
        for vehicle_id in vehicle_ids:
            mean_waiting_time_network += self.sumo_interface.vehicle.getWaitingTime(vehicle_

        # Calculate trip completion rate, total started trips/ arrived trips
        trip_completion_rate = (self.total_arrived_vehicles / self.total_trips) * 100

        # play around with the weights for optimal performance and so that all terms have re
        w1 = 1
        w2 = 1
        w3 = 1

        #reward = -w1*  mean_waiting_time_network + w2 * trip_completion_rate + w3* self.cyc
        reward = [mean_waiting_time_network, trip_completion_rate,  self.cycle_mean_speed]


        return reward

    def save_data(self, filename):
        """Append collected data to a CSV file continuously."""
        df = pd.DataFrame(self.data, columns=["state", "policy", "reward", "traffic_dist", "
        # Use 'mode="a"' to append, and 'header=False' to avoid writing the header repeatedl
        df.to_csv(filename, mode='a', header=False, index=False)
```

```
            # Clear data after saving to prevent duplicate entries
            self.data = []

    def is_traffic_light_on_perimeter(self, tl_id):
            """
            Checks if a traffic light is on the perimeter
            :param tl_id: The ID of the traffic light.
            :return: True if the tl is on the perimeter, False otherwise.

            Note that this approach does not work for non-straight roads

            """
            lane_coordinates = []
            x_coordinates = []
            y_coordinates = []

            # Get a list of lanes controlled by the tl
            controlled_lanes = traci.trafficlight.getControlledLanes(tl_id)

            # Loop over all lanes and get their start and end coordinates. (all lanes are straig
            for lane in controlled_lanes:
                lane_coordinates.append(traci.lane.getShape(lane))

            # Get the x and y coordinates of the end points of the lane (where the tl is),
            for x in range(len(controlled_lanes)):
                x_coordinates.append(lane_coordinates[x][-1][0])
                y_coordinates.append(lane_coordinates[x][-1][1])

            x_coordinate_tl = np.mean(x_coordinates)
            y_coordinate_tl = np.mean(y_coordinates)


            xmin, ymin, xmax, ymax = self.perim_box
            coordinates_tl = [x_coordinate_tl, y_coordinate_tl]

            # Define the corners of the perimeter
            corners = [
                (xmin, ymin),  # Bottom-left corner
                (xmin, ymax),  # Top-left corner
                (xmax, ymax),  # Top-right corner
                (xmax, ymin)   # Bottom-right corner
            ]

            # Check if the TL is located at any of the corners, corners will not be considered a
            for corner in corners:
                if np.isclose(coordinates_tl[0], corner[0], atol=10) and np.isclose(coordinates_
                    # print('tl id at corner is', tl_id)
                    return False, None  # TL is at the corner, return False

            # Define the sides of the perimeter box as LineString objects
```

```
        lines = [
            LineString([(xmin, ymin), (xmin, ymax)]),  # Left side
            LineString([(xmin, ymax), (xmax, ymax)]),  # Top side
            LineString([(xmax, ymax), (xmax, ymin)]),  # Right side
            LineString([(xmax, ymin), (xmin, ymin)])   # Bottom side
        ]


        # Depending on which iteration of line the statement is true, the position of the tl
        # Each loop of the line in lines correspond to the next lines above so left, top, ri
        # Check if the point is near any of the lines
        line_counter = 0
        for line in lines:
            if self.point_near_line(coordinates_tl, line):
                on_line = True
                return on_line, line_counter
            else:
                on_line = False
                line_counter += 1

        return on_line, line_counter

    def is_vehicle_inside_perimeter(self, vehicle_id):
        """
        Check if a vehicle is inside the perimeter.

        :param vehicle_id: The ID of the vehicle.
        :return: True if the vehicle is inside the perimeter, False otherwise.
        """
        # x, y coordinate of vehicle
        x, y = traci.vehicle.getPosition(vehicle_id)
        # coordinates of the perimeter
        xmin, ymin, xmax, ymax = self.perim_box
        # check if a vehicle is within the perimeter
        return xmin <= x <= xmax and ymin <= y <= ymax

    def point_near_line(self, point, line, tolerance=20.0):
        """
        Checks if the point is within a certain tolerance of a line.

        :param point: A tuple (x, y) representing the point.
        :param line: A LineString object representing the line.
        :param tolerance: The tolerance within which the point should be from the line (in m
        :return: True if the point is within the tolerance of the line, False otherwise.
        """
        point_obj = Point(point)
        # determines the euclidian distance from the point to the line
        distance = point_obj.distance(line)
        # If the distance is more than the tolerance it will return False, else True
        return distance <= tolerance

    def get_inflowing_outflowing_vehicles(self):
        """
            Get the count of vehicles inflowing into the perimeter and outflowing out of the
            Return: number of inflowing vehicles (int)
            Return: number of outflowing vehicles (int)
        """
        inflow_count = 0
        outflow_count = 0

        # Track the vehicles' previous positions (inside or outside perimeter)
        if not hasattr(self, 'previous_positions'):
            self.previous_positions = {}

        for vehicle_id in self.vehicle_ids:
            # Check if the vehicle is currently inside the perimeter
            is_inside_now = self.is_vehicle_inside_perimeter(vehicle_id)
            was_inside_before = self.previous_positions.get(vehicle_id, False)

            # Count inflow if the vehicle was outside before but is now inside
            if not was_inside_before and is_inside_now:
                inflow_count += 1
            # Count outflow if the vehicle was inside before but is now outside
            if was_inside_before and not is_inside_now:
                outflow_count += 1

            # Update the previous position tracking
            self.previous_positions[vehicle_id] = is_inside_now

        return inflow_count, outflow_count


    def init_correct_phase_states(self, phases):
            """
            Modifies the states of all traffic light phases by changing all 'g' (green with
            We only want G (green without confliction)
            Parameters:
            - phases (list of self.sumo_interface.trafficlight.Phase): A list of traffic lig

            Returns:
            - list of self.sumo_interface.trafficlight.Phase: The modified list of traffic l
            """

            # all conflicted greens should be removed already in netedit
            corrected_phases = []
            for phase in phases:
                # Modify the phase state to change all 'g' to 'r'
                new_state = phase.state.replace('g', 'r')
                # Create a new phase with the modified state and original duration
                corrected_phase = self.sumo_interface.trafficlight.Phase(phase.duration, new
                corrected_phases.append(corrected_phase)
            return corrected_phases

sumoBinary = sumolib.checkBinary('sumo') # change to sumo-gui if you want to use gui. gui ca
sumo_cmd = [
    sumoBinary,
    "-n", "sumo_rl/nets/PC-Alex/City_network.net.xml",
    "-r", "sumo_rl/nets/PC-Alex/custom_route_NN.rou.xml" ,
    "--fcd-output", "fcd.xml",
    "--vehroute-output", "vehroutes.xml",
    "--default.speeddev", "0",
    "--no-step-log",
    "--summary-output", "summary_output.xml",
    "--load-state", "City_warm_up.xml" # using load_state with city warm up directly uses th
    # state of city_warm_up as starting point of the simulation which means no additional wa
 ]


data_collector = SumoDataCollector(sumo_cmd)


def initialize_settings(run_training=True):
    """
    Initialize_settings for either training or validation based on the input.

    Parameters:
    run_training (bool): If True, run training simulations. If False, run validation simulat
    """
    if run_training:
        # Training settings
        #phase  = 90s, yellow time = 3*4=12, remaining time 78

        policy_settings = [
            [29, 10, 29, 10],  # standard traffic light configurations
            [31, 8, 31, 8],    # extended duration for main phases
            [27, 12, 27, 12],  # extended duration for the secondary phases
            [32, 11, 24, 11],  # extended duration for phase into perimeter
            [24, 11, 32, 11],  # extended duration for phase out of perimeter
            [30, 11, 26, 11],  # equally divided phase durations
            [28, 11, 28, 11],  # standard traffic light configurations
            [31, 8, 31, 8],    # extended duration for main phases
            [25, 14, 25, 14],  # extended duration for the secondary phases
            [35, 11, 21, 11],  # extended duration for phase into perimeter
            [21, 11, 35, 11],  # extended duration for phase out of perimeter
            [19.5, 19.5, 19.5, 19.5],
            [26, 13, 26, 13],
            [29, 10, 26, 13],
            [28, 13, 25, 12],
            [34, 9, 25, 10],
            [33, 9, 26, 10],
            [32, 10, 26, 10],
            [30, 12, 26, 10],
            [27, 11, 27, 13],
            [25, 11, 30, 12],
            [28, 12, 26, 12],
            [33, 9, 24, 12],
            [32, 9, 25, 12],
            [31, 10, 25, 12],
            [26.5, 12.5, 26.5, 12.5],

        ]
        traffic_dist_array =  ['exp(0.0041)','exp(0.00435)', 'exp(0.00445)', 'exp(0.00425)',
                               'exp(0.0044)','exp(0.0043)', 'exp(0.0045)', 'exp(0.0046)', 'e
                                 'exp(0.00475)',  'exp(0.00485)',  'exp(0.00415)',
'exp(0.00435)']
        route_count_array = [1000] # this is kept constant, instead traffic intensity is cha
        # if this is changed as well the traffic intensity changes too drastically
    else:
        # Validation settings
        policy_settings = [
            [20, 15, 20, 23],
            [18, 18, 21, 21],
            [17, 19, 20, 22],
            [24, 12, 18, 24],
            [19, 20, 19, 20],
            [15, 17, 23, 23],
            [21, 11, 23, 23],
            [22, 14, 19, 23],
            [16, 18, 22, 22],
            [23, 9, 23, 23]
        ]
        traffic_dist_array = ['exp(0.0047)', 'exp(0.0042)','exp(0.0049)', 'exp(0.00428', 'ex
        route_count_array = [1000]
    return policy_settings, traffic_dist_array, route_count_array


run_training = True # True means training data will be created, false means validation data

# modify the name of the save file to match the iter run
if run_training is True:
    file_name = 'sumo_training_data_iter5.csv'
else:
    file_name = 'sumo_validation_data_iter5.csv'

policy_settings, traffic_dist_array, route_count_array = initialize_settings(run_training)

data_collector.set_simulation_settings(policy_settings, traffic_dist_array, route_count_arra
```

```
import matplotlib.pyplot as plt
```

```python
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import train_test_split, GridSearchCV, RandomizedSearchCV, cros
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_absolute_error
import numpy as np
import pandas as pd
import ast
import re
from joblib import dump

# Load and preprocess the data
train_filename = "sumo_training_data2.csv"
train_df = pd.read_csv(train_filename, names=["state", "policy", "reward", "traffic_dist", "

valid_filename = "sumo_validation_data_testing.csv"
valid_df = pd.read_csv(valid_filename, names=["state", "policy", "reward", "traffic_dist", "


# Function to format and convert state vector strings
def format_state_vector(state_str):
    # Replace spaces with commas while ignoring scientific notation parts
    state = re.sub(r'(?<=\d)\s+(?=[\d\-])', ', ', state_str)
    return state
train_df["state"] = train_df["state"].apply(format_state_vector)
valid_df["state"] = train_df["state"].apply(format_state_vector)

# Define various subsets of state vector indices
state_index_combinations = [
    [0,2,3,4,5,6,8,9,10,12,13], #avg
    [1,2,3,4,5,7,8,9,11,12,13], #total
    [1, 3, 4, 5, 8, 9, 12, 13], # intuitively most important
    [5], # only avg speed
    [5,8,9], # avg speed, queue inside and outside
    [3,4,5], # vehicles inside, outside, avg speed
]

def preprocess_data(df, state_indices=None):
    """ Preprocess data with optional state vector subset selection. """
    def fix_format(data_string):
        data_string = data_string.strip()
        data_string = re.sub(r'\s+', ' ', data_string)
        fixed_string = re.sub(r'(\d)\s+(\d)', r'\1, \2', data_string)
        return fixed_string

    df["state"] = df["state"].apply(fix_format).apply(ast.literal_eval)
    df["policy"] = df["policy"].apply(fix_format).apply(ast.literal_eval)

    def extract_inner_value(value):
        return float(value[len("exp("):-1])
    df['traffic_dist'] = df['traffic_dist'].apply(extract_inner_value)


    states = df["state"].apply(lambda x: [x[i] for i in state_indices] if state_indices else
    policies = df["policy"].tolist()
    rewards = df["reward"].tolist()
    traffic_dist = np.array(df["traffic_dist"].tolist()).reshape(-1, 1)

    X = np.hstack((states, policies, traffic_dist))
    reward_number = 2
    y = np.array([ast.literal_eval(reward)[reward_number] for reward in rewards], dtype=np.f

    return X, y

# Set up scalers
x_scaler = MinMaxScaler()
y_scaler = MinMaxScaler()

def run_optimize_hyperparameter(X_train, y_train, X_valid, y_valid):
    """ Optimize the NN hyperparameters with random search and return best model and perform
    model = MLPRegressor(max_iter=5000, random_state=42, early_stopping=True)

    # Define the parameter distribution for Random Search
    param_dist = {
        'hidden_layer_sizes': [(128, 128, 64, 32), (128, 64, 32, 16), (256, 128, 64, 32, 16)
        'activation': ['relu'],
        'solver': ['adam'],
        'alpha': np.logspace(-4, -1, 4),
        'learning_rate': ['constant', 'adaptive'],
        'batch_size': [32, 64, 128],
        'early_stopping': [True]
    }

    # Set up Random Search
    random_search = RandomizedSearchCV(
        estimator=model,
        param_distributions=param_dist,
        scoring='neg_mean_squared_error',
        cv=3,
        n_iter=200,  # Number of random combinations to try
        verbose=2,
        random_state=42,
        n_jobs=-1
    )

    random_search.fit(X_train, y_train)

    # Get the best model and evaluate on validation set
    best_model = random_search.best_estimator_
    best_model.fit(X_train, y_train)  # Fit best model on full training data

    predictions = best_model.predict(X_valid)
    original_scale_predictions = y_scaler.inverse_transform(predictions.reshape(-1, 1)).flat
    original_scale_y_valid = y_scaler.inverse_transform(y_valid.reshape(-1, 1)).flatten()

    mae = mean_absolute_error(original_scale_y_valid, original_scale_predictions)
    rmse = np.sqrt(np.mean((original_scale_y_valid - original_scale_predictions) ** 2))

    return random_search.best_params_, mae, rmse

# Experiment with different state subsets
results = []
for state_indices in state_index_combinations:
    print(f"Testing state indices: {state_indices}")

    # Preprocess data with the selected subset of state indices
    X_train, y_train = preprocess_data(train_df, state_indices)
    X_valid, y_valid = preprocess_data(valid_df, state_indices)

    # Scale the data
    X_train = x_scaler.fit_transform(X_train)
    X_valid = x_scaler.transform(X_valid)
    y_train = y_scaler.fit_transform(y_train.reshape(-1, 1)).flatten()
    y_valid = y_scaler.transform(y_valid.reshape(-1, 1)).flatten()

    # Run NN training and get performance metrics
    best_params, mae, rmse = run_optimize_hyperparameter(X_train, y_train, X_valid, y_valid)
    results.append({
        'state_indices': state_indices,
        'MAE': mae,
        'RMSE': rmse,
        'best_params': best_params
    })
    # print(f"MAE: {mae}, RMSE: {rmse}")


# Convert results to a DataFrame and display
results_df = pd.DataFrame(results)

# Find the top 3 rows with the lowest MAE
top_3_mae = results_df.nsmallest(3, 'MAE')[['state_indices', 'MAE', 'RMSE', 'best_params']]

# Find the top 3 rows with the lowest RMSE
top_3_rmse = results_df.nsmallest(3, 'RMSE')[['state_indices', 'MAE', 'RMSE', 'best_params']

# Save the top 3 MAE and top 3 RMSE results to CSV files
top_3_mae.to_csv("top_3_mae_results.csv", index=False)
top_3_rmse.to_csv("top_3_rmse_results.csv", index=False)

# Save the results DataFrame to a CSV file
results_df.to_csv("nn_optimization_results.csv", index=False)
```

```

# Display the results
print("Top 3 by MAE:")
print(top_3_mae)

print("\nTop 3 by RMSE:")
print(top_3_rmse)

print(results_df)
```

```
import gymnasium
import gymnasium.spaces.discrete
import traci
import numpy as np
from stable_baselines3 import PPO
import os
import time
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'  # Suppress INFO and WARNING logs
import tensorflow as tf
from stable_baselines3.common.callbacks import BaseCallback
tf.debugging.set_log_device_placement(True)
import matplotlib.pyplot as plt
from shapely.geometry import Point, LineString
from stable_baselines3.common.vec_env import SubprocVecEnv
from stable_baselines3.common.monitor import Monitor
import sumolib
import re
import generate_trips_NN
# import uuid
#import self.sumo_interface as tc


# class AvgSpeedLoggingCallback(BaseCallback):
#     def __init__(self, summary_writer, verbose=0):
#         super(AvgSpeedLoggingCallback, self).__init__(verbose)
#         self.summary_writer = summary_writer

#     def _on_step(self):

#         avg_speeds = []
#         for env in self.model.get_env().envs:  # Get all environments
#             avg_speeds.append(env.sumo_env.self.avg_speed)  # Assuming `avg_speed` attribu

#         # Calculate the average speed across all environments
#         avg_speed_across_envs = sum(avg_speeds) / len(avg_speeds)

#         # Log the average speed into TensorBoard using the provided summary writer
#         with self.summary_writer.as_default():
#             tf.summary.scalar('avg_speed_across_envs', avg_speed_across_envs, step=self.ep
```

```python


class SumoEnvironment():
    """
        Environment for SUMO
        Class that contains all functions that interact with SUMO
    """

    def __init__(self, sumo_cfg, max_steps, rank, use_libsumo=False):

        # Choose between sumo and libsumo
        self.use_libsumo = use_libsumo

        # Import the correct module based on `use_libsumo`
        if self.use_libsumo:
            import libsumo as sumo_interface
            print('using libsumo')
        else:
            import traci as sumo_interface
            print('using traci')
        self.sumo_interface = sumo_interface

        # main config file provided
        self.sumo_cfg = sumo_cfg
        # max seconds of simulation
        self.max_steps = max_steps
        self.current_step = 0
        self.avg_speeds_per_step = []
        self.all_avg_speeds = []
        self.use_gui = False
        self.episode_counter = 0
        self.phases = []
        self.avg_speed_per_episode = []
        self.previous_phase = {}
        self.log_file_path = "avg_speed_log_phases_simple.txt"  # Path to the log file
        # xmin, ymin, xmax, ymax coordinates of the perimeter in meters
        self.perim_box = 0, -200, 800, 600 #modify this if the permiter changes
        self.num_rewards = 0
        self.reward_mean = 0
        self.reward_var = 0
        self.avg_speed_network = 0
        self.min_green_time = 8
        self.on_line_tl_ids = []
        self.avg_speed_over_simulation = []
        self.episode_90_sec_mean = []
        self.rank = rank
        self.vehicle_data = []
        self.avg_speed = []
        self.avg_speed_90sec = []
        self.avg_speed_per_sec = []
        self.mean_waiting_time_90_sec = []
        self.mean_waiting_per_sec = []
        self.waiting_time_per_second = []
        self.change_in_waiting_time_per_second = []
        self.reward_list = []
        self.model = None #later defined in set_model
        self.prev_waiting_time = 0
        self.change_in_waiting_time = []
        self.speed_inside_perim_90_sec = []
        self.speed_outside_perim_90_sec = []
        self.avg_speed_inside_perim_per_sec = []
        self.avg_speed_outside_perim_per_sec = []
        self.avg_speed_inside_perim = []
        self.avg_speed_outside_perim = []
        self.start_time = 0
        self.end_time = 0
        self.total_arrived_vehicles = 0
        self.total_departed_vehicles = 0
        self.traffic_light_data = {}
        self.traffic_lights = []
        self.times_reset_is_called = 0
        #self.create_traffic()

        super(SumoEnvironment, self).__init__()

        # Initialize log file
        with open(self.log_file_path, 'w') as f:
            f.write("Episode,Average Speed\n")  # Write header


    def set_model(self, model):
        """Sets the model to the environment, so that we can log PPO-related information in
        self.model = model

    @staticmethod
    def make_sumo_env(sumo_cfg, max_steps, rank, seed=0):
        """
        Utility function to create a sumo environment and set the seed for it.
        :param sumo_cfg: SUMO config file path
        :param max_steps: maximum steps per episode
        :param rank: rank of the environment (index)
        :param seed: random seed for the environment
        :return: the environment instance
        """
        def _init():
            env = SumoGymEnv(sumo_cfg, max_steps, rank)
            env = Monitor(env)  # Wrapping the environment with Monitor
            env.rank = rank
            env.reset(seed=seed + rank)  # Set a different seed for each environment
            return env
            return _init


    def reset(self, *, seed=None, options=None):
        """
        Reset function that resets the SUMO environment
        ****implement starting sim x amount of steps in, add seed, options******

        Parameters:
        - seed: optional specific seed for the simulation run
        - options:

        Returns:
        - state: state of the environment
        - info: mandatory param for RL algo, can be empty dict
        """
        print(f"-----------------Reset called for episode {self.episode_counter}")
        # if the reset function is called before the start of the sim self.sumo_interface co
        # if reset is called at the end of an episode the self.sumo_interface connection mus
        if self.episode_counter >= 1:
            # this loop only enters after first episode is completed, and thus start_time is
            # var used before defined can thus be ignored
            #print('episode duration is', time.time() - self.start_time)
            self.sumo_interface.close()
        # else:
        #     """For more stochasticity in the simulation create traffic could be called eac
        #     self.create_traffic()

        if self.current_step == 0:
            self.start_time = time.time()

        #reset variables:
        self.total_arrived_vehicles = 0
        self.total_departed_vehicles = 0
        self.current_step = 0

        #self.start_time = time.time
        # Run the simulation with or without gui depending on the value of self.use_gui whic
        # sumo_binary = "sumo-gui" if self.use_gui else "sumo"
        # if sumo_binary == "sumo-gui" and self.use_libsumo == True:
        #     print(' no gui possible with libsumo')

        # Build the command to start SUMO
        # sumo_cmd = [sumo_binary, "-c", self.sumo_cfg, "--no-warnings", "true", "--log", "l


        print(' new routing used')
        self.times_reset_is_called +=1
        # print(' episode couner', self.episode_counter)
        # uncomment "-r", "sumo_rl/nets/PC-Alex/unique_routes.rou.xml", to use original rout

        # used for libumo but does not seem to be faster
        sumoBinary = sumolib.checkBinary('sumo-gui') if self.use_gui else  sumolib.checkBina
        sumo_cmd = [
            sumoBinary,
            "-n", "sumo_rl/nets/PC-Alex/City_network.net.xml",
            "-r", "sumo_rl/nets/PC-Alex/custom_route_NN.rou.xml",
            "--fcd-output", "fcd.xml",
            "--vehroute-output", "vehroutes.xml",
            "--default.speeddev", "0",
            "--no-step-log",
            "--load-state", "City_warm_up.xml"]

        # Set random seed if provided
        if seed is not None:
            sumo_cmd.extend(["--seed", str(seed)])  # Add seed option to the SUMO command

        # Start SUMO
        self.sumo_interface.start(sumo_cmd)

        if self.episode_counter == 0:
            self.traffic_lights = self.sumo_interface.trafficlight.getIDList()
            for tl_id in self.traffic_lights:
                on_line, line_counter = self.is_traffic_light_on_perimeter(tl_id)
                if on_line == True:
                    # Store the data of tl on perim in the dictionary with tl_id as the key
                    self.traffic_light_data[tl_id] = {
                        'line_counter': line_counter
                    }

        """"Warm up np longer required, it is now loaded in using sumo_cmd load"""

        # Initialize the current step and increase the episode counter
        self.current_step = 0
        self.episode_counter += 1

        # Get the state of the environment
        state = self.get_state()

        # Info is a mandatory return argument for PPO and STB3, an empty dict is sufficicent
        info = {}
        return state, info

    def create_traffic(self):
        """"
        Generates the traffic flows for the simulation.
        param: trip_count: how many trips are present during the simulation (how busy the ne
        param: fringe_factor: The ratio between traffic spawning at the outside of the simul

        """
```

```python

        # Now call the function as needed in the current script
        network_file = "sumo_rl/nets/PC-Alex/city_network.net.xml"
        output_file = "sumo_rl/nets/PC-Alex/custom_route_NN.rou.xml"
        fringe_factor = 1
        begin_time = 0
        end_time = 3600
        traffic_dist = 'exp(0.0044)'
        trip_count = 1000

        print(' -----------------------output file is', output_file)
        # Call the function from the other script
        generate_trips_NN.run_full_pipeline(network_file, output_file, fringe_factor, begin_

    def step(self, action):
        """
        Reset function that resets the SUMO environment
        ***Remove the additional way to get avg speed after error testing*****

        Parameters:
        - seed: optional specific seed for the simulation run
        - options:

        Returns:
        - state: state of the environment
        - info: mandatory param for RL algo, can be empty dict
        """

        # adjust the action such that the cycle time remains constant
        # Base phase durations (without actions applied)
        # Hard coded, should change if netedit durations are changed or extract from sumo

        if np.isnan(action).any():
            print(' nan in action space')
        phase_durations = np.array([28, 11, 28, 11])

        # # Compute the adjusted phase durations using the action values
        adjusted_durations = phase_durations * action

        # # Compute the sum of the adjusted phase durations
        sum_adjusted_durations = np.sum(adjusted_durations)

        # # total sum of phase durations (cycle time is sum_of_phase_durations + 4*3 (yellow
        sum_of_phase_durations = sum(phase_durations)

        # # Scale the actions so that the sum of adjusted durations equals 78 seconds
        scaled_actions = action * (sum_of_phase_durations / sum_adjusted_durations)

        # Apply action
        self.apply_action(scaled_actions)

        # Run simulation for 90 steps
        timestep = 90
        start_time_90_sec_loop = time.time()
        done = False
        for _ in range(timestep):
            self.sumo_interface.simulationStep()
            # stop the simulation if it is empty
            if self.sumo_interface.vehicle.getIDCount() == 0:
                done = True
                print('------steps required to stop is', self.current_step)
                break

            self.current_step += 1
            arrived_vehicles = self.sumo_interface.simulation.getArrivedIDList()
            self.total_arrived_vehicles += len(arrived_vehicles)
            departed_vehicles = self.sumo_interface.simulation.getDepartedIDList()
            self.total_departed_vehicles += len(departed_vehicles)

            # # Get newly departed and arrived vehicles
            # new_vehicles = self.sumo_interface.simulation.getDepartedIDList()
            # print(' new vehicles are', new_vehicles)
            # arrived_vehicles = self.sumo_interface.simulation.getArrivedIDList()
            # print('arrived vehicles are', arrived_vehicles)

            # # Subscribes speed and position for new vehicles
            # for vehicle_id in new_vehicles:
            #     print(' new vehicle id is', vehicle_id)
            #     print(' tc.var speed is', tc.VAR_SPEED)
            #     print('tc.VAR_POSITION' , tc.VAR_POSITION)
            #     self.sumo_interface.vehicle.subscribe(vehicle_id, [tc.VAR_SPEED, tc.VAR_PO
            #     print(' subscribe check', self.sumo_interface.vehicle.getSubscriptionResul

            # # Unsubscribe from vehicles that have arrived
            # for vehicle_id in arrived_vehicles:
            #     # Check if the vehicle has an active subscription before attempting to uns
            #     subscription_results = self.sumo_interface.vehicle.getSubscriptionResults(
            #     if subscription_results:
            #         print('if statement test')
            #         self.sumo_interface.vehicle.unsubscribe(vehicle_id)

            # # Retrieve and use the subscribed data for all active vSehicles
            # for vehicle_id in new_vehicles:
            #     vehicle_data = self.sumo_interface.vehicle.getSubscriptionResults(vehicle_
            #     if vehicle_data:
            #         speed = vehicle_data[tc.VAR_SPEED]
            #         # keep track of the speed of all cars
            #         avg_speed_array.append(speed)
            # # save the avg speed over all cars for each time step
            # self.avg_speed_90sec.append(np.mean(avg_speed_array))

            vehicle_ids = self.sumo_interface.vehicle.getIDList()
            speed_per_vehicle = []
            waiting_time_vehicle = []

            car_id_inside_perim = []
            car_id_outside_perim = []
            speed_inside_perim = []
            speed_outside_perim = []
            # Accumulate the total speed of all vehicles and count the vehicles
            for veh_id in vehicle_ids:
                # speed of all vehicles
                speed_per_vehicle.append(self.sumo_interface.vehicle.getSpeed(veh_id))
                waiting_time_vehicle.append(self.sumo_interface.vehicle.getWaitingTime(veh_i
                if self.is_vehicle_inside_perimeter(veh_id):
                    car_id_inside_perim.append(veh_id)
                    speed_inside_perim.append(self.sumo_interface.vehicle.getSpeed(veh_id))
                else:
                    car_id_outside_perim.append(veh_id)
                    speed_outside_perim.append(self.sumo_interface.vehicle.getSpeed(veh_id))

            # Save the avg waiting time per second
            self.waiting_time_per_second.append(np.mean(waiting_time_vehicle))

            # mean of all vehicles per sec
            self.avg_speed_per_sec.append(np.mean(speed_per_vehicle))

            self.mean_waiting_per_sec.append(np.mean(waiting_time_vehicle))

            if len(speed_inside_perim) >0:
                self.avg_speed_inside_perim_per_sec.append(np.mean(speed_inside_perim))
            else:
                self.avg_speed_inside_perim_per_sec.append(0)

            if len(speed_outside_perim) >0:
                self.avg_speed_outside_perim_per_sec.append(np.mean(speed_outside_perim))
            else:
                self.avg_speed_outside_perim_per_sec.append(0)


        # ***outside the 90 sec loop ****
        for tl_id, data in self.traffic_light_data.items():
                line_counter = data['line_counter']  # Extract the line_counter array
                controlled_lanes = self.sumo_interface.trafficlight.getControlledLanes(tl_id
                # for some reason SUMO takes duplicates, these are removed by conversion to
                unique_controlled_lanes = tuple(set(controlled_lanes))
                # the lanes should also be order in a logical order so that the correct lane
                sorted_lanes = self.sort_lanes_clockwise(unique_controlled_lanes)
                #west side tl
                if line_counter == 0:
```

```
                vehicles_on_entering_lane = []
                vehicles_on_sec_link = []

                # phases are as follows, starting at north and going clockwise:
                # north 1,2,3
                # east 3,4,5
                # south 6,7,8
                # west 9, 10, 11

                # controlled lanes are slightly different:
                # north 1,2
                # east 3,4
                # south 5,6
                # west 7,8

                # save the lanes which go into the perim in the tl data dict if it's not
                # does not change over the duration of the sim, so must only be filled o

                # fill up the dictionary 'self.traffic_light_data[]
                # dict will contain: {line_counter, lanes_into_perim, vehicles_on_enteri
                if 'lanes_into_perim' not in self.traffic_light_data[tl_id] or not self.
                    self.traffic_light_data[tl_id].setdefault('lanes_into_perim', [])
                lanes_into_perim = [sorted_lanes[4], sorted_lanes[5]]
                self.traffic_light_data[tl_id]['lanes_into_perim'].extend(lanes_into_per

                # Loop through each lane in lanes_into_perim and get the vehicles that e
                for lane in lanes_into_perim:
                    if len(self.sumo_interface.lane.getLastStepVehicleIDs(lane)) > 0:
                        vehicles_on_entering_lane.extend(self.sumo_interface.lane.getLas

                # save the vehicle ids of the vehicles on entering and sec lanes
                # key only needs to be created once
                if 'vehicles_on_entering_lane' not in self.traffic_light_data[tl_id] or
                    self.traffic_light_data[tl_id].setdefault('vehicles_on_entering_lane
                self.traffic_light_data[tl_id]['vehicles_on_entering_lane'].extend(vehic

                # loop over all veh on the entering lane
                inflow_enter = 0
                queue_enter = 0
                for vehid in vehicles_on_entering_lane:
                    # check which ones are not queueing
                    if self.sumo_interface.vehicle.getSpeed(vehid) > 0.1:
                        inflow_enter += 1
                    # remainder of cars are queuing
                    else:
                        queue_enter += 1
                # save the amount of vehicles that queue to enter the pc and on the seco
                # key only needs to be created once
                if 'queue_enter' not in self.traffic_light_data[tl_id] or not self.traff
                    self.traffic_light_data[tl_id].setdefault('queue_enter', [])
                    self.traffic_light_data[tl_id].setdefault('inflow_enter', [])
                self.traffic_light_data[tl_id]['queue_enter'] = queue_enter
                self.traffic_light_data[tl_id]['inflow_enter'] = inflow_enter

            #north side tl
            if line_counter == 1:
                vehicles_on_entering_lane = []
                vehicles_on_sec_link = []
                # fill up the dictionary 'self.traffic_light_data[]
                # dict will contain: {line_counter, lanes_into_perim, vehicles_on_enteri
                if 'lanes_into_perim' not in self.traffic_light_data[tl_id] or not self.
                    self.traffic_light_data[tl_id].setdefault('lanes_into_perim', [])
                lanes_into_perim = [sorted_lanes[6], sorted_lanes[7]]
                self.traffic_light_data[tl_id]['lanes_into_perim'].extend(lanes_into_per

                # Loop through each lane in lanes_into_perim and get the vehicles that e
                for lane in lanes_into_perim:
                    if len(self.sumo_interface.lane.getLastStepVehicleIDs(lane)) > 0:
                        vehicles_on_entering_lane.extend(self.sumo_interface.lane.getLas

                # save the vehicle ids of the vehicles on entering and sec lanes
                # key only needs to be created once
                if 'vehicles_on_entering_lane' not in self.traffic_light_data[tl_id] or
                    self.traffic_light_data[tl_id].setdefault('vehicles_on_entering_lane
                self.traffic_light_data[tl_id]['vehicles_on_entering_lane'].extend(vehic

                # loop over all veh on the entering lane
                inflow_enter = 0
                queue_enter = 0
                for vehid in vehicles_on_entering_lane:
                    # check which ones are not queueing
                    if self.sumo_interface.vehicle.getSpeed(vehid) > 0.1:
                        inflow_enter += 1
                    # remainder of cars are queuing
                    else:
                        queue_enter += 1
                # save the amount of vehicles that queue to enter the pc and on the seco
                # key only needs to be created once
                if 'queue_enter' not in self.traffic_light_data[tl_id] or not self.traff
                    self.traffic_light_data[tl_id].setdefault('queue_enter', [])
                    self.traffic_light_data[tl_id].setdefault('inflow_enter', [])
                self.traffic_light_data[tl_id]['queue_enter'] = queue_enter
                self.traffic_light_data[tl_id]['inflow_enter'] = inflow_enter

            # east side tl
            if line_counter == 2:
                vehicles_on_entering_lane = []
                vehicles_on_sec_link = []
                # fill up the dictionary 'self.traffic_light_data[]
                # dict will contain: {line_counter, lanes_into_perim, vehicles_on_enteri
                if 'lanes_into_perim' not in self.traffic_light_data[tl_id] or not self.
                    self.traffic_light_data[tl_id].setdefault('lanes_into_perim', [])
                lanes_into_perim = [sorted_lanes[0], sorted_lanes[1]]
                self.traffic_light_data[tl_id]['lanes_into_perim'].extend(lanes_into_per

                # Loop through each lane in lanes_into_perim and get the vehicles that e
                for lane in lanes_into_perim:
                    if len(self.sumo_interface.lane.getLastStepVehicleIDs(lane)) > 0:
                        vehicles_on_entering_lane.extend(self.sumo_interface.lane.getLas

                # save the vehicle ids of the vehicles on entering and sec lanes
                # key only needs to be created once
                if 'vehicles_on_entering_lane' not in self.traffic_light_data[tl_id] or
                    self.traffic_light_data[tl_id].setdefault('vehicles_on_entering_lane
                self.traffic_light_data[tl_id]['vehicles_on_entering_lane'].extend(vehic

                # loop over all veh on the entering lane
                inflow_enter = 0
                queue_enter = 0
                for vehid in vehicles_on_entering_lane:
                    # check which ones are not queueing
                    if self.sumo_interface.vehicle.getSpeed(vehid) > 0.1:
                        inflow_enter += 1
                    # remainder of cars are queuing
                    else:
                        queue_enter += 1
                # save the amount of vehicles that queue to enter the pc and on the seco
                # key only needs to be created once
                if 'queue_enter' not in self.traffic_light_data[tl_id] or not self.traff
                    self.traffic_light_data[tl_id].setdefault('queue_enter', [])
                    self.traffic_light_data[tl_id].setdefault('inflow_enter', [])
                self.traffic_light_data[tl_id]['queue_enter'] = queue_enter
                self.traffic_light_data[tl_id]['inflow_enter'] = inflow_enter

            #south side tl
            if line_counter == 3:
                vehicles_on_entering_lane = []
                vehicles_on_sec_link = []
                # fill up the dictionary 'self.traffic_light_data[]
                # dict will contain: {line_counter, lanes_into_perim, vehicles_on_enteri
                if 'lanes_into_perim' not in self.traffic_light_data[tl_id] or not self.
                    # print('self.traffic_light_data[tl_id]' , self.traffic_light_data[t
                    self.traffic_light_data[tl_id].setdefault('lanes_into_perim', [])
                lanes_into_perim = [sorted_lanes[2], sorted_lanes[3]]
                self.traffic_light_data[tl_id]['lanes_into_perim'].extend(lanes_into_per

                # Loop through each lane in lanes_into_perim and get the vehicles that e
                for lane in lanes_into_perim:
                    if len(self.sumo_interface.lane.getLastStepVehicleIDs(lane)) > 0:
                        vehicles_on_entering_lane.extend(self.sumo_interface.lane.getLas

                # save the vehicle ids of the vehicles on entering and sec lanes
                # key only needs to be created once
                if 'vehicles_on_entering_lane' not in self.traffic_light_data[tl_id] or
                    self.traffic_light_data[tl_id].setdefault('vehicles_on_entering_lane
                self.traffic_light_data[tl_id]['vehicles_on_entering_lane'].extend(vehic

                # loop over all veh on the entering lane
                inflow_enter = 0
                queue_enter = 0
                for vehid in vehicles_on_entering_lane:
                    # check which ones are not queueing
                    if self.sumo_interface.vehicle.getSpeed(vehid) > 0.1:
                        inflow_enter += 1
                    # remainder of cars are queuing
                    else:
                        queue_enter += 1

                # save the amount of vehicles that queue to enter the pc and on the seco
                # key only needs to be created once
                if 'queue_enter' not in self.traffic_light_data[tl_id] or not self.traff
                    self.traffic_light_data[tl_id].setdefault('queue_enter', [])
                    self.traffic_light_data[tl_id].setdefault('inflow_enter', [])
                self.traffic_light_data[tl_id]['queue_enter'] = queue_enter
                self.traffic_light_data[tl_id]['inflow_enter'] = inflow_enter

    # avg speed of all vehicles over 90 sec (one step)
    self.episode_90_sec_mean.append(np.mean(self.avg_speed_per_sec))

    print('******** 90 sec mean speed is *********', self.episode_90_sec_mean[-1])

    # prevent NaN is needed
    if len(self.mean_waiting_per_sec) == 0:
        self.mean_waiting_time_90_sec = [0]
    else:
        self.mean_waiting_time_90_sec.append(np.mean(self.mean_waiting_per_sec))

    # change in waiting time is mean waiting time over 90 sec minus waiting time on prev
    self.change_in_waiting_time.append(np.mean(self.mean_waiting_per_sec) - self.prev_wa
    self.prev_waiting_time = (np.mean(self.mean_waiting_per_sec))

    # store the avg speed inside and outside of the perim per step (90 sec)
    if len(self.avg_speed_inside_perim_per_sec)>0:
        self.avg_speed_inside_perim.append(np.mean(self.avg_speed_inside_perim_per_sec))
    if len(self.avg_speed_outside_perim_per_sec) >0:
        self.avg_speed_outside_perim.append(np.mean(self.avg_speed_outside_perim_per_sec

    # clear so that the list can be refilled each step
    self.avg_speed_per_sec.clear()
    self.mean_waiting_per_sec.clear()
```

```
        self.avg_speed_inside_perim_per_sec.clear()
        self.avg_speed_outside_perim_per_sec.clear()

        state = self.get_state()
        reward = self.compute_reward()

        # make sure not to overwrite the done given by the empty sim check
        if done == False:
            done = self.current_step >= self.max_steps
        # if the network is empty the simulation stops

        end_time_90_sec_loop = time.time()
        info = {}

        truncated = False
        if done:
            # save the avg speed per episode
            self.avg_speed = np.mean(self.episode_90_sec_mean)

            # Log avg speed to file
            with open(self.log_file_path, 'a') as f:
                f.write(f"{self.episode_counter},{np.mean(self.avg_speed)}\n")
            print(' avg speed episode is', self.avg_speed, flush=True)
            self.avg_speeds_per_step.clear()
            self.episode_90_sec_mean.clear()

            self.end_time = time.time()
            print('episode time is' , self.end_time - self.start_time)

        # Check for NaNs in the observation
        for i, value in enumerate(state):
            if np.isnan(value):
                print(f"NaN found in observation at index {i}: {value}")
        return state, reward, done, truncated, info

    def reorder_lane_pairs(self, lanes):
        """Reorder each pair of lanes such that lanes ending in '_0' come before '_1'."""
        reordered_lanes = []
        for i in range(0, len(lanes), 2):
            lane_0 = lanes[i]
            lane_1 = lanes[i+1]

            # Check the numeric suffixes and reorder
            if '_0' in lane_0:
                reordered_lanes.append(lane_0)
                reordered_lanes.append(lane_1)
            else:
                reordered_lanes.append(lane_1)
                reordered_lanes.append(lane_0)

```

```python
            return reordered_lanes

    # get controlled lanes takes the lanes in a random order which is not useful
    # the following couple functions use the name indexing of the lanes to order the control
    # they are ordered clockwise starting at east with 0 index followed by 1 index
    # 0 is the most right lane (from direction of traffci) and 1 is the lane to the left of
    def get_direction_suffix(self, lane_name):
        """Extract the directional suffix from a lane name (U, D, L, R)."""
        if 'U' in lane_name:
            return 'U'  # Upper part on the X-plane
        elif 'D' in lane_name:
            return 'D'  # Lower part on the X-plane
        elif 'L' in lane_name:
            return 'L'  # Left part on the Y-plane
        elif 'R' in lane_name:
            return 'R'  # Right part on the Y-plane
        return None


    def extract_coordinates(self, lane_name):
        """Extract the X and Y coordinates from the lane name."""
        # Split by 'Y' to separate the X and Y values.
        # Example: 'X1Y2.5' -> X=1, Y=2.5
        parts = lane_name.split('Y')
        x_coord = float(re.findall(r"[-+]?\d*\.\d+|\d+", parts[0][1:])[0])
  # Extract number from 'X' part
        y_part = re.findall(r"[-+]?\d*\.\d+|\d+", parts[1])[0]  # Extract number before dire
        y_coord = float(y_part)
        return x_coord, y_coord

    def clockwise_sort_key(self, lane_name):
        """Generate a sort key based on direction and coordinates."""
        # Clockwise order starting from Up ('U') -> Right ('R') -> Down ('D') -> Left ('L')
        direction_order = {'U': 0, 'R': 1, 'D': 2, 'L': 3}
        direction = self.get_direction_suffix(lane_name)

        # Sort by direction first, then by coordinates if the direction is the same
        x, y = self.extract_coordinates(lane_name)
        return (direction_order[direction], x, y)

    def sort_lanes_clockwise(self, lanes):
        """Sort lanes clockwise starting from east and reorder lane pairs."""
        # First sort the lanes clockwise
        sorted_lanes = sorted(lanes, key=self.clockwise_sort_key)

        # Then reorder the lanes to ensure '_0' comes before '_1'
        final_lanes = self.reorder_lane_pairs(sorted_lanes)

        return final_lanes



    def get_state(self):
        """
        get_state function that returns the current state of the simulation step
        State consists of:
        -avg speed of the complete network
        -agv speed outside of the perimeter
        -avg speed inside the perimeter
        -total waiting time outside
        -change in waiting time

        Returns:
        - state: state of the environment as a np.float32 array consisting of the above ment
        """

        # the previously mentioned values are not yet filled in the step function
        # when resetting this loop is executed once
        if self.current_step == 0:
            total_vehicles_inside = 0
            total_vehicles_outside = 0
            mean_waiting_time_network = 0

            car_id_inside_perim = []
            car_id_outside_perim = []
            speed_inside_perim = []
            speed_outside_perim = []
            vehicle_ids = self.sumo_interface.vehicle.getIDList()


            # Calculate the total speed and number of vehicles inside and outside perimeter
            for vehicle_id in vehicle_ids:
                mean_waiting_time_network += self.sumo_interface.vehicle.getWaitingTime(vehi
                if self.is_vehicle_inside_perimeter(vehicle_id):
                    car_id_inside_perim.append(vehicle_id)
                    speed_inside_perim.append(self.sumo_interface.vehicle.getSpeed(vehicle_i
                    total_vehicles_inside +=1
                else:
                    car_id_outside_perim.append(vehicle_id)
                    speed_outside_perim.append(self.sumo_interface.vehicle.getSpeed(vehicle_
                    total_vehicles_outside += 1

            # Calculate the average speed
            avg_speed_inside_perim = np.mean(speed_inside_perim) if total_vehicles_inside >
            avg_speed_outside_perim = np.mean(speed_outside_perim) if total_vehicles_outside
            avg_speed_network = (np.mean(speed_inside_perim)+np.mean(speed_outside_perim)/2)
            change_in_waiting_time = 0 #no change as there is no previous step
        else:
            # to get more accurate values, the calculations are done in the step function
            avg_speed_network = self.episode_90_sec_mean[-1]
            avg_speed_inside_perim = self.avg_speed_inside_perim[-1]
            avg_speed_outside_perim = self.avg_speed_outside_perim[-1]
            mean_waiting_time_network = self.mean_waiting_time_90_sec[-1]
            change_in_waiting_time = self.change_in_waiting_time[-1]

        state = np.array([avg_speed_network, avg_speed_outside_perim, avg_speed_inside_perim

        # Replace NaN values with 0, on rare ocasions one of these is empty due to a car bei
        # so simulation will go on but some division by 0 errors can occur
        state = np.nan_to_num(state, nan=0.0)

        return state

    def init_correct_phase_states(self, phases):
        """
        Modifies the states of all traffic light phases by changing all 'g' (green with conf
        We only want G (green without confliction)
        Parameters:
        - phases (list of self.sumo_interface.trafficlight.Phase): A list of traffic light p

        Returns:
        - list of self.sumo_interface.trafficlight.Phase: The modified list of traffic light
        """

        # all conflicted greens should be removed already in netedit
        corrected_phases = []
        for phase in phases:
            # Modify the phase state to change all 'g' to 'r'
            new_state = phase.state.replace('g', 'r')
            # Create a new phase with the modified state and original duration
            corrected_phase = self.sumo_interface.trafficlight.Phase(phase.duration, new_sta
            corrected_phases.append(corrected_phase)
        return corrected_phases

    def apply_action(self, action):

        """
        Modifies the action order to allign correctly for tl on the perimeter according to t
        Then forwards the action to self.set_red_green_phases to implement the new tl config

        First action value is always inflow/outflow direction of PN forward+right turn
        Second action value is always inflow/outflow direction of PN left turn
        Third action is perpendicular forward+right turn
        Fourth action is perpendicular left turn
        Since the action is given to all tl lights on the perim and the phases always start
        the order must be changed depending on the position of the tl on the perim

        :param action: action given by rl algo
        """
        # Change traffic light configurations based on action
        traffic_lights = self.sumo_interface.trafficlight.getIDList()

        for tl in traffic_lights:
            #checks if the tl is on perimeter, else no modification, SUMO preset phases are
            # line counter is used to determine which line passed the " True" value for the
            on_line, line_counter = self.is_traffic_light_on_perimeter(tl)
            # only modify tl which are on perim line
            if on_line is True:
                #west side tl
                if line_counter == 0:
                    # Create a copy of the action array for the modified action
                    modified_action = action.copy()
                    # Swap the first and third values
                    modified_action[0], modified_action[2] = action[2], action[0]
                    # Swap the second and fourth values
                    modified_action[1], modified_action[3] = action[3], action[1]
                    # print('modified action west side tl', modified_action)
                    self.set_red_green_phases(modified_action, tl)
                # north side tl
                if line_counter ==1:
                    # already correct order
                    self.set_red_green_phases(action, tl)
                # east side tl
                if line_counter ==2:
                    # Create a copy of the action array for the modified action
                    modified_action = action.copy()
                    # Swap the first and third values
                    modified_action[0], modified_action[2] = action[2], action[0]
                    # Swap the second and fourth values
                    modified_action[1], modified_action[3] = action[3], action[1]
                    self.set_red_green_phases(modified_action, tl)
                # south side tl
                if line_counter ==3:
                    # already correct
                    self.set_red_green_phases(action, tl)

    def set_red_green_phases(self, action, tl_id):
        """
        Applies the action given by the rl algo for all tl on the perimeter (this check is a

        First action value is always inflow/outflow direction of protected network (PN) forw
        Second action value is always inflow/outflow direction of PN left turn
        Third action is perpendicular forward+right turn
        Fourth action is perpendicular left turn
        Since the action is given to all tl lights on the perim and the phases always start
        the order must be changed depending on the position of the tl on the perim

        :param action: action given by rl algo
        """

        # keep track of the original program which will be modified so that we prevent modif
        if not hasattr(self, 'original_program'):
            # Get the current (original) program logic
            current_program_tuple = self.sumo_interface.trafficlight.getAllProgramLogics(tl_
            #print('Original TL logic:', current_program_tuple[0])

            current_program = current_program_tuple[0]

            # Correct the phase states
            corrected_phases = self.init_correct_phase_states(current_program.phases)

            # Create a new traffic light program with the corrected phases
            corrected_program = self.sumo_interface.trafficlight.Logic(
                current_program.programID,
                current_program.type,
                current_program.subParameter,
                corrected_phases
            )

            # Store the modified program
            # modified is removing the conflicting green phases, action is NOT yet applied
            self.original_program = corrected_program
        else:
            # Use the stored original program for modification
            current_program = self.original_program

        action_counter = 0
        modified_phases = []
        for phase in current_program.phases:
            # If the phase contains yellow lights ('y'), skip modifying it
            if 'y' in phase.state or 'Y' in phase.state:
                modified_phases.append(phase)  # Keep the yellow phase unchanged
                continue
            # Modify the duration of the non yellow phases
            if action_counter < len(action):
                new_duration = phase.duration * action[action_counter]
                # prevent durations being smaller than the min green time (should not be pos
                if new_duration <= self.min_green_time:
                    new_duration = self.min_green_time
                action_counter += 1
            else:
                new_duration = phase.duration
            # Create a new phase with the modified duration and the same state
            modified_phases.append(self.sumo_interface.trafficlight.Phase(new_duration, phas


        # Create a new traffic light program with the modified phases
        program = self.sumo_interface.trafficlight.Logic("0", 0, 0, modified_phases)

        # Set the new traffic light program
        self.sumo_interface.trafficlight.setProgramLogic(tl_id, program)

    def is_traffic_light_on_perimeter(self, tl_id):
        """
        Checks if a traffic light is on the perimeter
        :param tl_id: The ID of the traffic light.
        :return: True if the tl is on the perimeter, False otherwise.

        Note that this approach does not work for non-straight roads

        """
        lane_coordinates = []
        x_coordinates = []
        y_coordinates = []

        # Get a list of lanes controlled by the tl
        controlled_lanes = self.sumo_interface.trafficlight.getControlledLanes(tl_id)

        # Loop over all lanes and get their start and end coordinates. (all lanes are straig
        for lane in controlled_lanes:
            lane_coordinates.append(self.sumo_interface.lane.getShape(lane))

        # Get the x and y coordinates of the end points of the lane (where the tl is),
        for x in range(len(controlled_lanes)):
            x_coordinates.append(lane_coordinates[x][-1][0])
            y_coordinates.append(lane_coordinates[x][-1][1])

        x_coordinate_tl = np.mean(x_coordinates)
        y_coordinate_tl = np.mean(y_coordinates)


        xmin, ymin, xmax, ymax = self.perim_box
        coordinates_tl = [x_coordinate_tl, y_coordinate_tl]

        # Define the corners of the perimeter
        corners = [
            (xmin, ymin),  # Bottom-left corner
            (xmin, ymax),  # Top-left corner
            (xmax, ymax),  # Top-right corner
            (xmax, ymin)   # Bottom-right corner
        ]

        # Check if the TL is located at any of the corners, corners will not be considered a
        for corner in corners:
            if np.isclose(coordinates_tl[0], corner[0], atol=10) and np.isclose(coordinates_
                # print('tl id at corner is', tl_id)
                return False, None  # TL is at the corner, return False

        # Define the sides of the perimeter box as LineString objects
        lines = [
            LineString([(xmin, ymin), (xmin, ymax)]),  # Left side
            LineString([(xmin, ymax), (xmax, ymax)]),  # Top side
            LineString([(xmax, ymax), (xmax, ymin)]),  # Right side
            LineString([(xmax, ymin), (xmin, ymin)])   # Bottom side
        ]


        # Depending on which iteration of line the statement is true, the position of the tl
        # Each loop of the line in lines correspond to the next lines above so left, top, ri
        # Check if the point is near any of the lines
        line_counter = 0
        for line in lines:
            if self.point_near_line(coordinates_tl, line):
                on_line = True
                return on_line, line_counter
            else:
                on_line = False
                line_counter += 1
        # (works)
        return on_line, line_counter

    def is_vehicle_inside_perimeter(self, vehicle_id):
        """
        Check if a vehicle is inside the perimeter.

        :param vehicle_id: The ID of the vehicle.
        :return: True if the vehicle is inside the perimeter, False otherwise.
        """
        # x, y coordinate of vehicle
        x, y = self.sumo_interface.vehicle.getPosition(vehicle_id)
        # coordinates of the perimeter
        xmin, ymin, xmax, ymax = self.perim_box
        # check if a vehicle is within the perimeter
        return xmin <= x <= xmax and ymin <= y <= ymax

    def point_near_line(self, point, line, tolerance=20.0):
        """
        Checks if the point is within a certain tolerance of a line.

        :param point: A tuple (x, y) representing the point.
        :param line: A LineString object representing the line.
        :param tolerance: The tolerance within which the point should be from the line (in m
        :return: True if the point is within the tolerance of the line, False otherwise.
        """
        point_obj = Point(point)
        # determines the euclidian distance from the point to the line
        distance = point_obj.distance(line)
        # If the distance is more than the tolerance it will return False, else True
        return distance <= tolerance
    def get_inflow_and_queue_enter_and_vehSec_and_queueSec (self):
        """
        Retrieves the amount of vehicles that are driving on a lane that enters the PN
        return: inflow_enter, queue enter, total_vehicles_in_sec_link, queu_sec (int, int, i
        """
        inflow_enter = 0
        queue_enter = 0

        for tl_id, data in self.traffic_light_data.items():
            inflow_enter += data.get('inflow_enter')
            queue_enter += data.get('queue_enter')

        return inflow_enter, queue_enter

    def normalize(self, value, min_val, max_val):
        """
        Normalizes the value given a min and max.

        :param value: desired value to normalize.
        :param min_value: min value.
        :param max_val: maximum value.

        :return: Normalized value
        """
        return (value - min_val) / (max_val - min_val)

    def compute_reward(self):
        """
        Computes the reward for a point in time in the simulation. This reward is then norma

        :return: Normalized reward
        """

        # Calculate trip completion rate
        # if self.total_departed_vehicles == 0:
        #     trip_completion_rate = 0.0  # Avoid division by zero if no vehicles have start
        # else:
        #     # 4000 comes from generate_trips.py " trip_count"
        #     trip_completion_rate = (self.total_arrived_vehicles / 4000) * 100

        # # two methods for trip completion rate:
        # # total arrived/ total departed so far
        # # toal arrived/total trips over whole sim

        # inflow_enter, queue_enter = self.get_inflow_and_queue_enter_and_vehSec_and_queueSe
        # vehicles_enter = inflow_enter + queue_enter

        # x1 = 0.9
        # x2 = -0.1
        # x3 = -0.1

        # # get the most recent value for the avg speed over 90 sec to find the avg speed ov

        # change_in_waiting_time = self.change_in_waiting_time[-1]

        # print(' TCR', trip_completion_rate)
        # # use tanh transform to make the value between -1 and 1. Shift the range to be bet
        # normalized_trip_completion_rate = (np.tanh(trip_completion_rate)+1)/2
        # normalized_vehicle_enter = (np.tanh(vehicles_enter)+1)/2

        # calculate reward
        # reward = x1*normalized_trip_completion_rate + x2*normalized_vehicle_enter + x3

        avg_speed = self.episode_90_sec_mean[-1]
        reward = avg_speed
        return reward

        # #basic reward
        # reward = avg_speed
        # self.reward_list.append(reward)
        # #print(' reward list is', self.reward_list)

        # return reward

        # min_speed = 0
        # max_speed = 8

        # min_waiting_time = 0
        # max_waiting_time = 3000

        # normalized_avg_speed = self.normalize(avg_speed, min_speed, max_speed )
        # normalized_waiting_time =self.normalize(change_in_waiting_time, min_waiting_time,
        # use normalized avg speed and waiting time in future

        # uncommend for weighted normalized reward
        # Raw reward calculation

        # reward = avg_speed #- 0.1 * change_in_waiting_time
        # reward  = avg_speed #- 0.1*normalized_waiting_time
        # return reward
        # # Update running mean and variance for normalization
        # self.num_rewards += 1
        # old_mean = self.reward_mean
        # self.reward_mean += (reward - self.reward_mean) / self.num_rewards
        # self.reward_var += (reward - old_mean) * (reward - self.reward_mean)

        # reward_std = np.sqrt(self.reward_var / (self.num_rewards - 1)) if self.num_rewards

        # # Normalize the reward using running mean and standard deviation
        # normalized_reward = -((reward - self.reward_mean) / (reward_std + 1e-8))

        # return normalized_reward

    def close(self):
        """" closes the self.sumo_interface connection """
        # close sumo connection
        self.sumo_interface.close()

    def plot_avg_speed_per_episode(self):
        """Plot the average speed per episode."""
        plt.figure(figsize=(10, 6))
        plt.plot(self.avg_speed_per_episode, marker='o', linestyle='-', color='b')
        plt.title('Average Speed per Episode')
        plt.xlabel('Episode')
        plt.ylabel('Average Speed')
        plt.grid(True)
        plt.show()


class SumoGymEnv(gymnasium.Env):
    """ This class does the RL part """

    def __init__(self, sumo_cfg, max_steps, rank):
        """"initializes the class"""
        super(SumoGymEnv, self).__init__()
        #print(' action space has been modified for error testing')
        #lower_limit_action = np.array([1, 1, 1, 1])
        #upper_limit_action = np.array([1, 1, 1, 1])
        lower_limit_action = np.array([0.5, 0.75, 0.5, 0.75])
        upper_limit_action = np.array([1.5, 1.8, 1.5, 1.8])
        self.sumo_env = SumoEnvironment(sumo_cfg, max_steps, rank)
        self.action_space = gymnasium.spaces.Box(low=lower_limit_action, high=upper_limit_ac
        self.observation_space = gymnasium.spaces.Box(low=0, high=np.inf, shape=(5,), dtype=


    def reset(self, *, seed=None, options=None):
        """" Reset function, calls the sumoEnvironment reset function"""
        # Seed the environment's RNGs if applicable
        if seed is not None:
            super().reset(seed=seed)
            np.random.seed(seed)
        return self.sumo_env.reset()

    def step(self, action):
        """ Step function, calls the SumoEnvironment step function"""
        return self.sumo_env.step(action)

    def close(self):
        """ close function, calls the SumoEnvironment close function"""
        self.sumo_env.close()
```

```


# class CustomTensorboardCallback(BaseCallback):
#     """
#     Custom callback for logging additional metrics to TensorBoard, such as policy loss, va
#     """
#     def __init__(self, verbose=0):
#         super(CustomTensorboardCallback, self).__init__(verbose)

#     def _on_step(self) -> bool:
#         print(' logging test print')
#         print(' self.model.policy is', self.model.policy)
#         # Access the model's rollout buffer to get the advantages, rewards, and actions
#         rollout_buffer = self.model.rollout_buffer

#         # Mean advantage, policy loss, and value loss are calculated during training
#         mean_advantage = rollout_buffer.advantages.mean().item()
#         mean_action = rollout_buffer.actions.mean().item()
#         # Retrieve the policy and value losses from the PPO training process
#         # Accessing the PPO losses requires some manual computation during the optimizatio
#         # Get the distribution (actor network output)
#         #actions_dist = self.model.policy.dist
#         if hasattr(self.model, 'policy'):
#             policy_loss = self.model.policy.loss.mean().item() if hasattr(self.model.polic
#             value_loss = self.model.policy.value_loss.mean().item() if hasattr(self.model.

#         else:
#             policy_loss = None
#             value_loss = None
#             mu, sigma = None, None

#         # Log the statistics to TensorBoard
#         if policy_loss is not None and value_loss is not None:
#             self.logger.record("ppo/policy_loss", policy_loss)
#             self.logger.record("ppo/value_loss", value_loss)

#         self.logger.record("ppo/mean_advantage", mean_advantage)
#         self.logger.record("ppo/mean_action", mean_action)
#         print('policy loss sent to tensorboard is', policy_loss, flush=True)
#         print('value loss sent to tensorboard is', value_loss, flush=True)
#         print('mean advantage sent to tensorboard is', mean_advantage, flush=True)
#         print('mean action sent to tensorboard is', mean_action, flush=True)
#         print('final mu sent to tensorboard is', mu[-1], flush=True)
#         print('final sigma sent to tensorboard is', sigma[-1], flush=True)

#         return True



if __name__ == '__main__':

    # Initialize SUMO gym environment
    sumo_cfg = "sumo_rl/nets/PC-Alex/city_network_main.sumocfg"
    max_steps = 3000    #max steps, warm up time is already considered in reset, multiple of
    # cars are spawned until 3600, the remainder is to empty the network
    print(' max steps has been changed to use dit4 reward')
    # env = SumoGymEnv(sumo_cfg, max_steps)

    # Number of parallel environments
    num_envs = 1
    print(' number of envs changed back to 1 for error testing')

    # Create the vectorized environment
    env_fns = [SumoEnvironment.make_sumo_env(sumo_cfg, max_steps, rank=i) for i in range(num

    # creates vectorized env
    env = SubprocVecEnv(env_fns)

    #experiment settings
    max_iters = 50
    TIMESTEPS = max_steps

    # logging settings
    log_name = f"PPO_city_simple_iters_{max_iters}_timesteps_{TIMESTEPS}"

    models_dir = f"models/{log_name}/"
    logdir = "logs_ppo_city/ppo_city_simple" + time.strftime("%Y-%m-%d_%H-%M-%S")

    if not os.path.exists(models_dir):
        os.makedirs(models_dir)

    if not os.path.exists(logdir):
        os.makedirs(logdir)

    # modify the neural network
    net_arch = [5, 256, 256, 256, 256, 4]


    # Initialize the PPO model with your environment
    # model = PPO('MlpPolicy', env, batch_size=32, ent_coef=0.1, learning_rate=0.001, n_step
verbose=1, tensorboard_log='logs_ppo_city/ppo_city_simple'+ time.strftime("%Y-%m-%d_%H-%M-%S
    model = PPO('MlpPolicy', env, batch_size=64, ent_coef=0.2, learning_rate=0.01, n_steps=1
verbose=1, tensorboard_log=logdir)


    # Set the model in one of the environments for logging
    env_fns[0]().sumo_env.set_model(model)  # Set the model in the first environment for log

    # TensorBoard logging setup

    summary_writer = tf.summary.create_file_writer(logdir)
    # Define the callback
    # avg_speed_logging_callback = AvgSpeedLoggingCallback(summary_writer=summary_writer)

    iters =0

    # Training loop
    while iters <= max_iters:
        # Start timing the episode
        start_time = time.time()
        iters += 1
        # model.learn(total_timesteps=TIMESTEPS, reset_num_timesteps=False, log_interval=1,
        model.learn(total_timesteps=TIMESTEPS, reset_num_timesteps=False, log_interval=1)
        #model.learn(total_timesteps=TIMESTEPS, reset_num_timesteps=False, tb_log_name=log_n
        # End timing the episode
        end_time = time.time()

        # Calculate and print the duration of the episode
        episode_duration = end_time - start_time
        print(f"Time taken for episode {iters}: {episode_duration:.2f} seconds")

        model.save(f"{models_dir}/{log_name}")



# logging residual code
# # Access the PPO's `rollout_buffer` and get the advantage from the buffer
#               advantages = self.model.rollout_buffer.advantages
#               mean_advantage = advantages.mean().item()  # Log the mean of the advantages

#               # Get policy loss and value loss
#               policy_loss = self.model.policy.loss
#               value_loss = self.model.policy.value_loss

#               # Log the policy (actions chosen by the agent)
#               observations = self.locals['rollout_buffer'].observations
#               actions = self.model.predict(observations, deterministic=False)[0]
# Get actions from the policy
#               mean_action = actions.mean()  # Optionally, log the mean of the actions taken


#               policy = self.model.policy

#               # Get the distribution (actor network output)
#               actions_dist = policy.dist
#               mu = actions_dist.mean
#               sigma = actions_dist.stddev

#                # Log to TensorBoard
#               with self.summary_writer.as_default():
```

```
#                   tf.summary.scalar('PPO/policy_loss', policy_loss.item(), step=self.current
#                   tf.summary.scalar('PPO/value_loss', value_loss.item(), step=self.current_s
#                   tf.summary.scalar('PPO/mean_advantage', mean_advantage, step=self.current_
#                   tf.summary.scalar('PPO/mean_action', mean_action, step=self.current_step)
#                   # Log the mean and stddev per action value (action array consists of 4 act
#                   for i in range(mu.shape[0]):
#                       tf.summary.scalar(f'PPO/action_mu_{i}', mu[i], step=self.current_step)
#                       tf.summary.scalar(f'PPO/action_sigma_{i}', sigma[i], step=self.current

#               print('policy loss sent to tensorboard is', policy_loss.item(), flush=True)
#               print('value loss sent to tensorboard is', value_loss.item(), flush=True)
#               print('mean advantage sent to tensorboard is', mean_advantage, flush=True)
#               print('mean action sent to tensorboard is', mean_action, flush=True)
#               print('final mu sent to tensorboard is', mu[-1], flush=True)
#               print('final sigma sent to tensorboard is', sigma[-1], flush=True)
```

```
from joblib import load, dump
import numpy as np
import ast
import pandas as pd
import re
import matplotlib.pyplot as plt
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import RobustScaler
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
from sklearn.model_selection import cross_val_score, KFold
from sklearn.preprocessing import MinMaxScaler
from scipy.stats import gaussian_kde
from sklearn.metrics import mean_absolute_error

"This file is used for fine tuning the NN, used in the iterative supervised learning approac


def load_existing_model(model_filename, x_scaler_filename, y_scaler_filename):
    """Load an existing model and scalers from saved files."""
    model = load(model_filename)
    x_scaler = load(x_scaler_filename)
    y_scaler = load( y_scaler_filename)
    return model, x_scaler, y_scaler

def preprocess_new_data(df, x_scaler, y_scaler):
    """Preprocess and scale new data to match existing model."""
    # Apply the same preprocessing steps as used in the original training data
    def fix_format(data_string):
        data_string = data_string.strip()
        data_string = re.sub(r'\s+', ' ', data_string)
        fixed_string = re.sub(r'(\d)\s+(\d)', r'\1, \2', data_string)
        return fixed_string

    df["state"] = df["state"].apply(fix_format).apply(ast.literal_eval)
    df["policy"] = df["policy"].apply(fix_format).apply(ast.literal_eval)

    def extract_inner_value(value):
        return float(value[len("exp("):-1])
    df['traffic_dist'] = df['traffic_dist'].apply(extract_inner_value)

    states = df["state"].tolist()
    policies = df["policy"].tolist()
    traffic_dist = np.array(df["traffic_dist"].tolist()).reshape(-1, 1)
    X_train = np.hstack((states, policies, traffic_dist))

    # mutliple rewards were attempted, this is the best performing one
    reward_number = 2
    y_train = np.array([ast.literal_eval(reward)[reward_number] for reward in df["reward"].t

    # Scale the new data using the original scalers
    X_train_scaled = x_scaler.transform(X_train)
    y_train_scaled = y_scaler.transform(y_train.reshape(-1, 1)).flatten()

    return X_train_scaled, y_train_scaled

def update_model_with_new_data(model, X_train, y_train, epochs=5):
    """Update the model using new data."""
    for epoch in range(epochs):
        model.partial_fit(X_train, y_train)  # Use partial_fit to update model with new data
        train_loss = np.mean((model.predict(X_train) - y_train) ** 2)
        print(f"Epoch {epoch + 1}/{epochs} - New Data Training Loss: {train_loss:.4f}")
    return model


"""" MODIFY HERE"""
# Load the existing model and scalers
model, x_scaler, y_scaler = load_existing_model('iter4_nn_model_rew2.joblib', 'x_scaler_rew2
# model, x_scaler, y_scaler = load_existing_model('iter4_nn_model_state2.joblib', 'x_scaler_

# Load and preprocess new data
new_data_filename = "sumo_training_data_iter5.csv"
new_validation_filename = "sumo_validation_data_iter5.csv"

# new_data_filename = "sumo_training_data_state1_iter5.csv"
# new_validation_filename = "sumo_validation_data_state1_iter5.csv"


""""""""

new_data_df = pd.read_csv(new_data_filename, names=["state", "policy", "reward", "traffic_di
new_validation_data_df = pd.read_csv(new_validation_filename, names=["state", "policy", "rew

X_train, y_train = preprocess_new_data(new_data_df, x_scaler, y_scaler)
X_valid, y_valid = preprocess_new_data(new_validation_data_df, x_scaler, y_scaler)

# Update model with new data
updated_model = update_model_with_new_data(model, X_train, y_train)

# save the updated model
dump(updated_model, 'iter5_nn_model_rew2.joblib')
dump(x_scaler, 'x_scaler_rew2_iter5.joblib')
dump(y_scaler, 'y_scaler_rew2_iter5.joblib')

# dump(updated_model, 'iter5_nn_model_state2.joblib')
# dump(x_scaler, 'x_scaler_state2_iter5.joblib')
# dump(y_scaler, 'y_scaler_state2_iter5.joblib')

# Implement cross-validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)
cross_val_scores = cross_val_score(model, X_train, y_train, cv=kf, scoring='neg_mean_squared

print("Cross-Validation MSE for each fold:", -cross_val_scores)
print("Cross-Validation Mean MSE:", -np.mean(cross_val_scores))

# Evaluate the model
train_score = model.score(X_train, y_train)
valid_score = model.score(X_valid, y_valid)
print(f"Training score: {train_score}")
print(f"Test score: {valid_score}")

# Inverse transform predictions for interpretation
predictions = model.predict(X_valid)
original_scale_predictions = y_scaler.inverse_transform(predictions.reshape(-1, 1)).flatten(
original_scale_y_valid = y_scaler.inverse_transform(y_valid.reshape(-1, 1)).flatten()

# Print the first 5 original-scale true values and predicted values
print("First 5 true values (original scale):", original_scale_y_valid[:5])

print("First 5 predicted values (original scale):", original_scale_predictions[:5])

plt.figure(figsize=(8, 6))
plt.scatter(original_scale_y_valid, original_scale_predictions, alpha=0.6)
plt.plot([original_scale_y_valid.min(), original_scale_y_valid.max()],
        [original_scale_y_valid.min(), original_scale_y_valid.max()], 'r--', lw=2)
plt.xlabel('True Values')
plt.ylabel('Predicted Values')
plt.title('True vs. Predicted Values')
plt.show()

mae = mean_absolute_error(original_scale_y_valid, original_scale_predictions)
print("Mean Absolute Error (MAE):", mae)
```

```

rmse = np.sqrt(np.mean((original_scale_y_valid - original_scale_predictions) ** 2))
print("Root Mean Squared Error (RMSE):", rmse)

# Calculate percentage errors
percentage_errors = np.abs((original_scale_y_valid - original_scale_predictions) / original_

# Perform Kernel Density Estimation to get a smooth curve
kde = gaussian_kde(percentage_errors, bw_method=0.3)  # Bandwidth can be adjusted for smooth
x_range = np.linspace(0, percentage_errors.max(), 100)

# Plot smooth density line of the percentage errors
plt.figure(figsize=(10, 6))
plt.plot(x_range, kde(x_range), label='Density of Percentage Error')
plt.xlabel("Percentage Error (%)")
plt.ylabel("Density")
plt.title("Density Plot of Percentage Errors in Validation Predictions")
plt.legend()
plt.grid(True)
plt.show()

# Calculate absolute errors for each data point
errors = np.abs(original_scale_y_valid - original_scale_predictions)

# Sort errors in descending order
sorted_errors = np.sort(errors)[::-1]

# Calculate cumulative contribution of errors as percentage
cumulative_error = np.cumsum(sorted_errors) / np.sum(sorted_errors) * 100

# Calculate percentage of data points
data_points_percentage = np.arange(1, len(errors) + 1) / len(errors) * 100

# Calculate absolute percentage errors
percentage_errors = np.abs((original_scale_y_valid - original_scale_predictions) / original_

# Define error thresholds
thresholds = np.arange(0, 50, 5)  # Adjust range and step as needed (e.g., up to 50% with 5%
percentages_below_threshold = [(percentage_errors <= t).mean() * 100 for t in thresholds]

# Plot the cumulative distribution of errors
plt.figure(figsize=(10, 6))
plt.plot(thresholds, percentages_below_threshold, marker='o', linestyle='-', color='b')
plt.xlabel('Error Threshold (%)')
plt.ylabel('Percentage of Predictions Below Threshold (%)')
plt.title('Cumulative Distribution of Prediction Errors')
plt.grid(True)
plt.ylim(0, 100)
plt.show()

```

```
# Calculate absolute percentage errors
absolute_percentage_errors = np.abs((original_scale_y_valid - original_scale_predictions) /

# Calculate the mean absolute percentage error
mape = np.mean(absolute_percentage_errors)
print("Mean Absolute Percentage Error (MAPE):", mape)
```