Realtime Vehicle Route Optimisation via DQN for Sustainable and Resilient Urban Transportation Network

Song Sang Koh

A thesis submitted in partial fulfilment of the requirements of
Liverpool John Moores University
for the degree of Doctor of Philosophy

January, 2020

Declaration of Authorship

I, Song Sang Koh, declare that this thesis titled, "Realtime Vehicle Route Optimisation via DQN for Sustainable and Resilient Urban Transportation Network" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: 

Date: 30th September 2019

LIVERPOOL JOHN MOORES UNIVERSITY

Abstract

Department of Computer Science

Doctor of Philosophy

Realtime Vehicle Route Optimisation via DQN for Sustainable and Resilient Urban Transportation Network

by Song Sang Koh

Traffic congestion has become one of the most serious contemporary city issues for urban transportation network as it leads to unnecessary high energy consumption, air pollution and extra travelling time. During this decade, many optimization algorithms have been designed to achieve the optimal usage of existing roadway capacity in cities to leverage the problem. However, it is still a challenging task for the vehicles to interact with the complex city environment in a real time manner. In this thesis, we propose a deep reinforcement learning (DRL) method to build a real-time intelligent vehicle navigation system for sustainable and resilient urban transportation network. We designed two rewards methods travel time based and vehicle emissions impact (VEI) based which aim to reduce the travel time for emergency vehicle (resilience), and reduce vehicle emissions for general vehicle (sustainability). In the experiment, several realistic traffic scenarios are simulated by SUMO to test the proposed navigation method. The experimental results have demonstrated the efficient convergence of the vehicle navigation agents and their effectiveness to make optimal decisions under the volatile traffic conditions. Travel time based reward schema perform better in reducing travel time however VEI based show better result in reducing vehicle emissions. Furthermore, the results also show that the proposed method has huge potential to provide a better navigation solution comparing with the benchmark routing optimisation algorithms.

Acknowledgements

It is an amazing journey for me to be able to complete my PhD program in Liverpool. My greatest thank to my dearest family, my mother and my sister who fully understand and always support me to do what I am interested in. I also want to particularly express my appreciation to my uncle in law Mr. Chow and my aunt Dr. Ng who had been guiding me to be a better person internally and externally.

During the last four years, I feel grateful to the people I have been working with in my PhD program. As my first supervisor, Dr. Bo Zhou is extremely helpful not only for my PhD program, and also my well-being in Liverpool. In fact, Dr. Bo Zhou was also the supervisor for my master degree, he is the person who brought me into research life and gave me the confidence to start my PhD. His kindness and patient helped me getting through so many challenges in the last four years. He also helped me to broaden my research vision. I could not express more grateful to have him as my supervisor. My sincere gratitude also goes to Dr. Fang Hui, for his valuable and elaborate comments on each of my significant experiment and submission. Without his suggestion and guideline, I would never be able to complete my PhD in this four years. His suggestion and opinion provided huge contribution for my research. In fact, he pushed me to become a better researcher, helped me to improve my technical and management skills and I will always appreciate it. My thank also goes to my co-supervisor Dr. Po Yang and Dr. Zaili Yang, for their kindness and help during my PhD program. They are always be there when I need them.

I highly appreciate the time in Liverpool John Moores University (LJMU) for my master and PhD program. I would like to thank all the staffs in LJMU, especially Tricia Waterson who had given me so many supports.

List of Publications

[**Journal**]

- **Song Sang Koh**, Bo Zhou, Hui Fang, Po Yang, Zaili Yang, Qiang Yang, Lin Guan, Real-time Deep Reinforcement Learning based Vehicle Navigation, Elsevier Soft Computing. [Submitted]

[**Conference**]

- **Song Sang Koh**, Bo Zhou, Po Yang, Zaili Yang, Hui Fang, Jianxin Feng, Reinforcement Learning for Vehicle Route Optimization in SUMO. HPCC/SmartCity/DSS 2018: 1468-1473, 2018
- **Song Sang Koh**, Bo Zhou, Po Yang, Zaili Yang, Study of Group Route Optimization for IoT Enabled Urban Transportation Network. iThings/-GreenCom/CPSCom/SmartData 2017: 888-893, 2017
- **Song Sang Koh**, Bo Zhou, Po Yang, Zaili Yang. A Survey on Urban Traffic Optimisation for Sustainable and Resilient Transportation Network. 2016 9th International Conference on Developments in eSystems Engineering (DeSE). IEEE, 2016

Contents

Declaration of Authorship i

Abstract ii

Acknowledgements iii

1 Introduction 1
- 1.1 Overview . . . 1
- 1.2 Research Motivation . . . 3
- 1.3 Research Aims and Objectives . . . 9
- 1.4 Research Novelty . . . 10
- 1.5 Thesis Structure . . . 12

2 Literature Review 14
- 2.1 Overview . . . 14
- 2.2 Shortest Path Algorithm . . . 14
  - 2.2.1 Dijkstra's Algorithm . . . 15
  - 2.2.2 Bellman-Ford's Algorithm . . . 18
  - 2.2.3 A-Star Algorithm . . . 20
  - 2.2.4 Heuristic Shortest Path Finding and Re-planning . . . 22
  - 2.2.5 Bidirectional Search . . . 23
  - 2.2.6 Ant-Colony Algorithm . . . 24
- 2.3 Traffic Management Systems . . . 26
  - 2.3.1 Traffic Signal Control System . . . 26
  - 2.3.2 Vehicle Routing Optimisation . . . 28
- 2.4 Summary of Limitations . . . 30

3 Deep Reinforcement Learning 32
- 3.1 Overview . . . 32
- 3.2 Reinforcement Learning . . . 32
  - 3.2.1 Markov Decision Process (MDP) . . . 32
  - 3.2.2 Q-Learning . . . 35
- 3.3 Deep Reinforcement Learning . . . 37
  - 3.3.1 Artificial Neural Networks . . . 37
  - 3.3.2 Action Selection Policy . . . 39
  - 3.3.3 Optimisation Algorithm . . . 40
  - 3.3.4 Deep Q-Network . . . 42
- 3.4 Deep Reinforcement Learning for Urban Traffic Optimisation . . . 44
  - 3.4.1 Intersection Traffic Control . . . 45
  - 3.4.2 Urban Traffic Prediction . . . 46

3.4.3 Motivation of using DQN in Vehicle Navigation 46
3.5 Limitation 47

4 Urban Traffic Simulation 48
4.1 Overview 48
4.2 Background 48
4.2.1 Traffic Simulation 48
4.2.2 Simulation Models and Approaches 49
4.2.3 Simulators Overview 52
4.3 Overview of SUMO 53
4.4 Additional Features 54
4.4.1 TraCI 54
4.4.2 Emissions 55

5 Preliminary Design and Experiment for Vehicle Route Optimisation 57
5.1 Overview 57
5.2 Markov Decision Process for Vehicle Route Optimisation Problem 57
5.2.1 Overview of Markov Chain 57
5.2.2 Apply Markov Chain Modelling in Urban Road Traffic Network 58
5.3 Reinforcement Learning for Vehicle Route Optimisation 60
5.3.1 Problem Statement 61
5.3.2 Key term definition of RL for Vehicle Route Optimisation 61
5.4 Experiment Evaluation 63
5.4.1 Experiment Setup 63
5.4.2 Experiment Implementation 64
5.4.3 Simulation Result 66
5.4.4 Discussion 68
5.5 Summary 68

6 The Proposed Framework and Structure Design 70
6.1 Overview 70
6.2 Proposed Framework for Vehicle Route Optimisation 70
6.2.1 Overview of Proposed Framework 71
6.2.2 Training Framework Structure 71
6.2.3 Training Framework Process Flow 73
6.3 The Design of DRL for Real-time Vehicle Route Optimisation . 74
6.3.1 Problem Statement 74
6.3.2 Vehicle Agent 75
6.3.3 State Space 76
6.3.4 Action Space 77
6.3.5 Reward Function 78
6.4 DRL method for Real-time Vehicle Route Optimisation 82
6.5 Deep Neural Network Architecture for Real-time Vehicle Route Optimisation 86
6.6 Summary 87

7 Experiment Implementation and Evaluation **89**
7.1 Overview . . . 89
7.2 Experiment Implementation . . . 89
7.2.1 Training Simulation Overview . . . 89
7.2.2 Scenario class definition . . . 91
7.2.3 Building a Simulation with SUMO . . . 91
7.2.4 Environment Class Definition . . . 94
7.2.5 Data Extraction and Pre-processing . . . 95
7.2.6 Benchmark Methods . . . 98
7.2.7 DRL Agent Class Definition . . . 98
7.2.8 DRL Agent Architecture . . . 99
7.2.9 Action Selection Policy . . . 100
7.3 Experimental Evaluation . . . 104
7.3.1 Toy Data . . . 104
7.3.2 Realistic scenario analysis . . . 106

8 Conclusion and Future Work **118**
8.1 Overview . . . 118
8.2 Problem Overview . . . 118
8.3 Contributions and Achievements . . . 119
8.4 Future work . . . 120
8.5 Summary . . . 121

Bibliography **123**

A Key Code Snippets for Vehicle Route Optimisation **132**
A.1 Scenario Class . . . 132
A.2 Environment Class . . . 135
A.3 DRL Agent Class . . . 144

List of Figures

1.1 Urban and rural populations of the world, 1950-2050 [119] . . . 4
1.2 Population and number of urban agglomerations of the world by size class of urban settlement, 1990, 2018 and 2030 [119] 4
1.3 Congestion Growth Trend – Hours of Delay per Auto Commuter [100] 5
1.4 Global transport CO2 emissions [23] 6
1.5 Souces of Congestion[109] 7
1.6 Examples of car navigation system 9

2.1 A (u_0, v_0) path of minimum weight [10]) 15
2.2 Illustration of Dijkstra's algorithm [85]) 17
2.3 Tracking nodes in A-Star algorithm and in Dijkstra's algorithm [85]) 21
2.4 An illustration of the bidirectional version of Dijkstra's algorithm [85] 24
2.5 Example of how the effect of laying/sensing pheromone during the forth and back journeys from the nest to food sources to determine shortest path between two nodes [28]) 25
2.6 3-tier system architecture of SCATS [103]) 27
2.7 Layout architecture for efficient dynamic traffic control system [103]) 28
2.8 Roadmap and its corresponding road network graph [21]) 29

3.1 Reinforcement Learning Design Flow 33
3.2 A Markov decision Process 34
3.3 The interaction between environment and agent in Q-Learning . 36
3.4 Single neuron output based on weight, input, bias and non-linear activation function 38
3.5 Function curves of sigmoid, Tanh and ReLU 38
3.6 Visualization of the structure of neural network with multiple hidden layers 39
3.7 DQN Learning Diagram 43
3.8 The traffic light control model in deep learning [21]) 45

4.1 Overview of traffic simulation models [14] 50
4.2 Space-discrete vs Space-continuous simulation 51
4.3 The relationship of traffic simulation models [15] 51

5.1 Sumo network 59
5.2 The edge network associated to the Sumo map shown in Figure 5.1 59

5.3 Edges in Sumo network that connected nodes shown in Figure 5.2 60
5.4 Dual Graph with rewards 62
5.5 SUMO urban road traffic network 64
5.6 Dual graph network of Figure 5.5 64
5.7 Cumulative rewards per training episode 67
5.8 Travel time per training episode 68

6.1 The proposed framework structure 72
6.2 The Framework consists of SUMO simulator, Middleware and RL Agent for the vehicle navigation task 73
6.3 Use case of vehicle route optimisation 75
6.4 Problem statement of the RL based multi-agents navigation . . 76
6.5 Problem statement of the RL based multi-agents navigation . . 78
6.6 State Matrix for network Figure 6.5 79
6.7 Virtualisation of reward calculation in urban network 80
6.8 The convergence graph for 2 proposed reward schemas in travel time and VEI 82
6.9 A popular single stream Q-network (top) and the dueling Q-network (bottom). The dueling network has two streams to separately estimate (scalar) state-value and the advantages for each action; the green output module combine both state-value and the advantages and output the Q-values. [114] 84
6.10 Comparison of different DQN methods 85
6.11 Euclidean distances based action selection policy 86
6.12 The structure of neural network in this experiment 87

7.1 Training Simulation Mechanism Flowchart 90
7.2 SUMO traffic simulation process diagram 92
7.3 (a) openstreetmap (b) sumo map 93
7.4 (a) NetEdit (b) sumo map 93
7.5 Three cases showing the limitations of the travel time calculation using SUMO 97
7.6 Overview of DQN architecture 100
7.7 Eval network architecture 101
7.8 Target Network architecture 102
7.9 Train and loss architecture 103
7.10 Simple map structure and mean step in 100 episodes 105
7.11 The simulated illustration of conventional Dijkstra/A* method and proposed method 106
7.12 Real world city map that are captured for SUMO simulation in this thesis 107
7.13 Convergence of the Avg. travel time in City Map 1 109
7.14 Convergence of the Avg. VEI in City Map 1 110
7.15 Convergence of the Avg. travel time in City Map 2 111
7.16 Convergence of the Avg. VEI in City Map 2 112
7.17 Convergence of the Avg. travel time in City Map 3 113
7.18 Convergence of the Avg. VEI in City Map 3 114

List of Tables

4.1 CPU and Memory performance in different traffic simulators [59] 52
4.2 Pollutants covered by models 56

5.1 Vehicle agent hyper parameters for intelligent navigation 66
5.2 The connected roads table 67

6.1 Vehicle agent hyper parameters for reward schemas comparison 82
6.2 Vehicle agent hyper parameters for intelligent navigation 86

7.1 Definition of vehicle type 93
7.2 The comparison of expected travel time calculation from SUMO and proposed approach 98
7.3 Maps information 107
7.4 The objective performance comparisons under various traffic conditions 116

List of Abbreviations

RL	**R**einforcement **L**earning
DRL	**D**eep **R**einforcement **L**earning
VEI	**V**ehicle **E**missions **I**mpact
RC	**R**ecurring **C**ongestion
NRC	**N**on-**R**ecurring **C**ongestion
SPP	**S**hortest **P**ath **P**roblem
DA	**D**ijkstra's **A**lgorithm
ARA	**A**nytime **R**eparing **A***
ADA	**A**nytime **D**ynamic **A***
SCATS	**S**ydney **C**oordiated **A**daptive **T**raffic **S**ystem
SCOOT	**S**plit **C**ycle **O**ffset **O**ptimisation **T**echnique
IP	**I**nternet **P**rotocol
VANETs	**V**ehicular **A**d-hoc **NET**works
ITLCS	**I**ntelligent **T**raffic **L**ight **C**ontrol **S**ystem
EDTCS	**E**fficient **D**ynamic **T**raffic **C**ontrol **S**ystem
TCU	**T**raffic **C**ontrol **U**nit
TMU	**T**raffic **M**onitor **U**nit
TSU	**R**oad **S**ide **U**nit
VSN	**V**ehicular **S**ensor **N**etwork
AP	**A**ccess **P**oint
VADD	**V**ehicle **A**ssisted **D**ata **D**elivery
TBD	**T**rajectory **B**ased **D**ata
MDP	**M**arkov **D**ecision **P**rocess
VI	**V**alue **I**teration
ANN	**A**rtificial **N**eural **N**etwork
CNN	**C**onvolutional **N**eural **N**etwork
RNN	**R**ecurrent **N**eural **N**etwork
MLP	**M**ulti-**L**ayer **P**erceptron
ReLU	**Re**ctifier **L**inear **U**nit
MSE	**M**ean **S**quare **E**rror
DQN	**D**eep **Q**-**N**etwork
TMC	**T**raffic **M**essage **C**hannel
TRSS	**T**ransportation **R**egulation **S**upport **S**ystem
APTS	**A**dvanced **P**ublic **T**ransportation **S**ystem
TraCI	**Tra**ffic **C**ontrol **I**nterface
ERD	**E**xploration **R**ate **D**ecay
ICE	**I**nternal **C**ombustion **E**ngine
GUI	**G**raphical **U**ser **I**nterface
GDUE	**G**awron's **D**ynamic **U**ser **E**quilibrium
DTA	**D**ynamic **T**raffic **A**ssignment

Chapter 1

Introduction

1.1 Overview

Over the last decade, sustainability and resilience have become a major consideration that cannot be neglected in urban development, where most major cities in the world are faced with challenges in increasing economic and environmental pressures associated with urbanisation. Although many definitions abound, the most often used definition of sustainability is that proposed by the Brundtland Commission [11], which defined sustainability as "the ability to produce and/ or maintain a desired set of conditions or things for some time into the future, not necessarily forever". Generally, sustainable development aims at creating and maintaining our options for prosperous social and economic development in future [33]. It emphasizes an optimal balance between social needs, economy and environment. Therefore, environmental, social, and economic concerns have to be integrated throughout the decision making process [30]. Resilience, has become a common term in risk analysis or risk management in highly complex and adaptive systems [76]. However, it has been defined in different ways. Essentially, resilience provides the capacity to absorb the shock and the availability of maintaining the function where unexpected events happen [41]. It also refers to the inherent ability and adaptive responses of systems that enable them to avoid potential losses [42], in order to protect and enhance people's live, secure development gains, foster an environment for investment, and drive positive change.

The urban transportation network is among the most complex and critical system of modern cities. It is essentially playing an important role in the day-to-day running of cities, directly influencing people's daily life and activities in all functions of society. It is linked to all aspects of urban life: leisure, education and business. Ensuring a comprehensive, accessible and integrated transportation network is essential to sustain social and economic development. The central role of transport networks in urban life means that any reduction in performance may compromise the city's operations across a number of sectors, causing large and costly disruptions. In order to make sure urban transportation will meet the basic needs of society and individuals without seriously damaging the environment and decreasing the safety level. Developing a sustainable and resilient urban transportation network has become a major challenge nowadays.

Sustainability and resilience are the key factors in urban transportation networks in order to make sure they will meet the basic needs of society and individuals without seriously damaging the environment and decreasing the safety level on the roadway. There are a variety of unexpected events that could put urban transportation at risk. For instance, some natural disasters including flooding, earthquake or heatwaves could totally paralyse the urban transportation network. Moreover, traffic congestion is also a major issue in urban planning, especially when the number of vehicles on the road keeps growing and the urban infrastructure is not upgraded at a comparable pace to accommodate the growing number of vehicles. Besides, although many different methods about route optimisation have been widely studied, most of them lack the capability of self-evolution based on the rapid nature of the transportation network and lack of consideration from the urban perspective in terms of sustainability and resilience. Therefore, to develop a sustainable and resilient urban transportation network has become a key topic in urban development nowadays.

For a sustainable and resilient transportation network, although there is no universally accepted definition, with the combination of sustainability and resilience concepts, it generally describes an intelligent transport network which is able to use the urban transportation network efficiently and meet the basic needs of individuals as well as the urban environment, and able to minimize the emissions and waste, increase road safety and limit fuel consumption [108] [46] [70]. Sustainability and resilience should be the main consideration of all strategies for modern urban transportation networks in the future.

Sustainable transportation systems offer greater flexibility and coordination, allowing users to be distributed across a diverse portfolio of transport options and to transfer easily from one mode to another when required. Better integrated systems provide a smoother, more efficient and user-friendly day-to-day service which is also better able to cope with the stresses associated with peak demand and strains of infrequent shocks or unforeseen events [83]. A resilient transportation system is essential to avoid such events. Recent evidence suggest that the frequency, extent and severity of extreme weather events is increasing around the world exposing transport infrastructure to more severe stresses and sudden (shock) events. Anticipating and preparing for the impacts of these stresses and shocks on the transportation systems is key to achieving and sustaining resilient urban mobility [54].

However, to make a sustainable and resilient urban transportation network is not easy. One of the major challenges is to resolve the traffic congestion. A better utilization of vehicles and a cost effective vehicle routing solution would more directly achieve sustainable transportation network schemes [69]. However, despite many problems, variations have been investigated and studied through many different approaches, due to the complexity of urban transportation, there is still a huge gap to be filled for the traffic congestion problem. Although travel time represents the major element that affects urban transportation performance, there are limited research studies on resolving the vehicle routing problem related to travel time dependency [31]. Therefore, route

optimisation is proposed recently as an approach to resolving the traffic congestion problem.

Traditionally, most of the past routing algorithms for traffic congestion problem are based on static approach and without considering the unexpected events in transportation network. They are lack of ability to deal with rapidly changes circumstance especially with the complex nature of the urban transportation. However, with the rapid development and recent success of machine learning technologies lately, a self-evolute route optimisation approach which is able to deal with real-time unexpected event for sustainable and resilient urban transportation network become possible. Hence, this chapter will introduce the research within this thesis, along with the research motivation, research aims, and objectives, research novelty, and lastly the overall structure of this thesis will be introduced.

1.2 Research Motivation

Globally, more people live in urban areas than in rural areas. According to World urbanization prospects in 2018 [119] as shown in Figure 1.1, a total of 55 percent of the world's population now live in urban areas in 2018, and by 2050, 66 percent of the world's population are projected to be urban. Furthermore, [119] as shown in Figure 1.2, in 1990 there were 10 cities with more than 10 million inhabitants, hosting 153 million people, which represents less than 7 per cent of the global urban population. In 2018, the number of megacities has tripled to 33. And by 2030, the number of megacities is expected to grow to 43. Urbanization has many positive impacts on human society. It creates an increasing number of better opportunities for jobs, education, and healthcare that more and more people are moving from the countryside to pursue. Urbanization makes the global distribution of population more concentrated in areas where fewer natural disasters occur, more food can be produced and more infrastructure can be built. Additionally, urbanization facilitates the requirement of modern industrialized society so that individuals cooperate with more people from diverse backgrounds. Consequently, the past 60 years of global urbanization have resulted in enormous economic growth, and concentration of population in densely populated cities.

However, with the growing population, urbanization leads to a series of unprecedented challenges including urban rural inequality and environmental damage. Cities around the world struggle to transform their infrastructure and make the changes in order to meet the daily basic needs economically and environmentally. It has caused severe damage to the environment such as increased consumption of natural resources [80], excess of air pollution, noise and dust [110], and raising hazardous waste on the urban population [81]. By achieving sustainability and resilience, urban development will move towards further economic and social progress, and at the same time strengthening environmental protection.

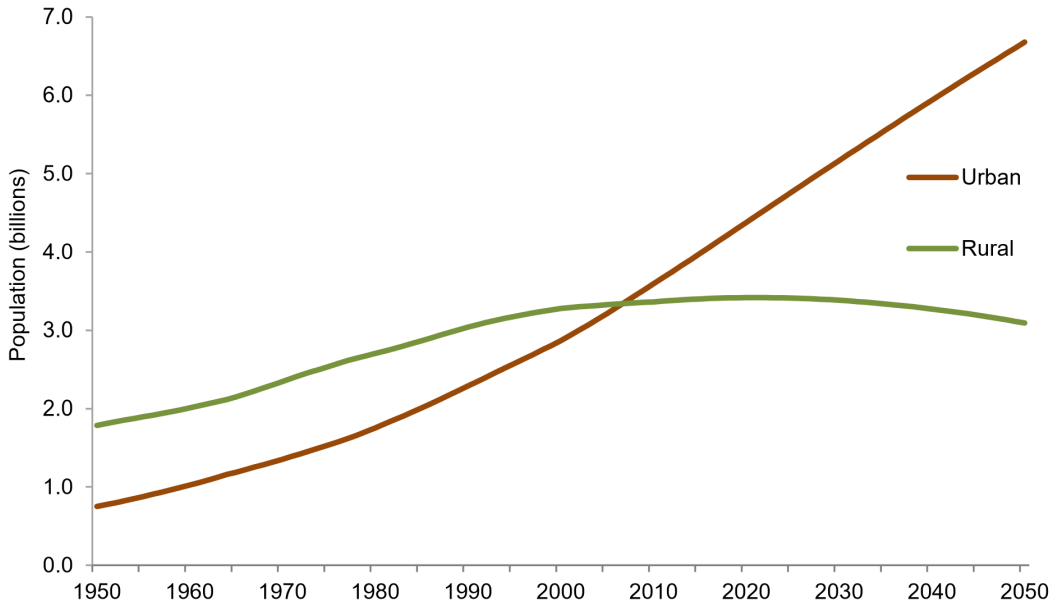


FIGURE 1.1: Urban and rural populations of the world, 1950-2050 [119]

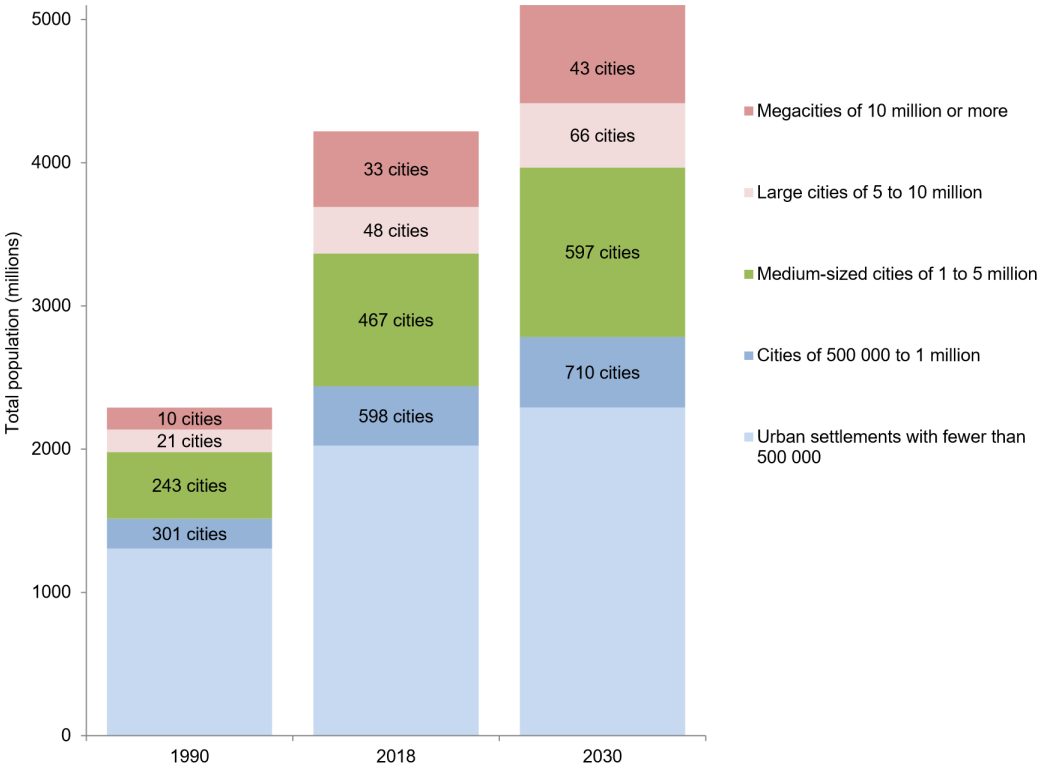


FIGURE 1.2: Population and number of urban agglomerations of the world by size class of urban settlement, 1990, 2018 and 2030 [119]

When considering the future development of the urban areas, especially in

the cities with high population, you find challenges dealing with sustainability and resilience. A sustainable and resilient transportation network entails an integrated system with social, environment and economic considerations. Therefore the range of issues are addressed widely to cover different categories. As shown in Figure 1, T.Litman and D.Burwell [70] listed a wide range of sustainability issues in social, environment and economic categories. However, they also acknowledged that although each issue fits into a specific category, practically they are potentially overlapped and dependent.

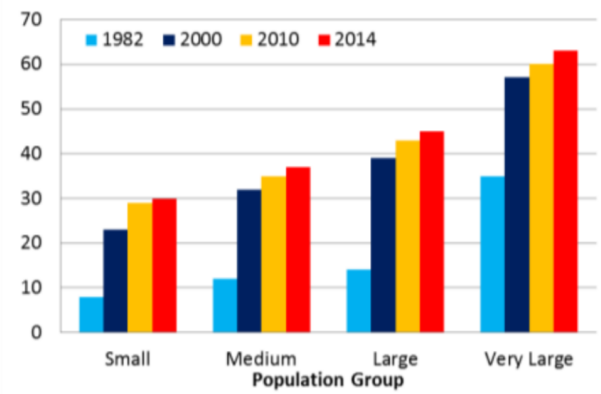


FIGURE 1.3: Congestion Growth Trend – Hours of Delay per Auto Commuter [100]

Traffic Congestion, one of the major challenges against sustainability and resilience will be particularly studied in this thesis. In recent years, traffic congestion in urban areas has become a serious problem due to the rapid development of urbanisation. It brings a major impact on urban transportation networks that leads to extra travelling hours, increased fuel consumptions and air pollution. It is totally the flip side of a sustainable and resilient transport network. A recent urban mobility report [100] states that in the year 2014 in the United States, the monetary loss due to traffic congestion was evaluated as $160 billion, representing 6.9 billion hours of extra travel time and 3.1 billion gallons of wasted fuel. As shown in Figure 1.3, larger cities with bigger population group face more serious traffic congestion problems. Besides, according to [23], CO2 emissions have more than doubled since the early seventies and increased by around 40% since 2000. As shown in Figure 1.4, transport accounted for one quarter of total emissions in 2016 at around 8 GtCO2, a level 71% higher than what was seen in 1990 and 74% of the emissions come from road transport.

Generally, the congestion can be categorized into recurring congestion (RC) and non-recurring congestion (NRC) [75]. Recurring traffic congestion is defined as a congestion that consistently happens at the same place during the same time each day. It refers to the congestion caused by the growing number of vehicles and a lagging city infrastructure with limited capacity. It usually is treated as a capacity problem and can be solved by increasing the roadway capacity. However, from a sustainability perspective, increasing capacity is not the efficient way to reduce traffic congestion socially, environmentally and

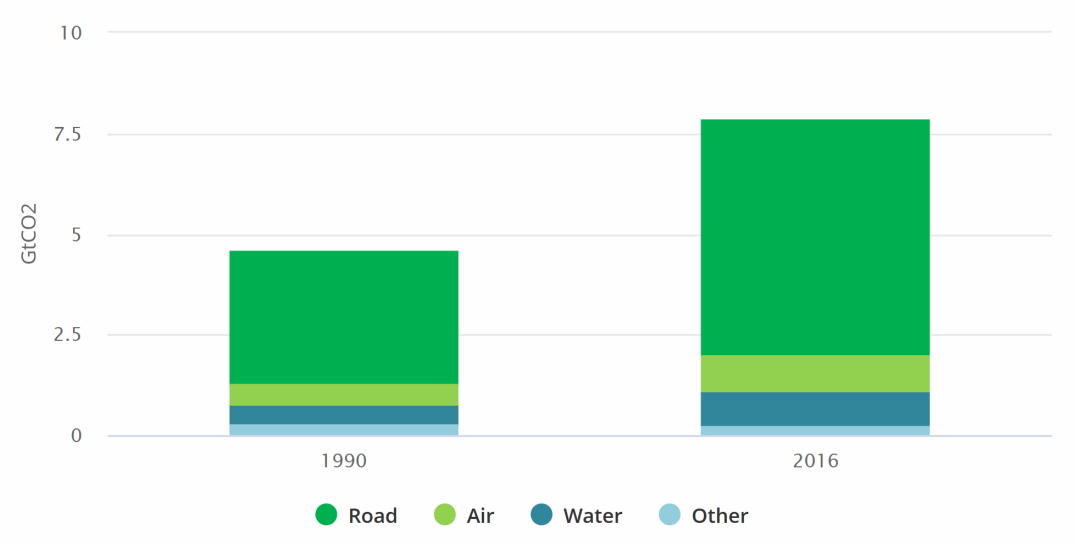


FIGURE 1.4: Global transport CO2 emissions [23]

economically [74]. Therefore an alternative route path optimisation is needed to distribute the traffic in order to reduce the traffic congestion.

Meanwhile, non-recurring traffic congestion occurs randomly without expectation. It refers to those congestion caused made by unexpected events, such as construction work, inclement weather, accidents, and special events [43]. The detection of non-recurring traffic congestion is critically more difficult compared to the recurring type because it requires real-time traffic information [75]. Improving roadway conditions does not seem to be a good option because of its unpredictable nature. In this case, a good vehicle traffic system is needed to control the traffic in dealing with unexpected events. Unsurprisingly, the NRC accounts for a larger proportion of traffic delays in urban areas compared to the RC due to its unpredictable nature [104]. As a result, significant attentions have been paid to addressing the NRC issues. Often these solutions targeting NRC issues will not require a huge financial investment in a city's infrastructure. In addition, another advantage of targeting on NRC is that most solutions to solve the RC issues rely on huge financial investment to increase the roadway capacity in cities.

A US Federal Highway Administration report [109] defined six sources of congestion as shown in Figure 1.5 which are (1) Bottleneck – vehicles stuck at narrow road; (2) Traffic Incidents – The delay of traffic because of vehicles crashing or spoil; (3) Work Zones – the road itself or a building that is beside a main road are under construction or maintenance activities; (4) Bad Weather – extreme weather such as heavy rain, snow, fog that can cause the congestion; (5) Poor Signal Timing – traffic light controller does not control the traffic signal efficiently with the time allocated for a signal not matching the traffic volume; (6) Special Event – Unexpected event that causes congestion, such as a marathon, car racing.

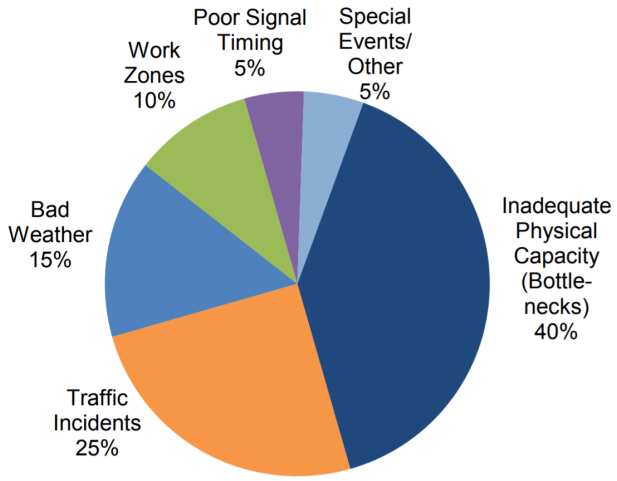


FIGURE 1.5: Souces of Congestion[109]

As it is up to 40% of congestion is caused by inadequate physical capacity. Urban road traffic congestion is mainly considered as the consequence of short supply in road capacity with respect to fast growing traffic demand. The modification of road infrastructure is not as flexible as traffic demand. One way to alleviate urban traffic congestion is the implementation of a public policy to restrict the growth of traffic demand.

The difficulty of deploying a centralized vehicle traffic management system is another issue for a sustainable and resilient transport network. An urban transportation network is usually inefficient due to the bad traffic management. Therefore, a centralised vehicle traffic management system which can supervise the routing process for local authorities by monitoring the urban traffic, and react instantly against unexpected event in the traffic, is the key to build an intelligent urban transport network [20]. However, a centralised vehicle traffic management system relies largely on the application of advanced technologies. Due to the high speed changes in traffic systems and the wide distribution of vehicles on the roadway [21], a centralized vehicle traffic management system requires sensing equipment deployed on each vehicle and advanced infrastructure on the roadway to adapt to the real-time traffic situation. Although some network technologies such as wireless sensor network are getting widely used, the lack of infrastructure in certain places may cause the delay in response from the central traffic control, when an accident or unplanned event takes place. It is especially more evident for those developing countries with high population density. Therefore, the main challenge of designing a centralised traffic management system relies on whether the infrastructure of the urban environment could support a reliable communication between vehicles and roadside infrastructure in order to provide rapid information sharing.

Vehicle route optimisation is another major topic in a sustainable and resilient transport network, Dijkstra [29] proposed static algorithm to find the shortest path without considering any external factor such as congestion, accident, or average vehicle speed. Despite the fact that [7] proposed an improved

approach with weight vertices, these methods are not practical enough as the traffic of transport networks is changing with time and specific constraints. Therefore, vehicle routing optimisation should always optimise the path continuously with the real-time information and adapt to the latest circumstance.

Kerr and Menadue [56] argue that planners and policy-makers are making efforts to adapt to various global processes that impact cities today. However, these are often related to spatial (i.e. urban densification), economic (i.e. economic crisis) and environmental (i.e. global warming) changes and tend to leave out the complex problems regarding social costs. Friesen et al. [34] stated that Technology will certainly play a major role in this transformation. Changes in consumption patterns can drive the creation of new technologies necessary for sustainability and their adoption and diffusion at the desired pace. Success in bringing about these changes will require substantial reorganisation of the economy and society and changes in lifestyles. Economic and financial incentives for the creation and adoption of new technologies will be needed which may include innovative policy reforms.

There are a number of techniques proposed to tackle the NRC problem although it is a challenging task given the unpredictable road conditions. One popular type of method focuses on detecting and predicting traffic congestions by utilising both the historical and real-time sensor data [123, 38]. These methods provide useful information for drivers to avoid accidents and roadblocks. However, when drivers receive the information on the road, most likely they have to make subjective decisions relatively quickly under an adversarial and stressful environment while driving. This may actually cause further severe issues in driving. Optimising traffic signal control and management is another promising way to alleviate NRC. In such work, optimisation algorithms, such as ant colony or genetic algorithm, are used to reduce the average number of vehicles waiting in the queue at junctions [105, 121]. Alternatively, vehicle routing and navigation systems show great potential from the personal level to solve the NRC problem. It assumes that the overall travelling time in NRC can be reduced significantly when each vehicle can make a better path planning to its destination [6].

Currently, an increasing number of consumers have taken an interest in car navigation equipment. Car navigation is no longer a luxury that is only for the rich. A car navigation system is offered as one of the many extras or as an advertising stunt of more and more "middle-class" cars. In the future, car navigation equipment may become as normal as air-conditioning. Of course, also other vehicles, such as trucks, buses and motorcycles can use car navigation. A navigation system offers the driver the possibility to be guided to his destination, by means of spoken and/or visual advices. Figure 1.6 shows the example of a car navigation system. Although in general common GPS applications such as Google map or Waze still rely on shortest path algorithm [63], they also allow drivers to access some real-time traffic information such as accidents, construction, road blocking etc, which could be biased and inaccurate due to the information relying heavily on human input.

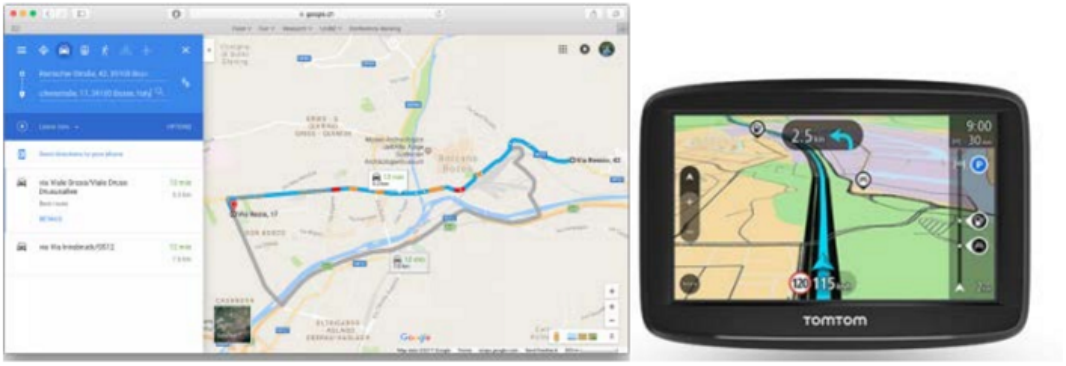


FIGURE 1.6: Examples of car navigation system

Although recent research on automated routing and navigation has achieved reasonable results, there are still several open questions need to be addressed:

1. Vehicle navigation is static and difficult to response to emergency situations.
2. Vehicle navigation does not consider different features in roadway conditions.
3. Vehicle navigation algorithm is lack of ability to self-evolution.

1.3 Research Aims and Objectives

This research aims to **address the urban traffic congestion problem by developing a novel, self-evolution and real-time vehicle route optimisation method,** which is able to navigate vehicles by learning and adapting to the complexity of the urban transportation network. In the term of self-evolution in vehicle route optimisation, it enables the ability to dynamically evolve or adapt in response to the unexpected. However, the real-time is a level of responsiveness to the events in urban transportation. By achieving that, a major city could fully utilise the capacity of its existing transportation network, and also reduce the travel time for individual road users.

There are two extended subsidiary aims for this research; 1) **Create an artificial environment to allow simulation for the testing of complex optimisation models for urban transportation networks in a reasonable time with minimum cost.** A good artificial environment is important in this research as running experiments with traffic in the real environment is just simply not practical. It also allows us to prove the concept of the proposed methodology and provide a stable environment for future research. 2) **Synthesise the optimisation methodology and artificial environment to support a decision making platform which can associate with sustainability and resilience of urban transportation network.** By combining the proposed optimisation methodology and enhanced environment, we could

take this research a step further toward sustainability and resilience, such as reducing the CO2 emissions or improving the safety of the urban transportation network.

These aims are achieved after completing the steps below:

1. Studied and investigated the current methods of route optimisation or traffic distribution for sustainable and resilient transportation networks, analysed their strengths and weaknesses and possible improvements.
2. Studied the latest version of SUMO simulator to understand its techniques and algorithms employed.
3. Evaluated a vehicle route optimisation methodology based on the research above.
4. Extended the vehicle optimisation methodology with the consideration of self-evolution and real-time responsiveness.
5. Enhanced the SUMO simulator to make it as an artificial environment for proposed optimisation methodology.
6. Implemented the proposed optimisation methodology in enhanced SUMO simulator with real urban map and evaluate the result.
7. Analysed the proposed optimisation methodology associated with sustainability and resilience.

1.4 Research Novelty

The contributions of this thesis are summarized as follows:

- **Design of a novel framework to facilitate the vehicle route optimisation research under complex urban transportation context.** This thesis proposes a novel framework to provide an accessible way to optimise the vehicle route planning problem using DRL methods. It enhances the SUMO simulator in order to make it more suitable for optimising vehicle route selection with DRL algorithms. The enhancements include providing an improved calculation method for expected travel time on a road depending on different circumstances, defining the segments in each edge to indicate the best timing for obtaining states and converting the SUMO network graph to a dual graph to model the states and actions in an urban network. The hand-designed controllers in the proposed framework enable the interaction between environment and external RL library through SUMO API TraCI, to allow model training in a rich environment with complex dynamics for vehicle route optimisation. Therefore, the DRL model could be trained across road networks of different size, density, number of edges and lanes. The demand traffic, network characteristic or vehicle behaviour in the experiments can be easily monitored and controlled. Besides, the state space and reward functions can

be constructed from the environment. This framework makes a more realistic and interactive environment by embedding the smart agents (DRL models) into the traffic simulator and the extensibility of the framework provides huge flexibility to extend the features of framework for future RL problems.

- **Design of effective observations, reward scheme and DRL algorithms to achieve efficient convergence of the DRL training** This thesis describes an effective observation as the representation of current traffic conditions within a specific area of the urban network. The representation variables contain multiple parameters reflecting the circumstances in the global urban transportation network to precisely describe the complexity of its dynamics. Besides, this thesis proposes an algorithm to measure the impact of individual vehicles on air pollutant emission called VEI. VEI takes several vehicle emissions as input such as nitrogen oxides, hydrocarbons, carbon monoxide, and particulate matter to compute the level of impact. Based on that, this thesis proposes 2 reward schemas to train the DRL model for vehicle route optimisation. One reward schema aims to reduce the total travel time of a vehicle, and another one aims to minimise the VEI from a vehicle. The results show both reward schemas are efficient to optimise vehicle route. Furthermore, a Euclidean distance based exploration method is proposed in this thesis to combine with the traditional ϵ greedy exploration, the result shows that it achieves more efficient convergence of DRL training than the traditional method.

- **Integration of the proposed vehicle route optimisation approach with real urban map to achieve a more sustainable and resilient urban transportation network.** The proposed vehicle route optimisation approach in this thesis aims to improve the sustainability and resilience in the urban transportation network. The two proposed reward schemas are applied in real urban networks with different sizes and different levels of demand traffic in order to evaluate their performance. Although both reward schemas are able to optimise vehicle route significantly to avoid traffic congestion, the results show the travel time based reward schema performs better in minimising the total travel time of individual vehicles, meanwhile the VEI based reward schema gets the least vehicle emissions to complete a trip. In terms of sustainability and resilience, the travel time based reward schema is suitable for emergency vehicles in the road such as ambulances, fire-fighting cars or police cars as those vehicles need to arrive at their destination as soon as possible in order to maintain the urban safety. However, the VEI based reward schema is suitable for general drivers in urban transportation networks to minimise the vehicle emissions for a more sustainable urban transportation network.

In order to achieve the novelty of this thesis, we overcome several technical challenges in our research. The first challenging part is to model the vehicle

navigation system as a Markov Decision Process in order to apply deep reinforcement learning method. Secondly, due to the limited hardware resources (only one GPU for whole training), a very efficient DRL elements need to be designed to reduce the model training time and achieve remarkable results. This including the designation of the state, action and rewards to form a DRL based vehicle navigation system for sustainable and resilient urban transportation network. Furthermore, another major challenge of our project is to create a framework between the traffic simulator and the DRL components in order to achieve the model training with virtual environment. Lastly, the challenge is to make sure the model training and testing are functioning seamlessly with multiple platforms and able to produce remarkable result for this project.

1.5 Thesis Structure

The remainder of the thesis is organised into the following 7 chapters and the stucture of this thesis is shown as follows:

- **Chapter 2: Literature Review** This chapter outlines a critical literature review on existing research on vehicle route optimisation. This in fact provides the motivation to carry out the following research by addressing the current limitations. This chapter presents the general algorithms for the shortest path problem, then it introduces the traffic management and the study of non-recurrent traffic congestion. Eventually, the limitations are summarised and in fact the current research gaps are providing motivation to carry out the following research.

- **Chapter 3: Deep Reinforcement Learning Background** This chapters presents the main technical concepts of Deep Reinforcement Learning (DRL) which is mainly to be used in this thesis in order to build the proposed framework for real-time vehicle route optimisation. Various deep reinforcement learning techniques are covered in here including Markov Decision Process, Q-learning, Deep Q-Network etc.

- **Chapter 4: Urban Traffic Simulation** This chapter is to introduce the background of urban traffic simulators. The general simulation models and approaches are overviewed and the comparison of different traffic simulator are presented. This chapter also briefly introduces the traffic simulator SUMO and some of its features which are heavily used in this thesis.

- **Chapter 5: Preliminary Design and Experiment for Vehicle Route Optimisation** This chapter presents how the reinforcement learning method is applied to solve vehicle navigation problem. It shows how an urban network could be modelled as a Markov chain process, how to define the problem statement and the key elements in reinforcement learning. This chapter also discusses the limitation of this approach and how it provides the motivation for the proposed approach in this thesis.

- **Chapter 6: Proposed Framework and Structure Design** This chapter presents the main contribution of this thesis - the proposed vehicle route optimisation mechanism by detailing its architecture and decision making process using a heuristic approach. Specifically, the proposed approach uses DQN algorithm to train a vehicle agent to make better routing decision for vehicle route optimisation in order to achieve the sustainable and resilient urban development goal.

- **Chapter 7: Experiment Implementation and Evaluation:** This chapter describes the experiment implementation of the real-time vehicle route optimisation for sustainable and resilient urban transportation networks. Firstly the python based classes and the components in the experiment are presented. The preparation works that need to be done before running the experiment are introduced with details, including map converting, demand traffic generation, data pre-processing, etc.

- **Chapter 8: Conclusion and Future Work** This chapter summarises the work of this thesis and describes the extent of the limitations overcome by this research. Furthermore, it reveals future directions of this work and how the research will be continued.

Chapter 2

Literature Review

2.1 Overview

This chapter outlines a critical literature review on existing research on vehicle route optimisation. The original idea for solving the problem of traffic congestion was via traffic control and optimisation in which a significant number of research works had conducted. This chapter starts with introducing the famous shortest path algorithm for vehicle route optimisation, including classic Dijkstra's algorithm, A* algorithm, ant-colony algorithm etc. Furthermore, traffic management systems for urban transportation are studied. Eventually, the limitations are summarised and in fact the current research gaps are providing motivation to carry out the following research.

2.2 Shortest Path Algorithm

The very first solution in the early years for the vehicle navigation problem was shortest path algorithm, which aims to find a path between two nodes with minimum travelling distance. Traditionally, traffic modelling approaches consider the road network as a directed acyclic graph, of which nodes represent junctions while edges represent roads and their corresponding driving directions. In graph theory, the shortest path problem (SPP) is the problem of finding a path between two nodes in a graph such that the sum of the weights of its constituent edges is minimized [10]. If the graph is connected, then there is always a finite shortest path between any pair of distinct nodes. This problem has been studied extensively in the past decades. In a navigation scenario, the shortest path problem typically aims to find the optimal path from the source (or actual position) $s \in V$ to destination (or target) $t \in V$ which exhibits minimal cost. Figure 2.1 shows an example of theshortest path problem. The best approach to find this path crucially depends on the edge cost features. Edge costs could be either a static constant values (shortest distance, shortest time, etc) or a complex function (travel time dependent on departure time). The shortest path problem essentially is an optimisation problem and it is widely applied in different areas, such as communications, game development, robot movement, vehicle navigation systems, etc. In this thesis the research study focuses on the area of transportation in urban environments (vehicle navigation systems).

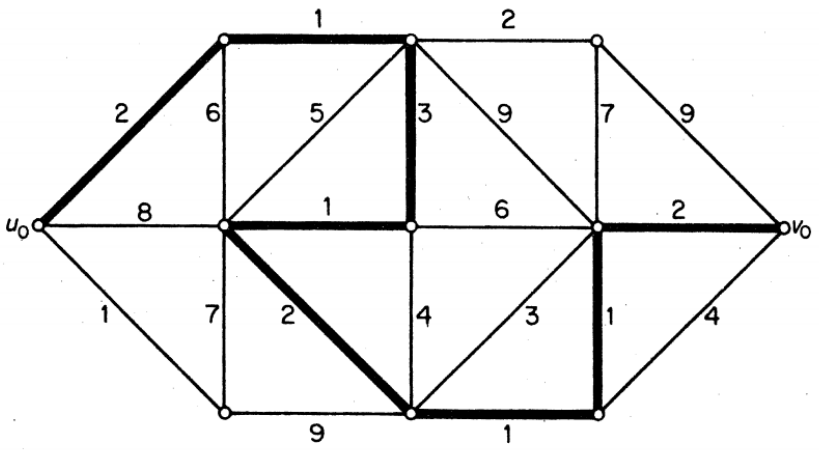


FIGURE 2.1: A (u_0, v_0) path of minimum weight [10])

There are two types of approaches for shortest path algorithm, which are one-directional search and bidirectional search. The one-directional search performs the search in one way, from the source node (or destination node), and then expanding the visited area around the source (or destination) node until the destination (or source) node is found. In other words, those algorithms only search from one direction and do nothing on the other direction. There are a wide-range of applications which are covered by this approach. However, in a navigation scenario, one-directional search could be inefficient when dealing with a large network which contains rapid, frequent changes as this approach need to re-search for an updated route from scratch when a vehicle deviates from its route or change its destination [77]. Therefore, to achieve better performance, the idea of the two way search (bidirectional search) approach which is to search in both directions simultaneously was proposed. In the following sub-sections, the well-known shortest path algorithms and the implementation of bidirectional search are briefly introduced.

2.2.1 Dijkstra's Algorithm

Dijkstra's algorithm was proposed by Dijkstra [29] in 1959 and is named after himself. It is a classical algorithms for the shortest path problem in terms of running time and is also used as the standard algorithm for evaluating other algorithms for the SP problem. It is used to find the shortest path from a single source node to all other nodes on a directed graph with non-negative edges cost only. Dijkstra's algorithm works on a static network where the edge weights on the network are static and deterministic. The idea of the algorithm is to exam the closest node to the start node, then update the distances from the source node to other nodes via their adjacent nodes using the labeling technique. However, it uses a weighted graph and a priority queue to decide which node contains minimal cost to be expanded first. The reached nodes are labelled as "visiting nodes" and stored in a priority queue with respect to their current distances from the source node.

Algorithm 1: Dijkstra's algorithm

```
1  Function Dijkstra(graph, source, target):
2      create unvisited vertex set Q
3      foreach vertex v in graph do
4          dist[v] ← ∞
5          prev[v] ← UNDEFINED
6          add v to Q
7      end
8      dist[source] = 0
9      while Q is not empty do
10         u ← vertex in Q with min dist[u]
11         if u = target then
12             return dict[], prev[]
13         else
14             remove u from Q
15         end
16         foreach neighbor v for u do
17             alt ← dist[u]+ length(u, v)
18             if alt < dist[v] then
19                 dist[v] ← alt
20                 prev[v] ← u
21             end
22         end
23     end
24     return dict[], prev[]
```

Algorithm 2: Recover shortest path from the array of previous nodes

```
1  Function (prev[], source, target):
2      p ← UNDEFINED
3      if prev[t] = NULL then
4          return p
5      end
6      i ← t
7      while i not equal s do
8          p ← {prev[i], i} ⊕ p
9          i ← prev[i]
10     end
11     return p
12 RecoverPath
```

The process of how Dijkstra's algorithm finds the shortest path is shown in Algorithm 1. Dijkstra's Algorithm (DA) takes 3 input elements: the graph, the source and target vertex. It returns 2 output elements: one array dist[] for retrieving the cost value for the shortest path. This array (dist[]) stores the minimum cost value from source vertex to one of the other vertices indicated in the array index, while the array prev[] is used for retrieving the shortest path

sequence. prev[] stores the predecessor vertex of a certain vertex given in the array index, according to the found shortest path. DA initializes the value of dist[] as infinity, the value of prev[] as empty for all vertices in the given graph, and adds all initialized vertices into a set Q, which is used for recording the unvisited vertices during the following execution process of DA. As the last step before the searching process of DA, the minimum cost value from source to destination, formalized as dist[source], is set to 0. The algorithm starts from the source vertex searching each of its neighbours by updating their dist[], then it moves on to one neighbouring vertex with the minimum dist[]. DA repeats this searching process iteratively until the target vertex is chosen as the current vertex or until all vertices are examined.

Dijkstra's algorithm can also be used to find shortest paths from one source node to many destination nodes on a given graph. In this case, the algorithm terminates when the priority queue is empty or when all the destination nodes are visited. The previous node of the node v on the shortest path from the source node to v is stored in Pre(v). By tracking backward the previous nodes of the visited nodes on the graph, the shortest path from the source node to any visited node can be explicitly recovered. This tracking procedure, namely RecoverPath(s,t,Pre[]), is expressed in Algorithm 2.

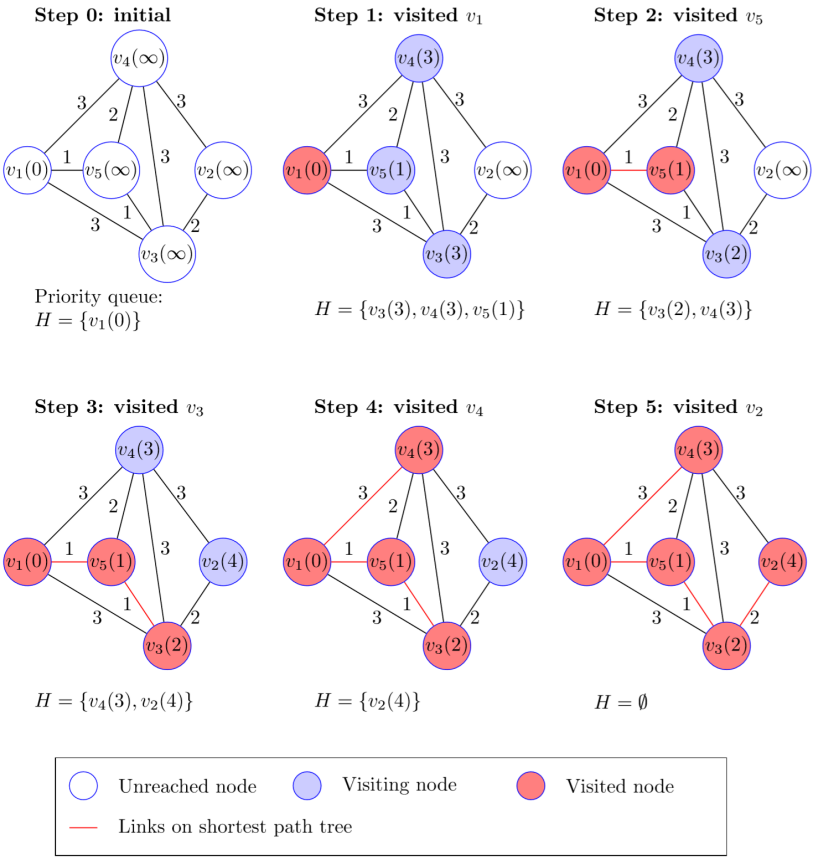


FIGURE 2.2: Illustration of Dijkstra's algorithm [85])

Figure 2.2 illustrates Dijkstra's algorithm to find the shortest path from the node v_1 to the node v_2. In the initial step, the current distances from v_1 to other nodes are set to infinite, but the distance from v_1 to v_1 is set to 0. The source node v_1 is then added to the priority queue H. In Step 1, the node v_1 is extracted from the list H and marked as visited node (in red). Then, the distances of its adjacent nodes, i.e. v_3, v_4 and v_5 are updated. Because $g(v_1, v_3) = \infty > g(v_1, v1) + w(v_1, v_3) = 3$, the current distance from v_1 to v_3 is updated to 3, i.e. $g(v_1, v_3) = 3$. The node v_3 is then marked as visiting node (in light blue) and added to the priority queue. Repeating the updating procedure for the node v_4 and v_5 we have $g(v_1, v_4) = 3, g(v_1, v_5) = 1$. After Step 1, the priority queue H has three visiting nodes v_3, v_4, v_5 with their current distances from v_1 as follow: $g(v_1, v_3) = 3, g(v_1, v_4) = 3, g(v_1, v_5) = 1$. The node v_5 is the node with minimum distance from the source node v_1.

In Step 2, since the visiting node on the priority queue with the smallest current distance is v_5, it is extracted from the priority queue and marked as the visited node. The remaining steps, i.e. Step 3, Step 4, and Step 5, repeat the updating procedure as in Step 1. The algorithm terminates at Step 5 when the destination node v_2 is visited. The shortest path tree from v_1 to all visited nodes is marked in red. Based on this tree, we can track backward to find the full shortest path from the source node v_1 to not only the destination node v_2 but all the visited nodes. For example, the shortest path from v_1 to v_2 is (v_1, v_5, v_3, v_2), the shortest path from v_1 to v_3 is (v_1, v_5, v_3).

The advantages of Dijkstra's algorithm are its simplicity and its easy implementation, therefore it is used widely in both academic research and real-world applications. For instance, Google Maps and most GPS navigation applications initially used Dijkstra's algorithm to find the most efficient route [63]. However, despite its simplicity, Dijkstra's algorithm is inefficient for large road networks. Therefore, based on the framework of Dijkstra's algorithm, many variants of this algorithm have been proposed in the following decades using techniques such as improving the data structure, introducing new heuristic functions, and reinterpreting the definition of the cost function.

2.2.2 Bellman-Ford's Algorithm

The Bellman-Ford algorithm is the common name for an algorithm for computing single-source shortest paths in directed graphs. It is based on where the graph G contains no negative cycles, the shortest path is always simple and therefore any optimal path $\pi(s, t)$ contains maximum n - 1 edges. The algorithm maintains for every node a (temporary) distance label d() initialized with ∞. Then the label d(s) is set to zero. In every round all edges get relaxed, i.e. for e = (v,w) $\in$ E the property d(v)+c(e) < d(w) is checked and if possible d(w) is updated to the new(smaller) distance value. Additionally a predecessor label is stored for each node. If relaxing an edge (v,w) leads to an update of d(w), the predecessor label of w changes to v. After performing n - 1 rounds of edge relaxations the distance label of every node v reflects the minimal distance from

s to v and the optimal path to t can be backtracked via the predecessor labels. In case there are negative cycles in G there maybe no shortest path, because there are paths with arbitrarily low costs (passing through the negative cycle repeatedly).

The Bellman-Ford can be adapted to decide whether G contains such a cycle by performing an additional round of edge relaxation. There exists a negative cycle in G, if the distance of any node changes in the n^th round. The run time for a single query is O(nm) as in each of the n - 1 rounds m edges are considered and edge relaxations can be performed in constant time (if distance and predecessor labels are stored e.g. in an array and a suitable graph representation is used). Observe that it is always sufficient to relax in every round only the edges incident to nodes that were updated in the last round (or adjacent to s in the first round). While this does not change the theoretical runtime it might significantly reduce the query time in practice.

Algorithm 3: Bellman-Ford algorithm

```
1  Function BellmanFord(graph, source, target):
2      foreach vertex v ∈ V[graph] do
3          dist[v] ← ∞
4      end
5      dist[source] = 0
6      for i ← 1 to |V[G]| do
7          relaxed ← FALSE
8          foreach neighbor v for u do
9              alt ← dist[u]+ length(u, v)
10             if alt < dist[v] then
11                 dist[v] ← alt
12                 relaxed ← TRUE
13             end
14         end
15         if relaxed = FALSE then
16             exit the loop
17         end
18     end
19     foreach neighbor v for u do
20         alt ← dist[u]+ length(u, v)
21         if alt < dist[v] then
22             return FALSE
23         end
24     end
25     return TRUE
```

The process of how the Bellman-Ford algorithm finds the shortest path is shown in Algorithm 3. Bellman-Ford also takes 3 input elements: the graph, the source and target vertex. It firstly initializes the value of dist[] as infinity, and sets the distance to source dist[source] as zero. After that the algorithm

starts from the source vertex searching each of its neighbours by updating their distance dist[]. Then it iteratively relaxes those estimates by finding new paths that are shorter than the previously overestimated paths. By doing this repeatedly for all vertices, the Bellman-Ford algorithm is able to guarantee that the end result is optimized.

2.2.3 A-Star Algorithm

Despite its simplicity, Dijkstra's algorithm is inefficient for large road networks. This is primarily due to shortest paths needing to be rapidly identified either because an immediate response is always required and also the shortest path needs to be re-computed repeatedly if the vehicle deviates from the pre-computed route or changes the desired destination. Hence the search for speed-up techniques is an important challenge for the purpose of reducing the execution time of the shortest path algorithms. One of the first approaches in this direction is the A* algorithm [44]. A* is the foundation for a heuristic search on the framework of Dijkstra's algorithm and widely used in artificial intelligence [86]. It is a greedy and goal-directed approach, that uses lower bounds on path weights to find a better processing order of the nodes, decreasing the search space significantly. A* does not rely on any preprocessing and hence is especially useful if edge costs might change over time.

The common idea of the Dijkstra's algorithm and A* algorithm is the labeling technique and updating current distances from the source node s to the adjacent nodes of the visited node. Both of the algorithms terminate when the destination node t is visited or when the queue of visiting nodes is empty. The difference between the algorithms is in the order of nodes to visit. In each iteration, Dijkstra's algorithm chooses a node v to visit with respect to only the current distance from the source node s to v, i.e. $g(s, v)$. The algorithm does not consider the potential distance from v to the destination node t. Thus, some nodes staying very far from the destination node may be visited before some closer nodes. This is said to be the disadvantage of Dijkstra's algorithm. In order to overcome that difficulty, A-Star algorithm requires a potential function, denoted $h(v, t)$ or $h(v)$ for short, to evaluate the potential distance from a node v to the destination node t. The function returns a lower bound on the shortest distance from a node to the destination node. The potential function is useful for knowing whether a node is close to or far away from the destination node. A popular lower bound of the shortest length between two nodes is the Euclidean distance—the length of the straight line between the nodes. The Euclidean is surely a lower bound on the shortest length of a path from v to t. Note that the value of the potential function at the destination node must be zero, i.e. $h(t, t) = 0$.

The A-Star algorithm chooses nodes to visit according to the sum of the current distance from the source node s and the lower bound distance to the destination node, i.e. nodes are visited orderly according to the sum $f(v) = g(s, v) + h(v, t)$. By choosing nodes to visit with respect to the value of $f(v)$,

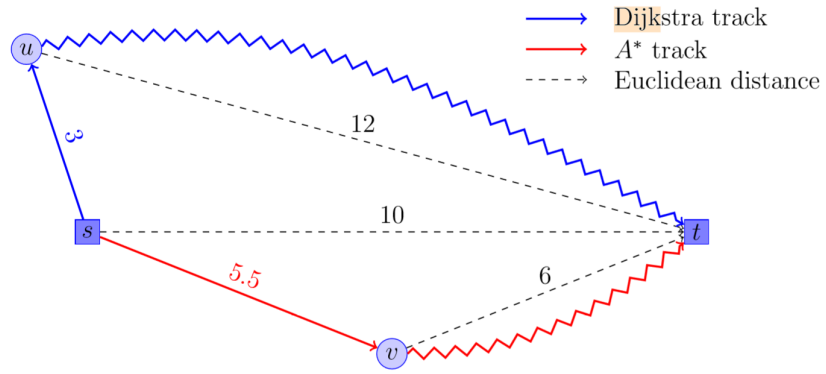


FIGURE 2.3: Tracking nodes in A-Star algorithm and in Dijkstra's algorithm [85])

A-Star algorithm may not need to visit nodes, that are very far from the destination node, thus the destination node is possibly reached after visiting fewer nodes than those of Dijkstra's algorithm. Figure 2.3 shows an example of the difference in the order of nodes to visit between Dijkstra's algorithm and the A-Star algorithm using Euclidean distance as a lower bound on the length of the shortest length of a path connecting two nodes. Both of the algorithms start at the source node s and then update the distance from s to the adjacent nodes u and v, i.e. $g(s,u) = 3$ and $g(s,v) = 5.5$. Because $g(s,u) < g(s,v)$, Dijkstra's algorithm chooses the node u to be the next visited node. However, it can easily be seen that the node u is further from the destination node than s. In other words, the direction from s to u goes away from t, and Dijkstra's algorithm cannot take this fact into account. The A-Star algorithm considers also the Euclidean distance from all the nodes to the destination node, i.e. $h(u,t) = 12 and h(v,t) = 6$. We have $f(u) = g(s,u) + h(u,t) = 15$ and $f(v) = g(s,v) + h(v,t) = 11.5$, thus the node v is chosen to be the next visited node, and it is actually closer to t than u.

Algorithm 4: A-star algorithm

```
1  Function AStar(graph, source, target):
2      create unvisited vertex set Q
3      foreach vertex v in graph do
4          dist[v] ← ∞
5          prev[v] ← UNDEFINED
6          add v to Q
7      end
8      dist[source] = 0
9      while Q is not empty do
10         u ← vertex in Q with min dist[u]
11         if u = target then
12             return dict[], prev[]
13         else
14             remove u from Q
15         end
16         foreach neighbor v for u do
17             alt ← dist[u]+ length(u, v)
18             if alt < dist[v] then
19                 dist[v] ← alt
20                 prev[v] ← u
21             end
22         end
23     end
24     return dict[], prev[]
```

The A-Star algorithm to find a shortest path from a source node *source* to a destination node *target* is described in Algorithm 4. It can be easily seen that A-Star algorithm considers more "the future", whereas Dijkstra's algorithm only focuses on the present. The A-Star algorithm is obviously the same as Dijkstra's algorithm when its potential function is equivalent to zero, i.e. $h(x,t) = 0$. Therefore, in theory the complexity of A-Star algorithm equals those of Dijkstra's algorithm. However, its average running time for querying a number of pairs of nodes is normally better than those of Dijkstra's algorithm. The running time of A-Star algorithm depends considerably on the potential function $h(x)$, i.e. the better lower bound the potential function $h(x,t)$ can give, the faster the algorithm reaches to the destination node. According to this feature, various algorithms based on A-Star algorithm have been developed by applying different potential functions, e.g., routing services in real transportation networks using the landmark technique or shortcut technique.

2.2.4 Heuristic Shortest Path Finding and Re-planning

In general, a heuristic is considered as a trade-off between computation time and optimality. Given a large-scale problem, heuristic-based methods are often used to provide a sub-optimal solution within an acceptable time range.

Unlike many other heuristic algorithms, A* with a well-designed, or more formally called admissible heuristic function can guarantee an optimal solution, but with a significantly reduced search space compared to Dijkstra's algorithm. For example, in a road network scenario, the heuristic function in A* can be implemented using Euclidean distance. As the geographical length of any possible routes between any O/D pair cannot be less than its corresponding Euclidean distance, this heuristic implementation is called admissible, which means it never overestimates the cost in practice.

In dynamic environments, where the edge cost is changing over time, the optimal route needs to be updated accordingly. The intuition to do re-planning is to run A* from scratch once the graph is updated. However, re-planning from scratch is a waste of computation when the changing environment has no effect or only has a minor effect on the previous optimal solution. D* Lite [58] is an efficient re-planning algorithm that only looks at certain areas that have their edge cost changed and repairs the previous route only if it is necessary. This process is achieved mainly by introducing a new heuristic function called "one-step look ahead cost", which is able to detect the changes in environment. Moreover, the whole search process of D* Lite is done in the reverse way from the target vertex to the current vertex, thus preventing a lot of computation on updating the estimated cost from the moving current vertex to the target. Compared to re-planning using A*, D* Lite is more efficient by nearly two orders of magnitude. In addition to the dynamic environment, the typical A* algorithm is also not applicable if a route solution is needed quickly in a complex environment, where the number of vertices and edges in the given graph is excessively large.

Anytime Repairing A* [66] (ARA*) solves this problem by using "inflation factor ϵ" to increase the output value of the admissible heuristic function in the typical A*. It is proven that maximally up to ϵ times computation cost could be saved when $\epsilon > 1$. The large ϵ is set, the faster the algorithm runs, and the worse the optimality of the route will be. ARA* trades off the speed and the optimality by decreasing the value of ϵ iteratively from a relatively large value, until it reaches the time threshold. Anytime Dynamic A* (AD*) [67] combines the advantages of the two algorithms to deal with the dynamic and complex environment in real-time.

2.2.5 Bidirectional Search

Early stage research in applying shortest path algorithms for vehicle road guidance consists of three major directions: bi-directional search, sub-goal search, and hierarchical search. Due to the lack of powerful computation capability and efficient geographical data techniques, these three directions have the same objective: reducing the search spaces. To achieve this objective, bi-directional search [53] starts the process from both directions in parallel, one from origin to destination, the other from destination to origin, until both search processes meet at the same vertex somewhere between origin and destination. The search

from the source node, called forward search, and from the destination node, called backward search. Algorithms following the bidirectional search have their own stopping conditions.

For instance, the bidirectional version of Dijkstra's algorithm, so-called bidirectional Dijkstra, terminates when there exists a node visited from both searching directions. The stopping conditions of bidirectional search using A-Star algorithm, namely bidirectional A-Star algorithm, is more complex since having a visited node in both directions does not guarantee that the shortest path is found [40].

As shown in Figure 2.4, The green ball illustrates the visited area in the forward search, while the red ball illustrates the visited area in the backward search. The balls are enlarged simultaneously until they meet each other at the node v_4, i.e. v_4 is visited in both directions. The shortest path from s to t is the combination of the shortest path from s to v_4, i.e. $p1 = (s, v11, v_4)$, and the shortest path from v_4 to t, i.e. $p2 = (v_4, t)$.

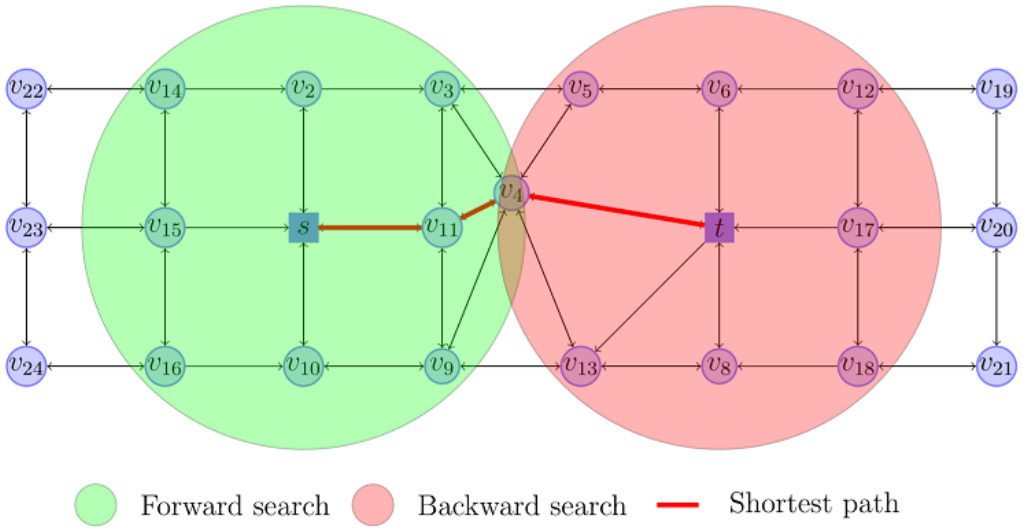


FIGURE 2.4: An illustration of the bidirectional version of Dijkstra's algorithm [85]

2.2.6 Ant-Colony Algorithm

Ant-colony algorithm was firstly proposed by Dorigo et al. [12] which was inspired by the natural behaviour performed by ants in finding food resources. In this natural behaviour, individual ants deposit on the ground a volatile chemical substance called pheromone when they are moving, forming in this way pheromone trails. Ants can smell pheromoneand, when choosing their way, they tend to choose, in all probability, the paths marked by stronger pheromone concentrations. In this way they create a sort of attractive potential field, the pheromone trails allows the ants to find their way back to food sources (or

to the nest). Moreover, they can be used by other ants to find the location of the food sources discovered by their nestmates. Previous experiments have proven that ants are able to find the shortest route between two individual sections. Therefore ant-colony algorithm is widely applied to the vehicle routing problem, although some modifications have been applied depending on different circumstances.

A simple transformation was applied to cost function $Desirability = 1/2_{cost}$. The equation alters the cost function into a desirability scale of range 0.5 to 1.15. This states that the edges with a higher cost will have a lower desirability to be selected while the edges with low cost will have higher desirability.

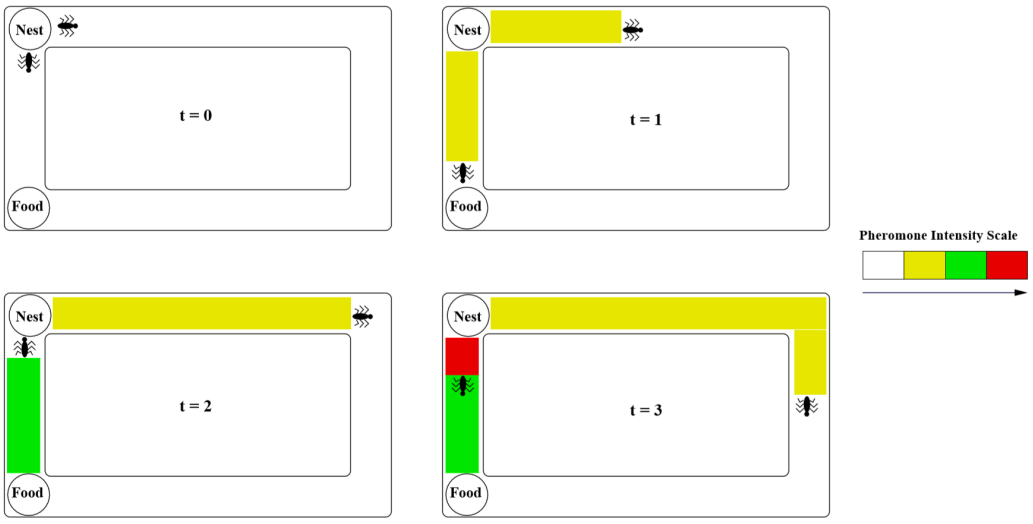


FIGURE 2.5: Example of how the effect of laying/sensing pheromone during the forth and back journeys from the nest to food sources to determine shortest path between two nodes [28])

Figure 2.5 shows in a schematic way how the effect of round-trip pheromone laying/sensing can easily determine the convergence of all the ants on the shortest distance between two available paths. At time t = 0 two ants leave the nest looking for food. According to the fact that no pheromone is present on the terrain at the nest site, the ants select randomly the path to follow. One ant chooses the longest and one the shortest path to the food. After one time unit, the ant who chose the shortest path arrives at the food reservoir. The other ant is still on its way. The intensity levels of the pheromone deposited on the terrain are shown, where the intensity scale on the right says that a darker colour means more pheromone. Pheromone evaporation is considered as negligible according to the time duration of the experiment. The ant already arrived at the food site must select the way to go back to the nest. According to the intensity levels of the pheromone near the food site, the ant decides to go back by moving along the same path, but in the opposite direction. Additional pheromone is therefore deposited on the shortest branch. At t = 2 the ant is back to the nest,

while the other ant is still moving towards the food along the longest path. At t = 3 another ant moves from the nest looking for food. Again, he/she selects the path according to the pheromone levels and, therefore, it is biased towards the choice of the shortest path. It is easy to imagine how the process iterates, bringing, in the end, the majority of the ants on the shortest path.

2.3 Traffic Management Systems

Agent technology is the key concept for implementing distributed artificial intelligence. Specifically, the paradigm of multi agent systems is well suited for the management of road traffic [20], as the road traffic network can be treated as a collective set of geographically distributed local areas, the traffic state is changing over time in each local area, and this change is sensitive to behaviours from any road network participants (i.e. drivers, pedestrians, traffic regulators, etc). This section divides the traffic management systems into two categories: traffic light signal control system and vehicle routing optimisation.

2.3.1 Traffic Signal Control System

Traffic light signal control is considered the most typical application of the multi-agent concept in road traffic management. The most widely deployed systems are Sydney Coordinated Adaptive Traffic System (SCATS) [103] and Split Cycle Offset Optimisation Technique (SCOOT) [48]. Both SCATS and SCOOT have a similar 3-tier hierarchy. Take SCATS for example, as shown in Figure 2.6. The basic agent in the bottom layer is each intersection, which is controlled and coordinated by a regional computer according to the real-time traffic information. All the regional computers are then organized by a central server for high level configuration and optimisation in a particular city. The agents here are regional computers controlling tens of intersections. The main differences between them are the mechanism of reaction to the real-time traffic information. When the traffic states are updated, SCATS chooses the best traffic light signal plan from several candidates that are configured manually in advance.

On the contrary, SCOOT can adjust all the related parameters (i.e. slip, cycle, offset, etc.) and provide an on-line optimised traffic signal plan. This difference is mainly due to the additional types of traffic data collectors (i.e. sensors and cameras) SCOOT has, while SCATS mainly relies on induction loops. More specifically, the different deployments of loop detectors, for example, have led to the aforementioned difference as well. SCATS installs one induction loop at the downstream for each lane to get the traffic information: occupancy, while SCOOT deploys two loop detectors on each lane, one in downstream, the other in upstream, so that it can retrieve traffic information like, queue length, speed, and occupancy. Therefore, more information allows SCOOT to tune the parameters in a finer granularity. Although SCOOT has more flexibility and advanced control mechanism, SCATS has less deployment cost and is proven

to have comparable effectiveness. The two systems have dominated the global market in urban traffic control during the last 4 decades.

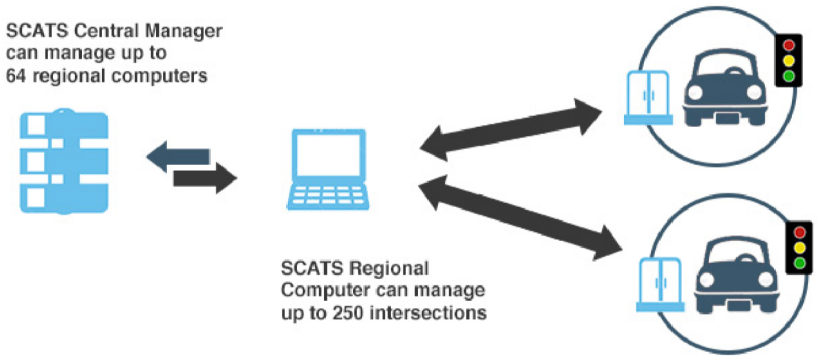


FIGURE 2.6: 3-tier system architecture of SCATS [103])

Traffic prediction technology frequently appears in the recent research on enhancing the multi-agent traffic signal control system. This prediction technology is driven by the increased number of types of collected traffic information from various deployed sensors. One typical example is InSync [87], which had been applied in 31 states and 2300 intersections in the U.S. up to November 2015. InSync was ranked the top in terms of waiting time reduction in several U.S. cities, as evaluated and compared in a survey [101] with four other popular systems. The traffic information collection of InSync is mostly done by Internet Protocol (IP) video cameras. This leads to a huge advantage as many useful pieces of microscopic information can be extracted such as the exact number of vehicles, speed for each particular vehicle, and even vehicle types. By taking advantages of this rich information, InSync can predict short-term traffic conditions to create so call "green tunnels" minimizing the number of stops for the longest platoon. Another way of collecting rich traffic information for predictive control is to use vehicular ad-hoc networks (VANETs), where vehicles are connected and periodically broadcast their states. VANETs are used in the approach proposed by K. Pandit [87] in which an online scheduling algorithm called "the oldest arrival first" is used. It is shown in the presented simulation results that approximately equal-sized platoons can be achieved with significantly reduced intersection delays, as compared to the state-of-the-art algorithm. VANETs are used in a predictive control method proposed by B. Asadi and A. Vahidi [4] that help to achieve minimum use of braking to improve fuel efficiency accordingly. Some pioneering work have tried to apply multi-agent reinforcement learning for adaptive traffic signal control.

Chao et al. [19] proposed an intelligent traffic management system based on RFID for determination of traffic flow. The proposed intelligent traffic light control system (ITLCS) uses an RFID system, which complies with the IEEE 802.11p protocol to detect the number of vehicles and find the time in seconds spent by vehicles on main roads and on side roads passing through the intersection throughout a period of green light. They used Zig Bee modules to send real time data like weather conditions and the vehicle registration information to the

regional control centre. The proposed system can perform remote transmission and reduce traffic accidents.

Traffic Signal Control System is also widely used for emergency units. Bharadwaj et al. [9] proposed an Efficient Dynamic Traffic Control System (EDTCS) to reduce vehicle traveling time and set the highest priority for emergency vehicles at intersections. EDTCS is composed of Traffic Control Unit (TCU), Traffic Monitor Unit (TMU) and Road Side Unit (RSU). Figure 6 shows the process at the intersection. All the vehicles in the sensor area are counted and emergency vehicles are identified by RFID tags. Those emergency vehicles are able to communicate with the RSU via RFID tags. If the RFID tag is positive, RSU will increase the emergency vehicle's number by one. Then the centralised traffic server will collect the number of both normal and emergency vehicles and switch the traffic signal to green for about 30 seconds for the side of emergency vehicles to go through the intersection.

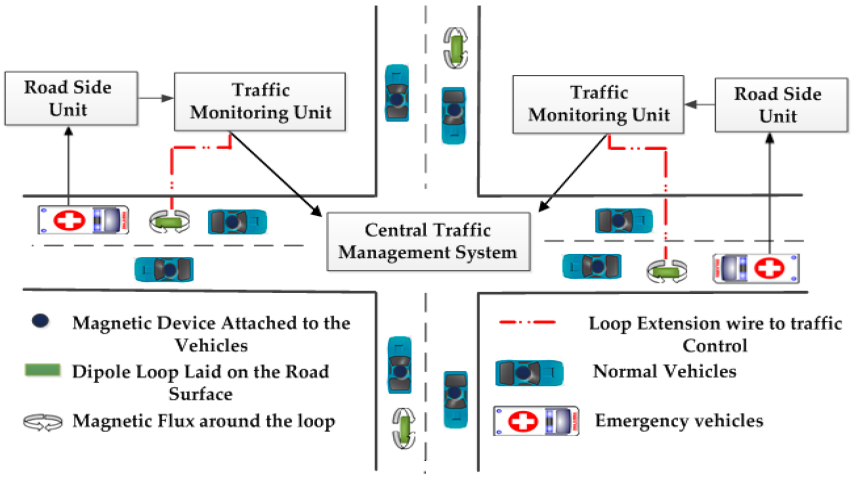


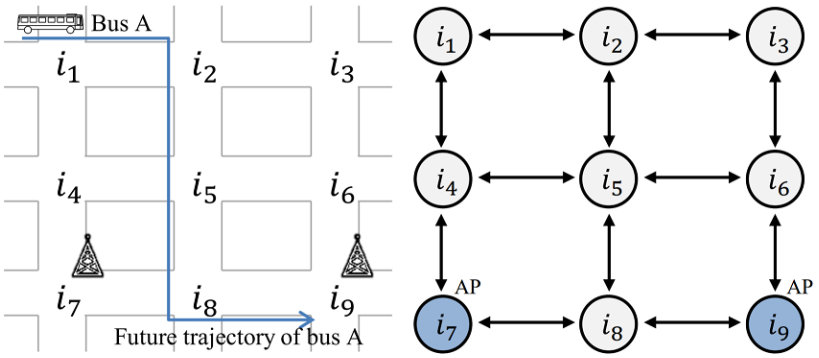
FIGURE 2.7: Layout architecture for efficient dynamic traffic control system [103])

2.3.2 Vehicle Routing Optimisation

Similar to the robotics research in the artificial intelligence area, most multiagent systems for vehicle route guidance consider each vehicle as an agent, then use different proposed coordination mechanisms to achieve a reduction of total travel cost (i.e. travel time, travel distance, fuel consumptions, etc.). For example, a decentralised delegate multi-agent system [22] is proposed to reduce the traffic congestion using anticipatory vehicle routing. The word "delegate" comes from the pheromones in the ant colony algorithm used for agents to exchange information. CARAVAN [27] puts vehicle agents into VANETs environment, and applies "virtual negotiation" to exchange route allocation cooperatively to achieve the reduction of total travel delay and communication overhead. Sejoon Lim [68] built a probabilistic path choice model based on a realistic dataset. In this model, each driver's route decision is regarded as a fractional flow. All vehicle agents in the same local area can exchange their route choice to achieve

UE or SO. Relying on a central server, participatory routing planning [118, 117] uses the previously planned routes to estimate future traffic conditions for the incoming routing requests. This routing collaboration among vehicle agents is done by the communication between the cloud server and in-vehicle mobile devices (i.e. smartphone).

Choi et al. [21] designed an optimal routing policy for the vehicular sensor network (VSN). This project developed a delay-optimal VSN routing algorithm by capturing three key features in urban VSNs: (i) vehicle traffic statistics, (ii) any cast routing and (iii) known future trajectories of vehicles such as buses. In Figure Figure 2.8(a), two Wi-Fi Access points (AP) are placed at the intersections i7, i9 and the path of bus A is the sequence of intersections, i1, i2, i5, i8 and i9. However, the Figure 2.8(b) represents a network that can be potentially used for delivering data packets, and the existence of data links in the network is highly uncertain. They conducted simulations on a GloMoSim simulator and compared the performance of Optimal VSN Data Forwarding with the Vehicle Assisted Data Delivery (VADD) algorithm and Trajectory Based Data (TBD) forwarding scheme. The simulation results show that the OVDF outperforms other algorithms.



(a) Roadmap (b) Road network graph G

FIGURE 2.8: Roadmap and its corresponding road network graph [21])

There are also some approaches which use the Ant-Colony Optimisation method for optimal routing. Nahar and Hashim [83] proposed a traffic congestion control method based on different preferences to create an optimal traffic system. These preferences allow the algorithm to reduce average travelling time by adjusting ant colony variables. Their results show that the number of ants is directly correlated with the algorithm performance. However, this method does not perform well when there is a small number of agents in the network. Kammoun et al. [83] however proposed an adaptive vehicle guidance system which is able to search the best path in smarter way by using real-time changes in the network. In order to achieve dynamic traffic control and improve driver request management, this method used three types of agents, namely, city agent, road supervisor agent and intelligent vehicle-ant agent. Besides, a multi-agent

evacuation model was introduced by Zong et al. [122] to minimise the total evacuation time for vehicles and balance traffic load. Experiments have shown that MAS is more effective than a single agent system.

Cong et al. [24] developed a model to optimise dynamic traffic routing by using a two-step approach: network pruning and network flow optimisation. In the network-pruning phase, ant pheromone is removed after the best route is found by the agents to increase the exploration rate. In the flow optimisation phase, which is based on ACO with the stench pheromone and coloured pheromone, the agents correspond to the links selected in the network-pruning phase only. Moreover, this two-step approach reduces the computational burden by addressing complex, dynamic traffic control problems. Kponyo et al. [24] proposed a distributed intelligent traffic system which uses vehicle average speed as a parameter to determine the traffic condition. This system guides cars to paths with low traffic. Therefore, this system selects the best path more efficiently in comparison with the scenario where the agents select their path randomly. Last but not least, BeeJamA [115] considers each junction-controlled region as an agent for traffic congestion problems. The agent in BeeJamA plays a role like a router in a computer network by keeping an updated routing table and assigns routes for vehicles. The coordination of agents mimics the process of bees foraging. Although these Ant-colony methods have achieved promising results, they did not perform well when it comes to a more realistic, complex and dynamic transportation system and lack the ability to deal with unexpected events instantly.

2.4 Summary of Limitations

As the investigated research problem in this thesis is how to efficiently reroute vehicles in order to significantly reduce non-recurrent congestions in urban areas, the limitations of the discussed related works, with regard to this problem, are summarised as follows:

- Limitations of shortest path algorithm: The information access and rerouting feedback process should be completed rapidly in order to respond to the constantly updated urban transportation network, due to the limited capability of existing techniques, shortest path algorithms struggle to deal with large network maps as their computing time could take too long time. Besides, they also lack the ability to self-adapt to the different situations in real urban transportation networks.
- Limitations of traffic management system: In general, traffic control focuses on reducing the waiting time at intersections, which is not directly correlated to minimising the total travel time. Moreover, the route choice for the whole trip is not always available while driving, especially when driving on a long trip or in unfamiliar areas. Additionally, vehicle-to-vehicle communication is not reliable when exchanging relatively long messages such as route choice information in real-time.

- Limitations of vehicle route optimisation: Current approaches of vehicle route optimisation are lack ability to self-evolve and adapt to highly complex traffic network. Besides, most studies focus on reducing travel time, however focusing on reducing vehicle emissions is important to achieve a more sustainable urban transportation network.

Traffic optimisation can potentially be more efficient if it combines with an intelligent vehicle navigation system in a complex traffic network via deep reinforcement learning methods. After identifying the challenges of traffic optimisation and discussing the strengths and weaknesses of the related works, the aforementioned limitations are addressed in the following chapters by the proposed framework that uses deep reinforcement learning for real-time vehicle route optimisation. In general, the proposed framework tends to navigate vehicle to the less congested road based on the observation of the vehicle agent. The vehicle agent is able to continuously learn the complex traffic patterns by receiving the reward. Therefore, it fits the rigorous real-time requirement of reducing non-recurrent congestions. Moreover, it also avoids complex and error-prone coordination mechanisms among vehicles by considering each junction and its controlled roads. Finally, NRR increases the practicability of research in reducing non-recurrent urban traffic congestion via the deep reinforcement learning method.

Chapter 3

Deep Reinforcement Learning

3.1 Overview

This sections covers the main technical concepts of Deep Reinforcement Learning (DRL) which is mainly to be used in this thesis in order to build the proposed framework for real-time vehicle route optimisation. The first section presents Reinforcement Learning which is essential for dynamic complex problems and its most popular technique, so called Q-Learning. The second section introduces the concept of Deep Learning and its impact nowadays. The third sections describes Deep Reinforcement Learning and the state of-the-art methods for Deep Q-Learning. Then the next section describe the improvement method of DQN. The last section describes the motivation to use DRL for the vehicle route optimisation problem and presents recent studies of urban transportation network.

3.2 Reinforcement Learning

A reinforcement learning (RL) method is able to gain knowledge or improve the performance by interacting with the environment itself. The theory of reinforcement learning is inspired by psychology that focuses on learning behaviour from rewards [73]. The reward is the positive or negative feedback based on the interaction of an artificial agent who executes an action and its environment. The goal of the agent is to learn which actions will lead to the highest reward in long run.

3.2.1 Markov Decision Process (MDP)

A Markov Decision Process (MDP) is a mathematical framework for optimizing decision-making under uncertainty. It is specified over an environment, where the goal is for an agent to reach some desired state. As such, the MDP formalizes a set of environmental states, a set of actions for the agent to take, a reward function that assigns a reward signal to the outcome of taking certain actions in certain states, and a transition function, that describes the change in the environment as a result of taking a certain action in a certain state. An MDP satisfies the Markov Property if the transition function depends only on the current state s and the taken action a. That is, the probability of moving from s to s' after taking a is dependent only on the current state. In mathematical terms, a state St has the Markov property, if and only if:

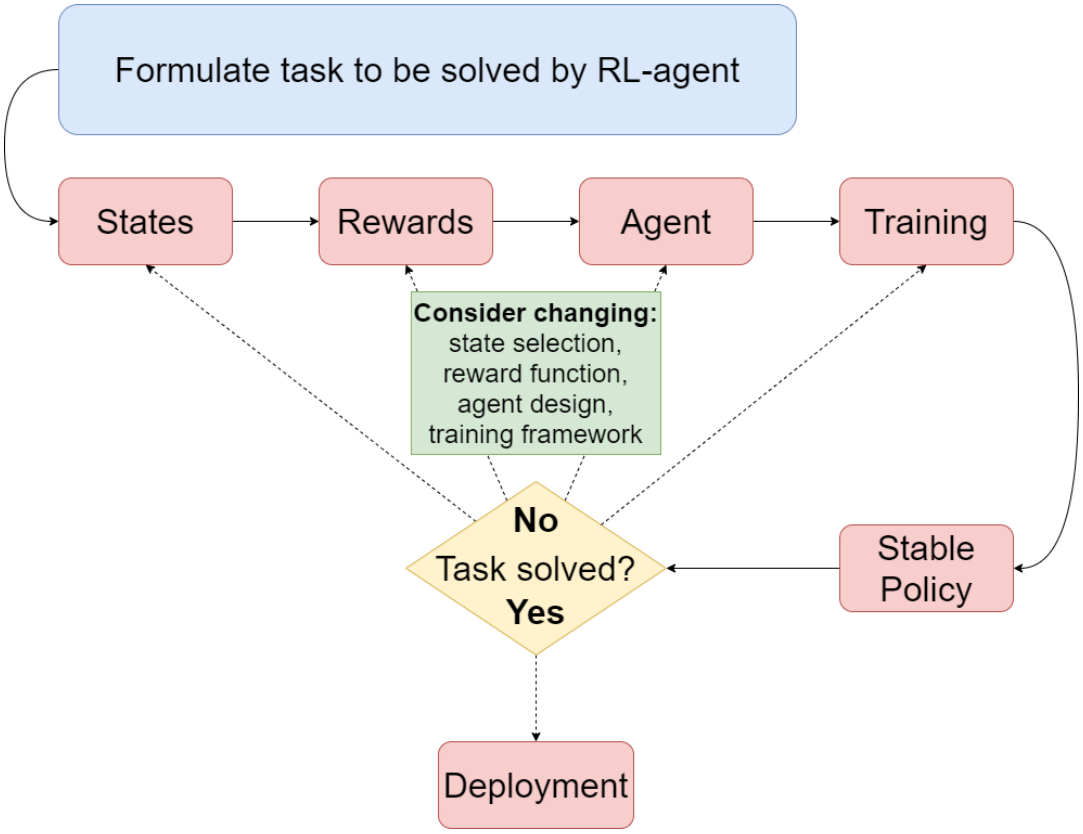


FIGURE 3.1: Reinforcement Learning Design Flow

$$P(s_{t+1}|s_t, a_t, r_t, s_{t-1}, a_{t-1}, r_{t-1}, ..., s_0, a_0, r_0) = P(s_{t+1}|s_t, a_t) \tag{3.1}$$

Formally, an MDP is a four-tuple < s, a, r, t > where

- s is the space of possible states
- a is the space of possible actions
- $r^a_{ss'}$ is the a reward function specifying the reward r for taking action a in state s and ending up in state s'
- $t^a_{ss'}$ is a transition function specifying the probability of taking action a in state s and ending up in state s'

The agent's goal is to maximize its reward over time, giving slightly more preference to short-term than to long-term reward. This goal is captured in the return, the discounted cumulative reward over time:

$$r_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{3.2}$$

where γ is a discount factor such that $0 < \gamma \leq 1$, meaning that future rewards are discounted exponentially.

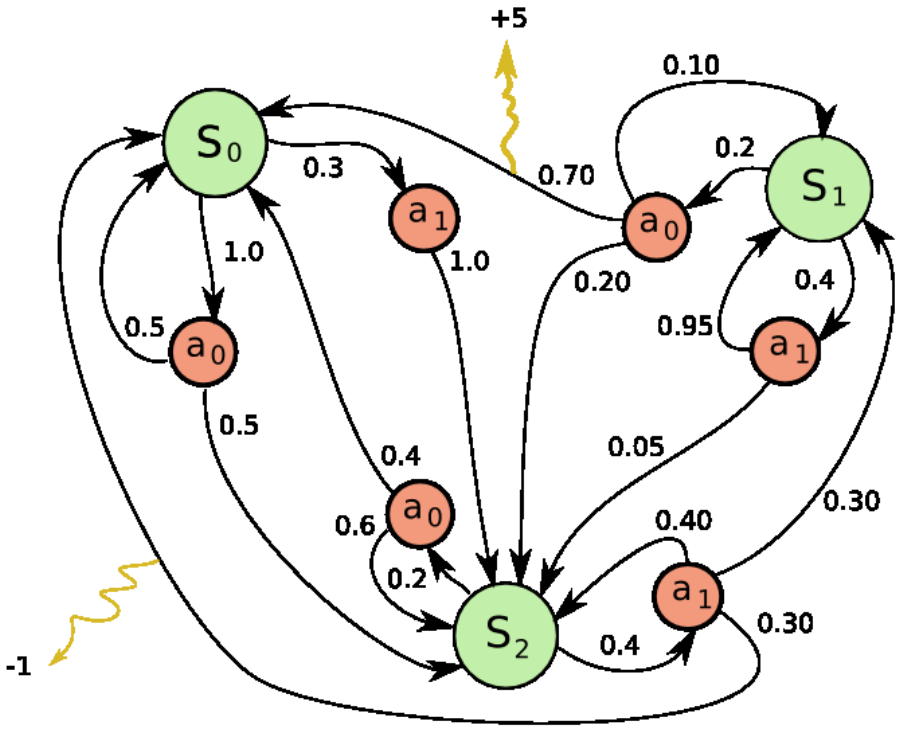


FIGURE 3.2: A Markov decision Process

The Figure 3.2 illustrates an example of the Markov Decision Process. The Markov Decision Process can be solved by Value Iteration (VI) which is an algorithm that finds the optimal value function (the expected discounted future reward of being in a state and behaving optimally from it), and consequentially the optimal policy. The central idea of the Value Iteration algorithm is the Bellman Equation, which states that the optimal value of a state is the value of the action with the maximum expected discounted future return (the action with maximum Q-value). And the Q-value for a state-action pair is defined as the expected value over all possible state transitions of the immediate reward summed with the discounted value of the resulting state. The formula is shown below:

$$V(s) = maxQ(s,a) \tag{3.3}$$

$$Q(s,a) = \sum_{s'} T(s^{'}|s,a)[R(s,a,s^{'}) + \gamma V(s^{'})] \tag{3.4}$$

In the case of Value Iteration, Bellman updates are performed in entire sweeps of the state space. That is, at the start, the value of all states is initialized to some arbitrary value. Then, the Bellman Equation updates the value function [28] estimate sweeping over the entire state space. These steps are repeated for

some fixed number of iterations or when the maximum change in the value function is small. The pseudocode of VI is shown in Algorithm 5.

Algorithm 5: Value Iteration

Initialize value function $V(s)$ arbitrarily for all state s. Repeat until convergence **foreach** *state* s **do**

  $V(s) = max \sum_{s'} T(s^{'}|s,a)[R(s,a,s^{'}) + \gamma V(s^{'})]$

end

The Value Iteration is as a planning algorithm that makes use of the Bellman Equation to estimate the Value function. However, if the probabilities or reward function is unknown, which is common in real systems, Value Iteration algorithms can no longer be computed. The root cause of this problem is that the planning algorithm need the access to a model of the world or at least a simulator. The other drawback of VI is that when state space is large or infinite, which may exceed the capability of modern computer.

3.2.2 Q-Learning

Unlike a planning algorithm, a learning algorithm like Q-learning [4] involves determining behaviour when the agent does not know how the world works and can learn how to behave from direct experience with the world. Figure 3.3 illustrates a typical example of how the agent interacts with the environment. Unlike the planning algorithm, the Learning agent has no predefined knowledge of the environment, which means the reward function and the transition function are unknown. Instead, the agent learns how to behave by interacting with the environment. As the name suggests, Q-learning estimates the optimal Q-values of a Markov Decision Process, which means that behaviour can be learned by taking actions greedily with respect to the learned Q-values. In the Q-learning algorithm, the most common way to choose an action in the current world state(s) is to use the greedy policy. ϵ is a fraction between 0 and 1. Based on the policy, the agent randomly selects among all actions a fraction of time, whereas the action with respect to the Q-value estimates a fraction of (1 - ϵ) time. The update rule for Q-learning is below.

$$Q(s,a) = Q(s,a) + \alpha[R(s,,a,s^{'}) + \gamma maxQ(s^{'},a^{'}) - Q(s,a)] \tag{3.5}$$

The Q-value is updated by the Q-value of the last state-action pair (s,a) with respect to the observed outcome state $s^{'}$ and direct reward $R(s,a,s^{'})$. The parameter between 0 and 1 stands for the learning rate.

The difference of update rules between Value Iteration and Q-learning algorithm is that the Q value of a state in VI is the maximum Q-value which is the expected sum of reward and discounted value of the next state, whereas the Q-value of Q-learning algorithm is the sum of rewards and discounted max

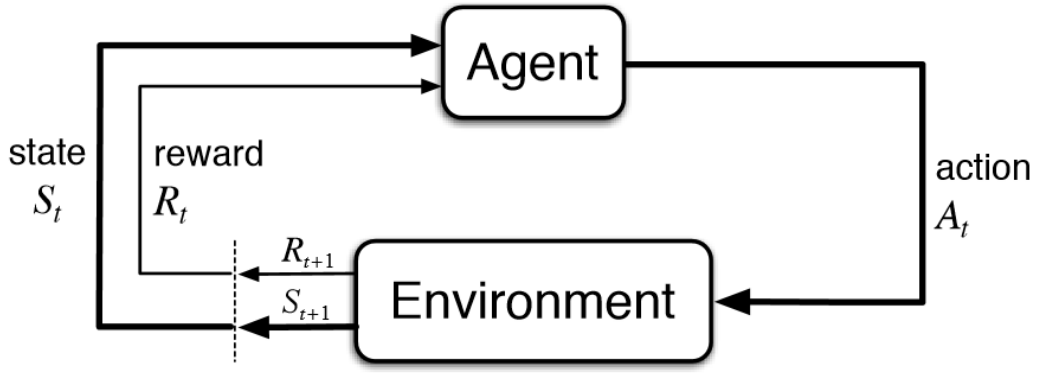


FIGURE 3.3: The interaction between environment and agent in Q-Learning

Q-value of the observed next state, which implies that we only use the states and rewards we happen to get by interacting with the environment. As long as we keep trying random actions on the same state, we could reach all possible states of next. After multiple times of aggregation, we should finally move close to the true Q-value. In order to have guaranteed convergence, some tips for the parameter setting could be very useful in practice. Firstly, the greedy policy should anneal linearly from 1.0 to a small fraction, for instance 0.1, over certain training steps, and fixed at the small fraction thereafter. This setting enables the agent to explore more action-state pairs at the beginning of the training, and reduce the randomization when the agent gains more experience. The other trick is slowly decreasing the learning rate α over time. The Q-learning algorithm can be summarized in the following pseudocode.

Algorithm 6: Value Iteration

1 Initialize Q-values $Q(s, a)$ arbitrarily for all state-action pairs. **while** *the learning is not terminated* **do**

2 Choose an action a in the current world state s based on current Q-value estimates Take an action a and observe the outcome state $s^{'}$ and reward $R(s, a, s^{'})$ Update $Q(s, a) = Q(s, a) + \alpha[R(s, , a, s^{'}) + \gamma maxQ(s^{'}, a^{'}) - Q(s, a)]$

3 **end**

As stated above, the basic idea of Q-learning is to estimate the action-value function by using the Bellman Equation as an iterative update. In that case, the value function converges to optimal the value function as the iteration i tends to infinity. However, it is impractical since action value function is estimated separately for each sequence without any generalization. Instead, it is common to use a function approximator to estimate the action-value function. The other problem is that traditional reinforcement learning algorithms heavily rely on the quality of handcrafted feature representations, which limits the application scope of these algorithms. There is no double that we could benefit more if features can be directly extracted from raw high-dimensional sensory inputs, for instance, the human-like visual and auditory information.

3.3 Deep Reinforcement Learning

The curse of dimensionality given by large state and action spaces make unfeasible to learn Q value estimates for each state and action pair independently as in normal tabular Q-Learning. Therefore, Deep Reinforcement Learning (DRL) models the components of RL with deep neural networks. The parameters of these networks are trained by gradient descent to minimize some suitable loss function.

3.3.1 Artificial Neural Networks

Artificial Neural Networks (ANN) was first designed in 1940s to mimic human brains neurons for learning from experiences. Since then, there have been various notable advances including the unsupervised learning, backpropagation, convolutional neural networks (CNN), recurrent neural networks (RNN), etc. The most general neural network structure is the Feed-forward neural network, also known as a Multilayer perceptron (MLP). A neural network consists of multiple artificial neurons that are connected in layers which can be divided into the input layer, hidden layer(s) and the output layer. It is a machine learning model parameterised by a set of parameters θ which maps the N-dimensional input layer $x = (x_1, x_2, ..., x_n)$, through the hidden layer(s) with activations, to a K-dimensional output layer $y = (y_1, y_2, ..., y_n)$. A hidden layer is composed by a set of artificial neurons, an artificial neuron is a mathematical construction that aims to mimic how neurons act in the human brain. As illustrated in Figure 3.4, an artificial neuron takes a number of weighted inputs $(w_1x_1, w_2x_2, ..., w_nx_n)$, sums them up with a bias term b and applies a non-linear activation function g to the sum. The single output y could be calculated by the following equation Equation 3.6:

$$y = g\left(\sum_{i=1}^{n} w_i x_i + b\right) \tag{3.6}$$

Activation functions play a key role in ANN. It generally is differentiable and non-linear which is applied by the neurons in the hidden layers can differ between networks and even between the layers within a single network. The three most well-known activation functions are the Logistic Sigmoid, the Hyperbolic tangent and Rectifier Linear Unit (ReLU). The function curves of these three activations are shown in Figure 3.5. There are no specific ways so far to determine which activation function is the best approach for ANN. Different application may get better performance in different activation functions. Therefore it is always dependent on trial and error. However, generally the ReLU activation was applied with more stable performance due to its positive impact on the different machine learning tasks it has been the default first choice when designing an ANN [91]. The ReLu activation function equation is:

$$relu(x) = max(0, x) \tag{3.7}$$

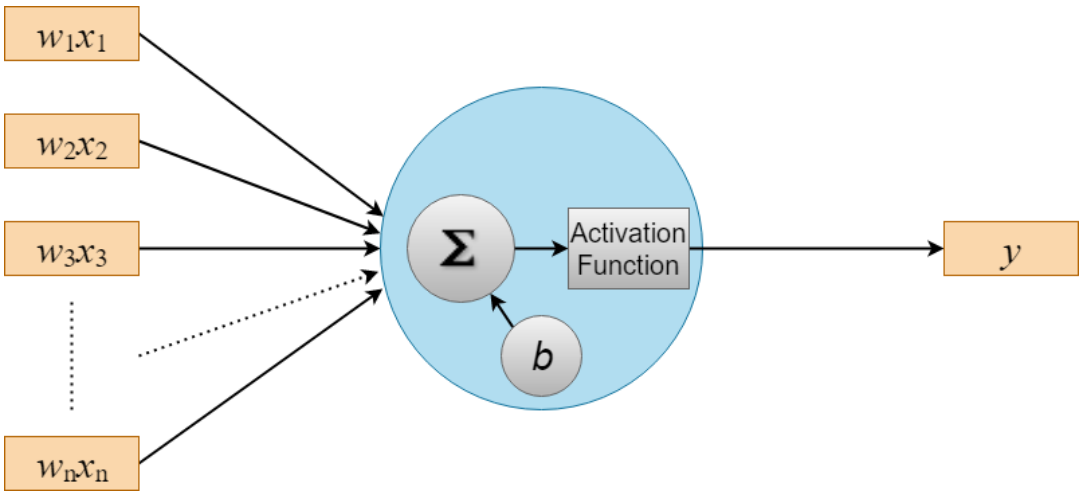


FIGURE 3.4: Single neuron output based on weight, input, bias and non-linear activation function

which contains the gradient:

$$\frac{d}{dx}relu(x) = \left\{ \begin{array}{ll} 0, & \text{if } x \leq 0 \\ 1, & \text{if } x > 0 \end{array} \right\}$$

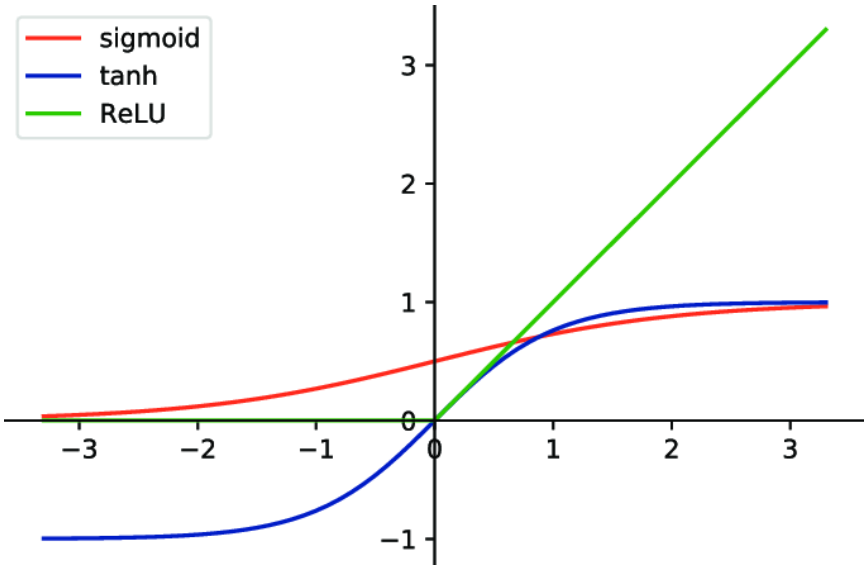


FIGURE 3.5: Function curves of sigmoid, Tanh and ReLU

In a multiple layers neural network, mapping the input vector $(x_1, x_2, ..., x_n)$ through the first hidden layer $h^{(0)}$ with weights $w^{(0)} \in \theta$, bias $b^{(0)} \in \theta$ and non-linear activation function $g^{(0)}$ results in the following equation Equation 3.8:

$$h^{(0)} = g^{(0)}\left(W^{(0)}x + b^{(0)}\right) \tag{3.8}$$

The output $h^{(0)}$ from Equation 3.8 can be used as input to the next layer, for example weights $w^{(0)} \in \theta$, bias $b^{(0)} \in \theta$ and non-linear activation function $g^{(0)}$ results in the following equation Equation 3.9:

$$h^{(1)} = g^{(1)}\left(W^{(1)}h^{(0)} + b^{(1)}\right) \tag{3.9}$$

$$h^{(1)} = g^{(1)}\left(W^{(1)}g^{(0)}\left(W^{(0)}x + b^{(0)}\right) + b^{(1)}\right) \tag{3.10}$$

And so on until forward the activation of the kth layer to the output layer as illustrated in Figure 3.6. Setting up the weights between the neurons can adapt how the input will be transformed by the neurons between the input and output in order to approximate the function $y = f(x;\theta)$. As the network grows deeper, the model can approximate more complex functions, but it also becomes harder to train. Therefore, much of the field of deep learning is trying to find more reliable and faster methods of training ANN.

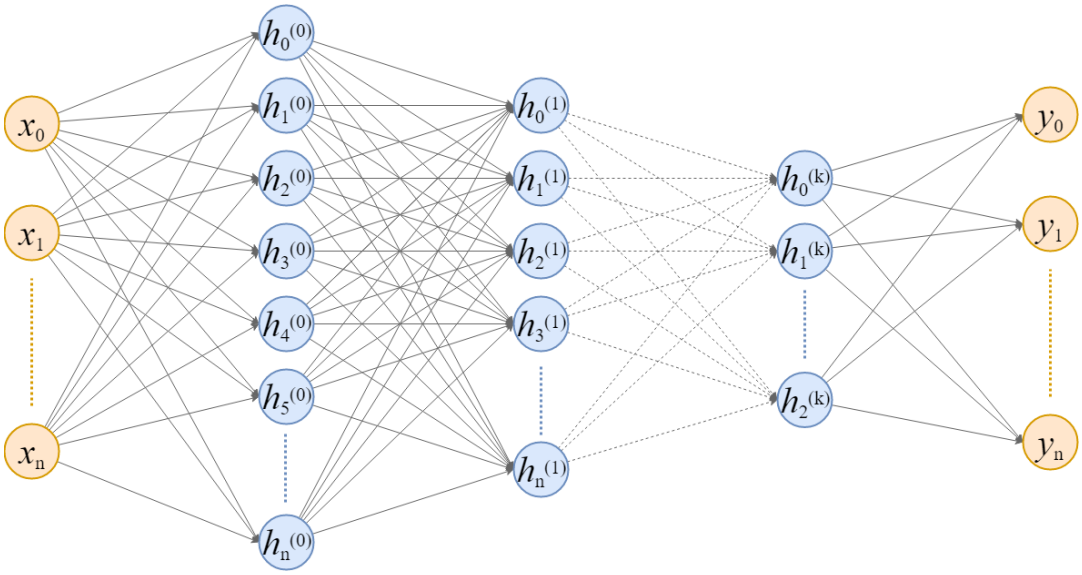


FIGURE 3.6: Visualization of the structure of neural network with multiple hidden layers

3.3.2 Action Selection Policy

The deep reinforcement learning algorithm does not specify what the agent should actually do. The agent learns the policy that can be used to determine an optimal action. There are two things that are useful for the agent to do:

- **Exploration:** For building a better estimate of the optimal policy. the agent should select a different action from the one that it currently thinks is best.
- **Exploitation:** For using the knowledge that it has found for the current state s by doing one of the actions a that maximizes the reward.

In DRL, exploration is due to randomness (or partly), therefore it also means the result always have huge difference in each run, however the experience from exploration could be very useful for training data set in DRL.

However, exploitation is based on policy, one of the simplest policies is the greedy policy, where the agent always chooses the action with the maximum expected return. The equation for greedy policy is shown below:

$$a = argmaxQ(A) \tag{3.11}$$

where $A \in a_1, a_2, ..., a_n$ is the collection of actions in particular state.

In order to add some exploration into the mix, the ϵ-greedy is stepped in. ϵ-greedy lets you decide what fraction of your decisions you want to spend exploring (ϵ) and what fraction you want to spend exploiting (1-ϵ)the best option so far. For instance, if ϵ is set as 0.4, then the RL agent will take the option that gave him the best average reward in the past 60% of the time and chose any other option 40% of the time. Typically, you want ϵ to be small so that you mostly exploit your experience, but also go explore occasionally from time to time. However, when a model just begin to start the learning process, ϵ will be set to a large number to encourage initial exploration and reduce it as you gather knowledge about the rewards.

3.3.3 Optimisation Algorithm

Essentially, training an ANN in order to approximate function $y = f(x; \theta)$ is changing the weights appropriately. To that end, some form of gradient descent is necessary. Firstly, a loss or cost function that quantifies the errors between the output of the neural network and the ground truth is defined. The objective is to minimise this quantity. The most common way to measure this is to calculate the mean squared error between the approximation and ground truth. The Mean Square Error (MSE) [26] is formally defined by the following equation:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(f(x;\theta) - \hat{y}_i)^2 \tag{3.12}$$

Backpropagation Neural networks can be trained using gradient descent methods - by minimizing the error function with respect to the parameters. To do so, the gradient of the error function is computed. Backpropagation is a method for passing the error in the output layer back through the individual nodes in the neural network. Since a neural network is essentially a hierarchy of nested functions, the chain rule can be used to compute the derivative of the error function with respect to the neural network weights.

Adagrad [20] proposed Adagrad which is a method which adaptively updates parameters based on a sum of squared gradients per parameter. It uses that value to normalize the learning rate before the update for each parameter i with the formula:

$$G_j^t = G_j^{(t-1)} + \left(\frac{\delta \ell}{\delta \theta_j^{(t-1)}} \right)^2 \tag{3.13}$$

$$\theta_j^t = \theta_j^{(t-1)} - \frac{\alpha}{G_j^t + c} \cdot \frac{\delta \ell_\theta}{\delta \theta_j^t} \tag{3.14}$$

where c is a small constant to prevent division by zero. The learning rate for each parameter is set adaptively based on past updates. If past gradients for parameter i were large, the learning rate for i is small and vice-versa. By dividing the learning rate by the sum of past square gradients, Adagrad removes the need for extensive learning rate tuning. Adagrad solved the problem of adaptively tuning the learning rate per parameter, but by dividing the learning rate by the sum of squared gradients, the learning rate diminishes too aggressively as time passes, since the sum keeps growing.

RMSProp has been developed independently from the need to solve Adagrads aggressive diminishing learning rates. [21] proposed RMSProp in order to solve that problem by defining an exponentially decaying average of squared gradients.

$$G_j^t = \gamma G_j^{(t-1)} + (1-\gamma) \left(\frac{\delta \ell}{\delta \theta_j^{(t-1)}} \right)^2 \tag{3.15}$$

where originally $\gamma = 0.9$.

Momemtum is an addition to the optimisation step that functions by increasing the strength of updates in directions that consistently lead to improvement. It does this by storing a variable v, the so-called velocity:

$$v^t = \mu v^{(t-1)} - \alpha \ell \tag{3.16}$$

where μ is the momentum coefficient. By using momentum, learning speeds up when gradients are following the loss curve down a slope.

ADAM [22] developed the Adaptive Moment Estimation (ADAM) algorithm by combining Adagrad and RMSProp with a new implementation of Momemtum. It uses a decaying average of squared gradients and a decaying average of past gradients:

$$m^t = \beta_1 m^{(t-1)}) + (1-\beta_1) \nabla \ell \theta \tag{3.17}$$

$$m^t = \beta_2 v^{(t-1)} + (1-\beta_2) \nabla \ell \theta^2 \tag{3.18}$$

where mt and vt are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively. β_1 and β_1 are hyper-parameters. Because mt and vt are initialized as zero vectors, this causes

a bias towards zero, especially during the initial time steps, and especially when the decay rates are small due to β_1 and β_2 are close to 1. In order to solve this issue, the first and second moment estimates are bias-corrected with:

$$\hat{m}^t = \frac{m^{(t-1)}}{1-\beta_1^{(t-1)}} \tag{3.19}$$

$$\hat{v}^t = \frac{v^{(t-1)}}{1-\beta_2^{(t-1)}} \tag{3.20}$$

and the final updating being:

$$\theta^t = \theta^{(t-1)} - \frac{\alpha \hat{m}^t}{\sqrt{\hat{v}^{(t)}+\epsilon}} \tag{3.21}$$

3.3.4 Deep Q-Network

[25] proposed Deep Q-Networks (DQN) as a technique to combine Q-Learning with deep neural networks where such technique proved to achieve super-human performance in several Atari Games. This benchmark has become in the most common one. Reinforcement learning is known to be unstable or even to diverge when a non-linear function approximator such as a neural network is used to represent the Q value. DQN addresses these instabilities by using two insights, experience replay and target network.

The experience replay has three main advantages. Firstly, it allows greater data efficiency because each step of experience is potentially used in many weight updates. Secondly, learning directly from consecutive samples is inefficient, due to the correlations between the samples; therefore, by randomizing the samples these correlations can be broken and the variance of the updates can be reduced. Thirdly, when learning on policy the current parameters determine the next data sample that the parameters are trained on. By using this technique the behaviour distribution is averaged over many of the prior states, stabilising the learning and avoiding fluctuations or divergence in the parameters.

The other technique which improves the stability of neural networks is to use a separate network to generating the targets yi during the Q-learning update. Specifically, every C updates the network Q is cloned in order to obtain a target network Q and use Q for generating the Q-learning targets yi for the following C updates to Q. By using this generations with older set of parameters it allows a delay between when an update is done to Q and when that update affects the targets yi which makes unlikely the presence of divergences or oscillations.

DQN parameterises an approximate value function $Q(s, a; \theta_i)$ ANN, where θ_i are the weights of the network at iteration i. The experience replay stores the agent's experiences $e_t = (s_t, a_t, r_t, s_{t+1})$ at each time step t in a dataset $D_t = e_t$.

e_t pooled over many episodes into a replay memory. Then, mini batches of experience drawn uniformly at random from the dataset D are applied as Q-updates during the training. The Q-learning update at iteration i follows the loss function:

$$L_i(\theta_i) = E_{(s,a,r,s')} \sim D \left(\underbrace{r + \gamma max Q(s^{'}, a^{'}; \theta_i^{-})}_{target} - Q(s, a; \theta_i) \right)^2 \tag{3.22}$$

where θ_i are the parameters of the Q-network at iteration i and θ_i^{-} are the target network parameters. The target network parameters are only updated with the Q-network parameters every C steps and are held fixed between individual updates. The Figure 3.7 illustrate the learning process of DQN and the Algorithm 7 states the procedure.

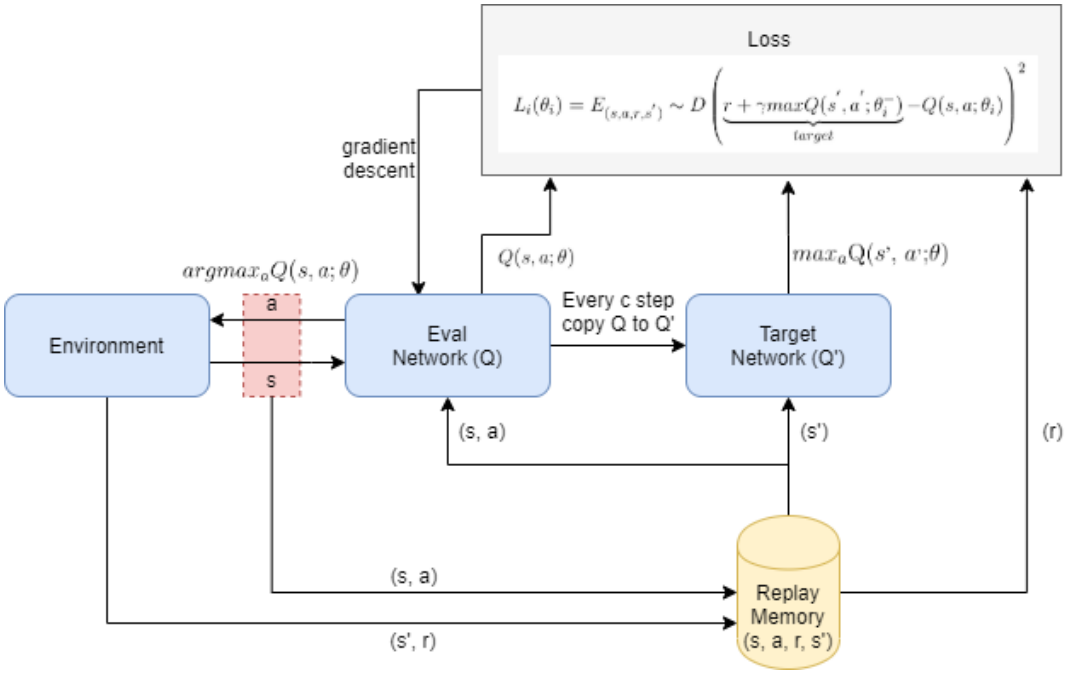


FIGURE 3.7: DQN Learning Diagram

Algorithm 7: Deep Q-Learning with memory replay

initialise replay memory D to capacity N
initialise action-value function Q with θ
initialise target-value function $\bar{Q}$ with $\theta^{-} = \theta$
for *episode=1, M* **do**
  for *t=1, T* **do**
    with probability ϵ select a random action a_t
    otherwise select $a_t = argmax_a Q(s, a; \theta)$
    execute action a, observe reward r and next state s'
    store transition (s, a, r, s') in D
    sample random minibatch of transitions from D
    if s'_j *is terminal* **then**
      $y_j \leftarrow r_j$
    else
      $y_j \leftarrow r_j + \gamma max_a \mathrm{Q}(s'_j, a'; \theta)$
    end
    perform a gradient descent step on $(y_j\text{-}\mathrm{Q}(s_j, a_j; \theta))^2$ with respect to the network parameters θ
    Every C steps reset $\bar{Q} \leftarrow$ Q
  end
end

Where D is the replay memory for each transition, N is the maximum number of the storage of D, Q and $\bar{Q}$ are action and target network which are with θ and θ^{-} as parameters.

3.4 Deep Reinforcement Learning for Urban Traffic Optimisation

With the rapid development and recent success of machine learning technologies lately, Deep RL techniques have demonstrated their ability to tackle a wide range of problems that were previously unsolved. Some of the most well-known achievements include attaining superhuman-level performance in playing ATARI games from the pixels, beating professional Go and Poker players etc. These achievements in popular games are important mainly because they show the potential of deep RL in a wide variety of complex and diverse tasks that require working from high-dimensional inputs. In fact, deep RL has also already shown lots of potential for real-world applications such as robotics, self-driving cars, online marketing, face-recognition etc. Virtual environments such as the ones developed by OpenAI [13] and DeepMind [8] will certainly allow to explore even further the limits of the deep RL algorithms. Therefore, many researches started to focus on solving traffic congestion problem based on the deep reinforcement learning method. M.G Karlaftis conducted an overview comparing statistical methods with neural networks in transportation related research and it demonstrated that solutions based on deep reinforcement learning are very promising [55]. Despite its potential, to the best of my knowledge, there is no

study relating vehicle route optimisation via the deep reinforcement learning method. Although there are some researches works focused on path planning, using BP neural network and fuzzy neural network learning method [84, 120], however, these approaches lacked the ability to deal with unexpected events such as road accidents. Currently, most of the studies focused on urban traffic prediction and intersection traffic control. This section introduces the previous works based on deep reinforcement learning in these two major categories.

3.4.1 Intersection Traffic Control

Various researches attempted to handle the traffic at the intersections by such methods as controlling the traffic light signal [92, 79, 37] and navigating vehicles at occluded intersections [50, 49, 51]. The first approach to the problem of traffic control using reinforcement learning was made by Thorpe and Anderson [106] in 1996. The state of the system was characterised by the number and positions of vehicles in the north, south, east and west lanes approaching the intersection. The actions consist of allowing either the vehicles on the north-south axis to pass, or those on East-West axis to pass. And the goal state is when the number of waiting cars is 0. A neural network was used to predict the waiting time of cars around a junction. In 2003, Abdulhai et al [2] showed that the use of reinforcement learning, especially the use of Q-learning is a promising approach to solve the urban traffic control problem. The state includes the duration of each phase and the queue lengths on the roads around the crossing. The actions were either extending the current phase of the traffic light or changing to the next phase. This approach has shown a good performance when used to control isolated traffic lights. An example of intersection traffic light control is illustrated in Figure 3.8, the agent observe the road traffic as the state and give the action accordingly based on the output of the neural network. Then the action will affect the road traffic and a reward could be retrieved. After that, the neural network will be trained based on the reward and improve the traffic light control policy.

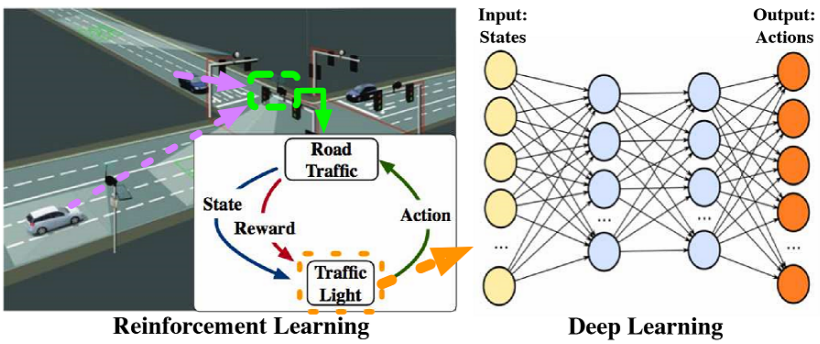


FIGURE 3.8: The traffic light control model in deep learning [21])

M. Weiring used multi-agent reinforcement learning to control a system of junctions [116]. A model-based reinforcement learning is used in this approach, and the system is modelled in its microscopic representation. They counted the frequency of every possible transition, and the sum of received rewards, corresponding to a specific action taken. Then, a maximum likelihood model is used for the estimation. The set of states was car-based: it included the road where the car is, its direction, its position in a queue, and its destination address. The goal was to minimise the total waiting time of all cars at each intersection, at every time step. An optimal control strategy for area traffic control can be obtained from this approach.

3.4.2 Urban Traffic Prediction

Most research in this area applies deep learning for traffic prediction [71, 93] or accident prediction [96, 104] in order to detect traffic congestions in advance. For traffic prediction, Yisheng Lv proposed a deep learning based traffic flow prediction method by using stacked autoencoder model to learn generic traffic flow features [71]. N. Polson also presented a deep learning predictor for spatial-temporal relations present in traffic speed measurements. It focuses on forecasting traffic flows that occur unexpectedly and hardly predictable such as special events or extreme weather [93]. Besides, Honglei Ren collected big traffic accident data and and analyzed the spatial and temporal patterns of traffic accident frequency for an accident risk predictor [96]. And FangZhou Sun proposed a deep neural network DxNAT to identiy non-recurring traffic congestion by converting traffic data in Traffic Message Channel (TMC) format to image, and use a convolutional neural network (CNN) to identify non-recurring traffic anomalies. Although these research studies showed great potential in traffic optimisation, they only solve half of the problem as drivers still need smarter navigation system to guide their vehicles to destinations based on the predicted results in order to optimise the traffic.

3.4.3 Motivation of using DQN in Vehicle Navigation

This subsection presents the motivation to apply Deep Q-learning in vehicle navigation for this research. In vehicle navigation, the driver often depend on the their driving experience (policy) to find out which way is faster. However, the policy could be optimised if the driver know better what to explore. As the DRL model been trained, based on that the driver therefore is able to know about the ground truth values of states and actions. In order to achieve that, DQN method contains several advantages in vehicle navigation system that show below:

- **Experience Replay:** During DQN training, thousands even millions transitions are stored into a buffer and sample a mini-batch of samples of size 32 from this buffer to train the deep neural network. This forms an input dataset which is stable enough for training. As the training process

randomly sample from the replay buffer, the data is more independent of each other closer to independent and identically distributed.

- **Target network:** DQN uses two deep networks during the learning process. The first one is called evaluate network which is to retrieve Q values while the second one is called target network which includes all updates in the training. After certain updates, the parameters in target network will be synchronised to evaluate network. The purpose is to fix the Q-value targets temporarily so we don't have a moving target to chase. In addition, parameter changes do not impact the evaluate network immediately and therefore even the input may not be entirely independent and identically distributed, it will not incorrectly magnify its effect.

With both experience replay and the target network in DQN, the learning process will have a more stable input and output to train the network and behaves more like supervised training for vehicle navigation.

3.5 Limitation

This section discusses the limitations of deep reinforcement learning for urban traffic optimisation.

- Currently there is much less work in the literature devoted to determining the appropriate reaction for vehicle route optimisation to reduce congestion due to unexpected events via deep reinforcement learning.
- The existing research urban traffic control and prediction are insufficient to reduce traffic congestion.
- There are lack of previous work to apply DRL methods in vehicle navigation system.

Chapter 4

Urban Traffic Simulation

4.1 Overview

This chapter presents the introduction of the urban traffic simulation. Simulation is widely applied in research as, most of the time running experiments in the real world is simply not practical and could waste too many resources. Three main categories of traffic simulation (microscopic, mesoscopic and macroscopic) which are designed for different requirements are covered in this topic. Furthermore, this chapter introduces different traffic simulators and compares their performance in speed. Lastly, the simulator that is used in this thesis is briefly introduced with its additional features such as TraCI and vehicle emissions.

4.2 Background

Simulation is a popular approach in computer science for research in different scientific problems by simulating an artificial environment. It brings the advantage of allowing the assessment of system's behaviour before it is deployed or produced in real life. It is able to characterise system's performance, test scientific models in order to prove or disprove their feasibility and correctness without any real implementation. Facilitating the increasing processing power possessed by computers, simulation allows us to test the complex scientific models in a reasonable time with minimum cost. Simulation techniques are based in mathematical models, which can take into account responses and constraints of the system to be simulated appropriately [89]. If the system's simulation is appropriate, we can then provide practical feedback to real systems, time compression or expansion, higher control, and lower costs.

4.2.1 Traffic Simulation

One of the most common systems that is widely studied by the scientific community using computer simulation is traffic simulation. Although there are some traffic related architectures which are tested on real system, for instance, the research in [5] developed and tested a Transportation Regulation Support System (TRSS) prototype on the Brussels transportation network. However, these models need to consider the system complexity that in many cases is hard to capture and they cannot be easily solved by using common sense, simple calculation, analytical methods, and direct experiments. Therefore, the real scale

testing, validation and assessment of output are extremely complicated, risky and expensive [39]. Traffic simulator which is able to create virtual scenarios is widely used in transportation research because, as mentioned before, running experiments with vehicles in the real world is simply not practical.

Traffic simulation is inherently complex, usually composed of diverse entities such as vehicles, road networks etc that reflect real transportation behaviours. In such cases, complex mathematical analyses in traffic simulation could be used to deal with traffic as a whole, using flow equations to describe multiple vehicles movements. Moreover, because of the time compressing characteristics that condense information and create hypothetical situations, traffic simulation can be used as a training set for real systems, and it allows the scientific community to compare different studies between new infrastructures, and control without interfering in the real system and wasting the resources [89]. In addition to all the aforementioned advantages, traffic simulator is a suitable and cost effective alternative tool for the scientific community, especially to validate and assess the performance of urban transport system.

4.2.2 Simulation Models and Approaches

Generally, there are 3 main categories to simulate traffic system (microscopic, mesoscopic, and macroscopic) and 2 main approaches (space-discrete and space-continuous) [95]. The overview of the three traffic simulation models is shown in Figure 4.1. Macroscopic mainly focuses on the movement of platoons of vehicles aggregate level, it aggregates the description of traffic flow and the relationships of traffic characteristics (speed, flow and density) [12]. This model does not consider the behaviour of a single vehicle but only the general evaluation of traffic flows in a network, which means it handles every vehicle in the same way and as a group in simulation. Moreover, macroscopic models the flow of traffic using high-level mathematical models often derived from fluid dynamics. Consequently, it can be used to predict the spatial and temporal congestion that is caused by the traffic demand or incidents in a road network [97]. As the macroscopic model does not require detailed modelling, it is often used for regional transportation planning [72]. It could perform fast, accurate simulation which is useful for the simulation of wide-area traffic systems, such as motorway networks and interregional road networks. However, this approach is not well suited to urban models as in real urban transportation there are many different types of vehicle driven by different individuals who have their own styles and behaviours [89]. The example of macroscopic simulation includes statistical dispersion models [35], free-way traffic model [62] etc

The Microscopic model however simulates the characteristics and interactions between individual vehicles. Essentially, the simulation produces the trajectories of each single vehicle moving across the network and primarily focuses on individual vehicle speeds and locations [72]. In this case, route options which are playing important role in simulating traffic are considered as they are becoming increasingly more complex in modern transport systems. The

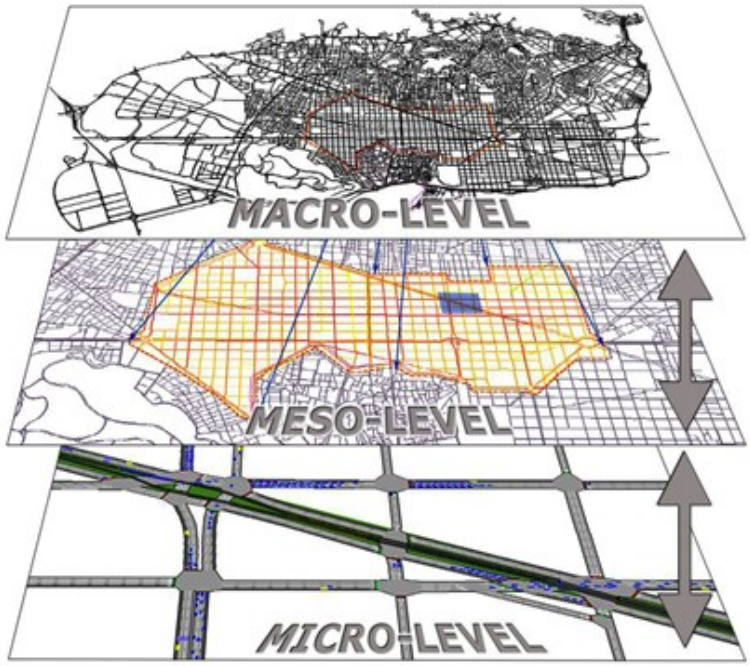


FIGURE 4.1: Overview of traffic simulation models [14]

simulation logic includes algorithms and rules that describe how vehicles move and interact. For instance, behaviour of individual vehicles in intersections is handled by car-following and lane-changing logics, which include acceleration, deceleration, lane changes and overtaking manoeuvres algorithms [97]. Therefore, microscopic can model traffic flow more realistically than macroscopic simulators do, due to the extra details added in modelling the simulated entities individually [89]. As they model individual entitles separately at a high level of detail, microscopic simulators are widely used to evaluate new traffic control and management technologies as well as performing analysis of existing traffic operations. Moreover, such microscopic models may be discrete in time and space using cellular automata or only discrete in time. Figure 4.2 shows the difference between space-discrete and space-continuous simulations. Although they are mainly applied to narrow-range transportation systems, with the increasing growth of computer processing power, microscopic simulations in big complex urban transportation become more common. Examples of microscopic simulation include cellular automata [102], multi-agent simulation [88], particle system simulation [98] etc.

In contrast, the mesoscopic model consists of the aspects of both macro and microscopic models. It fills the gap between the aggregate level approach of macroscopic models and the individual interactions of the microscopic ones by describing the traffic entities at a high level of detail, while their behaviour and interactions are designed at a lower level of detail [15]. In mesoscopic simulation, vehicles can be grouped in packets, which are routed throughout the network and are treated as one entity. Another paradigm is that of individual vehicles that are grouped into cells to control their behaviour. The cells traverse the link and vehicles can enter and leave cells when needed, but not overtake [89].

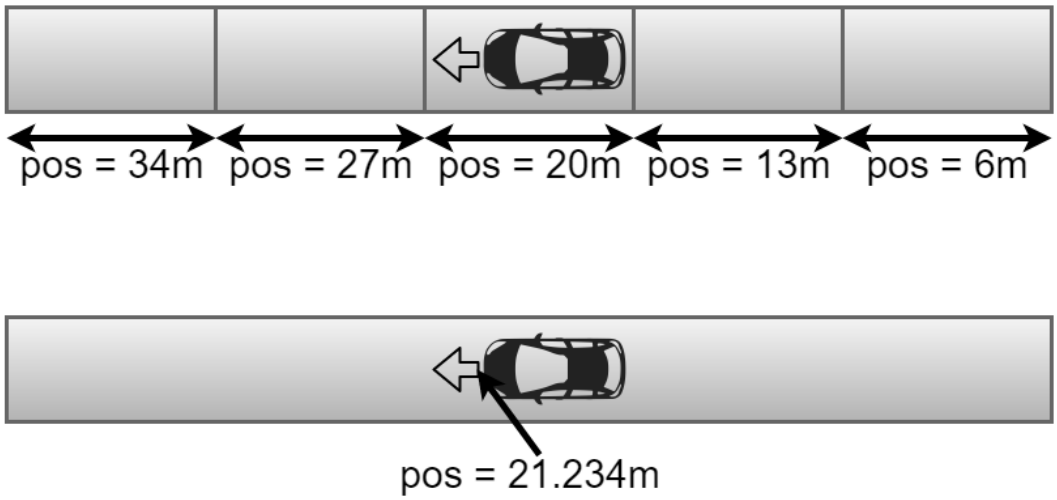


FIGURE 4.2: Space-discrete vs Space-continuous simulation

Mesoscopic simulators are useful for system wide evaluation of transit operations and Advanced Public Transportation Systems (APTS), as they are for general traffic [107].

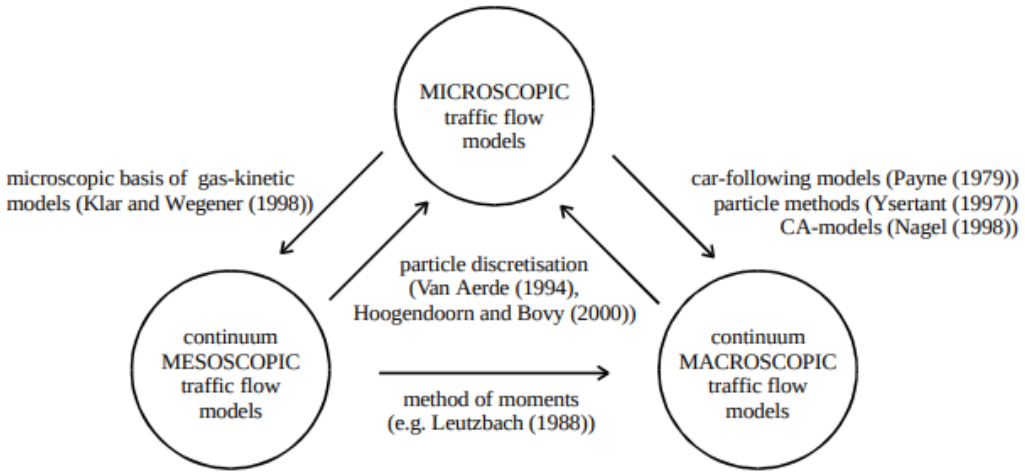


FIGURE 4.3: The relationship of traffic simulation models [15]

The research from [15] studied links between the three traffic simulation models. Figure 4.3 shows an overview of these relations. Klar and Wegener [57] describe a hierarchy of models: the authors present a simple microscopic flow model that is used to determine gas-kinetic flow equations. These are subsequently transformed into a mesoscopic traffic flow model. The derivation of the original Payne model [90] can be considered as an example of 'degeneration' of microscopic flow model to a macroscopic flow model. Nagel [82] shows the relation between CA-models and the simple wave model. Application of the method of moments (e.g. Leutzbach [64]) yields macroscopic equations from mesoscopic traffic flow models. Van Aerde [111], used a particle discretisation

	Sumo	**Paramics Modeller**	**Aimsun**	**SimTraffic**	**CORSIM TRAFVU**
CPU Usage	Between 5-17%, depending on the number of vehicles currently running on the traffic network	Constant 50%	Between 25-40%, depending on the number of vehicles and the scenario currently simulated	Constant 50%	Constant 50%
Memory Usage	Between 12-16 MB, depending on the traffic network	Between 40-140 MB, depending on the traffic network and the graphic models used	Between 30-40 MB, depending on the traffic network	Around 35 MB, does not depend much on the traffic network simulated	Between 28-32 MB, depending on the traffic network
Simulation Output	Included simulation output through generating output files.	Included tools to statistically represent what is happening in the simulated traffic network	Includes more than 20 different view styles for graphical representation of statistical information about the traffic and the events occurring in the ongoing simulation of the traffic network	Lack of information because demo version only reveals limited report	Did not include any type of output files.

TABLE 4.1: CPU and Memory performance in different traffic simulators [59]

method. Application of particle discretisation methods to derive microscopic models from gas-kinetic equations has recently been reported by Hoogendoorn and Bovy [45], who applied the method to gaskinetic equations describing pedestrian flows. Regarding the relation between microscopic and macroscopic traffic flow models, Del Castillo [18] proposes a car-following model, the three parameters of which can be determined directly from speed-density data.

4.2.3 Simulators Overview

There are several widely used traffic simulators, including Quadstone Paramics [16], VISSIM [32], AIMSUN [17], MATSIM [47] and SUMO [61]. These simulators provide different features and models for commercial and research purposes. G.Kotusevski et al. carried out a comprehensive comparison of these simulators with their features, characteristics and limitations [59]. Among them, SUMO comes with outstanding ability to simulate a very large and complex transportation network of up to 10,000 edges (roads).

G. Kotusevski and K.A. Hawick [59] reviewed different traffic simulators by using a machine with Intel Core 2 Duo Extreme processors running at 2.8GHz with 6MB of cache memory and 2GB of RAM memory. Table 4.1 provides information about the CPU and Memory performance of the software applications while they were actively simulating a traffic network.

In simulation, most of the simulators generally enables most precise modelling and simulation, with an emphasis on providing a high level of realism, concerning both a network and vehicles/drivers. However, such precision is at a price of low simulation speed [72]. In contrast, SUMO uses a space-continuous car following model, and enables simulation for large, even regional, networks but, at the same time, offers high speed simulation. Therefore, an alternative to these simulators is SUMO that can be considered as a reasonable compromise.

In our research, SUMO becomes an obvious option as the traffic simulator for DRL based vehicle navigation system. SUMO provides several advantages as shown below:

- **Open-source:** SUMO is an open-source software which means it is totally free and we are able to customise its features in order to apply DRL method.
- **Useful Output Files:** SUMO provides meaningful output files for each simulation. Those files contain useful data for DRL learning process.
- **Speed and Stability** SUMO provides stable processing speed while processing traffic simulation, even for a very large traffic networks (Up to 10000 edges).
- **API Available:** SUMO provides API for user to gain real time information from a running traffic simulation. This allows 3rd party libraries to integrate with the traffic simulation in real time.

4.3 Overview of SUMO

The German Aerospace Center (DLR) started the development of the open source traffic simulation package SUMO back in 2001. Since then SUMO has evolved into a full featured suite of traffic modeling utilities including a road network capable of reading different source formats, demand generation and routing utilities from various input sources (origin destination matrices, traffic counts, etc.), a high performance simulation usable for single junctions as well as whole cities including a "remote control" interface (TraCI) to adapt the simulation online.

SUMO was started to be implemented in 2001, with a first open source release in 2002. There are two reasons for making the work available as open source. The first is the wish to support the traffic simulation community with a free tool into which their own algorithms can be implemented. While there are some open source traffic simulations available, most of them have been implemented within a student thesis and got unsupported afterwards. A major drawback – besides reinventing the wheel – is the almost non-exist comparability of the implemented models or algorithms. A common simulation platform should be of benefit here. The second reason for making the simulation open source was the wish to gain support from other institutions.

SUMO is not only a traffic simulation, but rather a suite of applications which help to prepare and to perform the simulation of traffic. As the traffic simulation "SUMO" requires the representation of road networks and traffic demand to simulate in an own format, both have to be imported or generated using different sources. SUMO is a purely microscopic traffic simulation. Each vehicle's details are given explicitly, defined at least by an identifier (name), the departure time, and the vehicle's route through the network. If wanted, each vehicle can be described in more detailed. The departure and arrival properties, such as the lane to use, the velocity, or the position can be defined. Each vehicle can get a type assigned which describes the vehicle's physical properties and the variables of the used movement model. Each vehicle can also be assigned to one of the available pollutant or noise emission classes. Additional variables allow the definition of the vehicle's appearance within the simulation's graphical user interface.

As SUMO is an open-source, microscopic, multi-model traffic and extensible simulator, it has been widely used in research projects with a worldwide community support. It allows the user to simulate specific traffic scenarios performing in given road maps. There are several reasons that SUMO is used as the simulator in our experiments: (i) it performs an optimized traffic distribution method based on vehicle types or driver behaviors in order to maximize the capacity of the urban transportation network; (2) it updates the vehicle's route in real-time when congestion or accident occurs; and (iii) it supports TraCI, a Python based API that allows a user to get traffic observations from the traffic simulation and control the simulation as it runs in response to the observations.

4.4 Additional Features

SUMO also provides several additional features for different purposes, as the modern transportation network is become more and more diverse. Currently, there are 8 additional features in SUMO which are emission model, electric vehicle, logistics, generic parameters, shapes visualisation, wireless device detection, emergency vehicles and simple platooning. In this section only two them (Emission and emergency vehicles) are briefly introduced because it is highly related to this thesis.

4.4.1 TraCI

Another major feature of SUMO is TraCI , which is a Python based API that treats the SUMO simulator as a server. It allows the users to gain real time information from a running traffic simulation, and modify the simulation correspondingly. TraCI enables third party systems (or libraries) to integrate with the SUMO traffic simulation at runtime. In our trainings, TraCI will play the role of the communicator between SUMO and the RL agent to achieve this interaction. It is able to retrieve every piece of information about the vehicles and

road maps in the simulation and provide the useful features for the RL agent to justify the states of the environment.

In 2006, the simulation was extended by the possibility to interact with an external application via a socket connection. This API, called "TraCI" for "Traffic Control Interface" was implemented by Axel Wegener and his colleagues at the University of Lübeck [18], and was made available as a part of SUMO's official release. Within the iTETRIS project, see Section IV.B, this API was reworked, integrating it closer into SUMO's architecture.

To enable on-line interaction, SUMO has to be started with an additional option, which obtains the port number to listen to. After the simulation has been loaded, SUMO starts to listen on this port for an incoming connection. After being connected, the client is responsible for triggering simulation steps in SUMO as well as for closing down the connection what also forces the simulation to quit. The client can access values from almost all simulation artifacts, such as intersections, edges, lanes, traffic lights, inductive loops, and of course vehicles. The client may also change values, for example instantiate a new traffic light program, change a vehicle's velocity or force it to change a lane. This allows complex interaction such as online synchronization of traffic lights or modeling special behavior of individual vehicles.

TraCI is not the only contribution to SUMO from other parties. SUMO Traffic Modeler allows to define a population for a given area and compute this population's mobility wishes which can be used as an input for the traffic simulation. The same is done by "activitygen" written by Piotr Woznica and Walter Bamberger from TU Munich. eWorld allows to set up further environmental characteristics, such as weather condition and visualizes a running, connected simulation.

4.4.2 Emissions

Air pollution is one of the most serious problems in the world and transport accounted for one quarter of total emissions in the world. And the development of technical solutions for critical systems usually includes a step where the solution is modelled and simulated. Fortunately, SUMO provides the feature for vehicular emissions modelling. The emission model has been performed within the projects "COLOMBO" [60] and "AMITRAN" [52]. SUMO includes the following emission models:

- HBEFA v2.1-based: A continuous reformulation of the HBEFA v2.1 emissions data base (open source)
- HBEFA v3.1-based: A continuous reformulation of the HBEFA v3.1 emissions data base (open source)
- PHEMlight, a derivation of the original PHEM emission model (closed source, commercial).

Model	Pollutant / Measurement					
	CO_2	CO	HC	NO_X	PM_X	Fuel Consumption
HBEFA v2.1-based	x	x	x	x	x	x
HBEFA v3.1-based	x	x	x	x	x	x
PHEMlight	x	x	x	x	x	x

TABLE 4.2: Pollutants covered by models

As SUMO's goal is to simulate real-world traffic in large areas, the model are capable to be used as a further measurement within the simulator. Moreover, not all available models cover all pollutants emitted by road traffic. Therefore the pollutants assumed to be needed should be defined. SUMO model the emission of CO, CO2, NOx, PMx, and HC, because these emissions are toxic (CO), cause cancer (PMx), are responsible for ground-level ozone increase and smog generation (NOx and HC) or are greenhouse gases (CO2). Additionally, the fuel consumption should have been modelled. The Table 4.2 shows the pollutants covered by models.

Chapter 5

Preliminary Design and Experiment for Vehicle Route Optimisation

5.1 Overview

This chapter presents how a RL method could work with SUMO simulator for vehicle route optimisation. We run an experiment with 2 maps in SUMO by applying a RL method to optimise the route of a single vehicle in a network. The experiment shows promising results in finding the best path and avoiding traffic congestion. Firstly, this chapter describes the Markov Decision Process in vehicle route optimisation, how the urban network is to be modelled as a dual graph by using Markov Chain. Then it introduces the problem definition and the key elements in RL. The experiment results then are evaluated and discussed. The limitations of RL are also covered in the summary of this chapter.

5.2 Markov Decision Process for Vehicle Route Optimisation Problem

Reinforcement Learning briefly is a paradigm of the Learning Process in which a learning agent learns, overtime, to behave optimally in a certain environment by interacting continuously with the environment. The agent during its course of learning experience various different situations in the environment it is in. As RL could be only applied in the Markov Decision Process (MDP), vehicle route optimisation need to be modelled as MDP. Inspired by [25], this section introduces how the Markov Chain can be used to model an urban road traffic network.

5.2.1 Overview of Markov Chain

A Markov chain is a discrete time stochastic process, in which the transition probabilities depend only on the state of the chain at the previous time step and not on the past history of the process. In a time homogeneous Markov chain these probabilities are further independent of time. Let us consider a Markov

chain on $n \in N$ states. For each pair of states, $i, j = 1, ..., n$, we denote by p_{ij} the probability of going from i to j in one time step. As we consider only finite state Markov chains, the matrix $P \in [0, 1]^{nxn}$ of elements p_{ij} together with the initial distribution vector fully describe the evolution of the Markov chain for all times.

A sumo network essentially is a set of nodes that can be connected by edges. In this case only directed edges are considered here. Let's again consider a finite set of n nodes, then for each pair $i, j = 1, ..., n$ an edge from node i to node j indicates that it is possible to make a direct transition from state i to state j. The case $i = j$ is considered like a self-loop. It is possible to give a weight to each edge that corresponds to the cost of using that edge. If the aggregate cost of all outgoing edges of each node is normed to 1 then the costs can be interpreted as the weight of using the corresponding edge to leave the node.

There is a strong link between Markov chains with finite state space and graphs. States of the chain can be associated with nodes in the graph and non-zero probabilities of transition between two states in the chain can be associated with directed edges between the corresponding nodes with the given probability as a weight. Graphs can thus be analysed using methods for Markov chains. An important property of a graph is connectivity. A directed graph is called strongly connected if starting from any node it is possible to reach any other node by following the edge. Strong connectivity holds if and only if the transition matrix of the corresponding Markov chain is irreducible. Throughout this part of the thesis we shall assume that all considered Markov chain transition matrices are irreducible which means all nodes in the graph are reachable.

5.2.2 Apply Markov Chain Modelling in Urban Road Traffic Network

To define a sumo network, nodes and edges are needed to represent as the junction and road respectively. Therefore, the simplest way for the graph corresponding to the road network is constructed in the following way. Each intersection in the road network is a node in our graph and there is an edge between two nodes if there is a road segment that connects the two corresponding intersections. Below shows the example: In Figure 5.1, the left subfigure is a normal urban road traffic network that run in SUMO simulator. The right subfigure however is the corresponding graph of the urban road traffic network. We will call this the primal graph [94] where junctions A, B, ..., F are connected by road segments. For instance, the road segment CD is the road that allows a car to go from C to D, and is different from DC which goes from D to C. Note that in this specific example, Node A to B and node E to F are one way direction, however the rest are two way directions.

The connection between a road network and a Markov chain is straightforward if a city map is interpreted as a directed graph, where nodes correspond

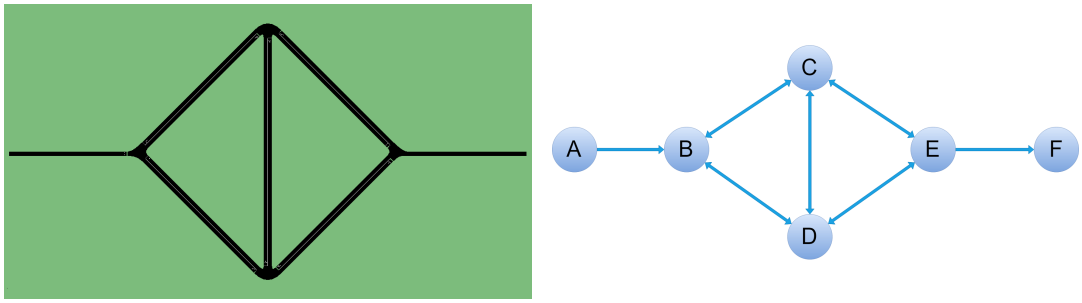


FIGURE 5.1: Sumo network

to junctions and edges to connecting roads. In the literature related to urban networks, compared to a primal graph, a dual graph however is a representation where the role of road and junctions is reversed (i.e. in the dual representation road correspond to nodes and junctions to edges). The dual graph corresponding to Figure 5.1 can be found in Figure 5.2. It can be noted that dual graphs carry more information than primal graphs. For instance, we can see from Figure 5.1 that cars are not allowed to perform u-turns at junction D, while the same information can not be recovered from the primal graph. The weights for the edges in the dual graph are given by the turning probabilities.

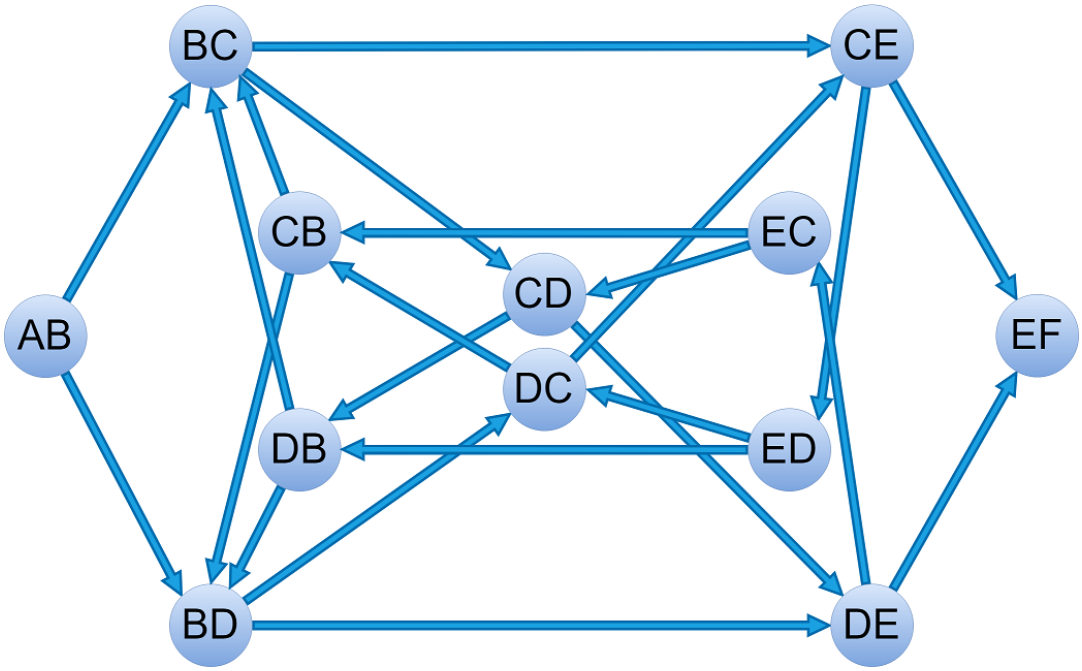


FIGURE 5.2: The edge network associated to the Sumo map shown in Figure 5.1

The first step to pass from a road network to a Markov chain is to transform the primal map into the dual one, where the nodes of the graph are represented by roads, as shown in Figure 5.2. The nodes of the dual network have been called XY intending that XY is the road that connects junction X to Y , where

X and Y were nodes in the primal network. The dual network is more convenient than the primal because it includes more information:

- In the primal network some edges should be inhibited depending on the edge of origin. For instance from Figure 5.1 it seems possible to go from node C to D and then to come back, while the more detailed dual network of Figure 5.2 shows that at the end of road CD turnaround is not permitted, and a longer route should be planned to enter road DC.
- A typical way of creating traffic flows is to exploit junction turning probabilities. The probability of choosing an out-going road at a junction clearly depends on the road segment of origin. This information is lost in the primal network.

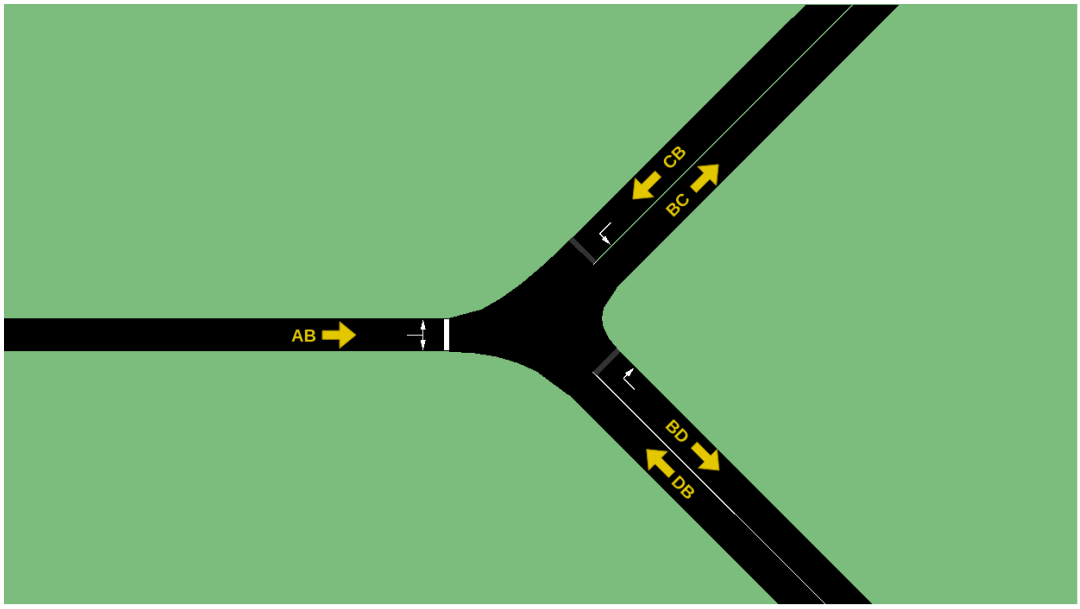


FIGURE 5.3: Edges in Sumo network that connected nodes shown in Figure 5.2

The dual network could work seamlessly with Sumo network. The figure 5.3 shows how a dual network is represented in the SUMO network that I designed for this experiment. Therefore, in principle, an urban road traffic network could be modelled as MDP and reinforcement learning could thus be applied.

5.3 Reinforcement Learning for Vehicle Route Optimisation

As with the complex nature of the urban transportation network, the vehicle navigation problem is a complex problem which involves many features that need to be considered, such as high-speed changes in traffic systems and the wide distribution of vehicles on the roadway. However, with the growing development of machine learning lately, reinforcement learning (RL) has led to very

promising results as a solution for complex systems. It also provides a potential mechanism for artificial agents to resolve the vehicle navigation problem. This section presents how reinforcement learning could be applied for vehicle route optimisation.

5.3.1 Problem Statement

In reinforcement learning, the goal is to learn a policy π that chooses actions at each time step in response to the current state thus the maximum total expected rewards could be received over all time. This is suitable for vehicle route optimisation in environments where global knowledge of the map is available nowadays. Inspired by recent success of reinforcement learning, a model free RL algorithm Q-Learning is adapted to deal with urban road traffic network as a Markov Decision Process for vehicle route optimisation.

Vehicle route optimisation requires prior knowledge or information of the urban road traffic network for navigation. Our problem is to find an optimal policy π for the agent to navigate *nav_veh* to its destination within this local environment. In other words, this experiment aims to use the Q-Learning method to obtain an efficient and self-learning based navigation system. Once *nav_veh* approaches a junction, an observed state s_t is derived from the current traffic environment to form the state space S ($s_t \in S$). The state s_t is fed into the agent v_i as the representation of current traffic observation. Based on the s_t, the agent v_i requires to select a decision from an action space A, where $a = \{a_1, a_2, ..., a_m\} \in A$ to perform re-routing for *nav_veh* in order to avoid traffic congestion. After taking an action based on current state s_t, the agent receives a reward $r_t(s_t, a_t)$ from the traffic environment. The following subsection describes the specific implementations of the action space A, state space S, reward R and the vehicle agent.

5.3.2 Key term definition of RL for Vehicle Route Optimisation

There are four key elements in the DRL system, named as vehicle agent, observation/state, action and reward scheme. The vehicle agent takes observations from the traffic environment as input and provides a recommended action as its output in order to maximize the final reward defined by reducing the travelling time to its destination. Their details are explained as follows:

State space: In RL, state is an observation encountered by the agent, then based on the observation an action is taken accordingly. The state space is the set of all possible situations for the vehicle agent. In this experiment, for simplicity purpose I only take the current road that the vehicle is on as the observation. Therefore the number of total possible states will be same as the total number of road in the network.

Action space: The agent observes the state (which road that vehicle is in) and it takes an action. As the purpose is to navigate the vehicle to its

destination, the actions in this experiment are to move the vehicle to the next road. In other words, the roads that are connected to the current road that the vehicle is on are the actions. Therefore, in Figure 5.2, a "state" is depicted as a node, while "action" is represented by the arrows that connect two nodes. In this particular network graph, each road connects to a maximum of two roads, therefore the possible actions of each state is 2.

Reward: To achieve the objective of this problem (To arrive at destination), a reward value to each action need to be set. The actions that lead immediately to the destination have an instant reward 10. The actions that lead to traffic congestion road have an instant reward -1. Other actions that not directly connected to the destination road have an instant reward 0. As shown in Figure 5.4, each arrow that represents action now contains an instant reward.

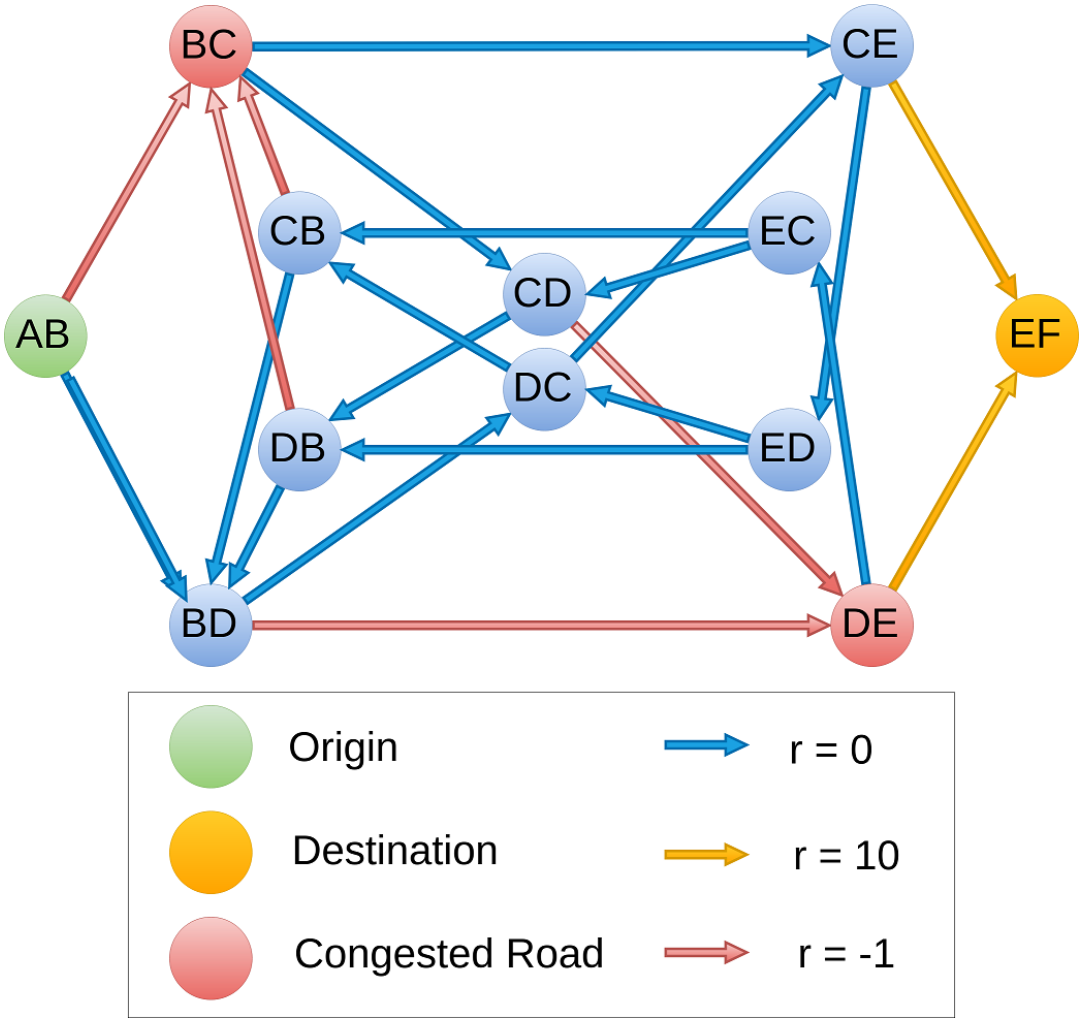


FIGURE 5.4: Dual Graph with rewards

Agent: The agent acts as a virtual robot interacting with the environment to learn the optimal route through experience by using the Q-Learning method.

A Q-table is created to store the Q-values that map to a $(state, action)$ combination. Firstly Q-table is initialised by all zeros then it starts exploring the network. For each state s, the agent selects any one among all possible actions for the current state and travels to the next stage s'. It then updates the Q-table with the action that has highest Q-value in the next stage by using the Equation 3.5 that was introduced in subsection 3.2.2. After that it sets the next state as current state. If the vehicle arrives at the destination, the agent ends the simulation and repeats the whole process. At the early stage, the agent has a high exploration rate to take more random actions to explore different routes to reach its destination. During the training it slowly takes actions based off rewards defined in the environment depends on the exploration rate. With enough training episode the agent is able to learn the optimal route to the destination once the Q-Table gets close enough to convergence.

5.4 Experiment Evaluation

This section introduces the concept of reinforcement learning for vehicle route optimisation through a simple but comprehensive experiment in SUMO simulator. This example describes a vehicle agent which uses unsupervised training to learn about an unknown road network. The idea is to learn from the model with optimal policy based on its observation. Each action that the agent has taken will lead to a reward or punishment with the new observation of the state. Through its learning progress, the agent learns an optimal routing policy to navigate a vehicle from origin to destination without encountering traffic congestion. The major goal of this experiment is to demonstrate, in a simplified environment, how RL techniques are applied to develop an efficient and safe approach for vehicle route optimisation. This section starts with the setup of the experiment, after that the implementation of the experiments are briefly described, including the hyper-parameter to represent the agent's policy. The result of the experiment then will be evaluated and analysed. At the end the summary and limitation of this experiment will be discussed.

5.4.1 Experiment Setup

The experiments are purely trained in SUMO simulator since it is a fast and efficient way of training and evaluating the model for urban transportation. The SUMO tool python API TraCI is used here to interact with the simulation environment. A SUMO network with corresponding graph design as shown in Figure 5.5 is used as the urban road traffic network in this experiment. The SUMO network then is converted to a Markov Chain dual graph as shown in Figure 5.6 by using the method that was described in subsection 5.2.2. Hence, each road in SUMO network becomes a node, and each junction in SUMO network becomes an edge.

A vehicle with RL agent, which is called *nav_veh*, is set to depart from node A to node J. In order to generate some traffic, various vehicles are inserted into

the network and are forced to stop for a certain number of time steps in order to create traffic congestion. Note that as illustrated in Figure 5.1, the destination of the vehicle in this experiment is node J, therefore in the dual graph, the corresponding destination node would be IJ (In another case it could be more than one road as long as it connects to node J, depending on the design of the urban road network), which is the only road that connects to node J in this case.

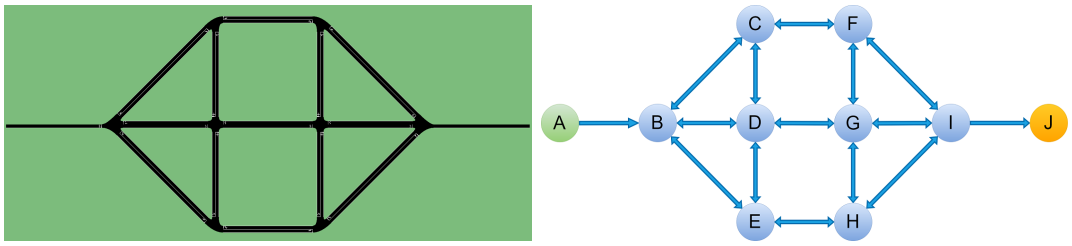


FIGURE 5.5: SUMO urban road traffic network

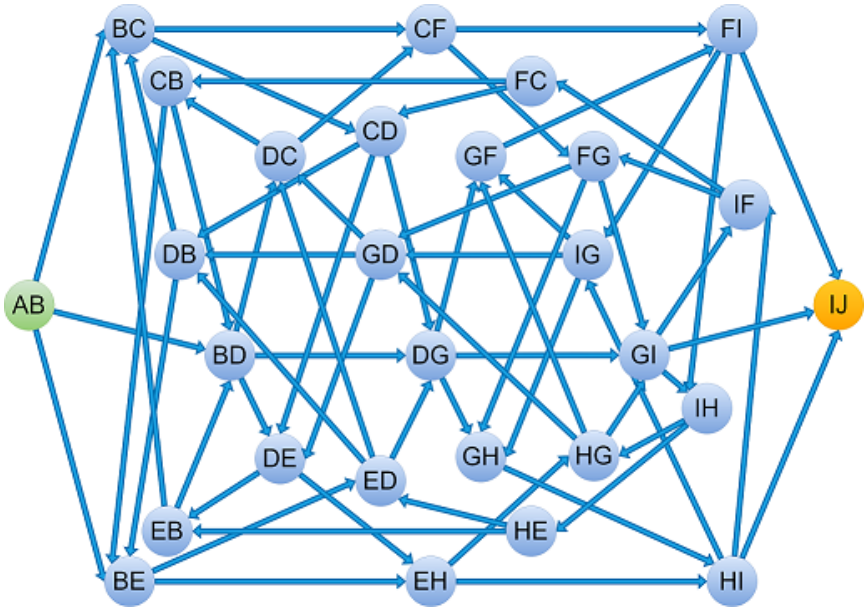


FIGURE 5.6: Dual graph network of Figure 5.5

5.4.2 Experiment Implementation

Two python classes are developed for this experiment. Python class *sumo_env* is to represents the environment that interfaces directly with SUMO TraCI API in order to recover parameters that are used in the simulation. The methods provided by *sumo_env* class are listed as follows:

- **get_edge_connection_info(self):** This method is to get and compile the connections between edges in the SUMO network and store the information in memory. The edge connection information is important in

this experiment as it is used as the reference to indicate which edge the actions in specific edge are linked to and determine the number of actions in each edge.

- **gen_demand_traffic(self):** This method is to generate the demand traffic for the simulation. It randomly injects various vehicles and forces them to stop for a certain amount of time in order to create traffic congestion.
- **is_state(self):** This method returns a boolean flag indicating if *nav_veh* is in a new state. This is for RL agent to determine the decision point to assign action for *nav_veh*.
- **reset(self)** This method is for resetting the simulation to the initial state in order to start a new episode. It returns to the initial state from the environment.
- **step(self, action):** This method takes an action a_t and transitions from state s_t to the new state s_{t+1}. It returns the new state s_{t+1}, the reward r and a boolean flag indicating if the simulation has finished.
- **get_obs(self):** This method gets the required parameters from the environment and build the state matrix.
- **get_reward(self):** This method calculates and returns the reward for the current state.

Another python class is called *RL_agent* which represents the RL agent that interacts with the environment and uses the implementation of the RL algorithm to make decisions for *nav_veh*. It implements the Q-Learning process by using the algorithm that was covered in subsection 3.2.2. The hyper-parameter that was used in this class are listed in Table 5.1. The methods provided by *RL_agent* class are listed as follows:

- **create_qtable(self, n_action):** This method is to create the Q-table for Q-learning RL method. It uses a well known python library *pandas* to create a two-dimensional data structure. The id is the state and the columns number is the same as the actions number, which to store the Q-value for each action in a specific state.
- **update_qtable(self, state, action, reward):** This method is to insert new data in Q-table or update the existing data in Q-table. All columns values are set to zero if it is a whole new state, otherwise it updates the Q-value accordingly based on Q-learning algorithm with the state, action, and reward parameters.
- **get_action(self, state):** This method returns an action value based on the state and Q-table.

Parameter	Value
Episodes	100
Learning Rate α	0.1
Exploration ϵ	0.1
Discount Factor γ	0.9

TABLE 5.1: Vehicle agent hyper parameters for intelligent navigation

A python script imports the aforementioned two classes above to implement the experiment. Initially, the SUMO network is loaded and the edge connection information is created and stored in memory. An example of edge connection information is shown in Table 5.2. A Q-table then is created for storing the Q-values. In every training episode, the demand traffic is generated, various vehicles are injected randomly into edges and stop for a certain time step, thus if *nav_veh* moves into those edges, it will be stuck in traffic congestion. When *nav_veh* is approaching the junction, a state will be observed and sent to the RL agent. If the state is a new state that never existed in Q-table, it will then be stored with all zeros in Q-table. A random action then will be selected and assigned to *nav_veh*. Otherwise, if the state is existing in Q-table, then a greedy policy ϵ is applied to determine if the agent chooses action randomly for exploration or choose the action with highest Q-value among action space to get more rewards. After executing the action, the *nav_veh* moves the target edge and an instant reward will be calculated and update the Q-value in Q-table by using the Q-learning algorithm that was described in Equation 3.5. This step is repeated until *nav_veh* reaches its destination. Then the simulation is terminated and runs a new episode to keep training the RL agent. The pseudocode of this experiment is presented in 8.

Algorithm 8: Q-Learning method

initialise Q-value table $Q(s, a)$ arbitrarily **for** *episode=1, M* **do**
  repeat
    observe state s
    with probability ϵ or s is a new state select a random action a_t
    otherwise select $a_t = argmax_a Q(s, a; \theta)$
    execute action a, observe reward r and next state s,
    $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma argmaxQ(s^{'}, a^{'}) - Q(s, a)]$ $s \leftarrow s^{'}$
  until *simulation is terminal*;
end

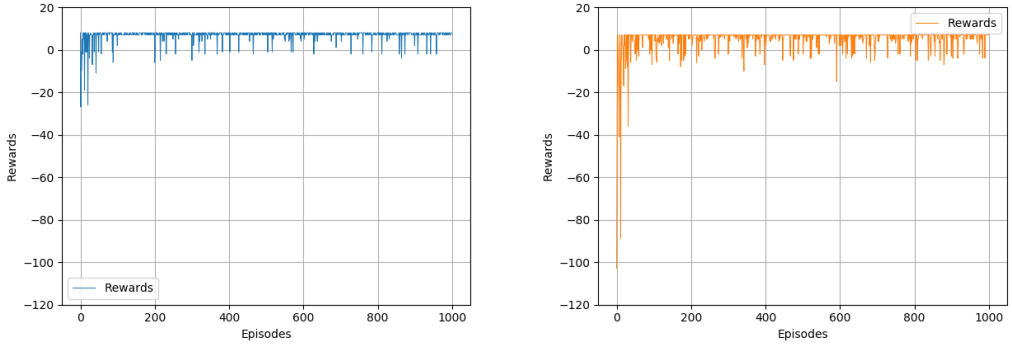
5.4.3 Simulation Result

The result of the experiment is promising. The Figure 5.7 shows the results of cumulative rewards that the RL agent received per episode. As illustrated in Figure 5.7(a), during its stabilisation point (approximately after episode 100), RL agent accomplishes positive rewards on average. According to the rewards

Road	Connected Road 1	Connected Road 2
AB	BC	BD
BC	CD	CE
BD	DC	DE
CB	BD	BC
DB	BC	BD
CD	DB	DE
DC	CB	CE
DC	CB	CE
CE	ED	EF
EC	CB	CD
ED	DC	DB
DE	EC	EF

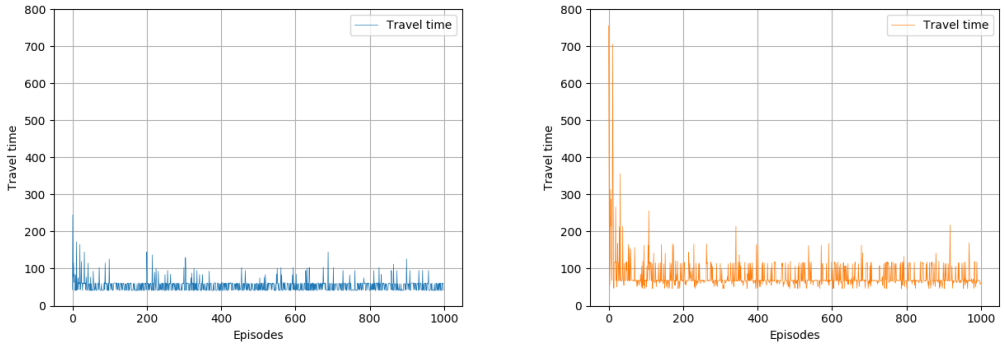
TABLE 5.2: The connected roads table

scheme in this experiment, this indicated that the RL agent manages to navigate *nav_veh* to its destination with fewer edges and without going through congested roads in the urban road traffic network. Figure 5.7(b) shows similar performance, although it took more episodes to reach the stabilisation point (approximately 250 episodes). It also has more fluctuations compared to Figure 5.7(a). Based on the observation we believe the reason is because the SUMO network that run in second scenario is bigger and has more edges than the first scenario. Therefore, the RL agent needs to navigate *nav_veh* through more edges to reach its destination. Thus, based on the greedy exploration rate ϵ, within one simulation in scenario 2, there are more chances that the RL agent decides to randomly select an action to explore the map, and those random actions could lead to the wrong route to the destination. Meanwhile, Figure 5.8 demonstrats the behaviour obtained in the travel time for *nav_veh* per episode. Both travel time reduce significantly after convergence, where the travel time are varying mostly between 40 and 60 time steps, and between 50 and 120 time steps respectively.



(A) Scenario 1 (B) Scenario 2

FIGURE 5.7: Cumulative rewards per training episode



(A) Scenario 1 (B) Scenario 2

FIGURE 5.8: Travel time per training episode

5.4.4 Discussion

Although the work done in this chapter has shown that Q-learning is a simple yet quite powerful algorithm to find an optimal route for vehicles in a transportation network, some challenges and limitations in this approach are worth discussing. Firstly, the real urban road traffic network is a lot bigger with more roads and junctions. Implementing the RL approach in a real urban map is needed. However, this means there could be a lot more states that the RL agent need to handle. Besides, the real urban transportation is a complex and rapidly changing system, and there are many features needing to be considered before making a decision to do vehicle navigation. The current state matrix designed for this experiment is insufficient for real urban transportation. However, the traditional Q-learning method struggles to deal with a large number of states, this is due to two main reasons, 1) as the number of states increases, the amount of required memory would increase to store and update Q-table; 2) the amount of time and power required to explore each state to create the required Q-table would be wasteful and unrealistic. Secondly, due to the complex nature of the urban transportation network, a dynamic reward function needs to be designed in order to improve the efficiency of policy learning. And last but not least, although the traditional exploration method in Q-learning makes the agent explore all possible states equally, in real urban transportation network this method could be using too much time and resources to explore some states and that obviously does not make sense.

5.5 Summary

This chapter has presented how to use the Markov Chain to model the Urban road traffic network in a dual graph. Dual graph is useful for RL method in vehicle route optimisation because it is able to present a more detailed graph and exploit junctions turning to probabilities. Besides, this chapter also has presented an experiment to combine the SUMO simulator and the development

of the RL agent to find the optimal route for vehicles in an urban road traffic network to avoid traffic congestion. Two SUMO maps with different scales are used in this experiment. The results show that with proper design of states, reward function and action, the RL method is able to optimise vehicle route in order to arrive at the destination with less travel time. However, RL methods could be inefficient when dealing with real large urban networks. Therefore the next chapter will introduce the main contribution of this thesis which brings the concept of the DRL method to overcome the limitation of the RL framework in vehicle route optimisation.

Chapter 6

The Proposed Framework and Structure Design

6.1 Overview

This chapter presents the proposed vehicle route optimisation mechanism by detailing its architecture and decision making process using a heuristic approach. Specifically, the proposed approach uses the DQN algorithm to train a vehicle agent to make better routing decisions for vehicle route optimisation in order to achieve the sustainable and resilient urban development goal. In other word, this approach aims to reduce the travel time for vehicle and the vehicles pollution for the urban transportation network. Firstly, the structure of the proposed training framework is presented. Furthermore, the key elements including two novel reward functions of DRL vehicle route optimisation are presented. Moreover, the DRL techniques and a distance based exploration schema that are used in the proposed approach are described. Finally, the comparison between different DRL techniques is conducted.

6.2 Proposed Framework for Vehicle Route Optimisation

In this thesis, an improved Deep Q-Learning Network (DQN) method [78] is proposed to train an intelligent agent to optimise vehicle route and navigate its vehicle to the destination and avoid congestion. This section is an introduction of the proposed framework for vehicle route optimisation from a system architecture perspective. The framework encapsulates the use of SUMO (Simulation of Urban Mobility) with Traffic Control Interface (TraCI) that establishes the connection with the DRL agent. It presents how the DRL agent is trained, demonstrating the various interactions between different components over the course of an experiment. Firstly, the structure of the training framework which include the traffic simulator, middleware and RL agent is overviewed. After that, the process of the training framework that establishes interaction between traffic simulator and RL agent is presented in detail.

6.2.1 Overview of Proposed Framework

The objective of the proposed framework is to provide an accessible way to optimise the vehicle route planning problem using DRL methods. The proposed framework is written by *python* which is able to create an environment encapsulating an MDP that defines a certain RL problem and handled by an external RL library. The environment is a class that provides an interface to initialize, reset and interfere the simulation, as well as various functions for receiving observations, executing actions and calculating rewards. In this work Tensorflow [1] which provides a number of built-in training algorithms is used as the machine learning application to support model training. This allows the evaluation of the performance of different DRL algorithms in a specific scenario.

The proposed framework aims to enhance SUMO simulator in order to make it more suitable for optimising vehicle route selection with DRL algorithms. To this end, the framework provides hand-designed controllers which enables interaction between the environment and external RL library through SUMO API TraCI, to allow model training in a rich environment with complex dynamics for vehicle route optimisation. A central focus in the design of the proposed framework is to step through the simulation in order to modify the demand traffic, network characteristic or vehicle behaviour within an experiment, along with an emphasis on enabling reinforcement learning control over an individual vehicle. Thus, it enables the training of policies across road networks of different size, density, number of edges and lanes. Moreover, the observation spaces and reward functions can be easily constructed from attributes of the environment. This makes the evaluation of the different scenarios straightforward after the models are trained.

Additionally, the proposed framework also supports the converting of OpenStreetMap to SUMO network. This enables vehicles and routing selection policies to be tested on real urban road networks seamlessly without designing a road network. Given different car-following models, vehicle types, speed limits, the framework makes it simple to evaluate traffic dynamics in different traffic circumstances. Furthermore, the extensibility of the framework provides huge flexibility to extend the features of the framework for future RL problems.

6.2.2 Training Framework Structure

As illustrated in Figure 6.1, a training framework is developed in order to provide several components to establish the interaction between the SUMO simulator and RL agent to run experiments. The training framework is designed to be modular and extensible and this makes it especially easy to extend the existing scenarios and environments to modify an experiment. The scenario is a pre-defined class designed to create the required files that define a simulation in SUMO. It holds various information about network, vehicle, etc and provides functions to convert an urban map (e.g. OpenStreetMap) to a SUMO network file and dynamically generate SUMO trips, routes and configuration files for traffic simulation.

The Environment is a class to directly interact with SUMO via SUMO API TraCI. It is mainly designed for controlling the SUMO simulator. It holds methods to initialise, interfere, step through and reset a simulation, including the definitions for the state spaces and action spaces, as well as the methods that aggregate information to calculate observations and rewards, and the action applicator to the simulation. Moreover, the environment allows adding multiple vehicle types or car-following models in an experiment. Thus the framework enables the straightforward use of diverse vehicle behaviour and configurations in SUMO and provides fully-functional environments for DRL problems.

Implicitly, the DRL agent is a class to import RL library Tensorflow which is supporting the implementation, training, and evaluation of reinforcement learning algorithms. It focuses on building, training and evaluating the deep neural network model for vehicle route optimisation policy. Based on the state spaces and action spaces that are defined in the environment, the DRL agent dynamically build the neural network with the corresponding input, output and number of neuron in hidden layers. It also holds the indicator flag of the certain DRL techniques in order to train the model with different DRL algorithms. The trained model is consistently saved as the checkpoint during the training. The saved model could be restored in a different scenario for evaluation or further training. It also could be used as the recovery when a failure of the training is detected.

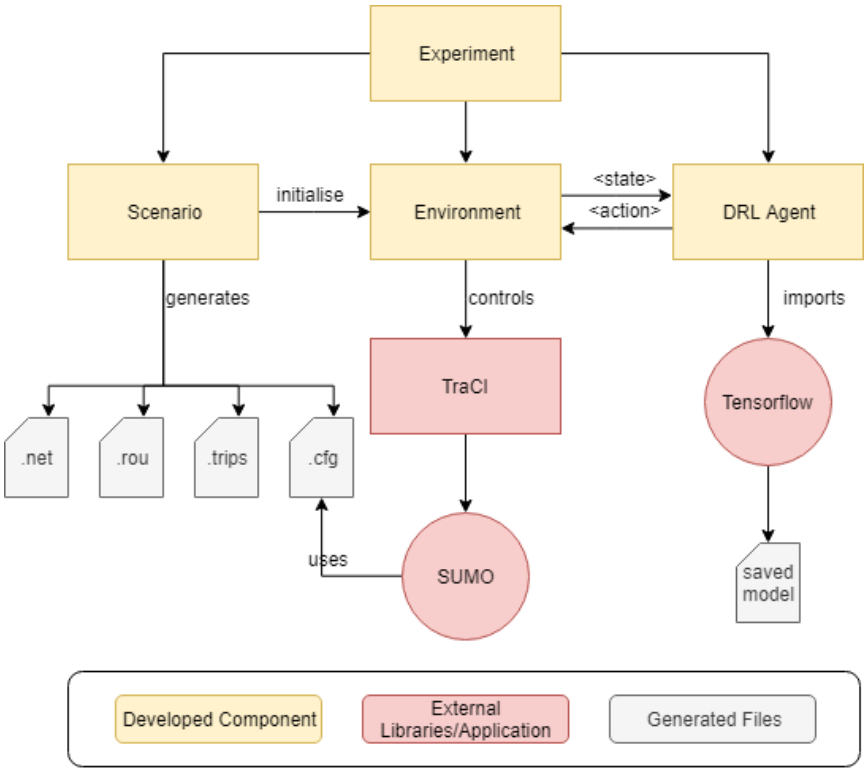


FIGURE 6.1: The proposed framework structure

6.2.3 Training Framework Process Flow

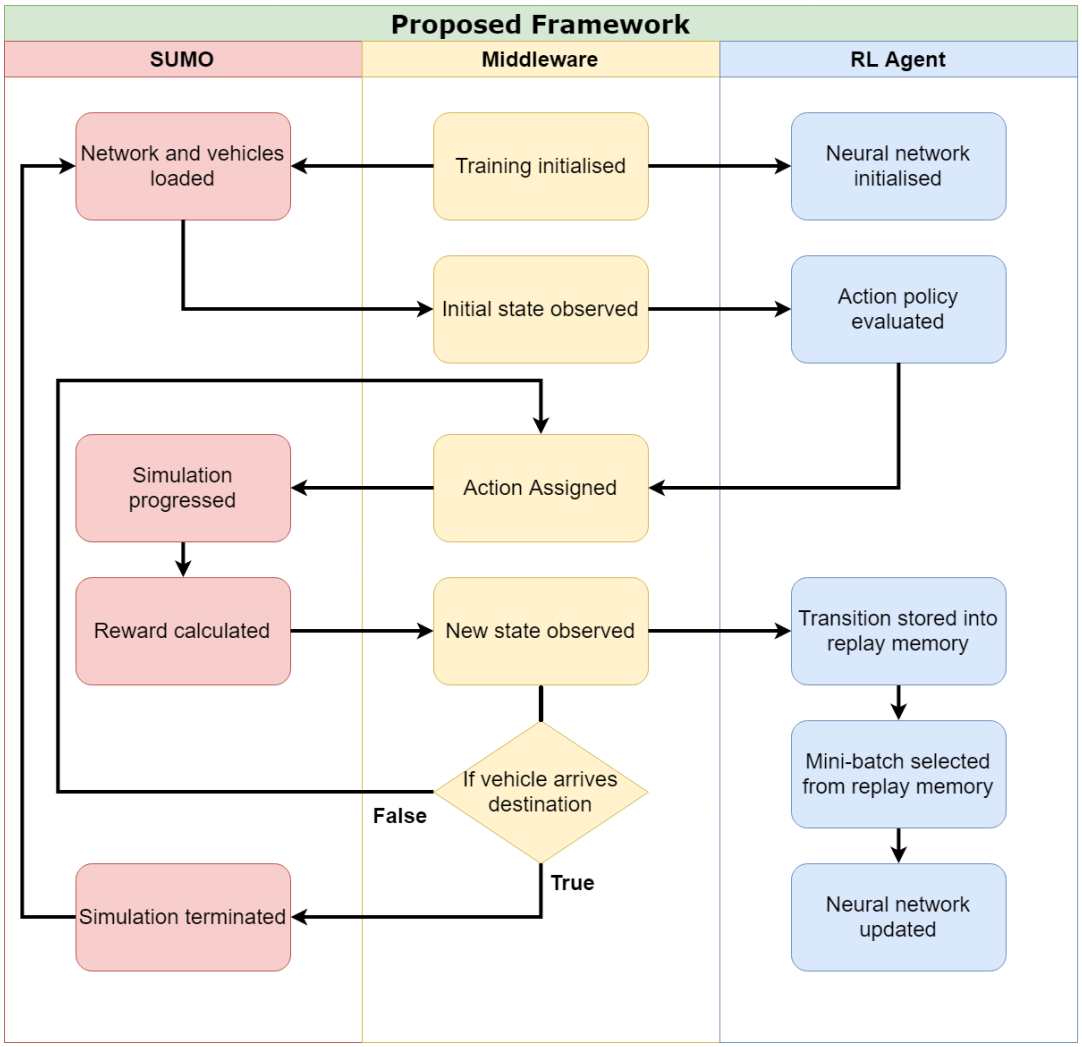


FIGURE 6.2: The Framework consists of SUMO simulator, Middleware and RL Agent for the vehicle navigation task

As shown in Figure 6.2, the designed framework consists of three parts: The first part is the SUMO, which is the environment simulator for training. The second part is the middleware that connects the SUMO environment with the reinforcement learning agent (RL agent), and the third part is the RL agent that consists of the reinforcement learning program, which is capable of internally maintaining and updating the policy, providing actions for the simulation.

The training framework coordinates the environment simulator SUMO with the observations, actions, and rewards needed and produced by the RL agent. When the training is being initialised, SUMO initialises the simulation by loading the required information such as the transportation network and vehicles information via TraCI. Note that the vehicle information includes the demand traffic for the simulation, and also a specific vehicle which is navigated by the RL agent, so called Agent Car. More details about the simulator SUMO and TraCI are discussed in subsection 4.4.1. In the meantime, RL agent is imported and a

neural network with corresponding structure associate with SUMO network is created and its parameters are initialised. After that, the simulation is started and the initial state is observed and forwarded to the RL agent. Based on the observation, the RL agent selects an action and sends it back to the middleware. In my experiment, actions are the edges that are connected to the current edge where Agent Car is. Subsequently, SUMO reroutes Agent Car accordingly based on the action from the RL agent. A reward then is calculated and forwarded to middleware with a new observation. This progress keeps going until Agent Car arrives at its destination to complete the simulation. In the meantime, in each training step, RL agent stores the required information (state, action, reward and new state) into replay memory as training data. When the replay memory has sufficient data, a mini-batch of transition is selected from replay memory to feed to neural network for optimising the policy.

The objective of this proposed framework is to train the neural network that is able to navigate a vehicle to find the best route to its destination and avoid congestion. Therefore, Once the model is trained, it can be evaluated in scenarios different from those in which they were trained, and making performance evaluation. The Figure 6.3 visualises all the use cases of the key actors in vehicle route optimisation. The driver or the user firstly plans the individual trip and send to vehicle agent that with the trained model. The vehicle agent then calculates the default route based on the trip info. The driver drives the vehicle by following the route and consistently updates the current vehicle position using GPS. When the vehicle is close to the junction, the vehicle agent makes the decision of the re-routing based on the observation of the traffic and sends the instruction to the driver. This process repeats until the driver reaches his destination. In order to justify the feasibility of our training framework, several experiments are run with different transportation maps and train the policy network with different levels of demand traffic, and compare their performances accordingly.

6.3 The Design of DRL for Real-time Vehicle Route Optimisation

This chapter outlines the approach used to design the different components of the vehicle route optimisation as an RL problem.

6.3.1 Problem Statement

Reinforcement learning improves system performance by means of taking real-time observations and evaluating the outcomes from actions under a complex environment. This is suitable to solve the navigation task as the nature of the traffic condition is dynamic and volatile. Inspired by recent success of deep reinforcement learning methods, an improved deep-Q learning method is adapted to deal with the real-world complexity given the fact that the navigation task can be modelled as a Markov Decision Process (MDP).

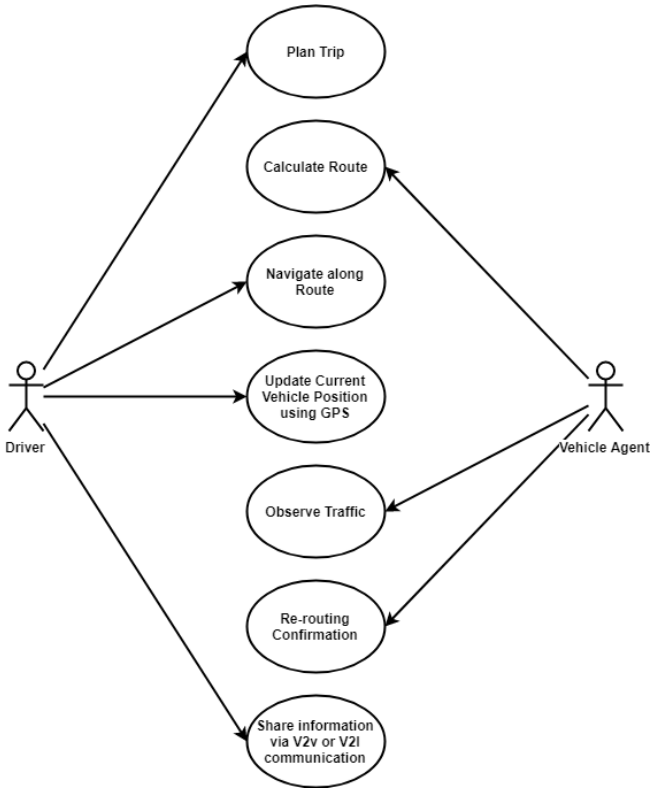


FIGURE 6.3: Use case of vehicle route optimisation

As illustrated in Figure 6.4, assume that each vehicle has an acting agent defined as $v \in \{v_1, v_2, ..., v_n\}$ to navigate the vehicle to reach its destination. Once the vehicle approaches a junction, an observed state s_t is derived from the current traffic environment to form the state space S ($s_t \in S$). The state s_t is fed into the agent v_i as the representation of current traffic observation. Based on the s_t, the agent v_i requires to select a decision from an action space A, where $a = \{a_1, a_2, ..., a_m\} \in A$. In addition, we defined a decision zone that has a fixed distance to each junction in order to make sure that the acting agent manages to change lane in our simulator to reach all possible actions in the action space. After taking an action based on current state s_t, the agent receives a reward $r_t(s_t, a_t)$ from the traffic environment. The goal of each reinforcement learning agent is to drive its vehicle to reach its destination as quickly as possible in order to achieve a higher reward. There are four key elements in the DRL system, named as vehicle agent, observation/state, action and reward scheme. The vehicle agent takes observation from the traffic environment as input and provides a recommended action as its output in order to maximize the final reward defined by reducing the travelling time to its destination. Their details are explained in the following subsections.

6.3.2 Vehicle Agent

Vehicle agent is a self-evolved neural network (NN) model, that takes traffic observations as its input and action decisions as its output. The number of

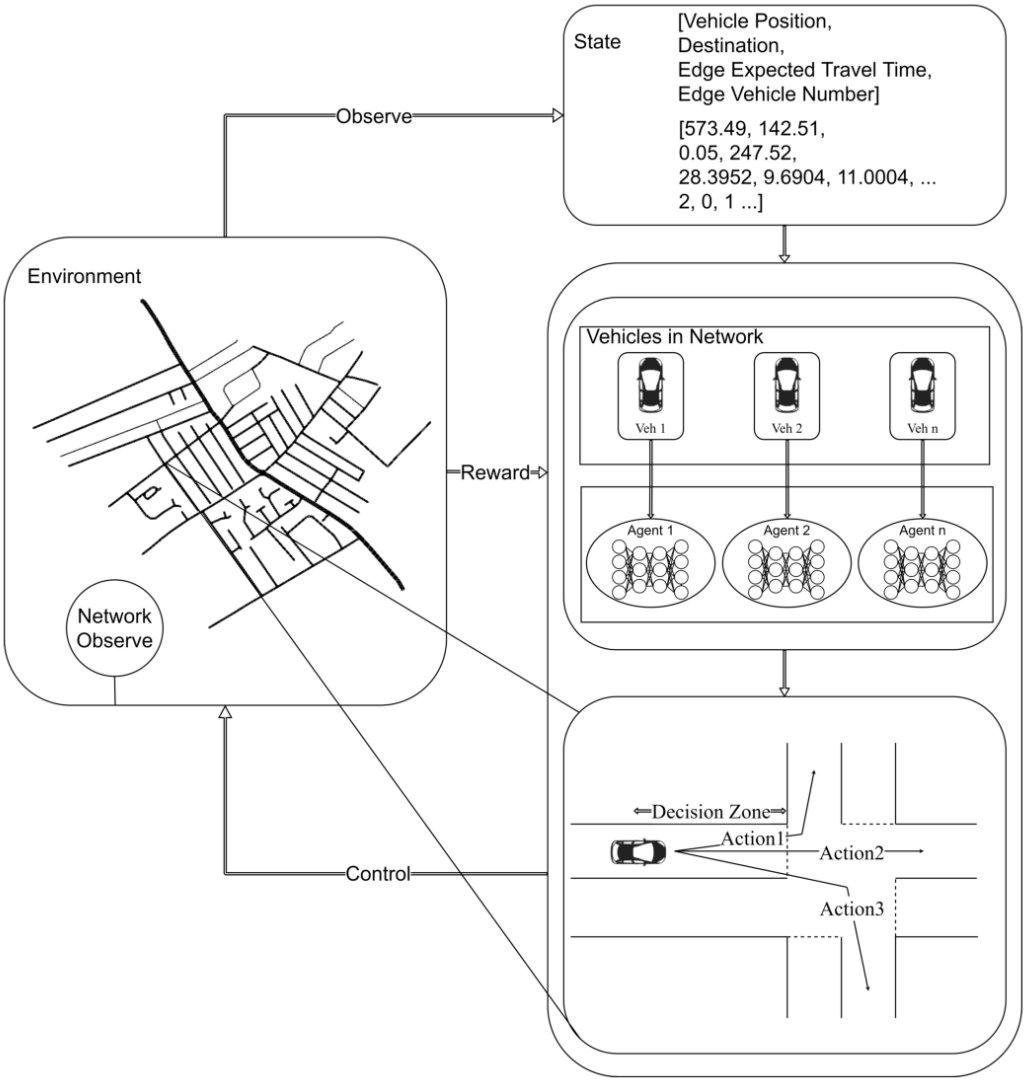


FIGURE 6.4: Problem statement of the RL based multi-agents navigation

inputs and outputs are dependent on the network characteristic. At its early training stage, the agent has a high exploration rate to take more random actions to explore different routes to reach its destination. During the training, a so-called exploration rate decay (ERD) is set to make the agent learn a more deterministic policy $\pi : S \rightarrow A$ so that the expectation of travel time under similar travel conditions can be reduced significantly. In terms of implementation, an acting agent is a four-layer NN with two hidden layers by using the Tensorflow library.

6.3.3 State Space

In order to allow the vehicle agent to makes a decision with the knowledge of the current traffic condition, state is an efficient representation of current traffic condition within a specific area of urban network. The representation variables

contain multiple parameters reflecting the circumstances in the global urban transportation network to precisely describe the complexity of its dynamics. Here, the state is defined as a vector with $[t_e, n_e, c_v, d_v]$ where the t_e is the expected travel time in the road, n_e is the numbers of vehicles in the road, and the c_v and d_v represent the current location of the agent and its destination.

Figure 6.5 illustrates an example of the state representation in a sample network and how the traffic conditions are observed and extracted. From the example, there are in total 8 roads in the network as $E \in \{AC, CA, BC, CB, CD, DC, CE, EC\}$. The traffic information is extracted in each road. Road DC has the most vehicles, which is 3 as $n_{DC} = 3$; road CA, EC have 2 as $n_{CA}, n_{EC} = 2$; road AC, BC, CD have 1 as $n_{AC}, n_{BC}, n_{CD} = 1$ and road BC has none as $n_{BC} = 0$. Meanwhile, in order to reduce the dimensionality of the state space, feature construction is applied for the calculation of expected travel time in road. Expected travel time in road is obtained from several features in the network, the calculation is shown by the following:

$$t_e = \left\{ \begin{array}{ll} \frac{l_e}{v_e}, & \text{if } n_e > 0 \\ \frac{l_e}{m_e}, & \text{if } n_e = 0 \end{array} \right\}$$

where l_e is the length of the road, v_e is the average driving speed in the road, m_e is the speed limit in the road and n_e is the number of vehicles in the road. Moreover, the state also takes the vehicle's current road and the destination road as the observation as well. As the roads in the network are label data, the coordinate of the middle point of each road is extracted as the observation. In the example, vehicle position x_v is the coordinate of the middle point of road AC which the vehicle agent (the red vehicle) is currently in. However, the destination y_v is the coordination middle point of road CE. The state matrix is shown in Figure 6.6. As the vehicle agent only requires the latest state to make a decision, I compartmentalise a segment in each edge, named as *Decision Zone* to indicate the best timing for obtaining state. The decision zone z_v is expressed as:

$$z_v = min[l_{edge}, v_{max} + v_{max} b(v) \tau_v] \tag{6.1}$$

where l_{edge} is the length of the edge, v_{max} is the maximum speed of the individual vehicle v, b_v is the deceleration function of vehicle v and τ is the driver's reaction time. The decision zone is determined by both vehicle type and driver's reaction time due to the safety considerations of urban transportation networks.

6.3.4 Action Space

Action in vehicle route optimisation refers to the navigation decision made by a vehicle agent. As mentioned before, when the vehicle agent arrives at the decision zone in a road, it observes the state S_{road} and then chooses one action $a_{road} \in A$. The action space A varies depending of the structure of network.

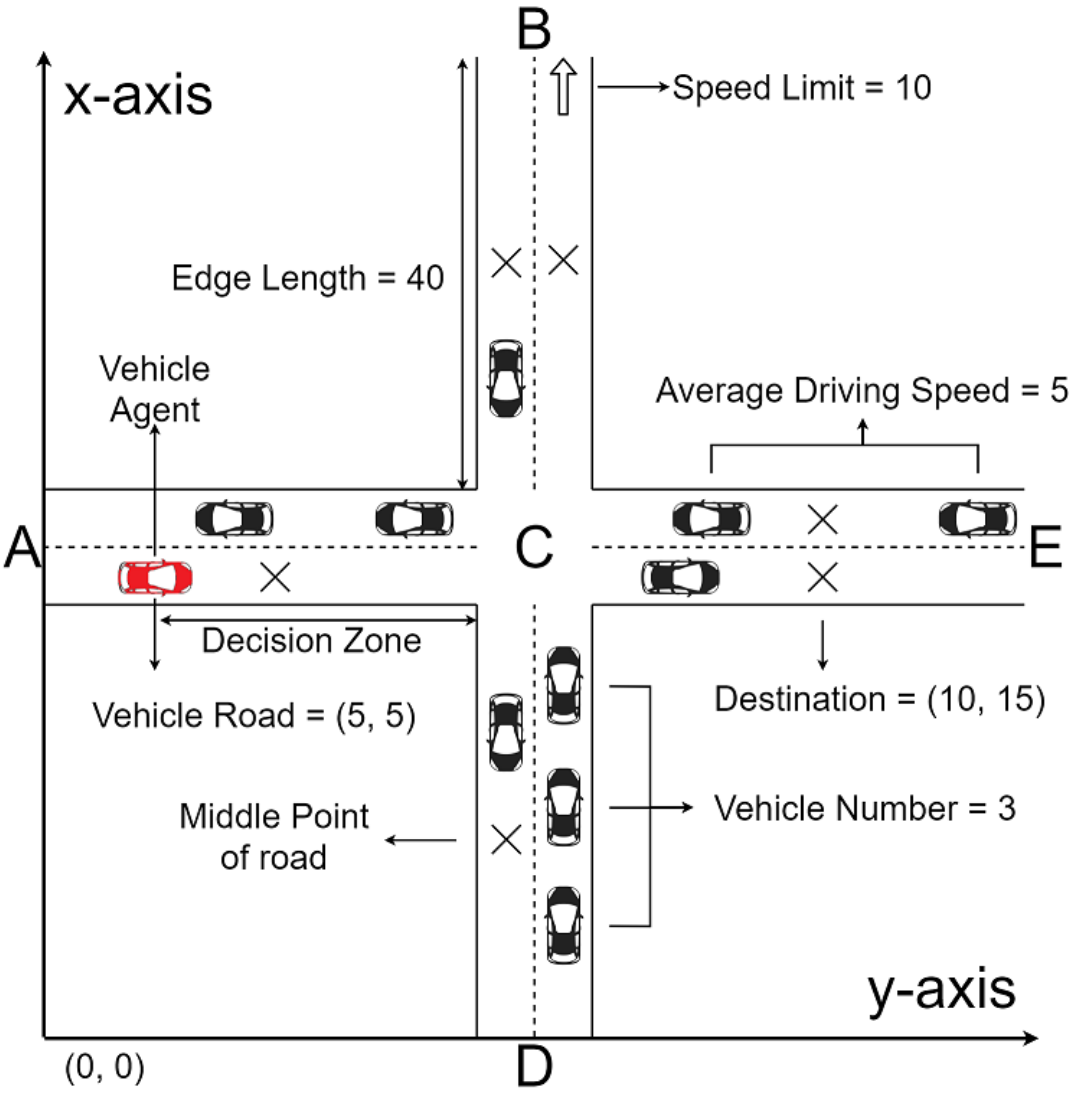


FIGURE 6.5: Problem statement of the RL based multi-agents navigation

The actions are discrete values corresponding to the decisions of navigating the vehicle to m connected edges from current edge. For example, the vehicle agent in road AC has actions which include left-turn, right-turn, go-straight or U-turn which link to road CB, CD, CE and CA as illustrated in Figure 6.5 if $m = 4$. Thus the action space is $A_{AC} \in \{e_{CB}, e_{CD}, e_{CE}, e_{CA}\}$. Meanwhile, if the vehicle agent is in road CA, the action space then is $A_{CA} \in \{e_{AC}\}$ as road CA only allows vehicle to make a U-turn to road AC. Finally, in the experiments with realistic city maps, m is set to the maximum number of connected edges in the maps.

6.3.5 Reward Function

Reward is the most important factor in the DRL system as it guides each agent to converge to an optimal policy π_θ by encouraging good actions made from the

	AC	CA	BC	CB	DC	CD	CE	EC
n	1	2	1	0	3	1	1	2
t	10	15	13.3	4	20	10	10	8

c_v	5	5
d_v	10	15

FIGURE 6.6: State Matrix for network Figure 6.5

function approximations. The definition of the reward function is one of the hardest parts of DRL. The overall principle of the reward setting in our work is to maximise the expected future discounted returns. the expected reward is described as:

$$r = \sum_{k=0}^{T} \gamma^k r_{s+k} \tag{6.2}$$

In order to improve the efficiency of vehicle routing, the first objective is to minimise the travel time. This is because travel time in vehicle routing is essentially important not only for normal vehicles but also for emergency vehicles in order to make urban transportation networks more resilient. However, the reward function can not be directly defined as total travel time because the travel time of a vehicle cannot be computed until it has completed the route, which leads to the problem of extremely delayed rewards. Fortunately, the total travel time can break down to travel time in each road of the route to avoid the long delay reward issues. Therefore, in the proposed system the reward function r_s sets the travel time in the road as an instant reward manner after the vehicle reaches the end of the road rather than the total travel time from origin to destination. Thus, the reward function can be formulated as:

$$r_{s_t} = -(t_{s_{t+1}} - t_{s_t}) \tag{6.3}$$

where $t_{s_{t+1}}$ is the total travelling time to the state s_{t+1} and t_{s_t} is the total travelling time to the state s_t.

In order to optimise vehicle routes towards sustainable and resilient urban transportation network, the travel time should not be the only factor that needs to be considered. Vehicle emissions affect local and regional air quality. Therefore, the reduction of vehicle emissions in urban networks is another important

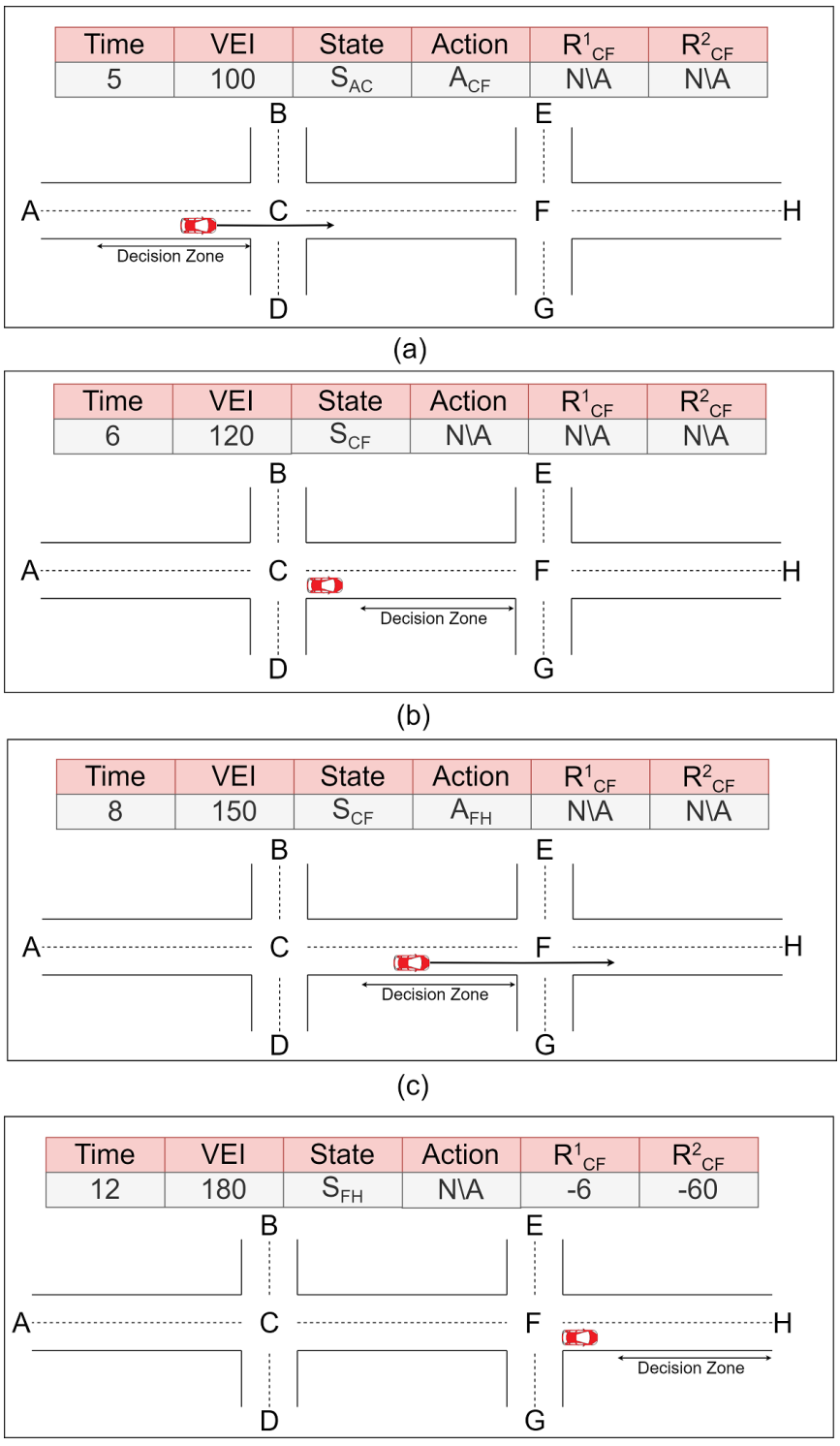


FIGURE 6.7: Virtualisation of reward calculation in urban network

optimised objective. The criteria pollutant emissions generated from fuel combustion by internal combustion engines (ICE) include nitrogen oxides NO and NO_2, together called NO_x, hydrocarbons HC and carbon monoxide CO, and particulate matter PM_x. A method to measure the environmental damage from

vehicle NO_x, HC, CO and PM_x emissions which is called Vehicle Environmental Impact (VEI) is proposed as another reward schema in this thesis. The idea of this reward schema is to optimise the vehicle route in order to minimise the VEI that is caused by vehicles. Thus reducing the vehicle emissions in urban networks for sustainable development. The VEI equation can be formulated as:

$$VEI = \sum_{k} \frac{E_k W_k}{M_k} \tag{6.4}$$

where k is vehicle emission as $\{NO_x, HC, CO, PM_x\} \in K$, E is the actual vehicle emission, M is the Euro 6 standard [65] for vehicle emission NO_x, HC, CO, PM_x and W is the damage impact weight value for NO_x, HC, CO, PM_x corresponding with Euro 6 emission standard. Similar to travel time reward schema, VEI reward schema takes the VEI in a road as an instant reward to avoid the extremely delayed reward problem. Thus, the VEI reward function is shown as follows:

$$r_{s_t} = -(VEI_{s_{t+1}} - VEI_{s_t}) \tag{6.5}$$

where $T_{s_{t+1}}$ is the total VEI of vehicle to the state S_{t+1} and T_{s_t} is the total VEI of vehicle to the state S_t.

The Figure 6.7 illustrates how travel time R^1 and VEI R^2 instant rewards in a state are calculated in a network. Assuming a vehicle's origin and destination are AC and FH respectively. In Figure 6.7(a), the vehicle arrives in the decision zone on road AC, the vehicle agent observes the state S_{AC} and makes action A_{CF} to road CF, note that currently the time step is 5 and the VEI for vehicle is 100. In Figure 6.7(b), vehicle arrives on road CF in time step 6 and the VEI is 120. In Figure 6.7(c), vehicle arrives in the decision zone in road CF, state S_{CF} is observed and action A_{FH} is decided. Note that until this stage, both travel time R^1 and VEI R^2 rewards are not yet computed as the vehicle has not completed road CF. In Figure 6.7(d), the vehicle arrives on road FH in time step 12 and total VEI is 180. In this stage since vehicle has completed road CF, both travel time R^1_{CF} and VEI R^2_{CF} can now be computed as $R^1_{CF} = -(12-6) = -6$ and $R^2_{CF} = -(180-120) = -60$. The transition $(S_{AC}, A_{CF}, R^1_{CF}, R^2_{CF}, S_{CF})$ then could be stored.

The two proposed reward schemas then are used to train the neural network and evaluate their performance in different scenarios for vehicle route optimisation. Table 6.1 shows the hyper-parameters used for the training. For the epsilon decay function in this research, in the beginning it starts with 1, where it indicates the probability that vehicle agent decides to explore the network instead of taking an action with highest Q-value. The rate will be reduced when the model is trained until 0.05, which indicates that there are 5% of time the vehicle agent will explore the network to looking for potential better path. The training showed promising results in vehicle route optimisation. As shown in Figure 6.8, both reward schemas start converging around episodes 480. Although both reward schemas are able to reduce travel time and VEI of a vehicle,

travel time reward schema had better performance in reducing the travel time. However VEI reward schema performed better in minimising the vehicle emission. Therefore, this experiment proved that both reward schemas could be chosen for vehicle route optimisation depending on the scenario. For instance, travel time reward schema could be applied for emergency vehicles as they always need to arrive at the destination with least travel time in order to maintain the resilience of the urban transportation network; VEI reward schema is more suitable for general vehicles for urban citizens in order to control the vehicle emissions for urban sustainable development.

Parameter	Value
Episodes	1000
Learning Rate	0.001
Exploration Rate (Decay Function)	$1.0 \rightarrow 0.05$
Target Network Update per learning step	3000
Discount Factor	0.99
Replay Memory Size	10000
Mini Batch for Update	32

TABLE 6.1: Vehicle agent hyper parameters for reward schemas comparison

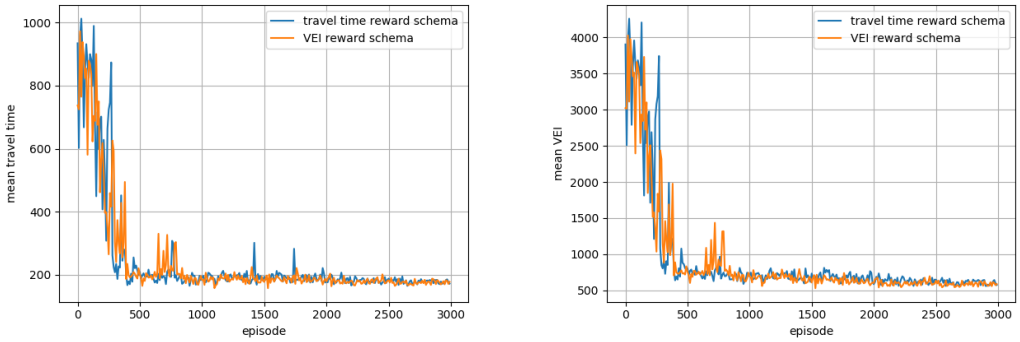


FIGURE 6.8: The convergence graph for 2 proposed reward schemas in travel time and VEI

6.4 DRL method for Real-time Vehicle Route Optimisation

In our work, an improved DQN architecture is designed for the real-time intelligent vehicle navigation. DQN is an online training method to maximise an action-value function $Q(s, a)$, defined in Eqn.6.6, that is an estimation of expected cumulated return from a sequential decision making. In the method, multi-layer neural networks are utilised as function approximators that map from a state to Q-values $Q(s, a) \approx Q(s, a|\theta)$.

$$Q(s,a) = E[R_t|s_t = s, a_t = a] \tag{6.6}$$

According to the Bellman equation, if the Q values for all actions in next state s_{t+1} are known in $Q^{\pi}(s_{t+1}, a_{t+1})$, the Q-value in current state is the summation of the immediate reward r_t and the maximum cumulated reward in the next step. Therefore, we can set the target maximum expected reward for current stage as $r_t + \gamma maxQ^{\pi}(s_{t+1}, a)$, where $0 < \gamma \leq 1$ is a discount factor. By updating the Q value iteratively, the expected return is defined in Eqn.6.7:

$$Q_{t+1}(s,a) = Q_t(s_t, a_t) + \alpha(r_t + \gamma maxQ_t(s_{t+1}, a) - Q_t(s_t, at)) \tag{6.7}$$

The parameter θ is trained by minimising the error between the expected cumulated return and the Q-value predicted by the agent. Same as the work in [78], two neural networks, including a target network and an online trained network, are adopted in our work. The target network is used to estimate the Q values and being updated after a certain number of episodes. The loss of an individual experience to train the online network is defined in Eqn.6.8:

$$L(\theta) = (r_t + \gamma maxQ(s_{t+1}, a|\theta^{-}) - Q(s_t, a|\theta))^2 \tag{6.8}$$

When sampling the eperiences (s_t, a_t, r_t, s_{t+1}) from replay memory, prioritised experience replay algorithm [99] is used in our work to update the DQN network θ. In the method, the probability p_t defined in Eqn.6.9 is calculated to increase the possibility of sampling experiences which are new in the memory for faster convergence.

$$p_t = \frac{1}{rank(i)} \tag{6.9}$$

Here rank(i) is the rank of transition i when the replay memory is sorted according to new or old degree.

The techniques used in Double DQNs [112] and Dueling DQN [113] are also implemented in our DRL networks. The double DQNs method is integrated into our method is for solving the Q value overestimation problem. The max operator in standard Q-learning and DQN uses the same values both to select and to evaluate an action. It is known the this maximization sometimes produces to learn unrealistically high action values which tends to prefer overestimated values over underestimated values, resulting in overoptimistic value estimations. To prevent this, Double Q-learning decouples the selection and the evaluation. In this algorithm, two value functions are learned by assigning each experience randomly to update one of the two value functions, such that there are two sets of weights, θ and $\theta^{'}$. For each update, one set of weights is used to determine the greedy policy and the other to determine its value. The difference with the Double Q-learning is that the weights of the second network $\theta_t^{'}$ are replaced with the weights of the target network θ_t^{-} for the evaluation of

the current greedy policy. The update to the target network works the same as normal DQN.

However, the dueling DQN is for achieving better convergence when presenting many similar-valued actions. is a technique proposed by [114] which computes separately the value V(s) and advantage A(s, a) functions that are represented by a duelling architecture that consists of two streams where each stream represents one of these functions. These two streams are combined by an convolutional layer to produce an estimate of the state-action value Q(s, a) as shown in Figure 6.9. The dueling network automatically produces separate estimates of the state value and advantage functions without supervision. Besides that, it can learn which states are valuable, without having to explore the consequence of each action for each state.

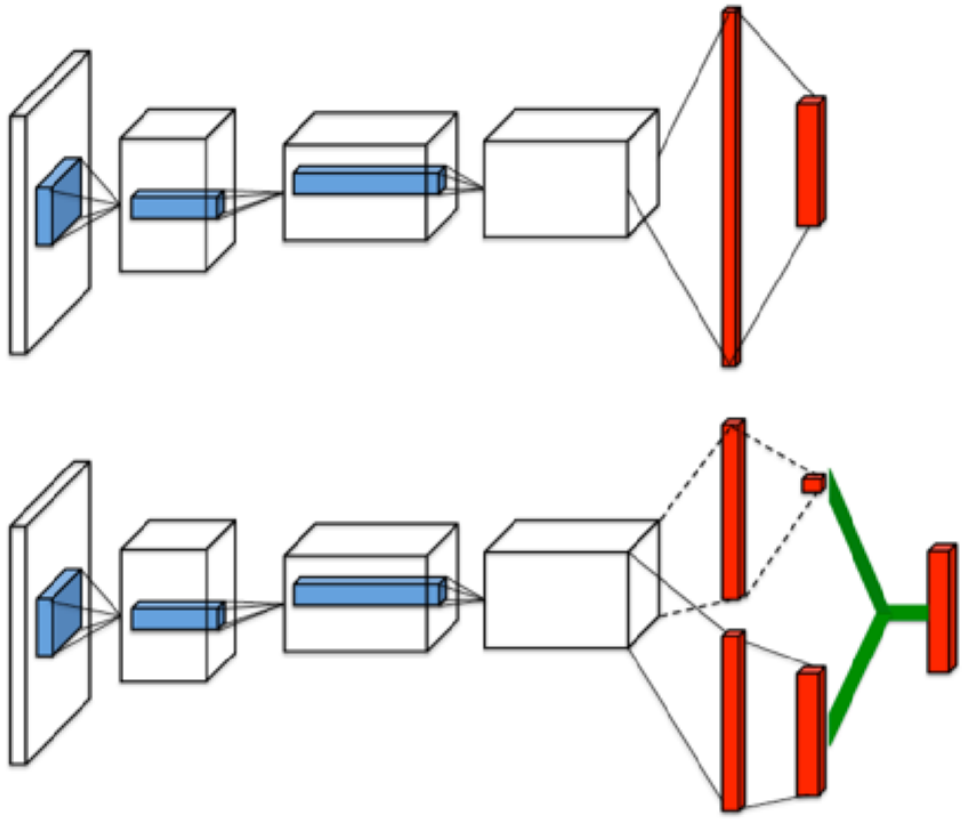


FIGURE 6.9: A popular single stream Q-network (top) and the dueling Q-network (bottom). The dueling network has two streams to separately estimate (scalar) state-value and the advantages for each action; the green output module combine both state-value and the advantages and output the Q-values. [114]

Therefore, this architecture helps us accelerate the training. We can calculate the value of a state without calculating the Q(s,a) for each action at that state. And it can help us find much more reliable Q values for each action by decoupling the estimation between two streams.

In the proposed method, a novel two-stage exploration scheme is designed to improve the network convergence as well as the converging speed. In the first stage, the conventional ϵ-greedy policy is used to control the ratio between exploration and decisions made by the current neural network. In the second stage of the scheme, a distance based method is used to replace the random selection for edge exploration. As illustrated in Figure 6.10, the agent vehicle is in decision zone (the vehicle in red colour), the edges with blue colour are the edges that link to agent vehicle current edge, as known as the possible actions of agent vehicle in this particular case. We calculate their Euclidean distances between the end of blue edges and the end of destination edge. After keeping all the Euclidean distances of each edge to the destination of the vehicle, where $\{d_1, d_2, ..., d_m\} \in D$ and m is the number of connected edges, the probability P is calculated from Eqn.6.10 to decide the explored edge selection.

$$P = softmax\left(\frac{D - \bar{D}}{\sigma}\right) \tag{6.10}$$

where $\bar{D}$ is the average value of the distances in D, and σ is its standard deviation of distances in D.

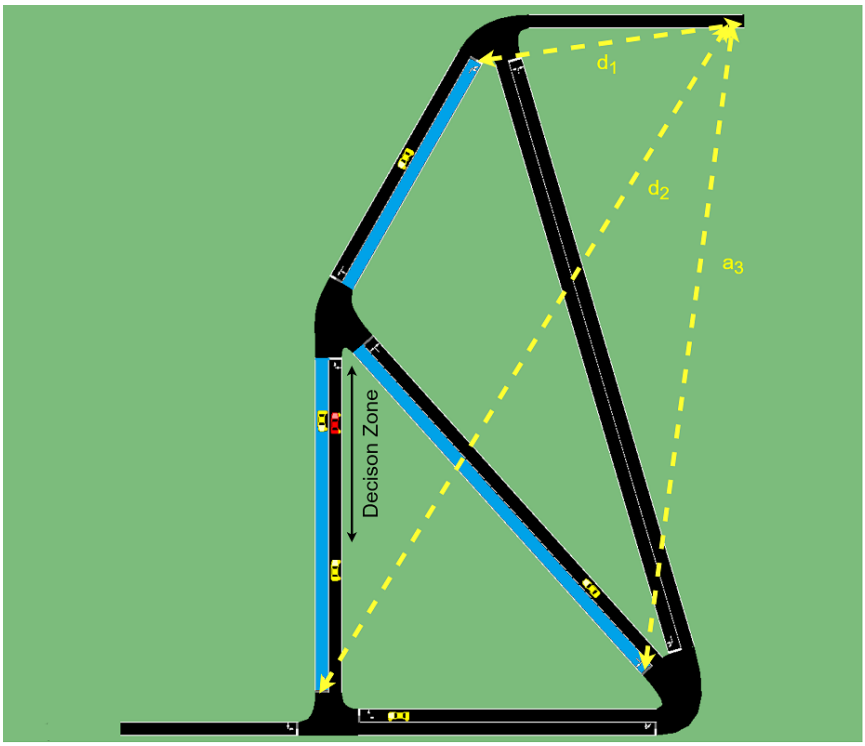


FIGURE 6.10: Comparison of different DQN methods

As illustrated in Fig.6.11, our proposed DRL method has the best convergence performance on both the converged travel time and the converging speed when compared to the traditional DQN and the combination of double DQN, dueling DQN method and priority experience replay (called Combo-DQN in our work). In the figure, it shows that the average converged travel time by using

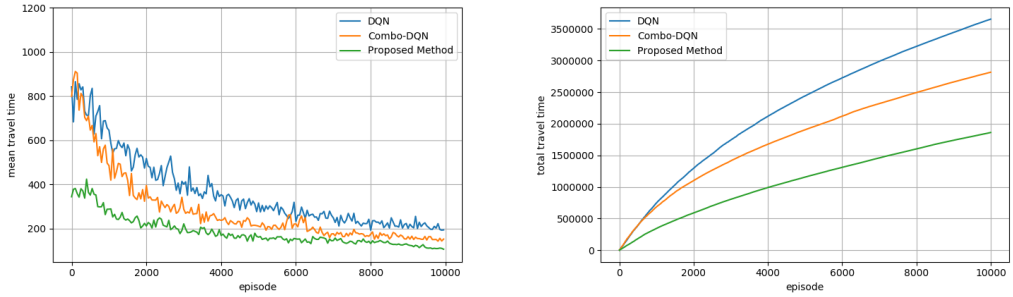


FIGURE 6.11: Euclidean distances based action selection policy

the proposed method after 10,000 epochs is 100 time steps which is reduced about 50% and 37% than the DQN and Combo-DQN respectively. Furthermore, it shows that the cumulative travel time of the proposed method during the training is also the best among the three methods. The training parameters are presented in Table.6.2.

Parameter	Value
Episodes	10000
Learning Rate	0.001
Exploration	$1.0 \rightarrow 0.05$
Target Network Update per learning step	3000
Discount Factor	0.99
Replay Memory Size	10000
Mini Batch for Update	32
Prioritisation Exponent	0.6
Prioritisation important Sampling	$0.4 \rightarrow 1.0$

TABLE 6.2: Vehicle agent hyper parameters for intelligent navigation

6.5 Deep Neural Network Architecture for Real-time Vehicle Route Optimisation

This section presents the deep neural networks structures that are used for the proposed framework. As mentioned in subsection 6.2.2, the RL agent build the neural network dynamically based on the urban network structure. This is because the state space and action space in our proposed framework varied depending on the total number of roads and the maximum number of connected roads in the urban network. The input number n_{input} for the deep neural network could be defined as follows:

$$n_{input} = 2n_{roads} + 4 \tag{6.11}$$

where n_{roads} is the total number of roads in an urban network. This is based on the design of the state in this proposed method, where we observed two features

(expected travel time and vehicle number) in each road, plus the coordination of the origin and the destination for the agent vehicle.

There are two deep neural networks for training a model. One is called *actor* network and another one is called *target* network. Both networks have the same structure but only *actor* network is trainable as *target* network is used to estimate the target Q-value for calculating loss. At certain steps the parameter θ of *actor* network will be copied to update the *target* network.

The deep neural network architecture for vehicle route optimisation contains five fully connected layers with two hidden layers and dueling structure. The first hidden layers has total 150 neurons and second hidden layers has total 100 neurons. Both hidden layers use Relu as the activation function. A dueling network splits into two streams of fully connected layers which are the advantage stream and value stream. The value stream only has one output and the advantage stream has the output of as many as the number of actions which is the maximum number of connected roads in the urban network. This deep neural network architecture is implemented for all the experiments performed in the next section. The Figure 6.12 shows the structure of the neural network in this experiment.

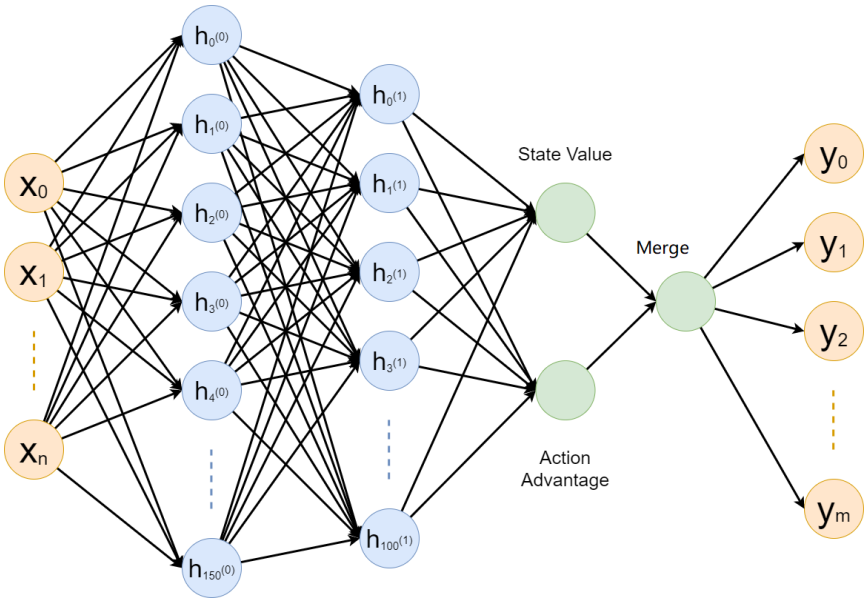


FIGURE 6.12: The structure of neural network in this experiment

6.6 Summary

In this chapter, the proposed real-time vehicle route optimisation based on the DQN method is introduced. Firstly the proposed framework presented how the work had been done to extend the existing applications in order to establishes interaction between the RL library and the traffic simulator for model training

and testing. Then the design of DRL for vehicle route optimisation is discussed. It described the key components of the proposed DRL approach such as vehicle agent, state and action spaces, reward function. In order to move towards a sustainable and resilient urban transportation network, a novel vehicle emission damage measurement algorithm called vehicle environment impact (VEI) is proposed. Based on that, two rewards schemas which aim to reduce the travel time or VEI are designed and compared where travel time based reward is suitable for first priority vehicle such as emergency vehicle, and VEI based reward which targets general vehicles in the urban transportation network. Moreover, the DQN techniques for the proposed approach are introduced and compared. The proposed approach combined several DQN techniques and a novel distance based method exploration schema to improve the training and exploration efficiency. Additionally, the deep neural network architecture that was designed for vehicle route optimisation is described. In the next chapter the implementation of the training and testing in three real urban networks with traffic simulator SUMO and RL framework Tensorflow will be presented.

Chapter 7

Experiment Implementation and Evaluation

7.1 Overview

This chapter describes the experiment implementation of the real-time vehicle route optimisation for the sustainable and resilient urban transportation network. Firstly the developed classes and the components in the experiment are presented. The preparation works that need to be done before running the experiment are introduced with details, including map converting, demand traffic generation, data pre-processing, etc. Moreover, the benchmark methods are introduced. Additionally, implementation of DRL agents and the communication between agents and the simulation environment are described. Lastly, the application of real-time vehicle route optimisation with the toy data and realistic scenario are evaluated and discussed.

7.2 Experiment Implementation

There are three aforementioned python classes are developed for this experiment which are *scenario*, *environment* and *DRL_agent*. This section introduces the components of the developed classes and presents the interaction and communication between them. This section also describes the preparation works and the enhancement of the existing framework.

7.2.1 Training Simulation Overview

This subsection presents the mechanism of the training simulation in this experiment. The overview of the three main classes process flowchart is illustrated in Figure 7.1. Initially, scenario class *scenario.py* converts the real urban map to SUMO map and generates the required vehicle types, trips and routes to create a configuration file for SUMO simulation. Environment class *sumo_env* then starts SUMO via TraCI by using the configuration file. The *sumo_env* stores the edge connection information and Euclidean distance to destination in memory, after that it starts the simulation and adds a vehicle "*nav_veh*" into the traffic simulation. Meanwhile, agent class *DRL_agent* is initialised and builds the corresponding DNN based on the road connection information. In environment class, the simulation starts after "*nav_veh*" is added. In every simulation

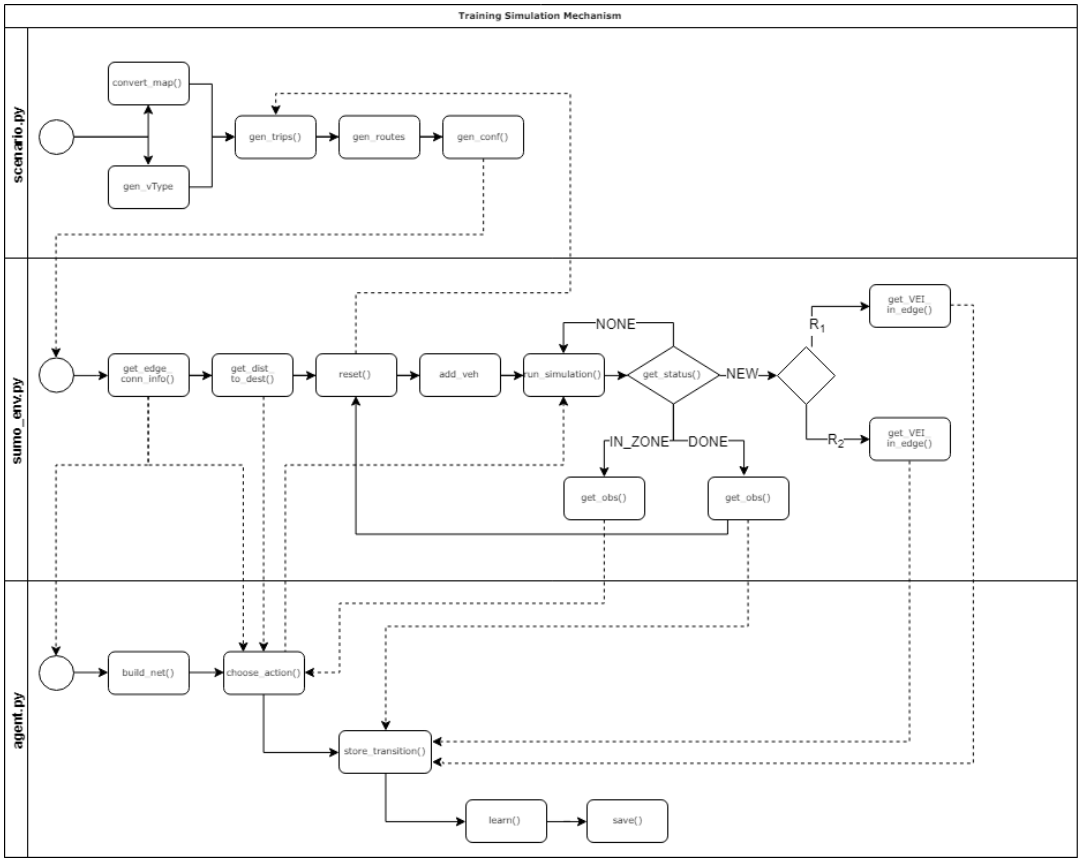


FIGURE 7.1: Training Simulation Mechanism Flowchart

step, *sumo_env* check the status of "*nav_veh*". There are four status defined in this proposed approach, which are IN_ZONE, NEW, DONE and NONE. The definitions of these 4 status are shown below:

- **IN_ZONE:** The *nav_veh* only is IN_ZONE status when it matches the following 3 conditions. 1) *nav_veh* is not in its destination road. 2) *nav_veh* front bumper is in decision zone of a road. 3) *nav_veh* has not been assigned a action to its next road by vehicle agent.
- **NEW:** This is only when *nav_veh* front bumper has just reached the new road.
- **DONE:** This is only when *nav_veh* has arrived at its destination (no longer exist in the network).
- **NONE:** This status indicates the rest of the situation except the above three special cases.

Environment keeps simulating until *nav_veh* is not NONE status. When *nav_veh* is IN_ZONE status, environment observes the state, and passes to agent class along with the road connection information and Euclidean distance to destination. Agent then choose an action and returns back to environment. Environment executes the action to *nav_veh* and monitors its status again. When status is NEW, the instant reward is computed depending on which

reward schema is used and passed to agent. Agent then stores the transition tuple $< s, a, r, s^{'} >$ as experience replay. Agent randomly selects mini batch from the experience replay as training data to train the DNN and save the model. When the status is DONE, environment observes the state again and sends to agent. After that it terminates the current simulation and resets the simulation to initial state. During this stage, scenario generates a new trips and routes files for the new simulation. And then the training process is repeated until the training episode is ended.

7.2.2 Scenario class definition

The *scenario* python class is mainly designed for creating the required files for SUMO simulator in order to run simulation in this experiment. The methods provided by *scenario* class are listed as follows:

- **convert_map(self, map):** This method is to convert OpenStreetMap (.osm) files to SUMO network file (.net.xml). It uses SUMO provided application *netconvert* to read common data like lists of edges and optional nodes from .osm file and convert it into a complete SUMO-network.
- **gen_vType(self, vType_list):** This method is to generate additional file (.add.xml) in SUMO to store the definitions of vehicle type that are used in this experiment.
- **gen_trips(self, duration, n_veh):** This method encapsulates SUMO provided *randomTrip* tool to generate the trip file (.trips.xml) which contains a list of origin/destination pair for individual vehicle.
- **gen_routes(self, method='dijkstra', accident=False):** This method is to generate routes file (.rou.xml) for simulation. It uses SUMO provided *duarouter* tool to define the route for each origin/destination pair in trips file accordingly.
- **gen_conf(self):** This method is to generate the SUMO configuration file (.cfg) for simulation. A sumo configuration file contains all the required parameters to start a simulation.

7.2.3 Building a Simulation with SUMO

This subsection presents the implementation of how to build a simulation for experiment. Figure 7.2 illustrates the SUMO traffic simulation process diagram.

Network building: To create a simulation for experiment, a network needs to be created. This experiment targets to optimise the vehicle route in urban transportation network, therefore all of the simulations are using real urban maps that have been converted from OpenStreetMap via *convert_map* method except the toy data simulation. OpenStreetMap is a valuable source for real-world map data which is totally free to be viewed and enhanced. For toy data simulation, the network is built by using a SUMO graphical network editor

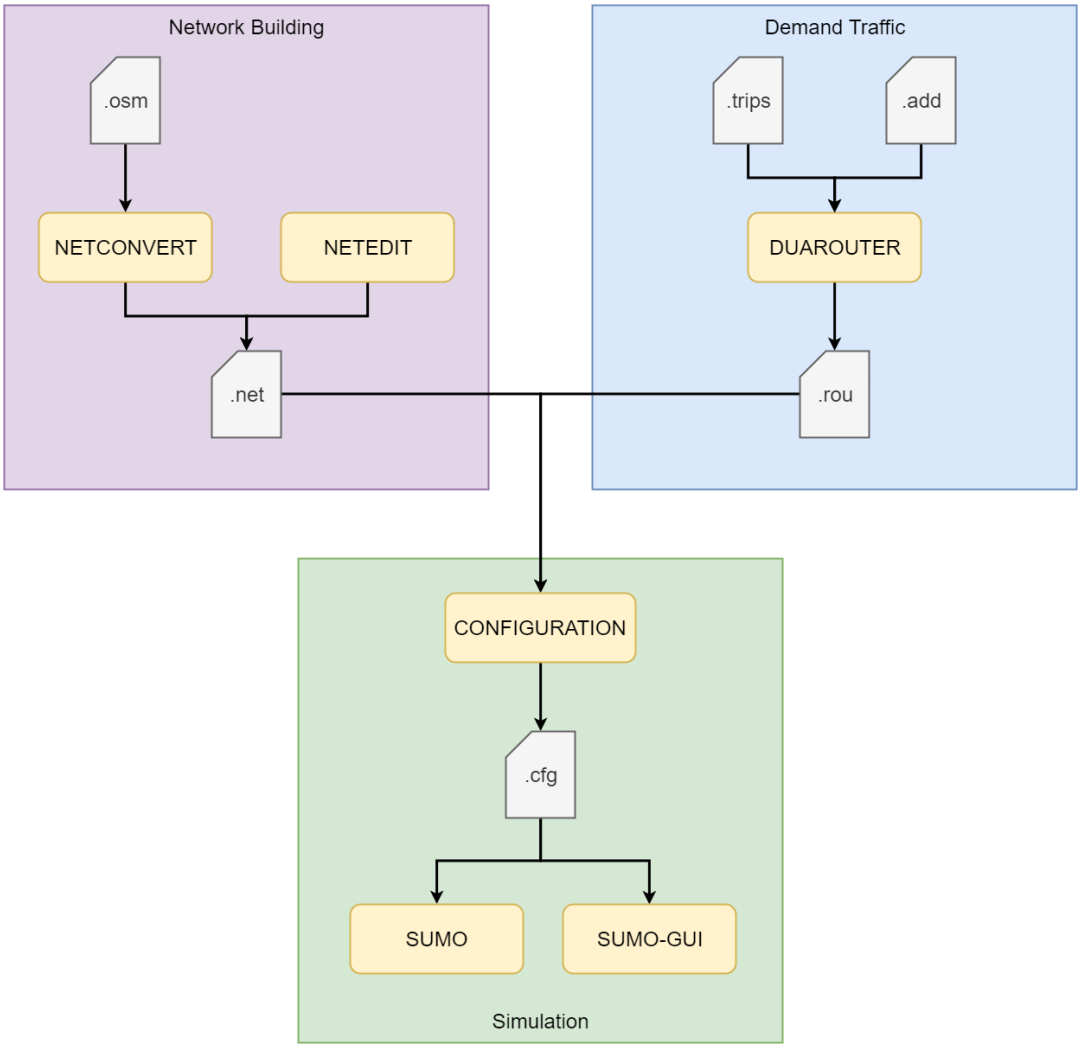


FIGURE 7.2: SUMO traffic simulation process diagram

NetEdit. Figure 7.3 and Figure 7.4 show the examples of converting an Open Street Map to SUMO map and the manual network designed by network editor *NetEdit*. Eventually, the SUMO networks are designed with the detail needed by microscopic road traffic simulations, which is ready for routing (navigation) purposes for individual vehicles.

Demand Traffic: Each vehicle in the SUMO simulation is defined explicitly since SUMO is a microscopic traffic simulator. They are given at least by a unique identifier, the departure time, and the vehicle's route through the SUMO network. A route is the complete list of connected edges between the origin/destination pair. A trip is defined as the trajectory of a single vehicle that contains the origin/destination pair and the departure time. The trip data is stored in *.trips.xml* file.

Moreover, vehicle's properties can be further categorised as vehicle type. The considered properties for the description of vehicle type in this experiment are described as follows:

- *id:* Unique identifier for this vehicle type.

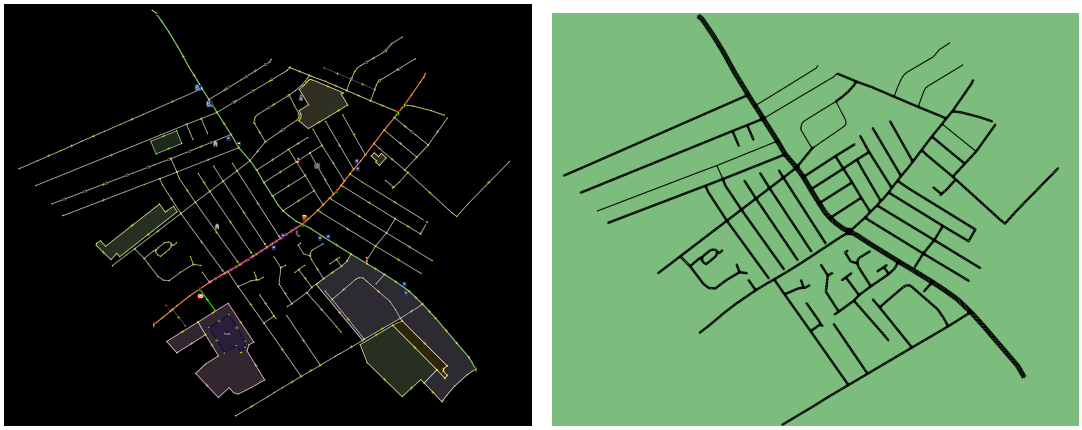


FIGURE 7.3: (a) openstreetmap (b) sumo map

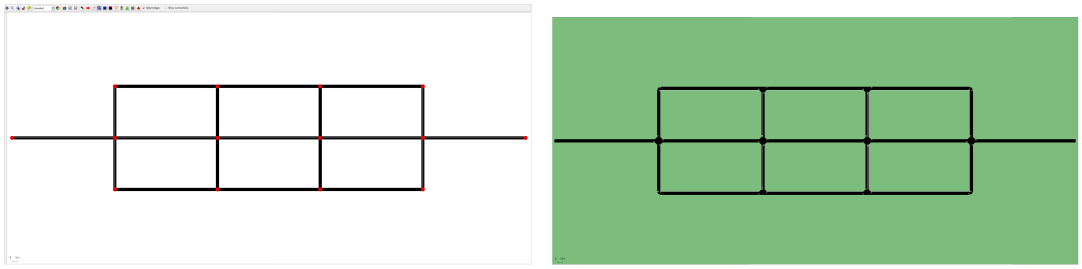


FIGURE 7.4: (a) NetEdit (b) sumo map

- *accel:* The acceleration ability of vehicles of the corresponding type.
- *decel:* The deceleration ability of vehicles of the corresponding type.
- *sigma:* An evaluation of the imperfection of the driver whose value is between 0 and 1.
- *maxspeed:* The maximum velocity of the vehicle.
- *color:* The colour for this vehicle type (only apply in SUMO-GUI).
- *probability:* The probability of the distribution for this vehicle type.

Two vehicle types are defined in this experiment which are "normal_car" and "truck". The definition details of these two vehicle types are displayed in table 7.1 and are stored in *.add.xml* file. Then, the *.trips.xml* and *.add.xml* are supplied to *gen_routes()* to generate the route file *.rou.xml* for traffic simulation.

Vehicle Type	Length	Accel	Decel	Sigma	Max Speed	Color	Probability
Normal Car	5.0	2.0	5.0	0.5	20.0	yellow	0.8
Truck	8.0	1.0	5.0	0.5	5.0	green	0.2

TABLE 7.1: Definition of vehicle type

Simulation: Once the the network file *.net.xml* and route file *.rou.xml* are ready, a SUMO configuration file *.cfg* could be created with the required parameters for traffic simulation. SUMO simulations only start when this configuration file is given. There are two types of traffic simulation provided by SUMO. The application called "*sumo*" is a pure command line application for efficient multi batch simulation. The application so called "*sumo-gui*" however offers a graphical user interface (GUI) rendering the simulation network and vehicles. For training speed wise, most of the training and testing are run in application "*sumo*". However, application "*sumo-gui*" will be used if the visual observation is needed.

7.2.4 Environment Class Definition

The environment class *sumo_env* is designed to encapsulate the use of SUMO simulator with TraCI. It enables the features to control SUMO including to initialise and interfere a simulation, define the state spaces and action spaces, reward calculation, and the action applicator to the simulation. The methods provided by *sumo_env* are presented as below:

- **get_edge_conn_info(self):** This method is to get and compile the connections between roads in SUMO network and store the information in memory. The road connection information is important in this experiment as it is used as the reference to indicate which road the actions in a specific road are linked to and determine the number of actions in each edge.
- **get_dist_to_dest(self):** This method is to calculate the Euclidean distances between the end point of each road to the destination and store the information in memory. The information is for the proposed Euclidean distances based exploration method.
- **reset(self)** This method is to reset the simulation to initial state in order to start a new episode. It returns to the initial state from the environment. An initial state will be returned in this method.
- **add_veh(self):** This method is to add the vehicle that uses vehicle agent for navigation. The id of the vehicle is a global value which is "*nav_veh*".
- **run_simulation(self, action=None)** This method is to process the simulation with or without applying an action. When *nav_veh* arrives in a decision zone of a road (method is_in_zone() is applied in here), this method returns an observation of the current state s_t, the number of available actions of current state, the reward $rt - 1$ for the previous state $st - 1$, and a flag to indicate the if *nav_veh* has arrived its destination. Note that unlike common DRL example, after applied action a_t, reward r_t does not return immediately. This is because the reward r_t could not be computed instantly.
- **get_status(self):** This method returns a status code to determine which status *nav_veh* is currently in.

- **get_obs(self):** This method gets the required parameters from environment and return the state matrix.
- **get_reward(self, done):** This method is to calculate the reward for *nav_veh*.
- **get_VEI(self):** This method is to calculate the VEI for *nav_veh* in specific road and return when VEI reward schema is applied. The equation for VEI is introduced in Equation 6.4.
- **get_emission(self):** This method return a list of total emissions (CO, HC, NOx, PMx) of *nav_veh*.

7.2.5 Data Extraction and Pre-processing

Environment class imports SUMO API TraCI to interact with SUMO. Although SUMO and TraCI can provide a powerful and high quality simulation for this experiment, there are still some issues on design and implementation that need to be further enhanced. This subsection presents how the environment class extracts and compile the data from SUMO simulation in order to compose the observation for state space, aggregate reward, and calculate vehicle emission and proposes an improved method to revise a problem in SUMO with infinity expected travel time. Besides, this subsection also presents the compilation of the data that is extracted from SUMO.

Data preprocessing is an integral step in DRL as the quality of data and the useful information that can be derived from it directly affects the ability of the trained model to learn. Therefore, it is extremely important to pre-process the data before feeding them into DRL agent class. One of the most common methods of data pre-processing is data scaling, which is a recommended pre-processing step when working with deep learning neural networks. It can be achieved by normalising real-valued input and output variables.

Normalisation is a technique often applied as part of data preparation for DRL. The goal of normalisation is to change the values of numeric columns in the dataset to a common scale, without breaking differences in the ranges of values [3]. For instance, rescaling of the data from the original range so that all values are within the range of 0 and 1. For DRL, normalisation is required when features have big different ranges. In this proposed approach, the four features which are expected travel time, vehicle number, the (x, y) coordinates of vehicle current road and the destination road have big different ranges. Consequently, normalisation is needed to be applied in order to make DRL achieve better performance. A typical way to perform normalisation is **Min Max Scaling**, a Min-Max scaling is done via the following equation:

$$x_{norm} = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{7.1}$$

where x_{min} and x_{max} are the minimum and maximum value of data set. From equation above x could only be in range 0 to 1 if its value is not outside the

bounds of the minimum and maximum values. Therefore, all the minimum and maximum values of the features need to be known in order to apply Min Max Scaling Normalisation. Fortunately, all these values could be computed from environment via SUMO API TraCI.

As mentioned above there are four features in state space, which are n_e the number of vehicle in the road, t_e the expected travel time in the road, and the c_v and d_v represent the current road of the agent and its destination. Data for feature n_e could be precisely retrieved via the method getLastStepVehicleNumber() in TraCI edge class. The absolute minimum value n_{min} is straightforward 0 which means that there is no vehicle in a road. However, its absolute maximum value means the biggest number of vehicles that could possibly occupy the longest road. The equation for computing n_{max} is shown as follows:

$$n_{max} = \frac{l_{max}}{\ell_{Vmin} + mingap_{Vmin}} \tag{7.2}$$

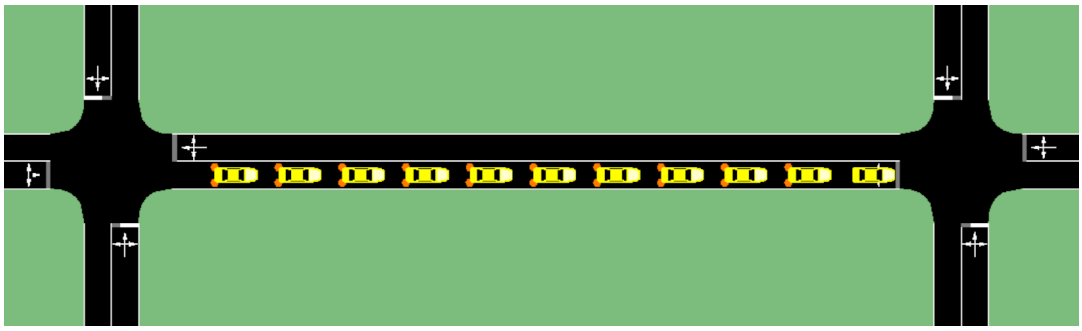
where l_{max} is the road with biggest space and ℓ_{Vmin} is the length of vehicle with shortest length and $mingap_{Vmin}$ is the minimum gap of the vehicle. The length and minimum gap of vehicle could be retrieved by mehtods getLength() and getMinGap() via TraCI vehicle class.

To retrieve the current road of the agent and its destination is also straightforward, TraCI edge class provides method getShape() to return the coordinates of the middle point of road. And the absolute minimum and maximum number for the (x, y) coordinates could be simply defined by extracting the coordinates of the network boundary, which could be retrieved by method getNetBoundary().

The data for t_e expected travel time in the road however is more tricky. Although TraCI provides method getTraveltime() in edge class, which calculates travel time by dividing the road length by the mean vehicle speed in the road. Therefore, this method is not truly accurate as it does not consider the infinity travel time problem when the mean vehicle speed is equal to zero. The disadvantages are presented based on three aforementioned cases when using SUMO as illustrated in Figure 7.5. In SUMO, all three cases return 1000000 expected travel time which are incorrect as they are totally different scenarios.

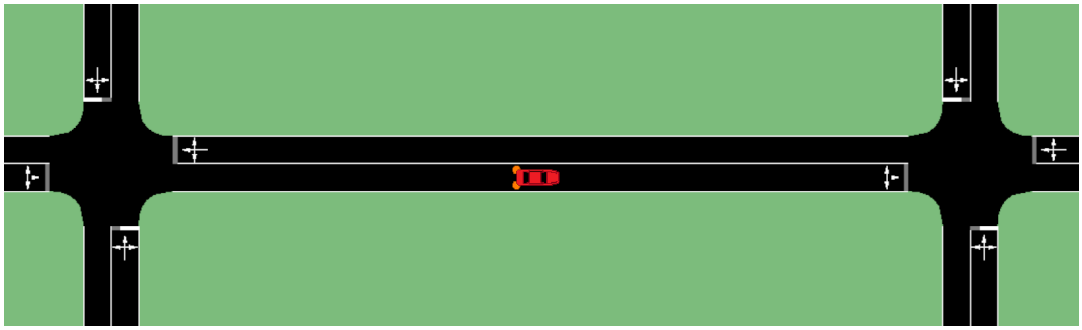
To overcome this problem, the following methods are applied. The first case scenario could be determined if getLastStepOccupancy() in lane via TraCI is 100%, for this case the expected travel time is still calculated by dividing the road length by the mean vehicle speed in the road. The length and mean vehicle speed could be retrieved by getLength() and getLastStepMeanSpeed() via traci lane class. A minimum mean vehicle speed 0.1 is set to prevent infinity travel time. Therefore, if two roads are both fully occupied, the expected travel time should be less in the road with shorter length, which reflects closer to the real scenario. In the second scenario, the road segment is divided according to its occupancy, and minimum vehicle speed 0.1 is set to the vehicle that is in the end of the road. Thus, for the unoccupied part we use the road maximum

speed to calculate the expected travel time while we use average vehicle speed for the last segment with vehicle. Then add both to get the final travel time. The road maximum speed can be retrieved by getMaxSpeed(). The third case works similarly to the second case by dividing the road segment. However, for the vehicle that stops in the middle of road at the accident, a constant "*accident clearance time*" which indicates how long this accident will take to be cleared, is used as the expected travel time. For normalisation, the absolute minimum value in expected travel time will be the smallest value among road length divided by road max speed. However, its absolute maximum value will be the longest road l_{max} divided by the minimum vehicle speed 0.1. Table 7.2 shows the comparison of the expected travel time.



(A) First case: road fully occupied

(B) Second case: Waiting in the junction



(C) Third case: Accident in the middle of road

FIGURE 7.5: Three cases showing the limitations of the travel time calculation using SUMO

To retrieve the action value, the approach is the same as the experiment in chapter 5. All the connected road details are stored in a python dictionary. The

Scenario	traci.edge.getTraveltime()	Improve Calculation
Case 1	1000000	856.0
Case 2	1000000	35.6
Case 3	1000000	55.7

TABLE 7.2: The comparison of expected travel time calculation from SUMO and proposed approach

connected roads could be provided by method getLinks() in TraCI lane class. Similarly, the Euclidean distances info between the end point of roads and destination are also stored in another python dictionary. Lastly, for travel time reward schema, the current time step could be retrieved by getTime() in TraCI simulation class. Meanwhile, for VEI reward schema, the vehicle emissions for CO, HC, NOx and PMx could be retrieved by methods getCOEmission(), getHCEmission(), getNOxEmission(), getPMxEmission() in TraCI vehicle class respectively.

7.2.6 Benchmark Methods

This section introduces the benchmark methods that are used in this experiment. The default traffic assignment method in SUMO is Gawron's dynamic user equilibrium (GDUE) [36]. GDUE uses Dynamic traffic assignment (DTA) to model the traffic via a discrete time dependent network. It assigns routes for all trips using some shortest path algorithms (e.g. Dijkstra algorithm or A* algorithm) as an initialisation step by taking the edge length as edge cost. After running the traffic simulation, it records the actual travel time on each edge, then uses the same shortest path algorithms to re-assign the routes. This step is done iteratively until the edge cost for all roads is relatively converged.

SUMO also provides automatic routing to perform dynamically routing in a running simulation. This routing approach works by giving some or all vehicles the capability to re-compute their route periodically or in specific time. The routing takes into account the current and recent state of traffic in the network and thus adapts to jams and other changes in the network. Based on this method, specific vehicle could be re-routed dynamically while a simulation is running. In order to further demonstrate the performance of the proposed method, the SUMO auto-routing methods which use Dijkstra and A* algorithm are applied to compare the performance in travel time and VEI with the trained models. The vehicle emissions are also recorded for comparison. The benchmark methods could be done via method rerouteTravelTime() in TraCI vehicle class. I name these two benchmark methods as dynamic-Dijkstra and dynamic-A* in this experiment.

7.2.7 DRL Agent Class Definition

The DRL agent class *agent* is to interact with the environment through the environment class *sumo_env* and use the implementation of the DRL algorithms

to train a model in order to optimise vehicle's routing selection. It imports Tensorflow library to implement the DQN learning process that described in chapter 3. The methods provided by *agent* are given as below:

- **build_net(self):** This method is to build the DNNs for agent. The number of input and output of the neural network varies depending on the network characteristic, such as the number of total roads, or the maximum connected roads. It created two DNNs which are *q_net* and *target_net*. More details of the implementation of the DNN will be presented in the next subsection.
- **store_transition(self, sim_data):** This method is to store the simulation transition tuple $< s, a, r, s^{'} >$ into memory. The parameter sim_data is an array which keeps the transition from environment and pass to agent.
- **choose_action(self, obs, n_actions, e_distances=None):** This method is to decide which action to take based on the observation of the environment. The exploration function with ϵ greedy value and the proposed Euclidean distances are applied here. The parameter obs is the observation from the environment, n_action is the number of available actions in this state, which is equal to the number of connected roads and e_distances is an array which keeps the Euclidean distances from the end point of the connected road to the destination. More details of the implementation of the action selection policy will be introduced in subsection 7.2.9.
- **learn(self):** This method is to train the agent to learn the policy by minimising the error in Bellman's equation on a batch sampled from experience replay buffer and compute TD-error. It is also responsible for updating exploration ϵ greedy value.

7.2.8 DRL Agent Architecture

This subsection presents the DQN architecture in DRL agent class. The architecture was built on top of the implementation of TensorFlow library. Figure 7.6 illustrates the overview of the DQL agent. Two DNNs textiteval_net and *target_net* are created where *eval_net* is for estimating the q value in each state to decide which action to apply, and textittarget_net output the target q value for calculating the loss for training. In every certain learning step, the parameters in *target_net* will be replaced by the parameters in *eval_net*. Moreover, the model will be saved by the save component.

Both *eval_net* and *target_net* have the same structure. As illustrated in Figure 7.7 and Figure 7.8, *eval_net* takes state s as input and *target_net* takes next state $s\_$ as input. Both DNN have 3 hidden layers, where l1 is the first layer that contains 150 neurons and l2 is the second layer that contains 100 neurons. Relu activation function is applied in these hidden two layers. The last layer is the dueling layer that has an advantage and value streams. Advantage estimates the advantage for each action and value estimates the value of specific state. The advantage and value streams then will compute the q value as the output.

Lastly, Figure 7.9 illustrates the component of train and loss in DRL agent class. Adam optimiser is applied in this experiment as the optimiser to minimise the loss. Gradient represents the gradient decent, beta1_power and beta2_power represent the hyper-parameter β_1 and β_2 for Adam optimiser.

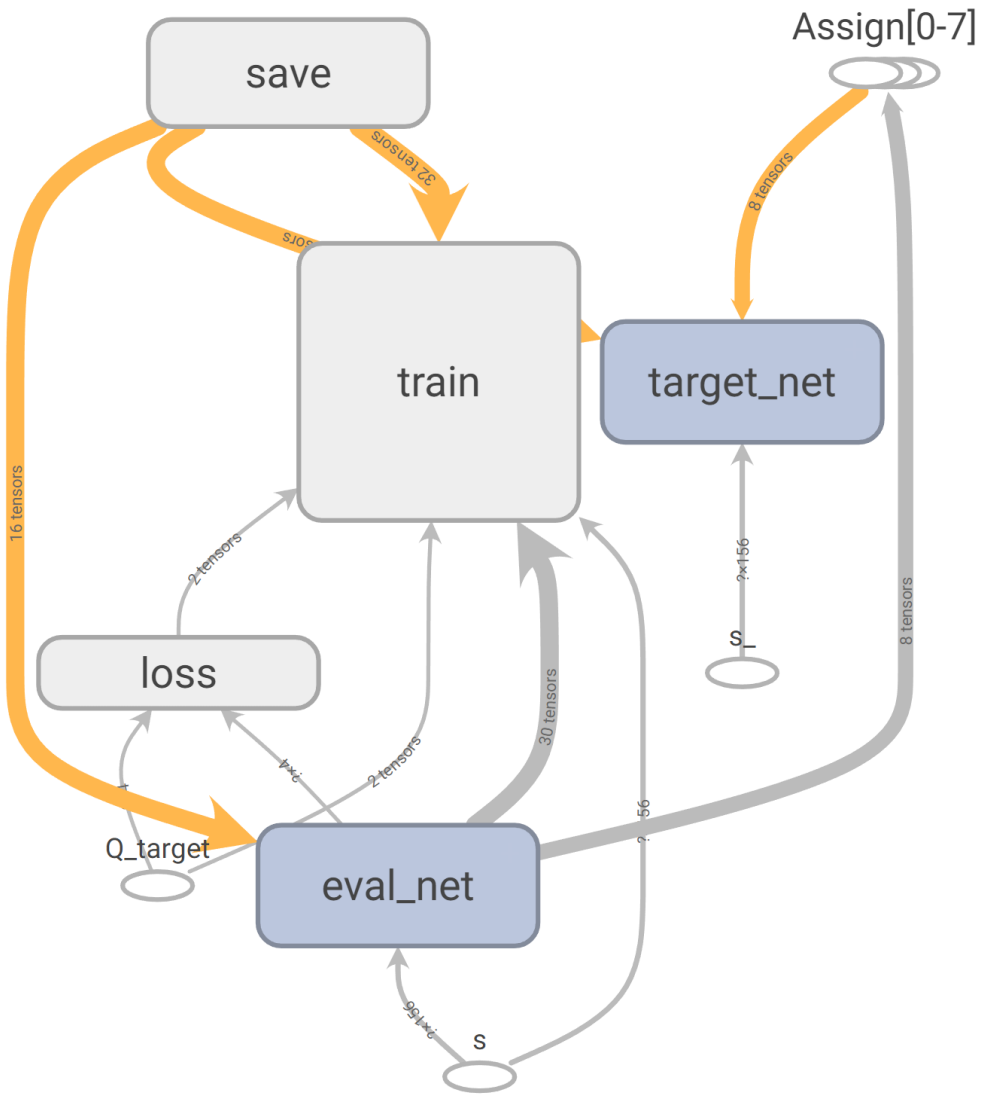


FIGURE 7.6: Overview of DQN architecture

7.2.9 Action Selection Policy

This subsection presents the implementation of the action selection policy in DRL agent. The DRL agent makes a decision accordingly based on the observation it receives. An aforementioned two-stage exploration scheme is designed to improve the network convergence as well as the converging speed. With probability ϵ, the agent decides to make exploration rather than take the action that is made by the current DNN. In this stage, there are two possible ways

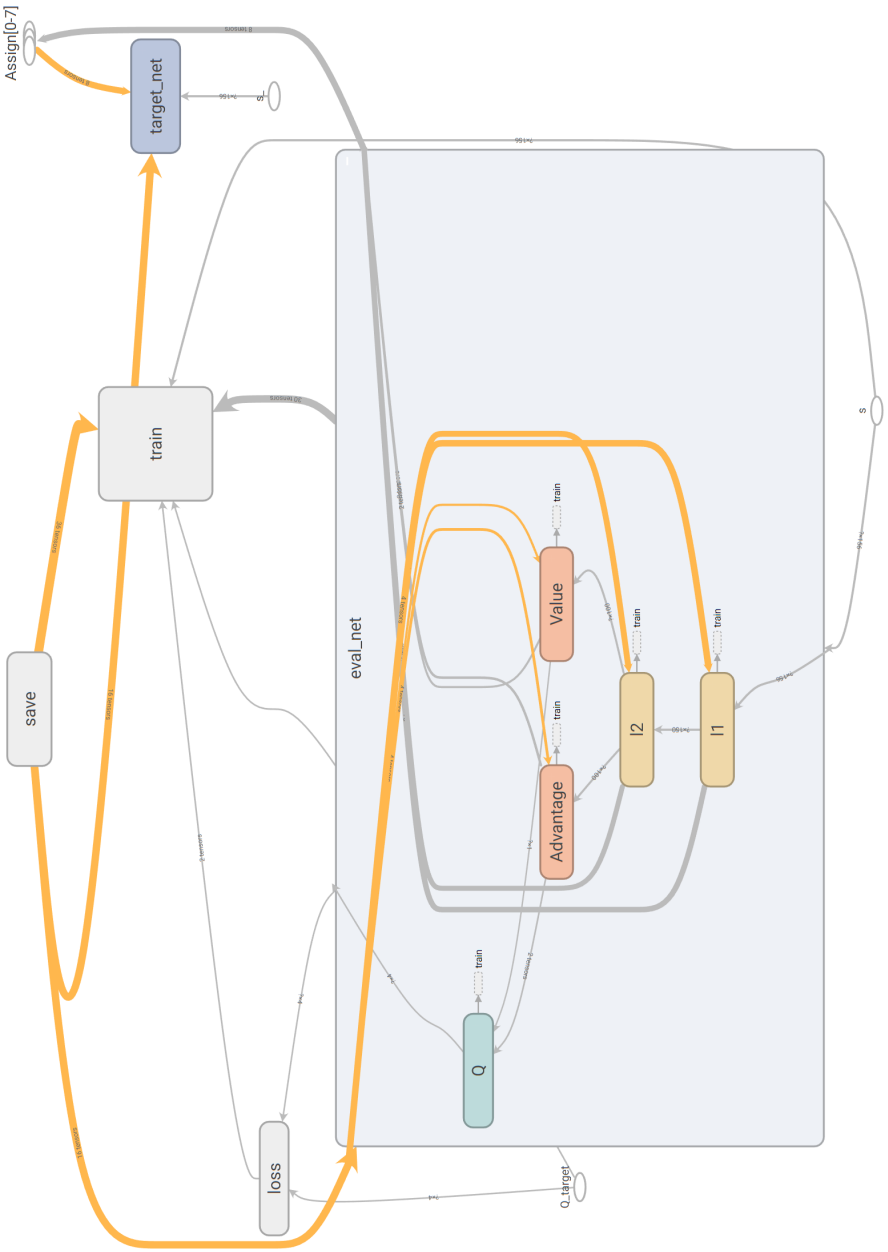


FIGURE 7.7: Eval network architecture

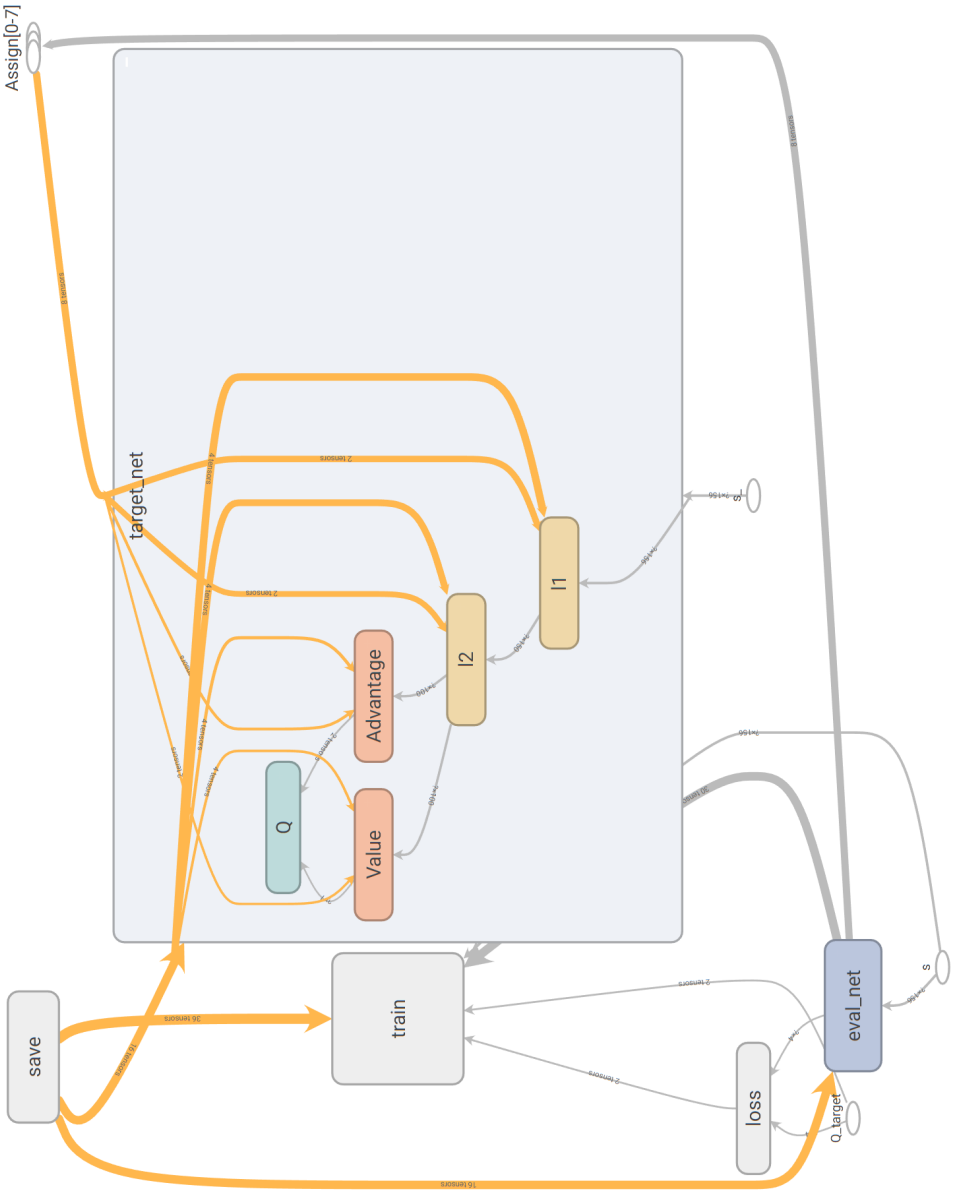


FIGURE 7.8: Target Network architecture

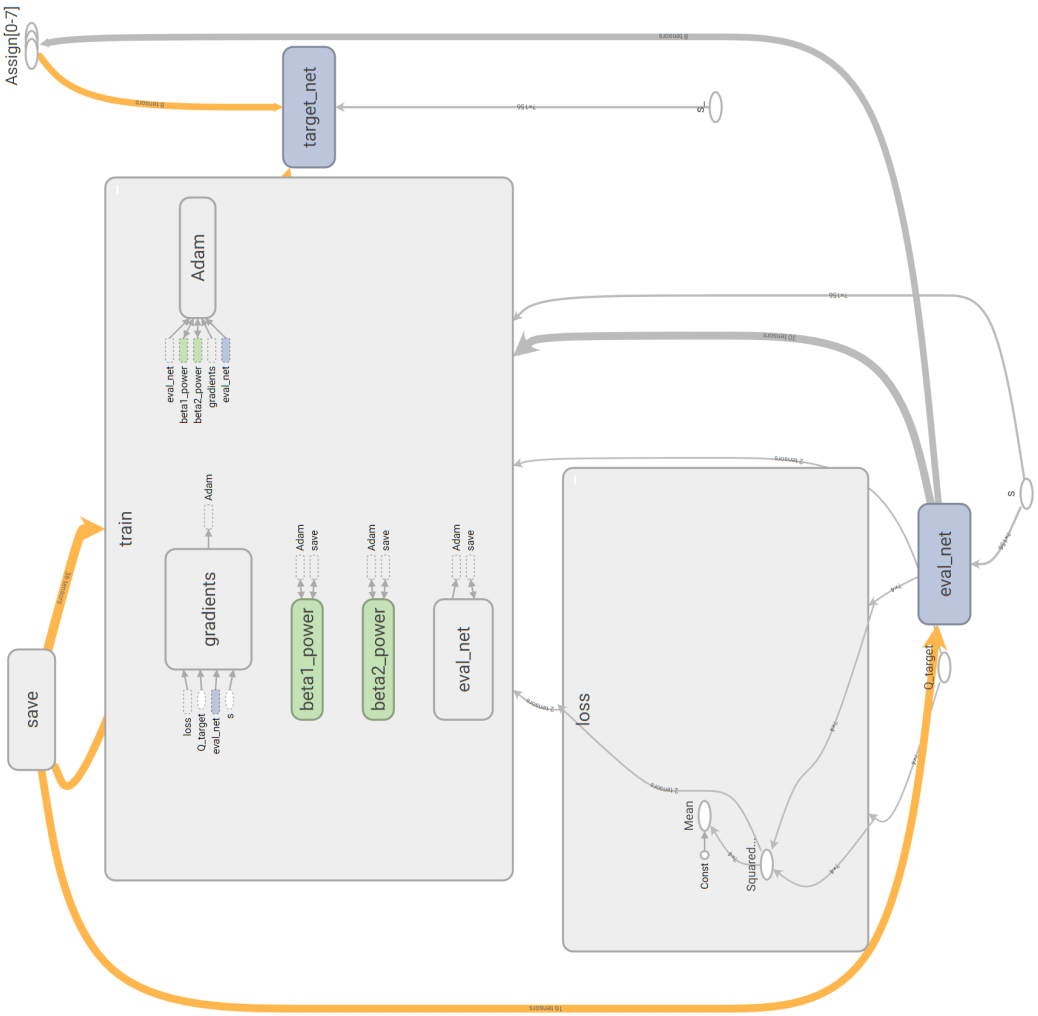


FIGURE 7.9: Train and loss architecture

to do exploration. One way is using the traditional way that selects an action with an even chance. In the other case, with a constant probability δ, the agent explores the network based on Euclidean distances policy. The road connection information and their Euclidean distances to a destination that is defined by the environment class will be passed to the DRL agent. Based on this information, the DRL agent computes the probabilities of selecting the connected road by following the algorithm that is presented in Equation 6.10. Consequently the DRL agent randomly selects a connected road as the explored road based on the probabilities that are just computed.

Moreover, the number of DNN outputs is dependent on the maximum number of connected roads in the network. However, in most cases roads in the network do not connect to the same number of roads in a network. Therefore, when the DRL agent decides to use the action that is based on DNN output, with the q values estimated by DNN, the DRL agent only selects the action with the highest q value among the number of connected roads and ignores the extras. For instance, assuming the maximum number of connected roads of a network is 4, and road AB only connects to road BC, BD and BE (3 actions). Based on the observation, DNN estimates 4 q-values $[q_1, q_2, q_3, q_4]$ as output. In this case, DRL agent only selects the highest q-value among $[q_1, q_2, q_3]$ and ignores the q_4 as it does not apply to any connected road.

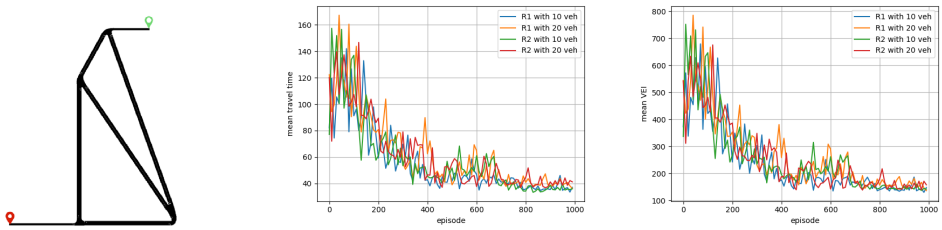
7.3 Experimental Evaluation

There are two subsections in the experimental evaluation: Firstly, two toy data maps are generated for testing the convergence of the intelligent agent. Additionally, the toy data simulation can further provide a tool to gain the insight of the decisions made by the intelligent agent during the navigation. Secondly, nine traffic conditions based on three regions in Liverpool city centre are simulated to demonstrate the efficiency of the DRLs with 2 proposed reward schemas. To further demonstrate the performance of the proposed method, the trained models are compared with the benchmark methods Dynamic-Dijkstra and Dynamic A* which are described in subsection 7.2.6 to evaluate their performance in vehicle route optimisation for sustainable and resilient urban transportation network.

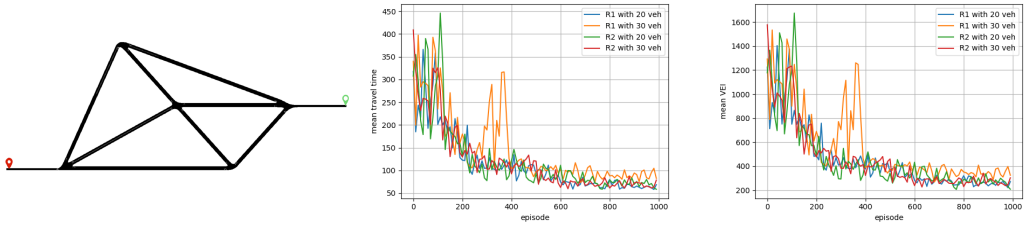
7.3.1 Toy Data

In the toy data experiments, two simple maps are built via SUMO graphical network editor *NetEdit* to train and test the intelligent vehicle agent. The first map has 12 edges and each edge connects with two other edges (m=2) as illustrated in Figure 7.10a. The second map has 18 edges and each edge connects with three other edges (m=3) as illustrated in Figure 7.10b. In the maps, two location icons (red and green icons) are used to show the starting point and the destination of the navigation tasks. In the experiments, two types of vehicles, normal cars and trucks, are injected into the maps to simulate realistic traffic

conditions by using *gen_trips()* and *gen_routes()* method in scenario class. The two types of vehicles have different maximum speeds, accelerations and decelerations. The number of vehicles is set at 10 and 20 and these vehicles are added into the maps randomly during the time stamp between 0 and 10.



(A) Simple map 1



(B) Simple map 2

FIGURE 7.10: Simple map structure and mean step in 100 episodes

In Figure 7.10a and Figure 7.10b, the middle and right sub-figures illustrate the convergence of the DRL agent during the training process with travel time based reward schema and VEI based reward schema. It demonstrates that all the cases converge in 1,000 episodes from random explorations. Two reward schemas have a similar convergence pattern and are able to significantly reduce the travel time and VEI. The converged mean travelling time is about 40 time steps in simple map 1 and about 70 time steps in simple map 2. Meanwhile the converged mean VEI is about 150 in simple map 1 and about 300 in simple map 2. Compared to the simple map 1, the average travel time of the agent trained in the simple map 2 is slightly longer as the agent has more action selections and state spaces at the start of the training stage, and simple map 2 has bigger map size with more roads. Therefore, simple map 2 has higher travel time and VEI level due to the different routing distance and traffic complexity. In the experiments, it is also found that the convergence of the run with more vehicles in the maps is slightly faster than the run with fewer vehicles.

When the decision network converges, it is capable of selecting optimal decisions to navigate its vehicle to the destination based on its observations of the current traffic states. The average decision making process timing is 2.7 millisecond and it means that the agent is suitable to make real-time decisions for the navigation task. The routing selected by using Dijkstra and A* methods is shown in Fig.7.11 (a). It is static and not able to be adapted to the volatile

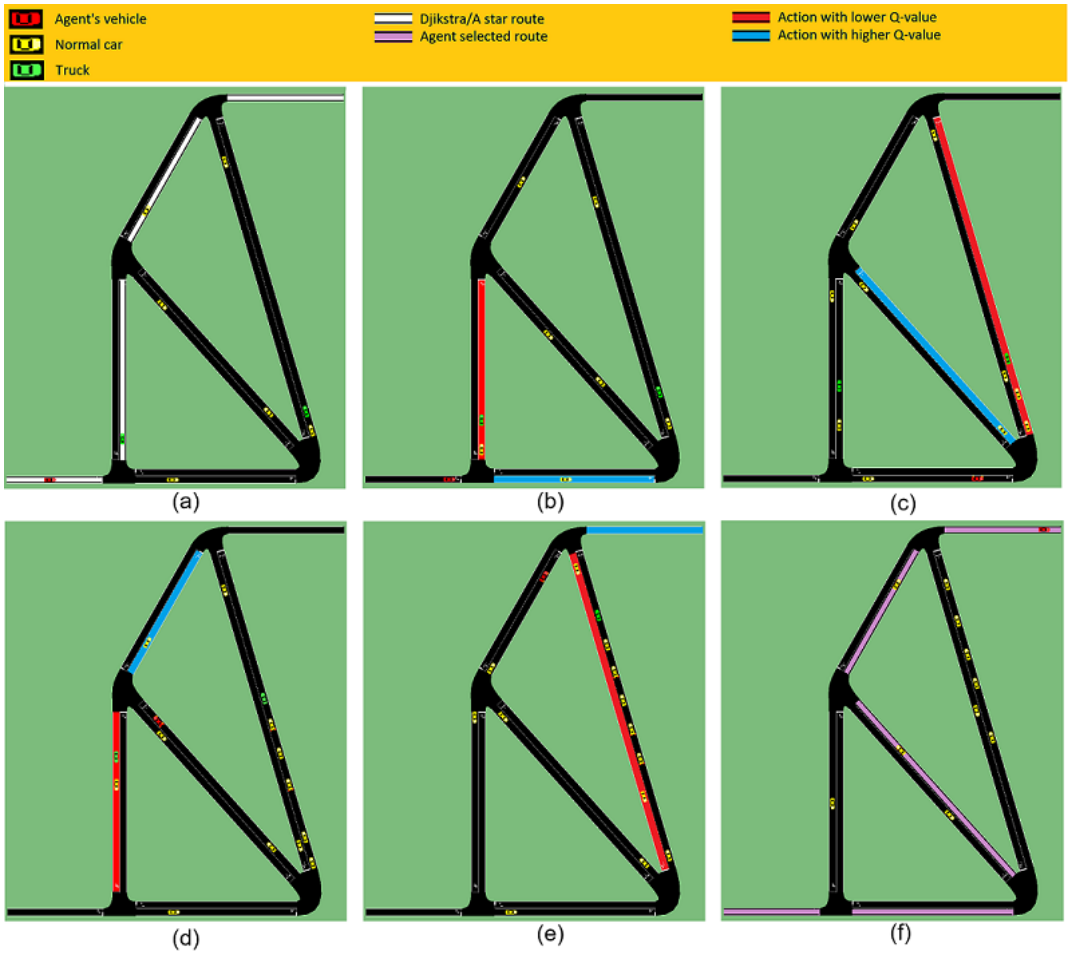
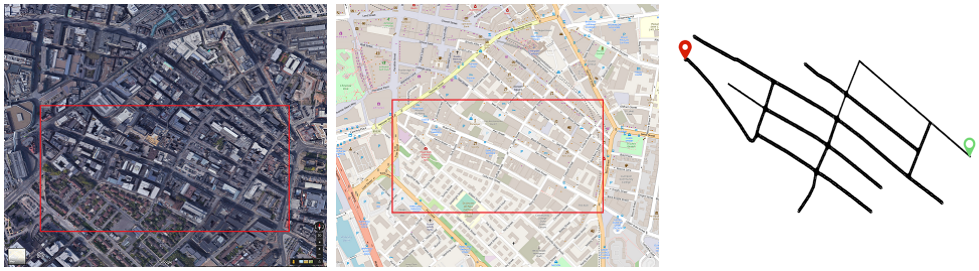


FIGURE 7.11: The simulated illustration of conventional Dijkstra/A* method and proposed method

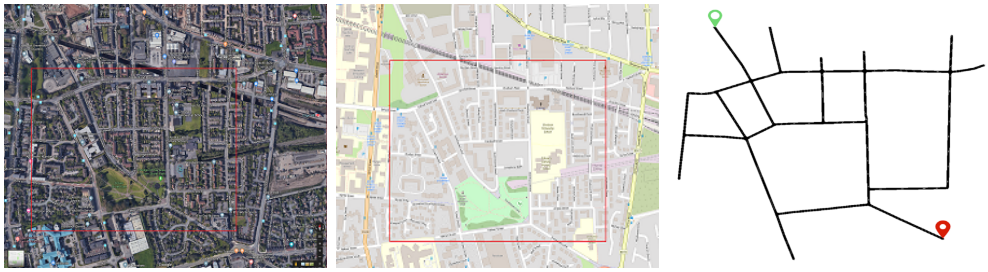
traffic states. However, the DRL based agent (illustrated from Fig.7.11 (b) to Fig.7.11 (e)) makes a flexible routing decision based on its observation when it approaches each decision zone. In Fig.7.11 (b), the Q value of the decision to travel straight is much higher than the Q value of the decision to turn left. It is consistent with an intuitive observation that a truck with lower speed is on the left edge. In Fig.7.11 (c), the selection of the left edge is reasonable as the vehicle number on the right edge is much more than the number on the left edge. The routing selected by the intelligent agent is illustrated in Fig.7.11 (f). In this demonstration, it takes 39 time steps to reach the destination by using the proposed algorithm while it takes 56 time steps when using the routing of Dijkstra and A*. In other words, the proposed navigation method improves 30.4% travel time in this case.

7.3.2 Realistic scenario analysis

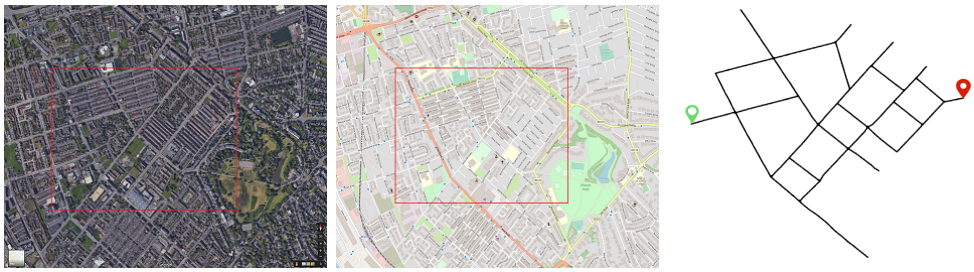
This thesis further tests the effectiveness of the framework in a more realistic scenario. In the experiments, two busy traffic regions in Liverpool city centre are selected on OpenStreetmap and converted into the SUMO maps in our



(A) Liverpool city map 1



(B) Liverpool city map 2



(C) Liverpool city map 3

FIGURE 7.12: Real world city map that are captured for SUMO simulation in this thesis

framework. The highlighted regions on GoogleMap, OpenStreetMap and SUMO generated maps are illustrated in Fig.7.12. In each map, three demand traffics with different number of vehicles are made accordingly to test the proposed DRL method in the integrated environment. The details of the map information are presented in Table 7.3.

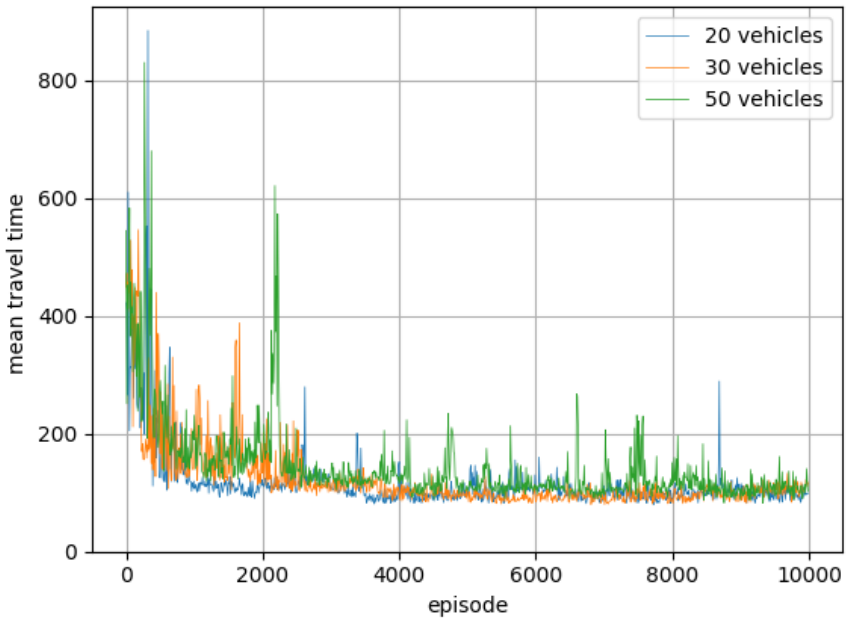
The convergence for two rewards schemas under different level of demand traffic is illustrated in Figure 7.13, Figure 7.14, Figure 7.15, Figure 7.16, Figure 7.17 and Figure 7.18. In these three maps, all the cases converge to certain levels ranging from 100 to 200 time steps or 500 to 600 VEI. This is highly correlated to the complexity of the maps as the converged average travelling time

	City Map 1	City Map 2	City Map 3
Total edges	40	60	80
Avg edge length (m)	107.79	143.23	173.95
Edge max speed range (mph)	20-30	30-50	30-50

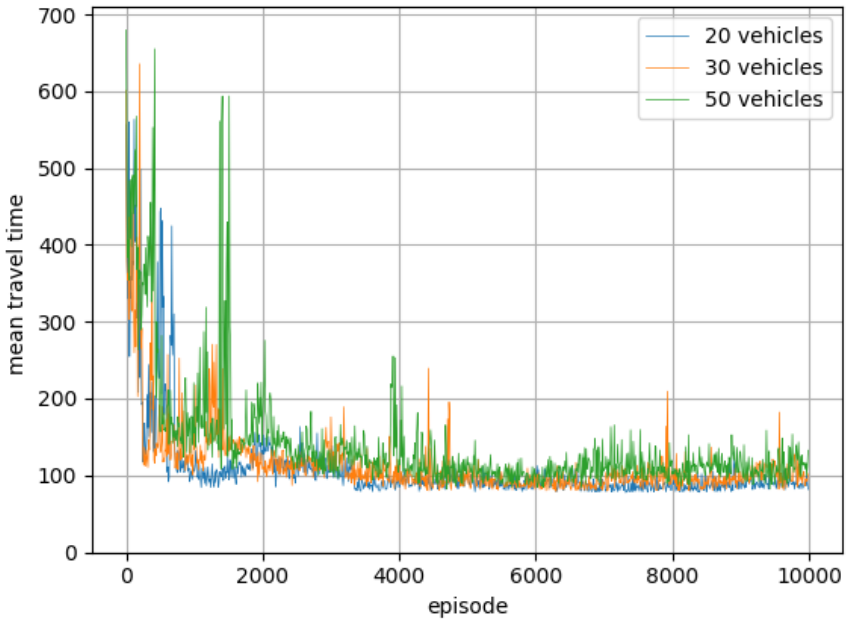
TABLE 7.3: Maps information

in the second map is much longer while compared to the time in the first map. Another finding is that the converging speed is relatively slower when there are more vehicles in the simulation due to the more volatile road conditions.

When the RL agent converges, 100 runs are made on each traffic condition to compare the two proposed reward schemas with the Dynamic-Dijkstra and Dynamic-A* methods objectively. The average travelling time, VEI and vehicle emissions of the runs and the standard deviation is presented in the Table 7.4. The results show that the proposed method outperforms the other algorithms in all traffic conditions. Furthermore, it is also proved that the improvement of the performance becomes much better when the road condition is more complex and volatile. The travel time based reward schema has the least average travel time compare to the others. However, VEI based reward schema outperform the others in the VEI and the vehicle emissions. For vehicle route navigation, it is very important to recognise the potential traffic congestion in order to arrive destination as fast as possible. From our experiment, we can see under a same city map, when the demand traffic is higher, it is more likely get sharply peak while training. This is because when the number of vehicle is bigger, more possibly certain roads in the map are having traffic congestion. And since the greedy policy is applied during the training, the vehicle agent might act greedy and randomly go into the road with heavy congestion and consequently cost much longer time than usual to arrive its destination. Besides, city map 3 had much bigger vibration in the beginning of convergence, the reason behind this is because city map 3 contain the highest number of edges and average edge length. Therefore while a vehicle goes on a wrong edge, it takes longer time than the other maps to finish that edge and arrive next junction to make another decision. According to the comparison table, in smallest map city map 1, our proposed method reduces at most 5.3%, 5.1% and 16.4% travel time with different demand traffic. However, in the largest map city map 3, our proposed method reduces at most 4.9%, 12.4% and 22.5% travel time in different demand traffic.

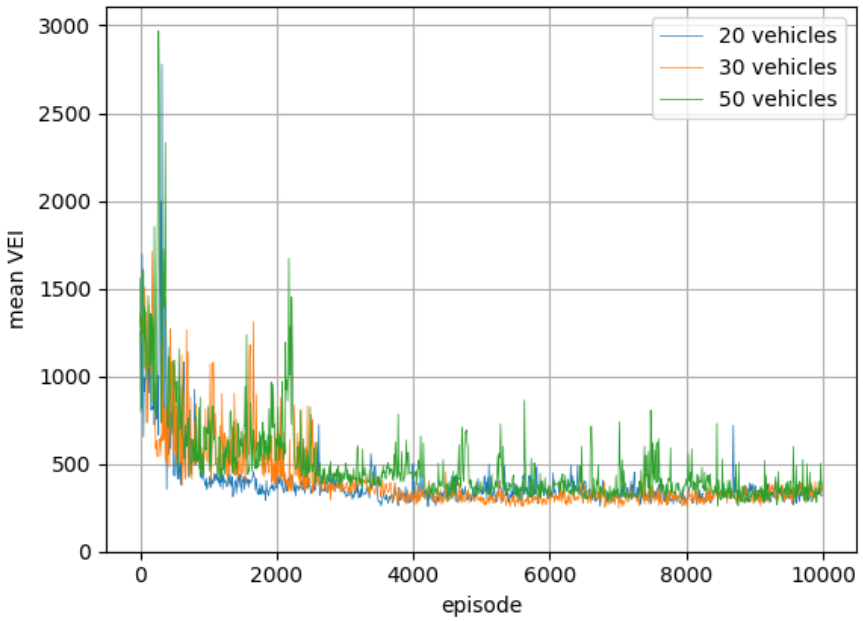


(A) travel time based reward schema

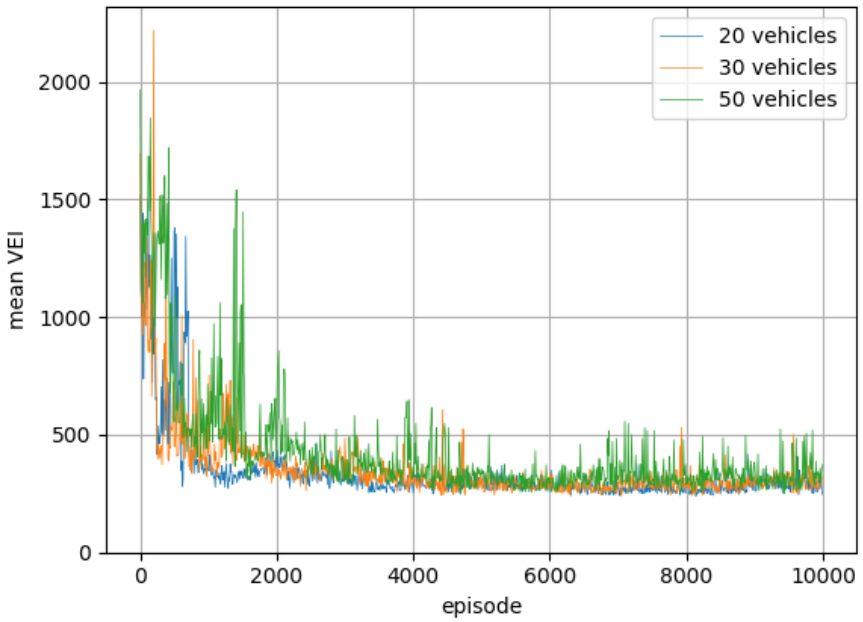


(B) VEI based reward schema

FIGURE 7.13: Convergence of the Avg. travel time in City Map 1

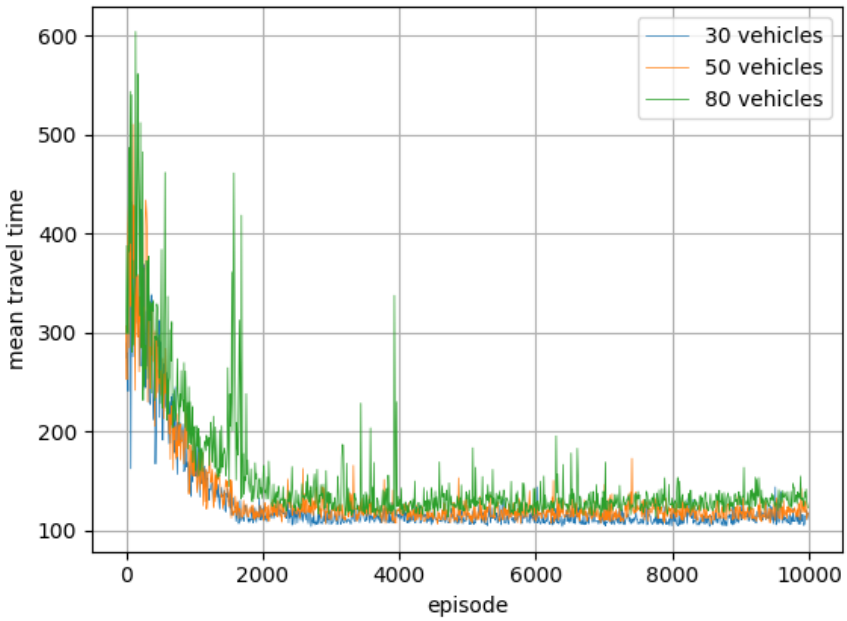


(A) travel time based reward schema

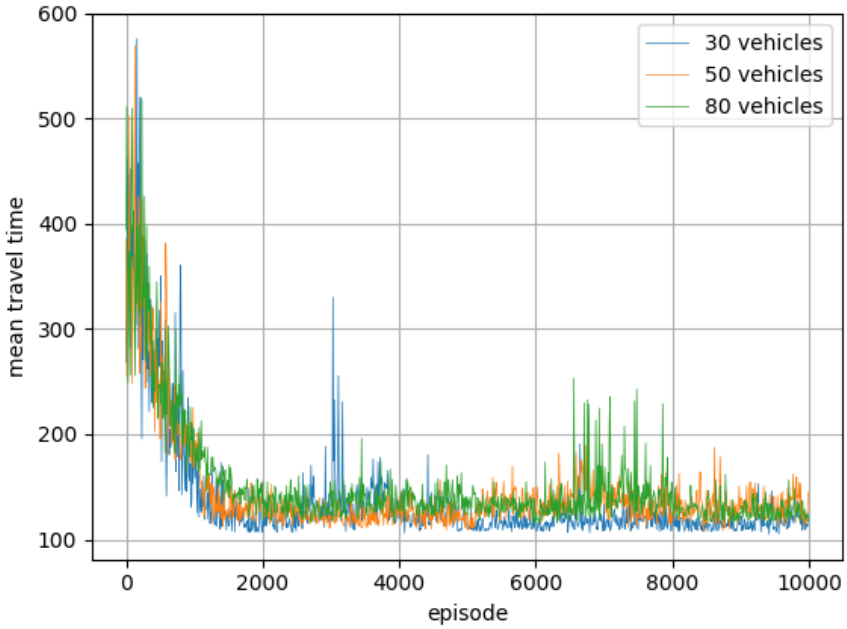


(B) VEI based reward schema

FIGURE 7.14: Convergence of the Avg. VEI in City Map 1

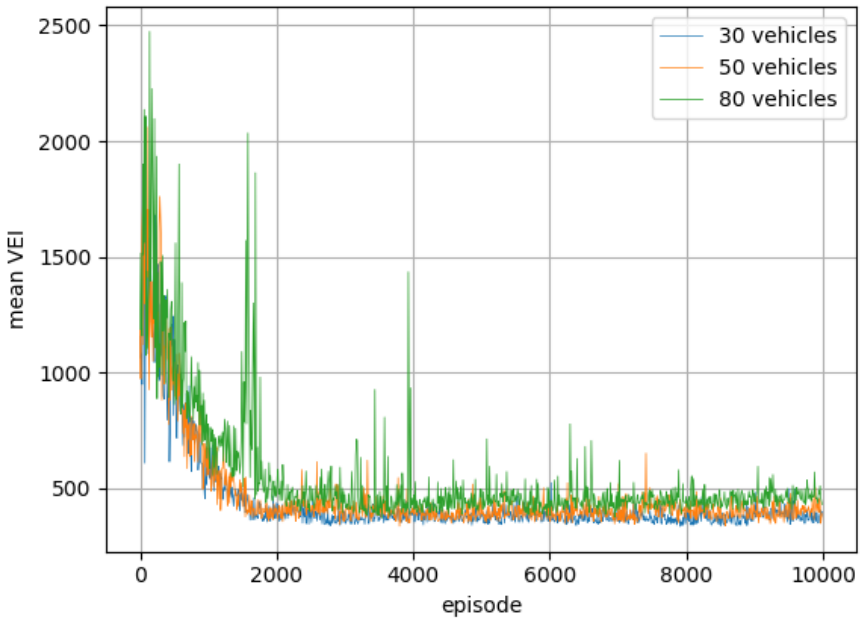


(A) travel time based reward schema

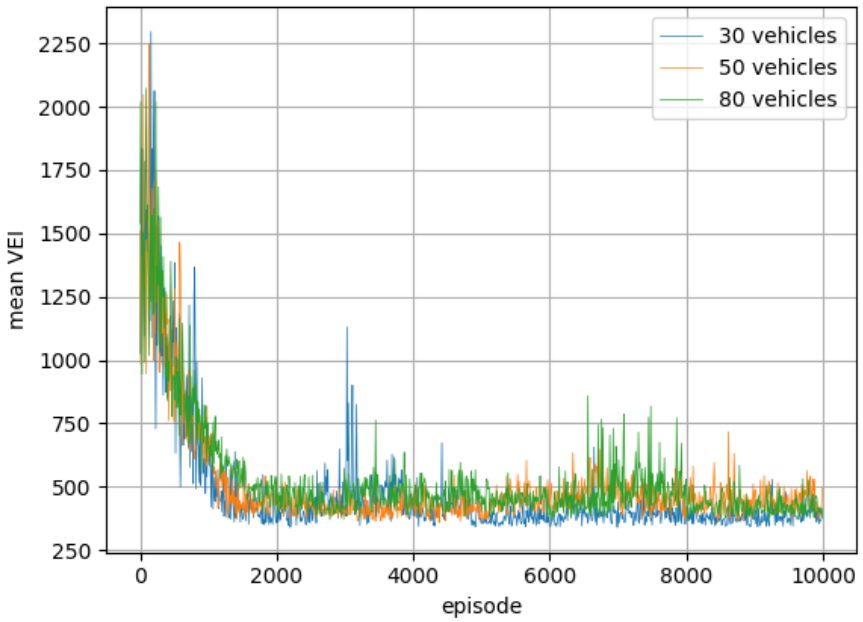


(B) VEI based reward schema

FIGURE 7.15: Convergence of the Avg. travel time in City Map 2

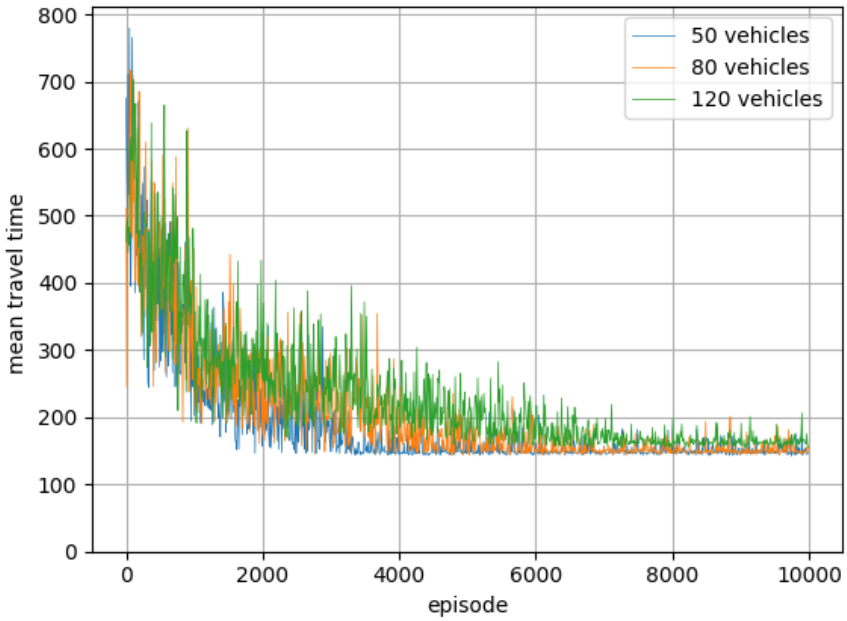


(A) travel time based reward schema

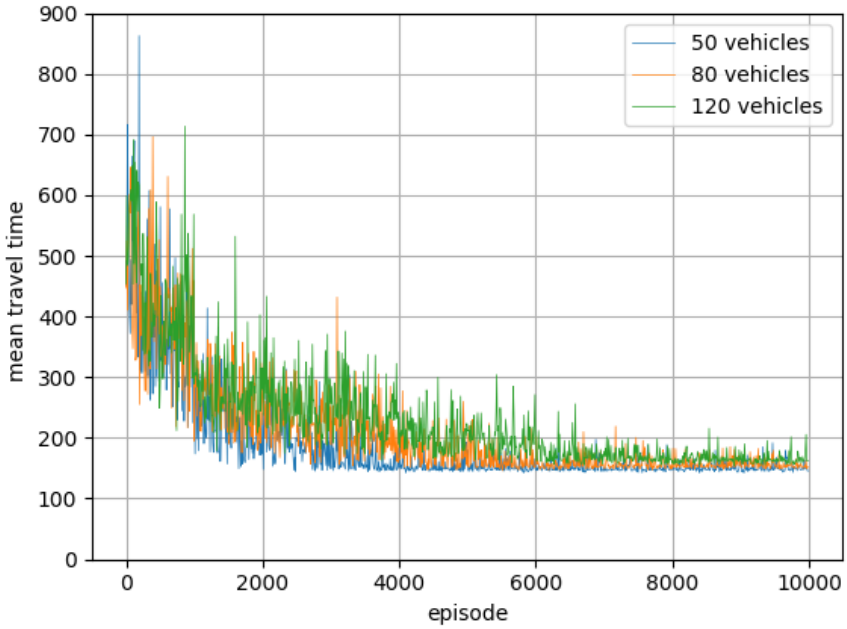


(B) VEI based reward schema

FIGURE 7.16: Convergence of the Avg. VEI in City Map 2

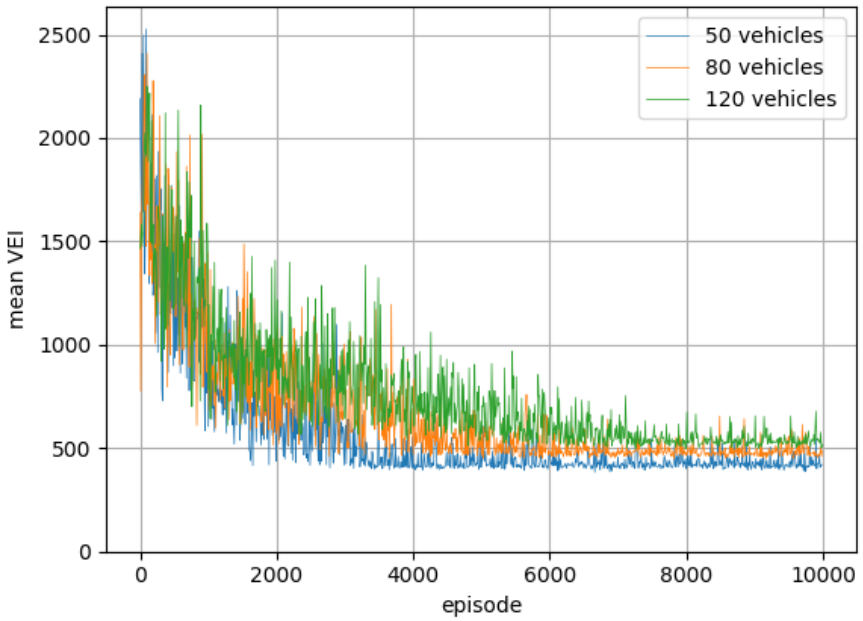


(A) travel time based reward schema

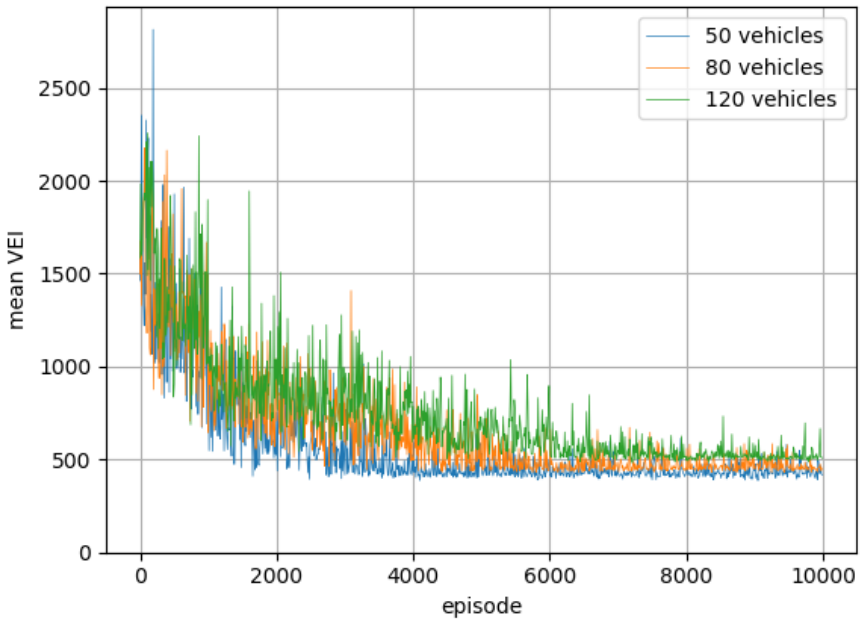


(B) VEI based reward schema

FIGURE 7.17: Convergence of the Avg. travel time in City Map 3



(A) travel time based reward schema



(B) VEI based reward schema

FIGURE 7.18: Convergence of the Avg. VEI in City Map 3

City Map 1							
Demand Traffic	Method	Travel Time	VEI	CO	HC	NOx	PMx
20 vehicles	Dynamic-Dijkstra	80.30 (±4.80)	288.29 (±23.11)	3924.11 (±430.28)	23.67 (±2.29)	101.62 (±6.75)	4.40 (±0.34)
	Dynamic-A*	83.00 (±6.32)	327.51 (±38.97)	4303.31 (±558.83)	25.86 (±3.16)	109.77 (±11.44)	4.78 (±0.55)
	Travel Time Based	**78.5 (±2.60)**	276.79 (±27.35)	3450.07 (±275.76)	20.98 (±1.63)	91.81 (±7.51)	3.90 (±0.39)
	VEI Based	80.0 (±3.70)	**247.57 (±14.69)**	**3231.44 (±291.04)**	**19.59 (±1.62)**	**84.50 (±5.29)**	**3.50 (±0.22)**
30 vehicles	Dynamic-Dijkstra	85.82 (±5.24)	286.08 (±21.17)	3942.32 (±444.53)	23.76 (±2.32)	101.59 (±5.93)	4.40 (±0.26)
	Dynamic-A*	92.54 (±5.11)	349.83 (±56.13)	4593.38 (±766.32)	27.68 (±4.40)	118.22 (±17.60)	5.11 (±0.79)
	Travel Time Based	**80.68 (±3.58)**	273.96 (±39.84)	3554.93 (±584.27)	21.49 (±3.28)	92.24 (±11.46)	3.89 (±0.56)
	VEI Based	81.62 (±3.99)	**247.08 (±12.63)**	**3239.62 (±287.96)**	**19.63 (±1.55)**	**84.46 (±4.49)**	**3.47 (±0.17)**
50 vehicles	Dynamic-Dijkstra	91.68 (±11.45)	316.58 (±37.67)	4853.75 (±1103.41)	28.30 (±5.51)	109.05 (±9.77)	4.77 (±0.49)
	Dynamic-A*	104.34 (±13.98)	368.86 (±59.79)	5825.03 (±1326.34)	33.33 (±6.92)	120.41 (±17.60)	5.28 (±0.85)
	Travel Time Based	**84.76 (±4.23)**	281.37 (±27.23)	3753.36 (±418.99)	22.50 (±2.35)	94.26 (±8.89)	3.97 (±0.45)
	VEI Based	86.64 (±4.06)	**261.01 (±15.89)**	**3566.23 (±264.34)**	**21.23 (±1.38)**	**86.96 (±3.99)**	**3.61 (±0.20)**

City Map 2							
Demand Traffic	Method	Travel Time	VEI	CO	HC	NOx	PMx
30 vehicles	Dynamic-Dijkstra	105.12 ((±4.49)	348.12 (±17.70)	4405.76 (±382.15)	27.23 (±1.97)	123.73 (±4.93)	5.24 (±0.24)
	Dynamic-A*	107.80 (±6.15)	347.91 (±30.50)	4386.96 (±670.74)	27.09 (±3.37)	122.64 (±6.91)	5.18 (±0.41)
	Travel Time Based	**103.46 (±2.23)**	339.87 (±13.93)	4044.67 (±311.56)	25.42 (±1.65)	120.54 (±4.45)	5.06 (±0.23)
	VEI Based	104.42 (±3.08)	**322.36 (±21.44)**	**3787.49 (±234.72)**	**24.00 (±1.23)**	**115.93 (±3.69)**	**4.82 (±0.19)**
50 vehicles	Dynamic-Dijkstra	109.44 (±9.60)	362.45 (±50.81)	4791.53 (±1253.79)	29.19 (±6.30)	127.47 (±12.11)	5.40 (±0.62)
	Dynamic-A*	110.74 (±10.27)	370.30 (±53.17)	4686.42 (±1076.19)	28.81 (±5.56)	129.13 (±12.93)	5.47 (±0.65)
	Travel Time Based	**106.04 (±3.57)**	341.92 (±21.62)	4076.18 (±305.60)	25.50 (±1.67)	119.27 (±5.57)	4.99 (±0.27)
	VEI Based	106.14 (±4.14)	**332.15 (±31.09)**	**3970.42 (±408.25)**	**24.91 (±2.15)**	**117.08 (±6.32)**	**4.89 (±0.31)**
80 vehicles	Dynamic-Dijkstra	127.54 (±8.33)	445.94 (±42.10)	7033.50 (±1271.80)	40.29 (±6.24)	145.32 (±10.07)	6.44 (±0.52)
	Dynamic-A*	124.16 (±14.67)	412.19 (±81.33)	5876.67 (±1807.11)	34.52 (±9.14)	135.24 (±19.64)	5.82 (±0.97)
	Travel Time Based	**118.36 (±7.59)**	392.87 (±32.44)	5080.88 (±1123.55)	30.59 (±5.49)	128.70 (±8.57)	5.48 (±0.47)
	VEI Based	121.3 (±9.73)	**385.36 (±27.75)**	**4966.95 (±837.62)**	**30.04 (±4.16)**	**128.45 (±7.99)**	**5.46 (±0.42)**

TABLE 7.4: The objective performance comparisons under various traffic conditions

City Map 3							
Demand Traffic	Method	Travel Time	VEI	CO	HC	NOx	PMx
50 vehicles	Dynamic-Dijkstra	149.86 (±12.45)	476.88 (±47.25)	6313.20 (±1110.57)	37.73 (±5.69)	156.16 (±12.79)	6.55 (±0.62)
	Dynamic-A*	151.00 (±9.99)	482.94 (±38.88)	6167.88 (±793.09)	36.94 (±4.12)	153.89 (±9.89)	6.41 (±0.49)
	Travel Time Based	**146.96 (±5.91)**	423.51 (±33.72)	4733.29 (±726.83)	29.83 (±3.79)	141.16 (±9.45)	5.68 (±0.44)
	VEI Based	148.22 (±5.97)	**416.73 (±24.23)**	**4610.60 (±350.12)**	**29.10 (±1.84)**	**138.25 (±6.54)**	**5.54 (±0.27)**
80 vehicles	Dynamic-Dijkstra	164.04 (±13.01)	521.94 (±34.04)	7423.78 (±959.05)	43.41 (±4.73)	167.75 (±8.34)	7.11 (±0.43)
	Dynamic-A*	168.82 (±17.28)	544.07 (±55.28)	7279.73 (±1325.27)	43.16 (±6.79)	174.42 (±16.55)	7.34 (±0.78)
	Travel Time Based	**148.64 (±6.31)**	477.58 (±38.69)	5397.59 (±578.58)	33.54 (±3.28)	154.14 (±12.63)	6.35 (±0.59)
	VEI Based	153.84 (±7.61)	**454.04 (±32.51)**	**5363.39 (±620.06)**	**32.97 (±3.23)**	**147.18 (±9.02)**	**6.02 (±0.42)**
120 vehicles	Dynamic-Dijkstra	193.56 (±9.60)	630.80 (±40.43)	10421.43 (±1174.41)	58.44 (±5.83)	194.66 (±10.06)	8.51 (±0.56)
	Dynamic-A*	203.22 (±28.70)	659.25 (±96.83)	10223.98 (±1390.89)	58.10 (±7.51)	203.17 (±33.08)	8.75 (±1.34)
	Travel Time Based	**160.62 (±8.83)**	523.54 (±40.52)	6295.46 (±957.38)	38.24 (±4.80)	165.26 (±9.31)	6.84 (±0.52)
	VEI Based	162.9 (±11.52)	**510.33 (±32.04)**	**6004.18 (±665.11)**	**36.76 (±3.40)**	**161.96 (±8.34)**	**6.63 (±0.40)**

Chapter 8

Conclusion and Future Work

8.1 Overview

This chapter concludes this thesis by recalling the investigated research problem, summarising the contributions and significant achievements, and then discusses the remaining research issues related to this work and propose the recommendations for future work.

8.2 Problem Overview

Global urbanisation has brought a lot of positive impacts on human society, it creates an increasing number of opportunities for jobs, education, and healthcare that benefit more and more people. In the past 60 years, global urbanisation has resulted in enormous economic growth, and concentration of population in densely populated cities. However, urban traffic congestion has risen as global urbanisation leads to the growing number of vehicles in urban transportation networks. It brings major impacts on urban transportation networks that lead to extra travelling hours, increased fuel consumption and vehicle emissions that cause air pollution.

Although there are various research projects that have been carried out to resolve recurrent traffic congestion, to the best of my knowledge, there is still a research gap to be filled for the non-recurrent traffic congestion problem. Non-recurrent traffic congestion is often caused by emergency events, such as accidents, road events and so on. Currently, most vehicle navigation systems struggle to response instantly to the non-recurrent congestion problem and lack the ability to self-evolve and to adapt rapidly to change in the urban transportation networks. Consequently, this causes serious traffic congestion problems and leads to environmental damage. Therefore, the objective of this thesis is to propose a self-evolution vehicle route optimisation approach by using the deep reinforcement learning method to re-route a vehicle to its destination and adapting the complexity of the urban transportation network in order to avoid traffic congestion.

8.3 Contributions and Achievements

To address the unexpected urban traffic problem for sustainable and resilient urban transportation network, this thesis proposes a self-evolution, novel adaptive approach for vehicle route optimisation via the DRL method which is able to navigate vehicle response to non-recurrent traffic congestion. There are three main contributions in this thesis as summarised as below:

- **Design of a novel framework to facilitate the vehicle route optimisation research under complex urban transportation context.** This thesis proposes a novel framework to provide an accessible way to optimise the vehicle route planning problem using DRL methods. It enhances SUMO simulator in order to make it more suitable for optimising vehicle route selection with DRL algorithms. The enhancements include providing an improved calculation method for expected travel time in a road depending on different circumstances, defining the segments in each edge to indicate the best timing for obtaining state and converting the SUMO network graph to a dual graph to model the states and actions in an urban network. The hand-designed controllers in the proposed framework enable the interaction between environment and external RL library through SUMO API TraCI, to allow model training in a rich environment with complex dynamics for vehicle route optimisation. Therefore, the DRL model could be trained across road networks of different size, density, number of edges and lanes. The demand traffic, network characteristic or vehicle behaviour in the experiments can be easily monitored and controlled. Besides, the state space and reward functions can be constructed from the environment. This framework makes a more realistic and interactive environment by embedding the smart agents (DRL models) into the traffic simulator and the extensibility of the framework provides huge flexibility to extend the features of the framework for future RL problems.

- **Design of effective observations, reward scheme and DRL algorithms to achieve efficient convergence of the DRL training** This thesis describes an effective observation as the representation of current traffic condition within a specific area of urban network. The representation variables contain multiple parameters reflecting the circumstances in the global urban transportation network to precisely describe the complexity of its dynamics. Besides, this thesis proposes an algorithm to measure the impact of individual vehicles on air pollutant emission called VEI. VEI takes several vehicle emissions as input such as nitrogen oxides, hydrocarbons, carbon monoxide, and particulate matter to compute the level of impact. Based on that, this thesis proposes two reward schemas to train the DRL model for vehicle route optimisation. One reward schema aims to reduce the total travel time of a vehicle, and another one aims to minimise the VEI from a vehicle. The results show both reward schemas are efficient to optimise the vehicle route. Furthermore, a Euclidean distance based exploration method is proposed in this thesis to combine with

the traditional ϵ greedy exploration, the result shows that it achieves more efficient convergence of DRL training than the traditional method.

- **Integration of the proposed vehicle route optimisation approach with real urban map to achieve a more sustainable and resilient urban transportation network.** The proposed vehicle route optimisation approach in this thesis aims to improve the sustainability and resilience in urban transportation networks. The two proposed reward schemas are applied in real urban networks with different size and different level of demand traffic in order to evaluate their performance. Although both reward schemas are able to optimise vehicle route significantly to avoid traffic congestion, the results show travel time based reward schema performed better in minimising the total travel time of individual vehicles, meanwhile the VEI based reward schema gets the least vehicle emissions to complete a trip. In term of sustainability and resilience, the travel time based reward schema is suitable for emergency vehicles in the road such as ambulances, fire-fighting cars or police cars as those vehicles need to arrive at their destination as soon as possible in order to maintain urban safety. However, the VEI based reward schema is suitable for general drivers in the urban transportation network to minimise the vehicle emissions for a more sustainable urban transportation network.

8.4 Future work

To further improve the proposed vehicle route optimisation for urban transportation network, several recommendations to overcome the limitations of this thesis are suggested as follows:

- **Improve the scalability of the proposed approach.** Although this thesis shows a significant reduction of vehicle travel time and vehicle emissions when applying the DRL method to train the model for vehicle agent, the model is trained by giving the individual network, with a specific size and number of roads, which means the model needs to be re-trained if a vehicle has a trip beyond the boundary of the trained network, or a new network. This potentially could bring the limitation of scalability for this proposed approach. A potential solution for this problem is to use convolutional neural networks to capture the features of the urban traffic condition. Nevertheless, more investigations are required against this limitation.

- **Optimise the proposed framework towards a more sustainable and resilient urban transportation network.** The two proposed travel time based and VEI based reward schemas in this thesis have achieved remarkable results for vehicle route optimisation in sustainable (VEI based) and resilient (travel time based) purposes respectively. To further improve the performance in term of sustainability and resilience, a few enhancements are worth developing in the future. With regard to sustainability, the proposed VEI reward schema only focuses on vehicle

emissions for air pollution. Some other pollution such as noise pollution should be considered in the future. Also, the special driving behaviours on the roadway for emergency vehicles should be included in the consideration such as exceeding the speed limit, overtaking on the right or driving in the opposite direction to the road etc when the DRL agent is making a decision. Although SUMO provides a certain level of emergency vehicle simulation, some capabilities are limited and need to be future developed. Training the model with the consideration of emission vehicle behaviour can achieve more accurate decision making for resilient purposes.

- **Investigate more features to represent a more realistic urban traffic condition and vehicle behaviour.** The features in the proposed approach have proven they are sufficient to optimise vehicle routes in urban transportation networks. However, in practice there are more factors that could affect the urban traffic condition which are worth investigating in the future. For instance, the road conditions (number of lanes, lane width, surface condition) that affect road capacity or the weather impact on travel time prediction. Besides, considering more features for individual vehicle characteristics, such as vehicle maximum speed, car following model, vehicle acceleration etc could potentially improve the efficiency when the DRL agent is making a decision for navigation.
- **Combine other DRL approaches on urban traffic optimisation in order to improve performance of vehicle route optimisation and eventually build an intelligent urban transportation network.** As mentioned before, there are several approaches focusing on using DRL methods to optimise urban traffic in several different areas, such as intersection traffic control and urban traffic prediction. Combining those techniques show great promising to achieve a much more efficient system for urban transportation networks. For instance, intersection traffic control like intelligent traffic light, speed limit control for bottlenecks could give the DRL agent a clearer insight of the expected travel time in a road; Meanwhile urban traffic prediction is able to provide the estimation of upcoming traffic congestion, which is useful for DRL agent to make decisions accordingly in order to avoid the traffic congestion.

8.5 Summary

This thesis proposed a novel DRL based solution for vehicle route optimisation to avoid traffic congestion. It takes the complex traffic conditions across the observed area into account, providing guidance to all the vehicles involved in order to maximize the efficiency of the transportation network. The improved design of a DQN architecture makes the proposed solution best suited for real-time vehicle route optimisation. The framework is built based on SUMO traffic simulator with an RL agent being able to observe and learn from the traffic conditions and instruct the simulated vehicles to different paths toward their destinations. The experiment results have shown that our solution outperformed both built-in navigation algorithms implemented by SUMO, whether it

is toy data in symbolic maps or realistic traffics in maps taken from the real world. As a proof of concept, to the best of my knowledge the work in this thesis is among the first to apply DRL solutions for vehicle route optimisation and provide a promising direction for future works.

Bibliography

[1] Martín Abadi et al. "Tensorflow: A system for large-scale machine learning". In: *12th Symposium on Operating Systems Design and Implementation*. 2016, pp. 265–283.

[2] Baher Abdulhai, Rob Pringle, and Grigoris J Karakoulas. "Reinforcement learning for true adaptive traffic signal control". In: *Journal of Transportation Engineering* 129.3 (2003), pp. 278–285.

[3] Luai Al Shalabi, Zyad Shaaban, and Basel Kasasbeh. "Data mining: A preprocessing engine". In: *Journal of Computer Science* 2.9 (2006), pp. 735–739.

[4] Behrang Asadi and Ardalan Vahidi. "Predictive cruise control: Utilizing upcoming traffic signal information for improving fuel economy and reducing trip time". In: *IEEE transactions on control systems technology* 19.3 (2010), pp. 707–714.

[5] Flavien Balbo and Suzanne Pinson. "Using intelligent agents for transportation regulation support system design". In: *Transportation Research part C: emerging technologies* 18.1 (2010), pp. 140–156.

[6] John S Baras, Xiaobo Tan, and Pedram Hovareshti. "Decentralized control of autonomous vehicles". In: *42nd IEEE International Conference on Decision and Control*. Vol. 2. IEEE. 2003, pp. 1532–1537.

[7] Michael Barbehenn. "A note on the complexity of Dijkstra's algorithm for graphs with weighted vertices". In: *IEEE transactions on computers* 47.2 (1998), p. 263.

[8] Charles Beattie et al. "Deepmind lab". In: *arXiv preprint arXiv:1612.03801* (2016).

[9] Richa Bharadwaj et al. "Efficient dynamic traffic control system using wireless sensor networks". In: *2013 International Conference on Recent Trends in Information Technology (ICRTIT)*. IEEE. 2013, pp. 668–673.

[10] John Adrian Bondy, Uppaluri Siva Ramachandra Murty, et al. *Graph theory with applications*. Vol. 290. Macmillan London, 1976.

[11] Iris Borowy. *Defining sustainable development for our common future: A history of the World Commission on Environment and Development (Brundtland Commission)*. Routledge, 2013.

[12] Sharon Adams Boxill and Lei Yu. "An evaluation of traffic simulation models for supporting its". In: *Houston, TX: Development Centre for Transportation Training and Research, Texas Southern University* (2000).

[13] Greg Brockman et al. "Openai gym". In: *arXiv preprint arXiv:1606.01540* (2016).

[14] *Bronx benefits from mesoscopic-microscopic modelling*. https://www.itsinternational.com/categories/gis-mapping/features/bronx-benefits-from-mesoscopic-microscopic-modelling/. Accessed: 2019-08-4.

[15] Wilco Burghout. "Hybrid microscopic-mesoscopic traffic simulation". PhD thesis. KTH, 2004.

[16] Gordon DB Cameron and Gordon ID Duncan. "PARAMICS—Parallel microscopic simulation of road traffic". In: *The Journal of Supercomputing* 10.1 (1996), pp. 25–53.

[17] Jordi Casas et al. "Traffic simulation with aimsun". In: *Fundamentals of traffic simulation*. Springer, 2010, pp. 173–232.

[18] J del Castillo. "A car following model-based on the lighthill-whitham theory". In: *Transportation and traffic theory* 13 (1996), pp. 517–538.

[19] Kuei-Hsiang Chao and Pi-Yun Chen. "An intelligent traffic flow control system based on radio frequency identification and wireless sensor networks". In: *International journal of distributed sensor networks* 10.5 (2014), p. 694545.

[20] Bo Chen and Harry H Cheng. "A review of the applications of agent technology in traffic and transportation systems". In: *IEEE Transactions on intelligent transportation systems* 11.2 (2010), pp. 485–497.

[21] Okyoung Choi et al. "Delay-optimal data forwarding in vehicular sensor networks". In: *IEEE transactions on vehicular technology* 65.8 (2015), pp. 6389–6402.

[22] Rutger Claes, Tom Holvoet, and Danny Weyns. "A decentralized approach for anticipatory vehicle routing using delegate multiagent systems". In: *IEEE Transactions on Intelligent Transportation Systems* 12.2 (2011), pp. 364–373.

[23] *CO2 emissions from fuel combustion 2018 overview*. https://www.iea.org/statistics/co2emissions/. Accessed: 2019-07-4.

[24] Zhe Cong, Bart De Schutter, and Robert Babuska. "A new ant colony routing approach with a trade-off between system and user optimum". In: *2011 14th International IEEE Conference on Intelligent Transportation Systems (ITSC)*. IEEE. 2011, pp. 1369–1374.

[25] Emanuele Crisostomi, Stephen Kirkland, and Robert Shorten. "A Google-like model of road network dynamics and its application to regulation and control". In: *International Journal of Control* 84.3 (2011), pp. 633–651.

[26] Frederik Michel Dekking et al. *A Modern Introduction to Probability and Statistics: Understanding why and how*. Springer Science & Business Media, 2005.

[27] Prajakta Desai et al. "CARAVAN: Congestion avoidance and route allocation using virtual agent negotiation". In: *IEEE Transactions on Intelligent Transportation Systems* 14.3 (2013), pp. 1197–1207.

[28] Gianni Di Caro. *Ant colony optimization and its application to adaptive routing in telecommunication networks*. Université libre de Bruxelles, 2004.

[29] Edsger W Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische mathematik* 1.1 (1959), pp. 269–271.

[30] Rachel Emas. "The concept of sustainable development: definition and defining principles". In: *Brief for GSDR* (2015), pp. 1–3.

[31] Diego Falsini, Angela Fumarola, and MM Schiraldi. "Sustainable trasportation systems: dynamic routing optimization for a last-mile distribution fleet". In: *Conference on sustainable development: the role of industrial engineering*. DIMEG Università di Bari. 2009, pp. 40–47.

[32] Martin Fellendorf. "VISSIM: A microscopic simulation tool to evaluate actuated signal control including bus priority". In: *64th Institute of Transportation Engineers Annual Meeting*. Vol. 32. Springer. 1994.

[33] C Folke et al. "Building adaptive capacity in a world of transformations". In: *Scientific Background Paper on Resilience for the process of The World Summit on Sustainable Development* (2002).

[34] Marc Friesen et al. "Vehicular traffic monitoring using bluetooth scanning over a wireless sensor network". In: *Canadian Journal of Electrical and Computer Engineering* 37.3 (2014), pp. 135–144.

[35] Donald T Gantz and James R Mekemson. "Flow profile comparison of a microscopic car-following model and a macroscopic platoon dispersion model for traffic simulation". In: *1990 Winter Simulation Conference Proceedings*. IEEE. 1990, pp. 770–774.

[36] Christian Gawron. "Simulation-Based Traffic Assignment. Computing user equilibria in large street networks". PhD thesis. Universität zu Köln, 1998.

[37] Wade Genders and Saiedeh Razavi. "Using a deep reinforcement learning agent for traffic signal control". In: *arXiv preprint arXiv:1611.01142* (2016).

[38] Amin Ghafouri et al. "Optimal detection of faulty traffic sensors used in route planning". In: *Proceedings of the 2nd International Workshop on Science of Smart City Operations and Platforms Engineering*. ACM. 2017, pp. 1–6.

[39] Nesrine Ghariani et al. "A survey of simulation platforms for the assessment of public transport control systems". In: *2014 International Conference on Advanced Logistics and Transport (ICALT)*. IEEE. 2014, pp. 85–90.

[40] Andrew V Goldberg and Chris Harrelson. "Computing the shortest path: A search meets graph theory". In: *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics. 2005, pp. 156–165.

[41] Lance H Gunderson and Lowell Pritchard. *Resilience and the behavior of large-scale systems*. Vol. 60. Island Press, 2012.

[42] Yacov Y Haimes. "On the definition of resilience in systems". In: *Risk Analysis: An International Journal* 29.4 (2009), pp. 498–501.

[43] Randolph W Hall. "Non-recurrent congestion: how big is the problem? Are traveler information systems the solution?" In: *Transportation Research Part C: Emerging Technologies* 1.1 (1993), pp. 89–103.

[44] Peter E Hart, Nils J Nilsson, and Bertram Raphael. "A formal basis for the heuristic determination of minimum cost paths". In: *IEEE transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107.

[45] Serge Hoogendoorn and Piet HL Bovy. "Gas-kinetic modeling and simulation of pedestrian flows". In: *Transportation Research Record* 1710.1 (2000), pp. 28–36.

[46] Remco Hoogma et al. *Experimenting for sustainable transport: the approach of strategic niche management*. Routledge, 2005.

[47] Andreas Horni, David Charypar, and Kay W Axhausen. "Variability in transport microsimulations investigated with the multi-agent transport simulation matsim". In: *Arbeitsberichte Verkehrs-und Raumplanung* 692 (2011).

[48] PB Hunt et al. "The SCOOT on-line traffic signal optimisation technique". In: *Traffic Engineering & Control* 23.4 (1982).

[49] David Isele, Akansel Cosgun, and Kikuo Fujimura. "Analyzing Knowledge Transfer in Deep Q-Networks for Autonomously Handling Multiple Intersections". In: *arXiv preprint arXiv:1705.01197* (2017).

[50] David Isele et al. "Navigating Intersections with Autonomous Vehicles using Deep Reinforcement Learning". In: *URL http://arxiv. org/abs/1705.01196* (2017).

[51] David Isele et al. "Navigating occluded intersections with autonomous vehicles using deep reinforcement learning". In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2018, pp. 2034–2039.

[52] Eline Jonkers et al. *Methodology and Framework Architecture for the Evaluation of Effects of ICT Measures on CO2 Emissions*. 2013.

[53] Hermann Kaindl and Gerhard Kainz. "Bidirectional heuristic search reconsidered". In: *Journal of Artificial Intelligence Research* 7 (1997), pp. 283–317.

[54] Habib M Kammoun et al. "An adaptive vehicle guidance system instigated from ant colony behavior". In: *2010 IEEE International Conference on Systems, Man and Cybernetics*. IEEE. 2010, pp. 2948–2955.

[55] Matthew G Karlaftis and Eleni I Vlahogianni. "Statistical methods versus neural networks in transportation research: Differences, similarities and some insights". In: *Transportation Research Part C: Emerging Technologies* 19.3 (2011), pp. 387–399.

[56] Lorraine Kerr and J Meandue. "Social change and social sustainability: challenges for the planning profession". In: *Planning pathways. Congress.* 2010.

[57] Axel Klar and Raimund Wegener. "A hierarchy of models for multilane vehicular traffic II: Numerical investigations". In: *SIAM Journal on Applied Mathematics* 59.3 (1998), pp. 1002–1011.

[58] Sven Koenig and Maxim Likhachev. "Improved fast replanning for robot navigation in unknown terrain". In: *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No. 02CH37292).* Vol. 1. IEEE. 2002, pp. 968–975.

[59] G Kotusevski and KA Hawick. "A review of traffic simulation software". In: (2009).

[60] Daniel Krajzewicz et al. "COLOMBO: Investigating the Potential of V2X for Traffic Management Purposes assuming low penetration Rates". In: *ITS Europe* (2013).

[61] Daniel Krajzewicz et al. "Recent development and applications of SUMO-Simulation of Urban MObility". In: *International Journal On Advances in Systems and Measurements* 5.3&4 (2012).

[62] Reinhart D Kuhne and Malte B Rodiger. "Macroscopic simulation model for freeway traffic with jams and stop-start waves". In: *1991 Winter Simulation Conference Proceedings.* IEEE. 1991, pp. 762–770.

[63] Daniel R Lanning, Gregory K Harrell, and Jin Wang. "Dijkstra's algorithm and Google maps". In: *Proceedings of the 2014 ACM Southeast Regional Conference.* ACM. 2014, p. 30.

[64] Wilhelm Leutzbach. *Introduction to the theory of traffic flow.* Vol. 47. Springer, 1988.

[65] NE Ligterink et al. *Investigations and real world emission performance of Euro 6 light-duty vehicles.* Delft: TNO, 2013.

[66] Maxim Likhachev, Geoffrey J Gordon, and Sebastian Thrun. "ARA*: Anytime A* with provable bounds on sub-optimality". In: *Advances in neural information processing systems.* 2004, pp. 767–774.

[67] Maxim Likhachev et al. "Anytime Dynamic A*: An Anytime, Replanning Algorithm." In: *ICAPS.* Vol. 5. 2005, pp. 262–271.

[68] Sejoon Lim and Daniela Rus. "Congestion-aware multi-agent path planning: distributed algorithm and applications". In: *The Computer Journal* 57.6 (2013), pp. 825–839.

[69] Canhong Lin et al. "Survey of green vehicle routing problem: past and future trends". In: *Expert systems with applications* 41.4 (2014), pp. 1118–1138.

[70] Todd Litman and David Burwell. "Issues in sustainable transportation". In: *International Journal of Global Environmental Issues* 6.4 (2006), pp. 331–347.

[71] Yisheng Lv et al. "Traffic flow prediction with big data: a deep learning approach". In: *IEEE Transactions on Intelligent Transportation Systems* 16.2 (2015), pp. 865–873.

[72] Michal Maciejewski. "A comparison of microscopic traffic flow simulation systems for an urban area". In: *Transport Problems* 5 (2010), pp. 27–38.

[73] Samiksha Mahajan. "Reinforcement learning: A review from a machine learning perspective". In: *International Journal* 4.8 (2014).

[74] Jessica McGroarty. "Neihoff Urban Studio–W10 January 29, 2010". In: (2010).

[75] Jessica McGroarty. "Recurring and non-recurring congestion: Causes, impacts, and solutions". In: *Neihoff Urban Studio–W10, University of Cincinnati* (2010).

[76] Sara Meerow, Joshua P Newell, and Melissa Stults. "Defining urban resilience: A review". In: *Landscape and urban planning* 147 (2016), pp. 38–49.

[77] Harvey J Miller, Shih-Lung Shaw, et al. *Geographic information systems for transportation: principles and applications*. Oxford University Press on Demand, 2001.

[78] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), p. 529.

[79] Seyed Sajad Mousavi, Michael Schukat, and Enda Howley. "Traffic light control using deep policy-gradient and value-function-based reinforcement learning". In: *IET Intelligent Transport Systems* 11.7 (2017), pp. 417–423.

[80] Shekhar Mukherji. *Migration and urban decay: Asian experiences*. Rawat, 2006.

[81] N Muthukumaran and NK Ambujam. "Wastewater treatment and management in urban areas—a case study of Tiruchirappalli City, Tamil Nadu, India". In: *Proceedings of the Third International Conference on Environment and Health, Chennai, India*. 2003, pp. 15–17.

[82] Kai Nagel. "From particle hopping models to traffic flow theory". In: *Transportation Research Record* 1644.1 (1998), pp. 1–9.

[83] Shahrizul Anuar Abu Nahar and Fazida Hanim Hashim. "Modelling and analysis of an efficient traffic network using ant colony optimization algorithm". In: *2011 Third International Conference on Computational Intelligence, Communication Systems and Networks*. IEEE. 2011, pp. 32–36.

[84] Jose E Naranjo et al. "Lane-change fuzzy control in autonomous vehicles for the overtaking maneuver". In: *IEEE Transactions on Intelligent Transportation Systems* 9.3 (2008), pp. 438–450.

[85] Tuan Nam Nguyen. "Solving Assignment and Routing Problems in Mixed Traffic Systems". PhD thesis. 2016.

[86] Nils J Nilsson. *Principles of artificial intelligence*. Morgan Kaufmann, 2014.

[87] Kartik Pandit et al. "Adaptive traffic signal control with vehicular ad hoc networks". In: *IEEE Transactions on Vehicular Technology* 62.4 (2013), pp. 1459–1471.

[88] Praveen Paruchuri, Alok Reddy Pullalarevu, and Kamalakar Karlapalem. "Multi agent simulation of unorganized traffic". In: *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1*. ACM. 2002, pp. 176–183.

[89] Lucio Sanchez Passos, Rosaldo JF Rossetti, and Zafeiris Kokkinogenis. "Towards the next-generation traffic simulation tools: a first appraisal". In: *6th Iberian Conference on Information Systems and Technologies (CISTI 2011)*. IEEE. 2011, pp. 1–6.

[90] Harold J Payne. "FREFLO: A macroscopic simulation model of freeway traffic". In: *Transportation Research Record* 722 (1979).

[91] Dabal Pedamonti. "Comparison of non-linear activation functions for deep neural networks on MNIST classification task". In: *arXiv preprint arXiv:1804.02763* (2018).

[92] Elise Van der Pol and Frans A Oliehoek. "Coordinated deep reinforcement learners for traffic light control". In: *Proceedings of Learning, Inference and Control of Multi-Agent Systems (at NIPS 2016)* (2016).

[93] N Polson and V Sokolov. "Deep learning predictors for traffic flows". In: *arXiv preprint arXiv:1604.04527* (2016).

[94] Sergio Porta, Paolo Crucitti, and Vito Latora. "The network analysis of urban streets: a dual approach". In: *Physica A: Statistical Mechanics and its Applications* 369.2 (2006), pp. 853–866.

[95] Nedal T Ratrout and Syed Masiur Rahman. "A comparative analysis of currently used microscopic and macroscopic traffic simulation software". In: *The Arabian Journal for Science and Engineering* 34.1B (2009), pp. 121–133.

[96] Honglei Ren et al. "A Deep Learning Approach to the Citywide Traffic Accident Risk Prediction". In: *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*. IEEE. 2018, pp. 3346–3351.

[97] Mustapha Saidallah, Abdeslam El Fergougui, and Abdelbaki Elbelrhiti Elalaoui. "A comparative study of urban road traffic simulators". In: *MATEC Web of Conferences*. Vol. 81. EDP Sciences. 2016, p. 05002.

[98] Lisa A Schaefer et al. "Application of a general particle system model to movement of pedestrians and vehicles". In: *1998 Winter Simulation Conference. Proceedings (Cat. No. 98CH36274)*. Vol. 2. IEEE. 1998, pp. 1155–1160.

[99] Tom Schaul et al. "Prioritized experience replay". In: *arXiv preprint arXiv:1511.05952* (2015).

[100] David Schrank et al. "2015 urban mobility scorecard". In: (2015).

[101] Matt Selinger and Luke Schmidt. "Adaptive traffic control systems in the United States". In: *HDR Engineering, Inc* (2009).

[102] Bruno Castro da Silva et al. "ITSUMO: an intelligent transportation system for urban mobility". In: *International Workshop on Innovative Internet Community Systems*. Springer. 2004, pp. 224–235.

[103] Arthur G Sims and Kenneth W Dobinson. "The Sydney coordinated adaptive traffic (SCAT) system philosophy and benefits". In: *IEEE Transactions on vehicular technology* 29.2 (1980), pp. 130–137.

[104] Fangzhou Sun, Abhishek Dubey, and Jules White. "DxNAT—Deep neural networks for explaining non-recurring traffic congestion". In: *2017 IEEE International Conference on Big Data (Big Data)*. IEEE. 2017, pp. 2141–2150.

[105] Wadhah Z Tareq and Rabah N Farhan. "Autonomic Traffic Lights Control Using Ant Colony Algorithm". In: *International Journal of Advances in Engineering & Technology* 5.1 (2012), p. 448.

[106] Thomas L Thorpe and Charles W Anderson. *Tra c light control using sarsa with three state representations*. Tech. rep. Citeseer, 1996.

[107] Tomer Toledo et al. "Mesoscopic simulation for transit operations". In: *Transportation Research Part C: Emerging Technologies* 18.6 (2010), pp. 896–908.

[108] Department for Transport. *Towards a sustainable transport system: supporting economic growth in a low carbon world*. Vol. 7226. The Stationery Office, 2007.

[109] Federal Highway Administration (US) and Federal Transit Administration (US). *2013 Status of the Nation's Highways, Bridges, and Transit Conditions & Performance Report to Congress*. Government Printing Office, 2017.

[110] S Uttara, Nishi Bhuvandas, Vanita Aggarwal, et al. "Impacts of urbanization on environment". In: *International Journal of Research in Engineering and Applied Sciences* 2.2 (2012), pp. 1637–1645.

[111] M Van Aerde et al. "INTEGRATION: An overview of traffic simulation features". In: *Transportation Research Records* (1996).

[112] Hado Van Hasselt, Arthur Guez, and David Silver. "Deep reinforcement learning with double q-learning". In: *Thirtieth AAAI Conference on Artificial Intelligence*. 2016.

[113] Ziyu Wang et al. "Dueling network architectures for deep reinforcement learning". In: *arXiv preprint arXiv:1511.06581* (2015).

[114] Ziyu Wang et al. "Dueling network architectures for deep reinforcement learning". In: *arXiv preprint arXiv:1511.06581* (2015).

[115] Horst F Wedde and Sebastian Senge. "BeeJamA: A distributed, self-adaptive vehicle routing guidance approach". In: *IEEE Transactions on Intelligent Transportation Systems* 14.4 (2013), pp. 1882–1895.

[116] MA Wiering et al. *Intelligent traffic light control*. 2004.

[117] David Wilkie, Cenk Baykal, and Ming C Lin. "Participatory route planning". In: *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM. 2014, pp. 213–222.

[118] David James Wilkie et al. "Self-aware traffic route planning". In: *Twenty-Fifth AAAI Conference on Artificial Intelligence*. 2011.

[119] *World Urbanization Prospects 2018*. https://population.un.org/wup/Publications/Files/WUP2018-Highlights.pdf.

[120] WANG Yanyang, WEI Tietao, and QU Xiangju. "Study of multi-objective fuzzy optimization for path planning". In: *Chinese Journal of Aeronautics* 25.1 (2012), pp. 51–56.

[121] Zhanhong Zhou and Ming Cai. "Intersection signal control multi-objective optimization based on genetic algorithm". In: *Journal of Traffic and Transportation Engineering (English Edition)* 1.2 (2014), pp. 153–158.

[122] Xinlu Zong et al. "Multi-ant colony system for evacuation routing problem with mixed traffic flow". In: *IEEE Congress on Evolutionary Computation*. IEEE. 2010, pp. 1–6.

[123] Nikolaos Zygouras et al. "Towards Detection of Faulty Traffic Sensors in Real-Time." In: *MUD@ ICML*. 2015, pp. 53–62.

Appendix A

Key Code Snippets for Vehicle Route Optimisation

A.1 Scenario Class

This section presents the python codes for the key functions in *scenario* class. *scenario* class is mainly designed for creating the required files for SUMO simulator in order to run simulation in this experiment. The python source code for *scenario* class is shown as follows:

```
import os
import numpy as np
import xml.etree.ElementTree as ET

NET_FILE = 'sumo_conf/net.net.xml'
TRIP_FILE = 'sumo_conf/trips.trips.xml'
ROUTE_FILE = 'sumo_conf/routes.rou.xml'
VTYPE_FILE = 'sumo_conf/vtype.add.xml'
SUMO_CONF = 'sumo_conf/basic.cfg'
DEFAULT_VTYPE_FILE =
    'D:/LJMU/PhD/SUMO_Projects/Deep_Learning/sumo_files/vtype.add.xml'
ACCIDENT_PROB = 0.01

class scenario():
    def __init__(self,
                 map,
                 duration,
                 n_veh,
                 vType_list=None,
                 accidents=False
                 ):
        self.n_veh = n_veh
        if map.lower().endswith('.osm'):
            self.network = self.convert_map(map)
        else:
            self.network = map

        if vType_list is not None:
            self.vType = self.gen_vType(vType_list)
```

```
29          else:
30              self.vType = DEFAULT_VTYPE_FILE
31
32          self.trips = self.gen_trips(duration, n_veh)
33          self.routes = self.gen_route(accidents)
34          self.conf = self.gen_conf()
35
36      def convert_map(self, map):
37          cmd = 'netconvert --osm ' + map + \
38                ' --geometry.remove' + \
39                ' --ramps.guess' + \
40                ' --roundabouts.guess' + \
41                ' --junctions.join' + \
42                ' -o ' + NET_FILE
43
44          os.system(cmd)
45
46          return NET_FILE
47
48      def gen_vType(self, vType_list):
49
50          with open(VTYPE_FILE, 'w+') as vtype_file:
51              vtype_file.write('<?xml version="1.0"?>\n')
52              vtype_file.write('<additional>\n')
53              vtype_file.write('\t<vTypeDistribution>\n')
54              for vType in vType_list:
55                  str = '<vType '
56                  for key, value in vType.items():
57                      str += key + '="' + value + '" '
58                  str += '/>'
59                  vtype_file.write('\t\t' + str + '\n')
60              vtype_file.write('\t</vTypeDistribution>\n')
61              vtype_file.write('</additional>\n')
62
63          return VTYPE_FILE
64
65      def gen_trips(self, duration, n_veh):
66          n = duration / n_veh
67          cmd = '\"\"%SUMO_HOME%tools\\randomTrips.py\" -n ' +
                self.network + \
68                ' --trip-attributes=\"type=\\\"normal_car\\\"\" ' + \
69                ' -b 0' + \
70                ' -e ' + str(duration) + \
71                ' -p ' + str(n) + \
72                ' --prefix veh_' + \
73                ' -o ' + TRIP_FILE + '\" '
74
75          os.system(cmd)
76
77          return TRIP_FILE
```

```
78
79      def gen_routes(self, accidents=False):
80          cmd = 'duarouter --route-files ' + TRIP_FILE + \
81                ' --additional-files ' + self.vType + \
82                ' -n ' + self.network + \
83                ' -o ' + ROUTE_FILE
84          os.system(cmd)
85
86          if accidents:
87              n_accidents = int(self.n_veh * ACCIDENT_PROB)
88              accidents_id_list = []
89              for i in range(n_accidents):
90                  while True:
91                      j = np.random.randint(self.n_veh)
92                      if j not in accidents_id_list:
93                          accidents_id_list.append(j)
94                          break
95
96              for i in accidents_id_list:
97                  root = ET.parse(ROUTE_FILE).getroot()
98                  veh = root.find('.//vehicle[@id="veh_' + i + '"]')
99                  route = veh.find('route').attrib['edges']
100                 route = route.split(' ')
101                 ET.SubElement(veh.find('route'), 'stop')
102                 veh.find('route').find('stop').set('lane',
                        route[np.random.randint(len(route))])
103                 veh.find('route').find('stop').set('endPos', '10')
104                 veh.find('route').find('stop').set('duration', '20')
105                 ET.ElementTree(root).write(open(ROUTE_FILE, 'w'),
                        encoding='unicode')
106
107         return ROUTE_FILE
108
109     def gen_conf(self):
110         with open(SUMO_CONF, 'w+') as conf:
111             conf.write('<?xml version="1.0"?>\n')
112             conf.write('<configuration>\n')
113             conf.write('\t<input>')
114             conf.write('\t\t<net-file value="' + self.network +
                    '"/>\n')
115             conf.write('\t\t<route-file value="' + self.routes +
                    '"/>\n')
116             conf.write('\t<\input>')
117             conf.write('<\configuration>\n')
118
119         return SUMO_CONF
```

A.2 Environment Class

This section presents the python codes for the key functions in *scenario* class. The environment class *sumo_env* is designed to encapsulate the use of SUMO simulator with TraCI. It enables the features to control SUMO including to initialise and interfere a simulation, define the state spaces and action spaces, reward calculation, and the action applicator to the simulation.

```
import traci
import numpy as np
import os
import math
from prettytable import PrettyTable
import xml.etree.ElementTree as ET
from sklearn import preprocessing
import statistics

SUMO_DIR = 'D:/LJMU/PhD/SUMO_Projects/Deep_Learning/sumo_files/'
TEMP_ROUTE_FILE = 'sumo_conf/temp.rou.xml'
TEMP_TRIP_FILE = 'sumo_conf/temp.trips.xml'
ROUTE_FILE = 'sumo_conf/network.rou.xml'
TRIP_FILE = 'sumo_conf/trip.trips.xml'
VTYPE_FILE =
    'D:/LJMU/PhD/SUMO_Projects/Deep_Learning/sumo_files/network.rou.xml'
ERROR_FILE = 'sumo_conf/error'
NAV_VEH_ID = 'nav_veh'
NAV_VEH_TYPE = 'nav_car'
STATUS_IN_ZONE = 'IN_ZONE'
STATUS_ARRIVED = 'ARRIVED'
ACCIDENT_CLEARANCE_TIME = 20

class Sumo():
    def __init__(self,
                 sumo_conf,
                 gui=False,
                 reward_method = 0, # 0 is travel time based, 1 is VEI
                     based
                 start_edge="",
                 end_edge="",
                 incl_travel_time=False,
                 incl_n_veh=False):

        sumo_cmd = 'sumo-gui' if gui else 'sumo'
        self.sumo_conf = sumo_conf
        self.cmd = [sumo_cmd, '-c', sumo_conf]
        self.start_edge = start_edge
        self.end_edge = end_edge
        self.incl_travel_time = incl_travel_time
        self.incl_n_veh = incl_n_veh
        self.reward_method = reward_method
```

```
41
42          # Run simulation once in order to get network information and
                features number
43          traci.start(['sumo', '-c', sumo_conf])
44          self.e_conn_dict, self.e_lane_dict, self.max_n_actions,
                self.edges = self.get_edge_conn_info()
45          self.e_distance_dest_dict = self.get_dist_to_dest()
46          self.n_features = (self.incl_travel_time + self.incl_n_veh) *
                len(self.edges) + 4
47          traci.close()
48
49          self.target = ""
50          self.veh_counter = 0
51
52          self.traci = traci
53
54      def get_edge_conn_info(self):
55          e_conn_dict = {}
56          e_lane_dict = {}
57
58          # max_link determines the number of actions in this network
59          max_link = 0
60          edges = []
61          self.shortest_travel_time = 1000
62          self.longest_travel_time = 0
63          for lane in traci.lane.getIDList():
64              if lane[:1] != ':':
65                  l_edge = traci.lane.getEdgeID(lane)
66                  e_length = traci.lane.getLength(lane)
67                  e_max_speed = traci.lane.getMaxSpeed(lane)
68
69                  if (e_length / e_max_speed) <
                        self.shortest_travel_time:
70                      self.shortest_travel_time = (e_length / e_max_speed)
71
72                  if (e_length / 0.1) > self.longest_travel_time:
73                      self.longest_travel_time = (e_length / 0.1)
74
75                  if l_edge not in edges:
76                      edges.append(l_edge)
77
78                  if l_edge in e_lane_dict:
79                      e_lane_dict[l_edge].append(lane)
80                  else:
81                      e_lane_dict[l_edge] = [lane]
82
83                  if traci.lane.getLinks(lane) != []:
84                      e_conn_dict[l_edge] = []
85                      if len(traci.lane.getLinks(lane)) > max_link:
86                          max_link = len(traci.lane.getLinks(lane))
```

```
87
88                  for i in range(len(traci.lane.getLinks(lane))):
89                      connected_lanes =
                            traci.lane.getLinks(lane)[i][0]
90                      connected_edges =
                            traci.lane.getEdgeID(connected_lanes)
91                      e_conn_dict[l_edge].append(connected_edges)
92
93          t = PrettyTable()
94          field_names = ['Edge_ID']
95          for i in range(max_link):
96              field_names.append('action_' + str(i))
97          t.field_names = field_names
98          for lane in traci.lane.getIDList():
99              if lane[:1] != ':':
100                 row = [None for i in range(max_link)]
101                 l_edge = traci.lane.getEdgeID(lane)
102                 if l_edge in e_conn_dict:
103                     for i in range(len(e_conn_dict[l_edge])):
104                         row[i] = e_conn_dict[l_edge][i]
105                 row = [l_edge] + row
106                 t.add_row(row)
107         with open('connected_edge.txt', 'w+') as f:
108             f.write(str(t))
109
110         return e_conn_dict, e_lane_dict, max_link, edges
111
112     def get_dist_to_dest(self):
113         e_distance_dest_dict = {}
114
115         for key, value in self.e_lane_dict.items():
116             l_end_pos_X, l_end_pos_Y =
                    traci.lane.getShape(value[0])[-1]
117             dest_lane = self.e_lane_dict[self.end_edge][0]
118             dest_pos_X, dest_pos_Y = traci.lane.getShape(dest_lane)[-1]
119             distance = math.hypot(dest_pos_X - l_end_pos_X, dest_pos_Y
                    - l_end_pos_Y)
120             e_distance_dest_dict[key] = distance
121
122         t = PrettyTable()
123         field_names = ['Edge_ID', 'distance to destination']
124         t.field_names = field_names
125         for key, value in e_distance_dest_dict.items():
126             t.add_row([key, value])
127         with open('distance_to_destination.txt', 'w+') as f:
128             f.write(str(t))
129
130         return e_distance_dest_dict
131
132     def reset(self):
```

```
133         traci.start(self.cmd)
134         self.add_veh(is_nav=True)
135         self.actual_route = [self.start_edge]
136         self.status = 'NONE'
137         self.CO = 0
138         self.HC = 0
139         self.NOX = 0
140         self.PMX = 0
141         self.total_CO = 0
142         self.total_HC = 0
143         self.total_NOX = 0
144         self.total_PMX = 0
145         self.time_in_edge = [0]
146         self.VEI_in_edge = []
147 
148         return self.run_simulation()
149 
150     def get_default_route(self, start_edge, end_edge):
151         root = ET.parse(self.sumo_conf)
152         for type_tag in root.findall('input/net-file'):
153             value = type_tag.get('value')
154 
155         with open(TEMP_TRIP_FILE, 'w+') as trip_file:
156             trip_file.write('<?xml version="1.0"?>\n')
157             trip_file.write('<trips>\n')
158             trip_file.write('\t<trip id="0" depart="0.00" from="' +
                    start_edge + '" to="' + end_edge + '" />\n')
159             trip_file.write('</trips>\n')
160 
161         cmd = 'duarouter --route-files ' + TEMP_TRIP_FILE + ' -n ' +
                value + ' -o ' + TEMP_ROUTE_FILE
162         if not os.system(cmd):
163             root = ET.parse(TEMP_ROUTE_FILE).getroot()
164             for type_tag in root.findall('vehicle/route'):
165                 value = type_tag.get('edges')
166             self.default_route = value.split(" ")
167             return value.split(" ")
168 
169         return None
170 
171     def add_veh(self, is_nav=False, veh_id=None, route=None,
            type=None):
172         if is_nav:
173             veh_id = NAV_VEH_ID
174             route = self.get_default_route(self.start_edge,
                        self.end_edge) if route == None else route
175             type = NAV_VEH_TYPE if type == None else type
176             route_id = veh_id + str(self.veh_counter)
177 
178             traci.route.add(route_id, route)
```

```
179             traci.vehicle.add(veh_id, route_id, type)
180             self.in_zone = None
181             self.actual_route.append(self.start_edge)
182             self.time_in_edge.append(0)
183 
184         else:
185             v_id = veh_id + str(self.veh_counter)
186             traci.route.add(v_id, route)
187             traci.vehicle.add(v_id, v_id, type)
188         self.veh_counter += 1
189 
190     def run_simulation(self, action=None):
191         v_edge = traci.vehicle.getRoadID(NAV_VEH_ID)
192         if action is not None:
193             self.target = self.e_conn_dict[v_edge][action]
194             traci.vehicle.changeTarget(NAV_VEH_ID, self.target)
195 
196         while True:
197             traci.simulationStep()
198             self.time_in_edge[-1] += 1
199             status = self.get_status()
200 
201             if NAV_VEH_ID in traci.vehicle.getIDList():
202                 self.get_emissions()
203 
204             if status == 'NEW':
205                 self.time_in_edge.append(0)
206                 self.VEI_in_edge.append(self.get_VEI())
207                 self.actual_route.append(v_edge)
208                 self.CO = 0
209                 self.HC = 0
210                 self.NOX = 0
211                 self.PMX = 0
212 
213             if status == 'IN_ZONE':
214                 done = 0
215                 obs, t = self.get_observation()
216                 v_edge = traci.vehicle.getRoadID(NAV_VEH_ID)
217                 n_actions = len(self.e_conn_dict[v_edge])
218                 reward = 0
219 
220                 return obs, t, n_actions, reward, done
221 
222             elif status == 'DONE':
223                 done = 1
224                 reward = self.get_reward(done)
225                 obs = np.zeros(self.n_features)
226                 t = ""
227                 n_actions = 0
228 
```

```
229             return obs, t, n_actions, reward, done
230
231     def get_status(self):
232         if NAV_VEH_ID in traci.vehicle.getIDList():
233             v_lane = traci.vehicle.getLaneID(NAV_VEH_ID)
234             if v_lane[:1] != ':':
235                 v_pos = traci.vehicle.getLanePosition(NAV_VEH_ID)
236                 l_len = traci.lane.getLength(v_lane)
237                 v_edge = traci.vehicle.getRoadID(NAV_VEH_ID)
238                 zone_length = self.get_decision_zone_length()
239
240                 if self.in_zone == None and \
241                         l_len - v_pos <= zone_length and \
242                         v_edge not in [self.end_edge]:
243                     self.in_zone = v_edge
244                     self.status = 'IN_ZONE'
245                     return 'IN_ZONE'
246
247                 elif self.in_zone != None and self.in_zone != v_edge:
248                     self.in_zone = None
249                     self.status = 'NEW'
250                     return 'NEW'
251
252             else:
253                 self.status = 'NONE'
254                 return 'NONE'
255
256         elif NAV_VEH_ID in traci.simulation.getArrivedIDList():
257             self.status = 'DONE'
258             return 'DONE'
259
260     def get_decision_zone_length(self):
261         v_lane = traci.vehicle.getLaneID(NAV_VEH_ID)
262         v_max_speed = traci.vehicle.getMaxSpeed(NAV_VEH_ID)
263         v_decel = traci.vehicle.getDecel(NAV_VEH_ID)
264         v_tau = traci.vehicle.getTau(NAV_VEH_ID)
265         l_len = traci.lane.getLength(v_lane)
266
267         zone_length = v_max_speed + (v_max_speed * v_decel * v_tau)
268         if l_len <= zone_length:
269             return l_len
270
271         return zone_length
272
273     def get_emissions(self):
274         self.CO += traci.vehicle.getCOEmission(NAV_VEH_ID)
275         self.HC += traci.vehicle.getHCEmission(NAV_VEH_ID)
276         self.NOX += traci.vehicle.getNOxEmission(NAV_VEH_ID)
277         self.PMX += traci.vehicle.getPMxEmission(NAV_VEH_ID)
278         self.total_CO += traci.vehicle.getCOEmission(NAV_VEH_ID)
```

```
279         self.total_HC += traci.vehicle.getHCEmission(NAV_VEH_ID)
280         self.total_NOX += traci.vehicle.getNOxEmission(NAV_VEH_ID)
281         self.total_PMX += traci.vehicle.getPMxEmission(NAV_VEH_ID)
282
283     def get_observation(self):
284         np_veh_pos = np.array([])
285         np_travel_time = np.array([])
286         np_n_veh = np.array([])
287         np_obs = np.array([])
288
289         l_ID = traci.vehicle.getLaneID(NAV_VEH_ID)
290         # v_pos_x, v_pos_y = traci.vehicle.getPosition(NAV_VEH_ID)
291         v_pos_x, v_pos_y = traci.lane.getShape(l_ID)[-1]
292         [min_x, min_y], [max_x, max_y] =
                traci.simulation.getNetBoundary()
293
294         l_ID = self.e_lane_dict[self.end_edge][0]
295         des_x, des_y = traci.lane.getShape(l_ID)[-1]
296         display_obs = 'Destination: ' + str([des_x, des_y]) + '\n'
297         display_obs += 'Vehicle Position: ' + str([v_pos_x, v_pos_y])
                + '\n'
298
299         min_x = des_x if des_x < min_x else min_x
300         min_y = des_y if des_y < min_y else min_y
301         max_x = des_x if des_x > max_x else max_x
302         max_y = des_y if des_y > max_y else max_y
303
304         des_x = (des_x - min_x) / (max_x - min_x)
305         des_y = (des_y - min_y) / (max_y - min_y)
306         v_pos_x = (v_pos_x - min_x) / (max_x - min_x)
307         v_pos_y = (v_pos_y - min_y) / (max_y - min_y)
308
309         np_veh_pos = np.append(np_veh_pos, [des_x, des_y])
310         np_veh_pos = np.append(np_veh_pos, [v_pos_x, v_pos_y])
311
312
313         np_obs = np.append(np_obs, np_veh_pos)
314
315         if self.incl_travel_time or self.incl_n_veh:
316             t = PrettyTable()
317             field_names = ['Edge ID']
318             for e_ID in traci.edge.getIDList():
319                 if e_ID[:1] != ':':
320                     row = [e_ID]
321
322                     if self.incl_travel_time:
323
324                         e_mean_speed =
                                traci.edge.getLastStepMeanSpeed(e_ID)
325                         e_lane = self.e_lane_dict[e_ID][0]
```

```
326                     e_length = traci.lane.getLength(e_lane)
327                     e_max_speed = traci.lane.getMaxSpeed(e_lane)
328                     if e_mean_speed == 0:
329                         if traci.edge.getLastStepOccupancy() == 100:
330                             e_travel_time = e_length / 0.1
331                         else:
332                             is_accident = True
333                             first_veh_pos = 0
334                             last_veh_pos = e_length
335                             for veh in
                                    traci.edge.getLastStepVehicleIDs():
336                                 v_lane_pos =
                                        traci.vehicle.getLanePosition(veh)
337                                 if v_lane_pos == e_length:
338                                     is_accident = False
339                                 if v_lane_pos > first_veh_pos:
340                                     first_veh_pos = v_lane_pos
341                                 if v_lane_pos < last_veh_pos:
342                                     last_veh_pos = v_lane_pos
343
344                             if is_accident:
345                                 e_travel_time = (last_veh_pos /
                                        e_max_speed) + \
346                                                 ACCIDENT_CLEARANCE_TIME
                                                    + \
347                                                 ((e_length -
                                                    first_veh_pos) /
                                                    e_max_speed)
348                             else:
349                                 e_travel_time = (last_veh_pos /
                                        e_max_speed) + \
350                                                 ((e_length -
                                                    last_veh_pos) / 0.1)
351                     else:
352                         e_travel_time =
                                traci.edge.getTraveltime(e_ID)
353
354                     e_travel_time = (e_travel_time -
                            self.shortest_travel_time) / \
355                                     (self.longest_travel_time -
                                        self.shortest_travel_time)
356
357                     np_travel_time = np.append(np_travel_time,
                            e_travel_time)
358                     field_names.append('expected travel time')
359                     row.append(e_mean_speed)
360
361                 if self.incl_n_veh:
362                     e_n_veh =
                            traci.edge.getLastStepVehicleNumber(e_ID)
```

```
363                     e_n_veh = e_n_veh / self.max_n_veh
364                     np_n_veh = np.append(np_n_veh, e_n_veh)
365                     field_names.append('vehicle number')
366                     row.append(e_n_veh)
367 
368                 if t.field_names == []:
369                     t.field_names = field_names
370                 t.add_row(row)
371 
372             display_obs += str(t)
373 
374             np_obs = np.append(np_obs, np_travel_time)
375             np_obs = np.append(np_obs, np_n_veh)
376         return np_obs, display_obs
377 
378     def get_reward(self, done):
379         if done:
380             reward = 1
381         else:
382             if self.reward_method == 0:
383                 reward = self.time_in_edge[-2] / -1
384 
385             else:
386                 reward = self.VEI_in_edge[-1] / -1
387 
388         return reward
389 
390     def get_VEI(self):
391         # Some values for calculation in this function is hardcoded by
                following the euro standard
392         # need to change when using different vehicle type
393 
394         w_CO = 1
395         w_HC = 1 / 0.068
396         w_NOX = 1 / 0.06
397         w_PMX = 1 / 0.005
398 
399         std_CO = 1000
400         std_HC = 68
401         std_NOX = 60
402         std_PMX = 5
403 
404         # convert from miles to km
405         CO_edge = self.CO * 1.6
406         HC_edge = self.HC * 1.6
407         NOX_edge = self.NOX * 1.6
408         PMX_edge = self.PMX * 1.6
409 
410         VEI_CO = (CO_edge * w_CO) / std_CO
411         VEI_HC = (HC_edge * w_HC) / std_HC
```

```
412         VEI_NOX = (NOX_edge * w_NOX) / std_NOX
413         VEI_PMX = (PMX_edge * w_PMX) / std_PMX
414 
415         VEI = VEI_CO + VEI_HC + VEI_NOX + VEI_PMX
416 
417         return VEI
418 
419     def step_count(self):
420         return traci.simulation.getTime()
421 
422     def close(self):
423         traci.close()
```

A.3 DRL Agent Class

The DRL agent class *agent* is to interact with the environment through the environment class *sumo_env* and use the implementation of the DRL algorithms to train a model in order to optimise vehicle's routing selection. It imports Tensorflow library to implement the DQN learning process.

```
1  import numpy as np
2  import tensorflow as tf
3  
4  class SumTree(object):
5      data_pointer = 0
6  
7      def __init__(self, capacity):
8          self.capacity = capacity
9          self.tree = np.zeros(2 * capacity - 1)
10         self.data = np.zeros(capacity, dtype=object)
11 
12     def add(self, p, data):
13         tree_idx = self.data_pointer + self.capacity - 1
14         self.data[self.data_pointer] = data
15         self.update(tree_idx, p)
16 
17         self.data_pointer += 1
18         if self.data_pointer >= self.capacity:
19             self.data_pointer = 0
20 
21     def update(self, tree_idx, p):
22         change = p - self.tree[tree_idx]
23         self.tree[tree_idx] = p
24 
25         while tree_idx != 0:
26             tree_idx = (tree_idx - 1) // 2
27             self.tree[tree_idx] += change
28 
29     def get_leaf(self, v):
```

```
        parent_idx = 0
        while True:
            cl_idx = 2 * parent_idx + 1
            cr_idx = cl_idx + 1
            if cl_idx >= len(self.tree):
                leaf_idx = parent_idx
                break
            else:
                if v <= self.tree[cl_idx]:
                    parent_idx = cl_idx
                else:
                    v -= self.tree[cl_idx]
                    parent_idx = cr_idx

        data_idx = leaf_idx - self.capacity + 1
        return leaf_idx, self.tree[leaf_idx], self.data[data_idx]

    @property
    def total_p(self):
        return self.tree[0]


class Memory(object):

    epsilon = 0.01 # small amount to avoid zero priority
    alpha = 0.2 # [0~1] convert the importance of TD error to priority
    beta = 0.4 # importance-sampling, from initial value increasing
        to 1
    beta_increment_per_sampling = 0.001
    abs_err_upper = 1. # clipped abs error

    def __init__(self, capacity):
        self.tree = SumTree(capacity)

    def store(self, transition):
        max_p = np.max(self.tree.tree[-self.tree.capacity:])
        if max_p == 0:
            max_p = self.abs_err_upper
        self.tree.add(max_p, transition)

    def sample(self, n):
        b_idx, b_memory, ISWeights = np.empty((n,), dtype=np.int32),
            np.empty((n, self.tree.data[0].size)), np.empty((n, 1))
        pri_seg = self.tree.total_p / n
        self.beta = np.min([1., self.beta +
            self.beta_increment_per_sampling])

        min_prob = np.min(self.tree.tree[-self.tree.capacity:]) /
            self.tree.total_p
        for i in range(n):
```

```
            a, b = pri_seg * i, pri_seg * (i + 1)
            v = np.random.uniform(a, b)
            idx, p, data = self.tree.get_leaf(v)
            prob = p / self.tree.total_p
            ISWeights[i, 0] = np.power(prob/min_prob, -self.beta)
            b_idx[i], b_memory[i, :] = idx, data
        return b_idx, b_memory, ISWeights

    def batch_update(self, tree_idx, abs_errors):
        abs_errors += self.epsilon
        clipped_errors = np.minimum(abs_errors, self.abs_err_upper)
        ps = np.power(clipped_errors, self.alpha)
        for ti, p in zip(tree_idx, ps):
            self.tree.update(ti, p)


class Agent:
    def __init__(
            self,
            n_features,
            n_actions,
            learning_rate=0.005,
            reward_decay=0.9,
            e_greedy=0.95,
            replace_target_iter=300,
            memory_size=10000,
            batch_size=32,
            e_greedy_increment=0.00001,
            output_graph=False,
            prioritized=True,
            double_q=True,
            dueling=True,
            sess=None,
            name='',
            saver=''
    ):
        config = tf.ConfigProto()
        config.gpu_options.allow_growth = True

        self.n_actions = n_actions
        self.n_features = n_features
        self.lr = learning_rate
        self.gamma = reward_decay
        self.epsilon_max = e_greedy
        self.replace_target_iter = replace_target_iter
        self.memory_size = memory_size
        self.batch_size = batch_size
        self.epsilon_increment = e_greedy_increment
        self.epsilon = 0 if e_greedy_increment is not None else
            self.epsilon_max
```

```
125
126         self.prioritized = prioritized
127         self.double_q = double_q
128         self.dueling = dueling
129         self.learn_step_counter = 0
130
131         self._build_net(name)
132         t_params = tf.get_collection('target_net_params')
133         e_params = tf.get_collection('eval_net_params')
134         self.replace_target_op = [tf.assign(t, e) for t, e in
                zip(t_params, e_params)]
135
136         if self.prioritized:
137             self.memory = Memory(capacity=memory_size)
138         else:
139             self.memory = np.zeros((self.memory_size, n_features*2+2))
140         self.saver = tf.train.Saver()
141         if sess is None:
142             self.sess = tf.Session(config=config)
143             self.sess.run(tf.global_variables_initializer())
144         else:
145             self.sess = sess
146         if saver != '':
147             self.saver.restore(self.sess,
                    'saver/my_policy_net_pg.ckpt')
148         self.cost_his = []
149
150
151     def _build_net(self, name):
152         def build_layers(s, c_names, n_l1, w_initializer,
                b_initializer, trainable, name):
153             with tf.variable_scope(name + 'l1'):
154                 w1 = tf.get_variable('w1', [self.n_features, n_l1],
                        initializer=w_initializer, collections=c_names,
                        trainable=trainable)
155                 b1 = tf.get_variable('b1', [1, n_l1],
                        initializer=b_initializer, collections=c_names,
                        trainable=trainable)
156                 l1 = tf.nn.relu(tf.matmul(s, w1) + b1)
157
158             with tf.variable_scope(name + 'l2'):
159                 w3 = tf.get_variable('w2', [150, 100],
                        initializer=w_initializer, collections=c_names,
                        trainable=trainable)
160                 b3 = tf.get_variable('b2', [1, 100],
                        initializer=b_initializer, collections=c_names,
                        trainable=trainable)
161                 l2 = tf.nn.relu(tf.matmul(l1, w3) + b3)
162
163             if self.dueling:
```

```
164                 with tf.variable_scope(name +'Value'):
165                     w2 = tf.get_variable('w2', [100, 1],
                            initializer=w_initializer, collections=c_names)
166                     b2 = tf.get_variable('b2', [1, 1],
                            initializer=b_initializer, collections=c_names)
167                     self.V = tf.matmul(l2, w2) + b2
168
169                 with tf.variable_scope(name + 'Advantage'):
170                     w2 = tf.get_variable('w2', [100, self.n_actions],
                            initializer=w_initializer, collections=c_names)
171                     b2 = tf.get_variable('b2', [1, self.n_actions],
                            initializer=b_initializer, collections=c_names)
172                     self.A = tf.matmul(l2, w2) + b2
173
174                 with tf.variable_scope(name + 'Q'):
175                     out = self.V + (self.A - tf.reduce_mean(self.A,
                            axis=1, keep_dims=True))
176             else:
177                 with tf.variable_scope(name + 'Q'):
178                     w2 = tf.get_variable('w2', [100, self.n_actions],
                            initializer=w_initializer, collections=c_names)
179                     b2 = tf.get_variable('b2', [1, self.n_actions],
                            initializer=b_initializer, collections=c_names)
180                     out = tf.matmul(l2, w2) + b2
181
182             return out
183
184         self.s = tf.placeholder(tf.float32, [None, self.n_features],
                name='s')
185         self.q_target = tf.placeholder(tf.float32, [None,
                self.n_actions], name=name+'Q_target')
186         if self.prioritized:
187             self.ISWeights = tf.placeholder(tf.float32, [None, 1],
                    name=name+'IS_weights')
188         with tf.variable_scope(name+'eval_net'):
189             c_names, n_l1, w_initializer, b_initializer = \
190                 ['eval_net_params', tf.GraphKeys.GLOBAL_VARIABLES],
                        150, \
191                 tf.random_normal_initializer(0., 0.3),
                        tf.constant_initializer(0.1)
192
193             self.q_eval = build_layers(self.s, c_names, n_l1,
                    w_initializer, b_initializer, True, name)
194
195         with tf.variable_scope(name+'loss'):
196             if self.prioritized:
197                 self.abs_errors = tf.reduce_sum(tf.abs(self.q_target -
                        self.q_eval), axis=1)
198                 self.loss = tf.reduce_mean(self.ISWeights *
                        tf.squared_difference(self.q_target, self.q_eval))
```

```
199             else:
200                 self.loss =
                        tf.reduce_mean(tf.squared_difference(self.q_target,
                        self.q_eval))
201         with tf.variable_scope(name+'train'):
202             self._train_op =
                    tf.train.RMSPropOptimizer(self.lr).minimize(self.loss)
203
204         self.s_ = tf.placeholder(tf.float32, [None, self.n_features],
                name='s_')
205         with tf.variable_scope(name+'target_net'):
206             c_names = ['target_net_params',
                        tf.GraphKeys.GLOBAL_VARIABLES]
207             self.q_next = build_layers(self.s_, c_names, n_l1,
                        w_initializer, b_initializer, False, name)
208
209     def store_transition(self, sim_data):
210         s = sim_data[0]
211         a = sim_data[1]
212         r = sim_data[2]
213         s_ = sim_data[3]
214         if self.prioritized:
215             transition = np.hstack((s, [a, r], s_))
216             self.memory.store(transition)
217         else:
218             if not hasattr(self, 'memory_counter'):
219                 self.memory_counter = 0
220             transition = np.hstack((s, [a, r], s_))
221             index = self.memory_counter % self.memory_size
222             self.memory[index, :] = transition
223             self.memory_counter += 1
224
225     def choose_action(self, observation, n_actions, current_edge,
            conn_edges, e_distance):
226         observation = observation[np.newaxis, :]
227         actions_value = self.sess.run(self.q_eval, feed_dict={self.s:
                observation})
228         if np.random.uniform() < self.epsilon:
229             action = np.argmax(actions_value[0][:n_actions])
230         else:
231             if n_actions > 1:
232                 if np.random.uniform() < 0.6:
233                     prob_explore = []
234                     for edge in conn_edges:
235                         distance = e_distance[edge]
236                         prob_explore.append(distance)
237                     prob_explore = np.array(prob_explore)
238                     prob_explore -= np.mean(prob_explore)
239                     prob_explore /= np.std(prob_explore)
240                     prob_explore = self.softmax(prob_explore)
```

```
241
242                 action = np.random.choice(range(n_actions),
                        p=prob_explore)
243             else:
244                 action = np.random.randint(0, n_actions)
245         else:
246             action = 0
247     return action, actions_value
248
249 def learn(self, episode, sim):
250     if self.learn_step_counter % self.replace_target_iter == 0:
251         self.sess.run(self.replace_target_op)
252         print('\ntarget_params_replaced\n')
253
254     if self.prioritized:
255         tree_idx, batch_memory, ISWeights =
                self.memory.sample(self.batch_size)
256     else:
257         sample_index = np.random.choice(self.memory_size,
                size=self.batch_size)
258         batch_memory = self.memory[sample_index, :]
259
260     q_next, q_eval4next = self.sess.run(
261         [self.q_next, self.q_eval],
262         feed_dict={self.s_: batch_memory[:, -self.n_features:],
263                    self.s: batch_memory[:, -self.n_features:]})
264     q_eval = self.sess.run(self.q_eval, {self.s: batch_memory[:,
            :self.n_features]})
265
266     q_target = q_eval.copy()
267     batch_index = np.arange(self.batch_size, dtype=np.int32)
268     eval_act_index = batch_memory[:, self.n_features].astype(int)
269     reward = batch_memory[:, self.n_features + 1]
270
271     if self.double_q:
272         max_act4next = np.argmax(q_eval4next, axis=1)
273         selected_q_next = q_next[batch_index, max_act4next]
274     else:
275         selected_q_next = np.max(q_next, axis=1)
276
277     q_target[batch_index, eval_act_index] = reward + self.gamma *
            selected_q_next
278     if self.prioritized:
279         _, abs_errors, self.cost = self.sess.run([self._train_op,
                self.abs_errors, self.loss],
280                                          feed_dict={self.s:
                                             batch_memory[:,
                                             :self.n_features],
281                                                    self.q_target:
                                                       q_target,
```

```
                                                self.ISWeights:
                                                    ISWeights})
            self.memory.batch_update(tree_idx, abs_errors)
        else:
            _, self.cost = self.sess.run([self._train_op, self.loss],
                                         feed_dict={self.s:
                                            batch_memory[:,
                                            :self.n_features],
                                                self.q_target:
                                                    q_target})

        self.cost_his.append(self.cost)
        self.epsilon = self.epsilon + self.epsilon_increment if
            self.epsilon < self.epsilon_max else self.epsilon_max
        self.learn_step_counter += 1

    def save(self):
        self.saver.save(self.sess, "saver/my_policy_net_pg.ckpt")

    def softmax(self, x):
        x = x - np.max(x)
        exp_x = np.exp(x)
        softmax = exp_x / np.sum(exp_x)
        return softmax
```