TECHNISCHE UNIVERSITÄT DRESDEN

ComNets
Deutsche Telekom Chair of Communication Networks

Faculty of Electrical and Computer Engineering Institute of Communcation Technology

Diploma Thesis

Deep Reinforcement Learning for Traffic Control

Johannes Valentin Stanislaus Busch

Born on: 31.07.1992 in Bergisch Gladbach
Course: Electrical Engineering
Discipline: Automation, Measurement and Control
Matriculation number: 3850449
Matriculation year: 2012

to achieve the academic degree

Diplomingenieur (Dipl.Ing.)

Supervisor

Dipl. Ing. Vincent Latzko

Supervising professor

Prof. Dr.-Ing. Dr. h.c. Frank Fitzek

Submitted on: 23.08.2019

TECHNISCHE UNIVERSITÄT DRESDEN
FAKULTÄT ELEKTROTECHNIK UND INFORMATIONSTECHNIK

Task for Diplom thesis

For Mr Johannes Busch ET/2010

Theme **Deep Reinforcement Learning for Traffic Control**

Task description

With global individual motorized traffic increasing, novel approaches to alleviate the congestion, delay, and pollution problems are needed. The intelligent control of traffic infrastructure provides an effective, easily implementable and cost efficient means towards a more efficient utilization of existing resources. With the emergence of fast and robust communication technology (especially 5G), arises the opportunity of transmitting relevant information about the dynamic state of the traffic network and leveraging this data in order to make well-informed control decisions in real-time. Due to the highly non-linear and stochastic nature of traffic systems, effective control has proven to be a non-trivial task and requires highly evolved, complex control strategies. Reinforcement Learning (RL) addresses the problem of how an agent that can observe its environment should act in order to maximize some reward function. In particular, the combination of RL and Deep Neural Networks – termed Deep Reinforcement Learning (DRL) – has emerged in recent years as a novel control technique for highly non-linear, stochastic and data-rich problems.
In this thesis, the suitability of the DRL framework for addressing the problem of dynamic control of traffic infrastructure (e.g. traffic lights) will be explored. For learning and evaluation of the control strategy, a traffic scenario in which the infrastructure can be interfaced by the DRL agent has to be developed and implemented. The evaluation of the benefit of communicating dynamic traffic information is of particular interest , enabling the agent to make more informed decisions, potentially moving towards an efficient solution of the traffic problem.
The thesis consists of the following tasks:

- Literature study on DRL algorithms and prior approaches to traffic control.
- Design of a traffic scenario that may showcase the mitigating effects on traffic congestion of the proposed control strategy.
- Development of one or several RL environments consisting of an observation space, an action space and a reward function; in particular, different observation spaces may model the availability of information and account for different degrees of communication.
- Selection of a DRL algorithm for the control of traffic infrastructure.
- Implementation of the proposed scenario and DRL agent in software.
- Evaluation of the performance of the proposed DRL agent and comparison of the benefits of different degrees of available information.

Supervisor: Dipl.-Ing. Vincent Latzko

Reviewer: Prof. Dr.-Ing. Dr. h.c. Frank Fitzek
Second Reviewer: Dr.-Ing. Roland Schingnitz

Started at: 15.03.2019
To be submitted by: 23.08.2019

The diplom thesis is written in English.

Prof. Dr.-Ing. Steffen Bernet
Vorsitzender des Prüfungsausschusses

Prof. Dr.-Ing. Dr. h.c. Frank Fitzek
Verantwortlicher Hochschullehrer

Statement of authorship

I hereby certify that I have authored this Diploma Thesis entitled *Deep Reinforcement Learning for Traffic Control* independently and without undue assistance from third parties. No other than the resources and references indicated in this thesis have been used. I have marked both literal and accordingly adopted quotations as such. There were no additional persons involved in the intellectual preparation of the present thesis. I am aware that violations of this declaration may lead to subsequent withdrawal of the degree.

Dresden, 23.08.2019

Johannes Valentin Stanislaus Busch

Kurzfassung

Die steigende Zahl an Fahrzeugen in Privatbesitz, in Kombination mit fortschreitender Urbanisierung, verursacht zunehmend Staus und Verzögerungen in Städten auf der ganzen Welt. Viele aktuelle Verkehrssteuerungen werden von Hand, mittels einfacher Heuristiken, sowie teilweise durch automatische Optimierungsalgorithmen, eingestellt. Daraus resultierende Steuerungen nutzen bestehende Verkehrsinfrastruktur oft ineffizient, und verursachen dadurch wirtschaftlichen sowie ökologischen Schaden. Die jüngste Entwicklung fortschrittlicher Fahrzeug-zu-Infrastruktur (V2I) Kommunikationstechnologie ermöglicht eine bessere Verkehrsregelung, da aktuelle Verkehrsdaten in die Regelungsentscheidungen einfließen können. Die gewaltigen Datenmengen in eine Kontrollentscheidung zu übersetzen, ist allerdings eine herausfordernde Aufgabe, und bedarf der Entwicklung neuartiger Regelungsansätze. In dieser Arbeit wird ein "Deep Reinforcement Learning (DRL)"-Ansatz entwickelt, der die Regelung von Lichtschaltanlagen, unter Kenntnis detaillierter Information des aktuellen Zustandes des Verkehrsnetzwerkes, ermöglicht. Unsere Methode optimiert einen zentralen Regler, der mehrere Lichtschaltanlagen gleichzeitig steuert, ohne dabei auf zusätzliche Koordinierungsalgorithmen zurückgreifen zu müssen. In einer Reihe von simulierten Szenarien zeigen wir, dass die Möglichkeit, seine Umgebung durch V2I Kommunikation wahrzunehmen, den Verkehrsregler befähigt, Staus zu reduzieren und somit dessen wirtschaftliche und ökologische Folgen zu mindern. Wir zeigen zudem, dass unser DRL Ansatz in der Lage ist, mehrere Zielkriterien gleichzeitig zu optimieren. Da unser Ansatz keine Modellannahmen treffen muss, der Aktionsraum inhärent sicher gestaltet ist, und realistische Verkehrsnetzwerke schnell modelliert, optimiert und getestet werden können, hat er das Potenzial, die Anwendung von DRL in tatsächlichen Verkehrssteuerungen zu ermöglichen.

Abstract

The rapid increase in privately owned vehicles per capita, alongside progressive urbanisation, causes rising levels of congestion and commuter delay in cities around the world. Current traffic control systems often rely on simple heuristics and hand optimisation, partly accompanied by automatic adaption, to determine reasonable traffic light signalling strategies. Implemented policies often fail to enable efficient utilisation of the road-traffic infrastructure – harming both economic competitiveness and environmental sustainability. The recent emergence of advanced Vehicle to Infrastructure (V2I) communication technologies presents a potential remedy, as it facilitates well-informed control decisions, based on the current traffic state. However, leveraging large amounts of state information is a challenging task that asks for novel control techniques. In this thesis, we describe the design of a Deep Reinforcement Learning (DRL) approach that allows the control of traffic scenarios under the knowledge of elaborate state-information. This approach centrally optimises the signalling strategy of multiple traffic lights, without the need for second-order coordination methods. In a series of simulated traffic scenarios, we show that the ability to observe its environment through V2I communication enables the control system to mitigate congestion and to alleviate its economic and environmental repercussions. We furthermore demonstrate that the DRL method, in contrast to many traditional methods, allows the joint optimisation of multiple objective-functions. The model-free nature of our approach, the inherent safety of control-actions, and the capacity to rapidly prototype and test a control-policy in realistic traffic networks could enable the deployment of DRL traffic-control methods in the real world.

Acknowledgements

I thank Vincent Latzko, my advisor, for his help and guidance, for providing me with everything necessary to complete this thesis and for always making time when I was in need of advice. Thanks to Frank Fitzek for making this thesis possible. Also, I thank Clara Costa Sala, Matthias Mozdzanowski and Peter Sossalla for proof-reading this thesis and for providing feedback and suggestions. Finally, I thank everyone at the Deutsche Telekom Chair of Communication Networks for making me feel welcome in the group as well as for many fruitful discussions.

Table of Contents

List of Figures

List of Tables

Acronyms

Acronym	Expansion
A2C	Advantage Actor-Critic
A3C	Asynchronous Advantage Actor-Critic
ACKTR	Actor-Critic using Knoecker-Factored Trust Region
ADAM	Adaptive Moment Estimation
ADPG-R	Asynchronous DPG with Variable Replay Steps
D4PG	Distributed Distributional Deterministic Policy Gradient
DDPG	Deep Deterministic Policy Gradient
DNN	Deep Neural Network
DPG	Deterministic Policy Gradient
DQL	Deep Q-Learning
DQN	Deep Q-Network
DRL	Deep Reinforcement Learning
GDP	Gross Domestic Product
GPI	Generalised Policy Iteration
IDM	Intelligent Driver Model
ITS	Intelligent Transportation System
MARL	Multi-Agent Reinforcement Learning
MDP	Markov Decision Process
ML	Machine Learning
MLP	Multilayer Perceptron
MOVA	Microprocessor Optimised Vehicle Actuation
MSE	Mean Squared Error
NN	Neural Network
OSM	Open Street Map
PG	Policy Gradient
POMDP	Partially Observable Markov Decision Process
PPO	Proximal Policy Optimisation
RL	Reinforcement Learning
SAC	Soft Actor-Critic
SCOOT	Split Cycle Offset Optimisation Technique

Acronym	Expansion
SGD	Stochastic Gradient Descent
SUMO	Simulation of Urban MObility
TD3	Twin-Delayed Deep Deterministic Policy Gradient
TDL	Temporal Difference Learning
TraCI	Traffic Control Interface
TRANSYT	Traffic Network Study Tool
TRPO	Trust Region Policy Optimisation
V2I	Vehicle to Infrastructure

Symbols

Symbol	Name	Description
a	action	The action that the agent executes
σ	activation function	The nonlinear activation function of a Perceptron
ρ	dynamics function	The dynamics of the Markov Decision Process; probabalistically maps from a tuple of a state/observation and an action to a tuple of a subsequent state and a reward
ε	epsilon	The probability of choosing a random action in epsilon-greedy policies
γ	discount factor	The factor that determines how much future rewards are valued
h	history	All past observations and states of the agent
T	horizon	The total number of timesteps of an episode in the environment; can be infinite
α	learning rate	The step-size for iterative approximation algorithms such as gradient descent
η	observation function	A probabilistic mapping from states to observations
o	observation	The observation that that the agent can perceive of its environment; in a fully observed setting, the observation equals the state
π	policy	The policy function that maps from states/observations to actions
Q	action-value function	The expected value of the future discounted return, when starting from a given state and executing a given action
g	discounted return	The sum of all future discounted rewards
r	reward	The obtained reward in a single timestep
s	state	The state of the environment of the agent
V	state-value function	The expected value of the future discounted return, when starting from a given state

1. Introduction

1.1. Motivation

Due to rising mobility demands, transportation emerges to be an economic key sector in developed nations, and inefficiencies in transportation networks result in high costs and commuter delay. In the European Union, the economic burden of traffic congestion accounts for approximately 1% of the annual Gross Domestic Product (GDP) and is expected to further increase within the next years (European Comission, 2017). In many metropolitan areas, the average commuter spends over 200 hours per year in congested traffic (Inrix, 2018). Furthermore, the heavy amounts of carbon and other emissions may have unpredictable and unprecedented environmental repercussions. Through the last decades, these problems have seen a rise in relevance, due to changing needs of transportation and a skyrocketing number of privately owned vehicles.

Different approaches towards mitigating these undesirable effects of modern traffic have been proposed, some of which require expensive and time-consuming remodelling of existing road infrastructure or major changes in traffic legislation. Many of these measures are difficult or even impossible to implement because of spatial, economic or environmental constraints, which are particularly rigid in crowded city centres.

Among the proposed solutions, a more efficient utilisation of existing infrastructure through the intelligent allocation of given resources is of particular interest, as it provides an effective, easily implementable and cost-efficient measure, which demands little changes to the physical road infrastructure. Current road traffic optimisation practice is a combination of hand-tuned policies with a small degree of automatic adaption (Richter et al., 2006). Many cities experiment with innovative mechanisms that aim to improve the efficacy of traffic control. However, most implemented measures are restricted to small, local adaptions of phase durations in traffic lights for a predefined sequence of traffic phases, based on historical data from past traffic counts or measurements of current traffic volume from inductive loop sensors. Furthermore, many systems independently optimise the schedule of single traffic lights, without considering their interplay with the surrounding traffic network. The origin of these techniques dates back to several decades ago, when information about the current state of the traffic system was sparse, and the optimisation of the traffic strategy was time-consuming due to slow hard- and software.

The recent emergence of fast and reliable communication interfaces between individual vehicles and the traffic infrastructure (especially 5G) provides an opportunity for a particularly

data-rich representation of the current traffic state, which might be leveraged to enable highly informed, near-optimal decisions in traffic-regulating infrastructure. However, due to the very high-dimensional state space and the complex, non-linear nature of traffic systems, finding an optimal solution of the given control problem for anything more than a small toy-problem quickly becomes infeasible. Novel control paradigms are therefore needed to cope with the staggering complexity of traffic control under the presence of detailed state-information and to effectively mitigate traffic congestion.

In recent years, the application of Machine Learning (ML) techniques to a broad range of problems has found traction in both academia and industry. In particular, the unreasonable effectiveness of Neural Network (NN) architectures and the backpropagation algorithm (Rumelhart et al., 1986) has encouraged researchers to tackle ever-more-complex problems with the help of learning algorithms. Some of the most celebrated of recent achievements stem from the field of Reinforcement Learning (RL), which addresses complex control problems with a learning-based approach. The combination of Reinforcement Learning and Deep Neural Networks (DNNs) — termed Deep Reinforcement Learning (DRL) — has been successfully applied to many challenging domains, ranging from arcade games (Mnih et al., 2015) over the ancient board-game of Go (Silver et al., 2016) to robotic control (Popov et al., 2017).

A supposed suitability of the RL framework to solve the traffic control problem led to the emergence of a plethora of research articles, that employ different approaches to model the traffic scenario and apply different RL algorithms to solve it (e.g. Richter et al., 2006; Salkham et al., 2008; Bakker et al., 2010; Prabuchandran et al., 2015; Casas, 2017; Mousavi et al., 2017). Many of these publications report that the respective RL approach proved to outperform other, traditional traffic control strategies by a significant margin.

Existing publications that combine RL with traffic signal control mostly assume traffic-state information that is collected through inductive loop sensors as well as a rich communication interface among the individual installations of the traffic infrastructure (e.g. Casas, 2017). Some works also utilise more advanced information about the traffic-state, like video data from a traffic camera that is installed above the intersection and gives a visual overview of the current traffic situation (e.g. Mousavi et al., 2017). However, relatively little effort has been put into the investigation of traffic systems that feature detailed information about individual vehicles, obtained through a rich communication interface between individual vehicles and the local traffic infrastructure.

1.2. Objectives

In this work, the Deep Reinforcement Learning framework will be applied to the complex problem of traffic infrastructure control. To this end, existing DRL algorithms will be analysed, and a suitable algorithm will be selected and adapted to the traffic control problem at hand.

To evaluate its ability in mitigating traffic congestion, the developed RL control system will be examined in a series of scenarios. These include a single isolated intersection scenario as well as several ensembles of connected intersections that require intricate coordination of traffic signalling strategies. The scenarios will be implemented in an existing traffic simulation framework, which needs to be capable of providing detailed information about the current state of individual vehicles. In order to apply the developed RL algorithm to the traffic scenario, a suitable hyper-parameter configuration of the algorithm has to be determined.

Of particular interest to this work, will be the benefit of available state-information of individual

vehicles that could, for example, be obtained through a Vehicle to Infrastructure (V2I) interface, which lets vehicles share information with the local traffic infrastructure in real-time. Therefore, two different observation-spaces will be developed, embodying the different degrees of available information. The advantages of a traffic infrastructure that features rich V2I communication, over a non-communicating infrastructure, will be assessed within the proposed traffic scenarios.

Furthermore, we will investigate the ability of the RL method to optimise multi-objective reward functions and, therefore, to fulfil the diverse requirements that we may ask of a traffic control system.

1.3. Structure

This thesis will be structured as follows: In chapter 2, we will give a general introduction to the Reinforcement Learning framework, that covers the field's basics, and then gradually introduce the current state-of-the-art. This introduction tries to be as complete and comprehensive as possible, while focusing on the methods and algorithms that will be used in this work. In chapter 3, the traffic control problem will be looked at in detail. We will further motivate the necessity of an intricate control paradigm and introduce some basic terminology. We will then discuss some traditional control approaches that are currently employed at intersections around the world. Subsequently, we will give a brief overview of traffic simulation and introduce the software libraries that will be made use of in this work. Finally, we review the emerging possibility of acquiring rich state information through V2I communication. In chapter 4, we take a deeper look at the expected advantages of using DRL for controlling traffic. Furthermore, we will analyse which factors might make traffic control with RL in general, and the setting of V2I in particular, a difficult problem. Subsequently, we will discuss several recent approaches towards the control of traffic, using RL methods. Then we will explain the proposed RL algorithm and define the full control environment, including the different observation-spaces that will be used to model infrastructure with and without V2I communication. Finally, we briefly discuss the possibility of deploying a RL traffic controller in a real-world traffic scenario. In chapter 5, several traffic scenarios will be proposed and evaluated. For each scenario, we will describe the individual setup in detail and define which behaviour we hope to witness. Subsequently, we will show and discuss the obtained results. Finally, in chapter 6, we will summarise our findings and examine which could be some promising further research directions.

2. Reinforcement Learning

In the following chapter, the ideas and principles of Reinforcement Learning (RL) that are needed to understand this thesis will be discussed. Starting from a brief introduction and a conceptual categorisation of RL within the Machine Learning framework, the basic principles and vocabulary of the RL problem will be introduced. Subsequently, the problem will be formalised and important solution strategies will be outlined. Finally, the RL framework will be put into the context of function approximation with Deep Neural Networks and several recent advancements in Deep Reinforcement Learning (DRL) will be explained. Through gradually integrating new concepts, we hope to give a gentle but concise introduction, that largely matches the historical evolution of the field. A review of the entire landscape of RL is beyond the scope of this work. The interested reader is therefore referred to Sutton and Barto, 2018 for a comprehensive introduction to the foundation of RL and to Lapan, 2018 for an applied study of latest trends in DRL.

2.1. Introduction to Reinforcement Learning

Machine Learning (ML) is a branch of the broad field of Artificial Intelligence, which is concerned with the study of algorithms and models that learn to perform a task through inference from data. Instead of explicit instructions towards solving the respective task, an ML algorithm describes how a statistical pattern can be deduced from examples, which then may be leveraged to reach said goal. Solution strategies are typically considered to fall into one of three different categories, depending on the nature of the example data:

1. In **Supervised Learning**, the ML model is trained to map from a given input to a given target. The target has to be provided by some external, knowledgeable supervisor. It is therefore used to find patterns that, generalising from the example data, may predict the targets of unseen data. A typical example is the classification of the content of images (e.g. He et al., 2015a).

2. In **Unsupervised Learning**, the algorithm is only given an input, but no corresponding target. Instead of predicting an output, it is used to model underlying statistical patterns in the input data. A typical application would be to identify clusters of similar shopping behaviour in customers of a retail store (e.g. Gil et al., 2009).

3. In **Reinforcement Learning**, the algorithm generates an output from its given input and receives an evaluation of its performance in the form of a reward signal. The amount of

supervision through the reward signal can be considered to be somewhere in between the full information of Supervised Learning and the complete lack of supervision of Unsupervised Learning. A typical example would be to learn how to play a range of arcade games from pixels (e.g. Mnih et al., 2015).

In contrast to the other two, that usually learn from a fixed, predefined dataset, Reinforcement Learning is mostly considered to continuously generate new data through interaction with its environment. Notably, this means that the statistical pattern of the training data in RL strongly depends on the intermediate solution of the algorithm and, therefore, is non-stationary.

Intuitively it may strike us, that this regime of learning through interaction most closely resembles our idea of how animals or humans learn. While a playing infant is not shown exactly how to behave and which actions to take (like in Supervised Learning), it certainly receives some feedback from the interaction with its environment, that lets it learn about causal relationships and the outcomes of its actions. In fact, there has been found considerable overlap between RL and research fields that are concerned with human and animal cognition, like Behavioural Psychology or Neuroscience (Sutton and Barto, 2018). In contrast to those fields however, RL explores a computational approach towards goal-directed problem solving and decision making through inference from data, rather than directly theorising about biologic learning. It therefore neglects the need to explain algorithms in terms of psychological or biochemical mechanisms and focuses on efficiently solving the given task.

In the following, the essential concepts and problems of the Reinforcement Learning framework are explained alongside with the standard terminology; new terminology will be printed in *italic* once and will be used naturally thereafter. A mathematical formalisation of these concepts will be presented in section 2.2.

The entity that takes *actions* within the *environment* to collect *experience* is called the *agent*. In order to choose from the available actions, the agent leverages some representation of the *state* of its environment, called the *observation*. The purpose of the agent is therefore to perform a mapping from its observations to a sequence of actions to execute. This mapping is called the agent's *policy*.

The ultimate goal of RL algorithms is to take the best of all possible actions in every situation. The quality of a policy is evaluated by some numerical *reward* signal, which is given after every action that the agent takes. For example, the reward signal for an agent that trades stocks might be the increase of its portfolio value; or an agent that plays computer games may receive a positive reward for every game that is won, a negative reward for every lost one and zero rewards in between. It is important to note that the performed actions may not only affect the immediate reward but also all subsequent rewards. In extreme cases, as the aforementioned game-playing agent, rewards might actually be so sparse that the immediate reward holds no information and the quality of an action cannot be evaluated until many timesteps later. Rather than maximising the immediate reward, the agent therefore strives to maximise the sum of all future rewards. Notably, this sum does not only depend on the current state and action but also on all subsequent ones. The learning algorithm is thus faced with the task of accrediting the obtained rewards in order to evaluate the quality of individual actions.

An important consequence of generating data through interaction is the need to trade off *exploration* and *exploitation*. The agent should exploit its knowledge by choosing actions that have proven to yield high rewards. However, it might well be that it has not yet discovered the optimal action and should therefore keep exploring by trying out other actions. This trade-off has to be carefully tuned as both exclusively exploring as well as exploiting will result in failure.

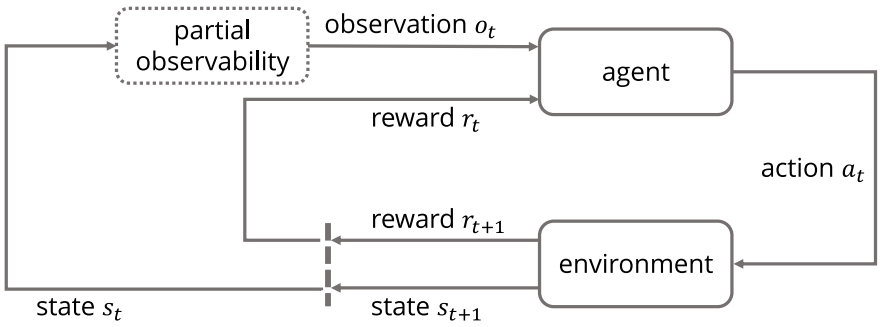


Figure 2.1.: The agent-environment loop of a Markov Decision Process. The agent can observe some partial information of the state of the environment. Leveraging this observation, the agent takes actions in order to influence its environment. The environment then returns a subsequent state and a numerical reward, which is used to evaluate the agent's actions.

On the one hand, actions may have to be tried several times in order to obtain a reasonable estimate of the expected reward, especially in environments where rewards are delayed and stochastic. On the other hand, without properly exploiting its current knowledge, the agent may never encounter certain states and therefore fails to learn to navigate these more advanced situations.

As we have seen, Reinforcement Learning faces some complex problems that do not usually occur in other Machine Learning branches, which can make RL problems very hard to solve. However, explicitly addressing these issues makes it a somewhat more complete framework, that can be applied to a wide variety of advanced control problems in an end-to-end fashion. This stands in contrast to many other approaches, that deal with subproblems without considering the bigger picture. For example, an RL algorithm might address the problem of navigating an autonomous car through urban traffic while avoiding pedestrians and other obstacles, directly mapping sensory input to control decisions. With Supervised Learning, on the other hand, one would address the classification of pedestrians from camera images as an isolated subproblem and then use this knowledge in a further signal processing chain. RL thus is an ambitious effort that just might take us a step closer towards building a truly 'intelligent' machine.

In the following section, the ideas that we have just introduced will be formalised in the mathematical framework of the Markov Decision Process.

2.2. Markov Decision Processes

Markov Decision Processes (MDPs) are formal descriptions of the tasks that RL algorithms try to solve. They formalise sequential decision making in problems where decisions not only influence immediate reward but also all future states and rewards (Sutton and Barto, 2018). As described in the preceding section, we consider an agent that senses the state s of its environment and takes influence on this environment through actions a. In the general case, the agent is not able to observe every detail of the state of its environment and only senses some partial information, called an observation o. To evaluate the quality of its actions, the agent receives some numerical reward r. Figure 2.1 shows this agent-environment interaction loop.

As most RL applications, we here consider the sequential case in that actions are taken at discrete timesteps $t = 0, 1, 2, \ldots$ and are followed by a reward and a subsequent observation. A

trajectory of the agent therefore consists of a sequence of observations, actions and rewards:

$$o_0, a_0, r_1, o_1, a_1, r_2, o_2, \ldots, a_{T-1}, r_T, o_T, \tag{2.1}$$

where the *horizon* T is the total length of the trajectory. Note that the horizon can potentially be infinite, resulting in a so-called *continuous* or *infinite-horizon* task (e.g. a robot that continuously tries to navigate the real world). If the trajectories naturally fall into separate *episodes* of finite but not necessarily equal length, it is called an *episodic* or *finite-horizon* task (e.g. a game of chess that ends when one player has won).

The fundamental assumption underlying the MDP — called the Markov property — is that the state fully identifies all knowable aspects of the environment that hold information for the future. Importantly, this means that an agent that observes the current state of the environment cannot improve its decisions through additionally considering past states or actions, as the current state includes all relevant, available information. Many RL applications consider the fully observable case in that the observation equals the state, which is seldom a reasonable assumptions. We will here describe the more general Partially Observable Markov Decision Process (POMDP) and include the fully observable setting as a special case in that the observation equals the state. Even though we may later ignore the difference between observation and state, we think that, when designing an RL application, it is crucial to bear in mind that they are not the same.

In a POMDP, the observation that the agent can sense of its environment consists of some aspects of the state; in general the observation therefore holds less information than the state itself. We can express the observation $o \in \mathcal{O}$ as a function of the state $s \in \mathcal{S}$, that may be non-deterministic:

$$\eta : \mathcal{S} \times \mathcal{O} \to [0, 1]. \tag{2.2}$$

We denote the probability distribution over observations, given a state by $\eta(o|s)$. As the Markov property does not necessarily hold for observations, an agent may make better decisions through taking into account all past observations and actions. We therefore consider the agents *history* $h_t = (o_0, a_0, o_1, a_2, \ldots, a_{t-1}, o_t)$ in the decision making process and define the policy π as a function that maps from a history h to a probability distribution over actions $a \in \mathcal{A}$:

$$\pi : \mathcal{H} \times \mathcal{A} \to [0, 1], \tag{2.3}$$

where $\mathcal{H}$ is the set of possible histories. The probability distribution over actions, given a history is denoted $\pi(a|h)$. It is worth noting that the space of available actions may, in general, depend on the state. However, here we will assume that the agent can always choose from a fixed set of actions, irrespective of its current state, but assigns zero probability to those actions that are unavailable. In the case of full observability, we can replace the history by the state, resulting in the policy $\pi(a|s)$.

Taking an action a_t in a state s_t takes the agent to the subsequent state s_{t+1} and returns the numerical reward r_{t+1}. We formalise this by introducing the dynamics function of the MDP:

$$\rho : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \times \mathbb{R} \to [0, 1], \tag{2.4}$$

that defines a joined probability distribution of the subsequent state and reward, conditioned on the previous state and action: $\rho(s_{t+1}, r_{t+1}|s_t, a_t)$. Note that, because of the Markov property, the dynamics function does not depend on any previous states or actions. Figure 2.2 visualises the described dependencies.

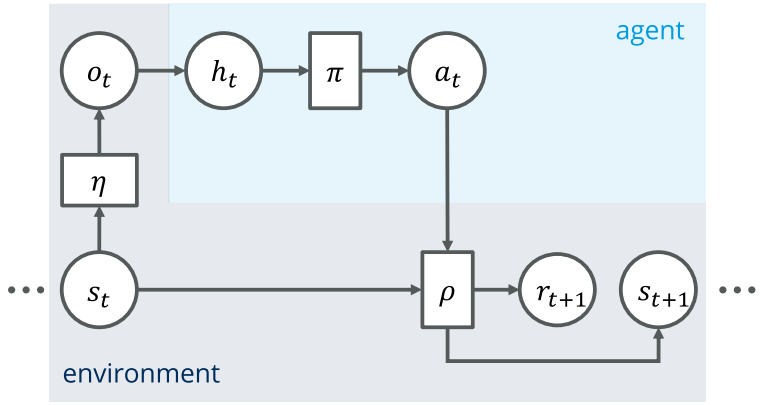


Figure 2.2.: Visualisation of the dependencies in a Partially Observable Markov Decision Process. Circles denote variables; squares denote probabilistic mappings. The observation function η maps from the environment state to the observation of the agent. Based on the history of previous observations and actions, the agent samples a new action from its policy π. The dynamics function ρ then determines the next state and the obtained reward.

As we have seen in section 2.1, choosing an action a_t that results in the highest possible reward r_{t+1} would be short-sighted, as an action can influence the subsequent state and thus all following rewards. A more natural goal would therefore be to maximise the sum of all future rewards: $\sum_{i=t+1}^{T} r_i$. For episodic tasks, this value is bounded as long as individual rewards r_t are not infinite. However, for continuing tasks, where $T \to \infty$, the sum of rewards is potentially unbounded, which makes it hard to maximise. We therefore introduce the concept of *discounting* and define the *discounted return* as:

$$g_t = \sum_{i=t+1}^{T} \gamma^{i-t-1} \cdot r_i, \tag{2.5}$$

where the *discount factor* $\gamma \in [0, 1]$ determines the degree of discounting. For bounded rewards $r_t \leq r_{max} \in \mathbb{R}$ and a discount factor $\gamma < 1$, the discounted return is bounded by $g_t \leq \frac{r_{max}}{1-\gamma}$. Note that $\gamma = 1$ results in the undiscounted case.

A byproduct of maximising the discounted return is that the immediate reward will be valued higher than rewards that are further in the future. In stochastic tasks, this might be an intuitively reasonable assumption, as the uncertainty about the reward, we expect to obtain, grows with temporal distance (after all, a bird in the hand is better than two in the bush). Furthermore, it may be desirable that the agent tries to obtain high rewards as fast as possible. In the extreme cases, a value of $\gamma = 0$ means that the agent will optimise only the reward that immediately follows its current action, whereas $\gamma = 1$ equally values all future rewards, irrespective of the time it takes to obtain them.

We can now formally define the discrete-time POMDP as the 6-tuple

$$(\mathcal{S}, \mathcal{A}, \mathcal{O}, \rho, \eta, \gamma), \tag{2.6}$$

with the set of states $\mathcal{S}$, the set of actions $\mathcal{A}$, the set of observations $\mathcal{O}$, the dynamics function ρ, the observation function η and the discount factor γ. Note that we do not assume all aspects of the MDP to be known to the agent; in some cases, the agent may know quite a bit about how rewards are generated or how the environment reacts to its actions, while in others it may know

close to nothing. For our purposes however, we will assume that the agent always knows the space of possible observations and actions as well as the discount factor (in fact, one could argue that the discount factor is internal to the agent and is not part of the MDP).

The Markov Decision Process is a flexible framework, that can model a versatile range of real-world problems. Even though it may appear limiting to formulate a task in terms of the MDP's three fundamental signals — states (respectively observation in the partially observable case), actions and rewards, in practice, these can account for a broad variety of different settings. States can be very low-level features, like the pixel-brightness of a camera image, or high-level information, like a long-term estimation of the development of an economy. Similarly, the action can be low-level, like the applied voltage to an electrical actuator, or high-level, like the decision of whether or not to learn to speak a new language. Finally, the reward signal needs to be engineered to fully encapsulate the goals of the agent. For example, a gambling agent would probably end up doing nothing when only trying to optimise its expected monetary outcome (since the dealer always wins in the long run). However, a real gambler might have other goals, like reaching the excitement and thrill of his hobby, or he may value the chance of winning higher than the risk of losing. On the other hand, one should be careful not to over-engineer the reward function; while it can help the agent to quickly find a good solution, when it obtains rewards for small sub-goals, it may find a way to gain high discounted returns without solving the task that we actually care about.

2.2.1. Value Functions

As we have discussed, our goal is to find a policy $\pi(a_t|h_t)$ that yields high discounted returns. Unfortunately, as the dynamics function ρ in general is stochastic, the discounted return g_t of some state s_t can only be known at the end of the episode. It is thus impossible to choose an action at timepoint t that reliably maximises the future realisations of the stochastic rewards. However, knowing the statistics of the reward signal, we may optimise the discounted return in expectation. We therefore introduce the *state-value function* as the expected value of the discounted return, given the current state:

$$V^\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\Big[\sum_{i=t+1}^{T} \gamma^{i-t-1} \cdot R_i \mid S_t = s\Big], \tag{2.7}$$

where uppercase symbols represent a random variable and lowercase symbols represent a specific realisation of the respective variable. The index π of the expectation reminds us that the actions are drawn from the policy π. The state-value function can be seen as a notion of how good it is to be in a certain state s in terms of future rewards under the policy π. Note that, in contrast to the discounted return g_t, the state-value function $V(s_t)$ can be known exactly at time point t, even for stochastic environments.

An important property of the state-value function is the recursive relationship:

$$\begin{aligned} V^\pi(s_t) &= \mathbb{E}_\pi\big[G_t \mid S_t = s_t\big] = \mathbb{E}_\pi\big[R_{t+1} + \gamma G_{t+1} \mid S_t = s_t\big] \\ &= \mathbb{E}_\pi\big[R_{t+1} \mid S_t = s_t\big] + \gamma \mathbb{E}_\pi\big[G_{t+1} \mid S_t = s_t\big] \\ &= \mathbb{E}_\pi\big[R_{t+1} \mid S_t = s_t\big] + \gamma \mathbb{E}_{s_{t+1}\sim\rho|\pi}\big[\,\mathbb{E}_\pi[G_{t+1} \mid S_{t+1} = s_{t+1}] \mid S_t = s_t\big] \\ &= \mathbb{E}_\pi\big[R_{t+1} \mid S_t = s_t\big] + \gamma \mathbb{E}_{\ s_{t+1}\sim\rho|\pi}\big[V^\pi(s_{t+1}) \mid S_t = s_t\big] \\ &= \mathbb{E}_\pi\big[R_{t+1} + \gamma V^\pi(S_{t+1}) \mid S_t = s_t\big], \end{aligned} \tag{2.8}$$

called the *Bellman equation for* V^π. The subscript $s_{t+1} \sim \rho|\pi$ denotes, in a slight abuse of notation, that the next state s_{t+1} is a random variable, drawn from the distribution $\rho(s_{t+1}, r_{t+1}|s_t, a_t \sim \pi)$. Note that we make use of the Markov property when we assume that the only relation between G_{t+1} and s_t is through the subsequent state s_{t+1}.

In analogy to the state-value function, we define the *action-value function* as the expected value of the discounted return, given the current state and the current action:

$$Q^\pi(s, a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\Big[\sum_{i=t+1}^{T} \gamma^{i-t-1} \cdot R_i \mid S_t = s, A_t = a\Big], \tag{2.9}$$

where the index π at the expectation denotes that, after choosing action a in timestep t, the policy π is followed. The action-value function can thus be seen as a notion of how good it is to take the action a in the state s under the policy π. The current action a can be of arbitrary probability under the policy π. Because of the common notation of the action-value function by the letter Q, the action-values are often called Q-values.

Knowing the action-value function, we can infer the state-value function through:

$$V^\pi(s) = \mathbb{E}_\pi\big[Q^\pi(s, a \sim \pi)\big] = \mathbb{E}_\pi\big[Q^\pi(s, A)\big], \tag{2.10}$$

where $a \sim \pi$ denotes that the current action a is a random variable that is, just as all following actions, drawn from the policy π. Vice versa, we can infer the action-value function from the state-value function through:

$$Q^\pi(s, a) = \mathbb{E}\big[R_{t+1} + \gamma V^\pi(S_{t+1}) \mid S_t = s, A_t = a\big]. \tag{2.11}$$

Note that we dropped the subscript π from the expectation as all dependence on the current policy is encapsulated in the state-value function of the subsequent state.

Similar to equation 2.8, we can write the *Bellman equation for* Q^π:

$$Q^\pi(s_t, a_t) = \mathbb{E}_\pi\big[R_{t+1} + \gamma Q^\pi(S_{t+1}, A_{t+1}) \mid S_t = s_t, A_t = a_t\big], \tag{2.12}$$

that recursively defines the action-value function.

Intuitively, knowing about the effects of our actions should enable us to choose those actions that maximise the expected discounted return. We will now explore how to select a policy that does so. For the sake of clarity of notation, we will assume the fully observable case in our further discussion of RL algorithms. In most cases, the formulation is easily generalised to the partially observable case, by replacing the agent's state with its history in the policy and the estimated value functions.

2.2.2. Acting Optimally

As discussed, a policy π is a function that maps from a state s (respectively the history h of observations and actions in the partially observable setting) to a probability distribution over the available actions a. For any MDP, there exists an infinite number of possible policies $\pi \in \Pi$, assigning arbitrary probabilities to all actions in all possible states. A specific policy can be seen as better than another one, if it tends to yield higher discounted returns. In particular, if we find

a policy π' that assigns equal action probabilities as another policy π in all states except s'

$$\pi'(s) = \pi(s),\ \forall s \in \mathcal{S} \setminus s' \tag{2.13}$$

but yields higher returns in the state s':

$$V^{\pi'}(s') = Q^{\pi}(s', a \sim \pi') > V^{\pi}(s'), \tag{2.14}$$

we say that the policy π' improves over π.

For any state of the MDP exists at least one *optimal action*, that yields the highest possible expected discounted return. In terms of the action-value function, the optimal action is $\arg\max_a Q^{\pi}(s, a)$. The policy that always chooses this action, and therefore maximises the state-value function, is called the *optimal policy*:

$$\pi^* = \arg\max_{\pi} V^{\pi}(s),\ \ \forall s \in \mathcal{S}. \tag{2.15}$$

In case there exists a state for that more than one action yields the highest possible expected discounted return, there is an infinite number of optimal policies, that assign arbitrary probabilities to the set of optimal actions. We will denote the state-value function and action-value function under the optimal policy by V^* and Q^*, respectively.

The optimal policy can also be defined in terms of the action-value function, as a mapping which always chooses the action that greedily maximises the action-value function:

$$\pi^* :\ \ \arg\max_{a} Q^*(s, a),\ \ \forall s \in \mathcal{S}. \tag{2.16}$$

The term greedy refers to the problem solving heuristic of choosing a locally optimal action, while not considering its long term implications. The beauty of optimal value functions is that choosing locally optimal actions is also the globally optimal solution, as the value function takes into account all future rewards.

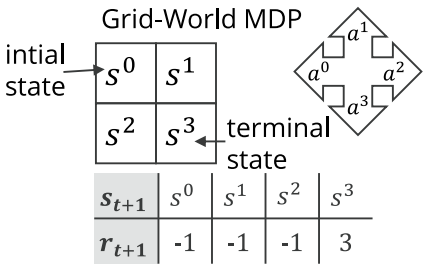
Using the optimal policy, we can rewrite equation 2.8 in the *Bellman optimality equation for* V^*:

$$V^*(s_t) = \max_{a_t} \mathbb{E}\big[R_{t+1} + \gamma V^*(S_{t+1}) \mid S_t = s_t, A_t = a_t\big] \tag{2.17}$$

and 2.12 in the *Bellman optimality equation for* Q^*:

$$Q^*(s_t, a_t) = \mathbb{E}\big[R_{t+1} + \gamma \max_{a_{t+1}} Q^*(S_{t+1}, a_{t+1}) \mid S_t = s_t, A_t = a_t\big]. \tag{2.18}$$

The result that we can maximise the expected discounted return through greedily choosing the action with the highest action-value is very important, as it tells us that acting optimally is relatively simple when we exactly know the optimal value function. However, in most cases, both value functions are unknown, which requires RL algorithms to estimate them and act optimally with respect to the estimated function. In the following sections, we will introduce several strategies to solve an MDP problem when the value functions are unknown.



s_{t+1}	s^0	s^1	s^2	s^3
r_{t+1}	-1	-1	-1	3

Optimal Value Functions

s_t	V^*
s^0	2
s^1	3
s^2	3

Q^*	a^0	a^1	a^2	a^3
s^0	1	1	2	2
s^1	1	2	2	3
s^2	2	1	3	2

Optimal Policy

π^*	a^0	a^1	a^2	a^3
s^0	0	0	0.5	0.5
s^1	0	0	0	1
s^2	0	0	1	0

Figure 2.3.: Minimal grid-world example of a tabular environment. The agent starts in state s^0 and can move from one square to the next along the chosen direction. Upon arriving on a new square, the agent gets a reward of -1; the only exeption is the terminal state s^3, where the agent receives a reward of 3 and the episode ends. The tables on the right show the optimal value functions for a discount factor $\gamma = 1$ as well as the optimal policy.

2.3. Tabular Learning

Thus far we have made no assumptions about the state- and action-spaces $\mathcal{S}$ and $\mathcal{A}$. The nature of these two spaces, however, has a strong influence on the available solution strategies. In particular, we may differentiate between the cases of finite and infinite sets $\mathcal{S}$ and $\mathcal{A}$. In the case of a finite state and action spaces, there exists a fixed number of possible states and available actions:

$$\mathcal{S} = [s^1, s^2, \dots, s^N],\ N \in \mathbb{N}, \tag{2.19}$$

$$\mathcal{A} = [a^1, a^2, \dots, a^M],\ M \in \mathbb{N}. \tag{2.20}$$

We will call this the case of a $discrete$ state- respectively action-space.

For infinite state- and action-spaces, we will focus on the case of states and actions being real-valued vectors — called the *continuous* case:

$$\mathcal{S} \subseteq \mathbb{R}^N,\ N \in \mathbb{N}, \tag{2.21}$$

$$\mathcal{A} \subseteq \mathbb{R}^M,\ M \in \mathbb{N}, \tag{2.22}$$

where $X \subseteq Y$ denotes that X is equal to, or is a convex subset of Y.

Of course, we are not limited to state- and action-spaces both being either discrete or continuous. We may encounter a continuous state-space, paired with a discrete action space or vice versa or even spaces that consist of some discrete and some continuous dimensions. In this section we will consider the case of both spaces being discrete. We will discuss the other cases in the following sections 2.5, 2.6 and 2.7.

If both the state- as well as the action-space are discrete, we can write the policy as a table, where the rows represent individual states, and the columns represent the available actions. Every entry then assigns a probability of choosing the respective action, when being in the respective state. Importantly, the tabular case allows us to represent all these functions exactly (at least up to the point of machine precision), while using a finite amount of memory.

Figure 2.3 shows an example of a tabular environment. In this kind of environment, typically called *grid-world*, the agent has to navigate through a two-dimensional field of discrete states, much like a checkers board. In the example, the agent starts out in the initial state s^0 and can decide in every timestep t, in which direction to go (left, up, right or down). The received reward

depends only on the subsequent state s_{t+1}, and the episode ends when the agent reaches the *terminal state* s^3. The tables on the right show the optimal state-value function and action-value function for a discount factor $\gamma = 1$ as well as the optimal policy. Because the episode ends after reaching the terminal state, no further rewards can be collected and no further action can be taken, which is why the tables do not include s^3. Of course, it would not be necessary to compute both the state-value function and the action-value function, as the MDP can be solved when knowing either one. Generally, the action-value function enables easier inference of the optimal policy, while the state-value function has the advantage of requiring less memory. Note that the depicted policy is just one of the optimal ones, as assigning arbitrary probabilities to actions a^2 and a^3 when being in state s^0 does not change the expected discounted return. We will later come back to this example to explain several concepts.

2.3.1. Dynamic Programming

Coming up with the correct value functions and the optimal policy in the example from figure 2.3 is trivial due to the very simple environment dynamics. Importantly, knowing the dynamics function ρ allowed us to figure out the value functions without actual interaction with the environment. Through the knowledge of the environment dynamics and the Bellman equation, we can formulate the problem as a system of linear equations, which can be solved by any suitable solver. In particular, methods of *Dynamic Programming* have been applied to solve MDPs, as they scale significantly better to large state spaces than other methods (Sutton and Barto, 2018).

The Bellman equation defines the value function of a state in terms of itself in a subsequent state, which is also unknown. This apparent catch-22 can be solved by the *Policy Evaluation* algorithm. Initially, this algorithm randomly guesses the tabular value function and then repeatedly applies:

$$\begin{aligned} \hat{V}^{\pi}_{k+1}(s) &= \mathbb{E}_{\pi}\left[R_{t+1} + \gamma \hat{V}^{\pi}_{k}(S_{t+1}) \mid S_t = s\right] \\ &= \sum_{a} \pi(a|s) \sum_{s',\, r} \rho(s', r|s, a)[r + \hat{V}^{\pi}_{k}(s')],\ \forall s \in \mathcal{S}, \end{aligned} \tag{2.23}$$

where $\hat{V}_k$ denotes the estimation of the state-value function at the k-th iteration of the Policy Evaluation algorithm. This equation is executed until the state-value function converges, meaning that the computed values no longer change and therefore $\hat{V}^{\pi}_{k} = V^{\pi}$. For the tabular case, the Policy Evaluation algorithm provably converges to the true state-value function (Bellman, 1958). This notion of iterative, recursive approximation is referred to as *bootstrapping* (originating from the notion of pulling oneself up by the bootstraps). In the following, two popular methods for solving MDPs with Dynamic Programming are introduced.

Policy Iteration

The *Policy Iteration* algorithm consists of two steps that are executed alternately. The first step is the Policy Evaluation algorithm (equation 2.23), that iteratively approximates state-value function for the current policy. As we usually cannot wait until the algorithm has fully converged, it is mostly run until the change in the approximated state-value function is sufficiently small and thus $\hat{V}^{\pi}_{k} \approx V^{\pi}$.

In the second step, called *Policy Improvement*, the current policy is improved towards choosing

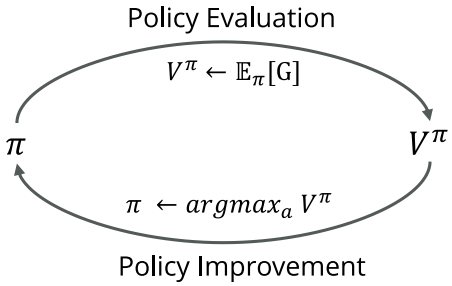


Figure 2.4.: The Policy Iteration algorithm alternately executes two steps. In the first step, called Policy Evaluation, the value function is approximated. In the second step, called Policy Improvement, the policy is optimised as to yield the highest expected return.

better actions – according to the estimated state-value function:

$$\begin{aligned}\pi_{l+1}(a|s) &= \arg\max_a \mathbb{E}\big[R_{t+1} + \gamma \hat{V}^{\pi_l}(S_{t+1}) \mid S_t = s, A_t = a\big] \\ &= \arg\max_a \sum_{s',\, r} \rho(s', r|s, a)[r + \hat{V}^{\pi_l}(s')],\ \forall s \in \mathcal{S}.\end{aligned} \tag{2.24}$$

The tabular nature of the dynamics function allows us here to replace the expectation by a summation over the finite number of possible subsequent state-reward pairings. The policy and state-value function usually are randomly initialised. The Policy Iteration algorithm provably converges to the optimal policy π^* (Bellman, 1958).

Figure 2.4 shows the concept of the Policy Iteration algorithm. This approach of alternating evaluation of the value function and improvement of the policy is common to most RL algorithms that approximate value functions.

Value Iteration

The *Value Iteration* algorithm, in contrast to Policy Iteration, does not apply equation 2.23 until convergence. Instead, it executes only only a single step of Policy Evaluation before the Policy Improvement step (equation 2.24). An equivalent interpretation is that Value Iteration iteratively approximates the optimal state-value function through applying the Bellman optimality equation (equation 2.17) to the current estimate:

$$\begin{aligned}\hat{V}^*_{k+1}(s) &= \max_a \mathbb{E}\big[R_{t+1} + \gamma \hat{V}^*_k(S_{t+1}) \mid S_t = s, A_t = a\big] \\ &= \max_a \sum_{s',\, r} \rho(s', r|s, a)[r + \hat{V}_k(s')],\ \forall s \in \mathcal{S},\end{aligned} \tag{2.25}$$

until the estimation has converged. After convergence, the optimal policy can be followed by greedily choosing the action that yields the highest expected discounted return (equation 2.24). Value Iteration has the same convergence guarantees as Policy Iteration but has been observed to converge much faster (Sutton and Barto, 2018).

The Policy Iteration and Value Iteration algorithms thus differ in the number of steps of Policy Evaluation that they execute before improving the policy. While Policy Iteration takes as many steps as needed for the value function to converge, the Value Iteration algorithm only takes a single step. The Generalised Policy Iteration (GPI) algorithm closes the gap between the two. In GPI, any number of Policy Evaluation steps is allowed before Policy Improvement. In particular, we may evaluate the state-value function of some states more often than of others. This is very

useful when certain states are barely ever visited under the current policy.

2.3.2. Monte Carlo Methods

So far we have assumed to have an explicit expression for the dynamics function. In most real-world cases, however, we cannot obtain an exact model of the environment dynamics. Even if we define the model ourselves, like in a complex simulation, it is often still infeasible to obtain the explicit form of the dynamics function that we would need in order to apply Dynamic Programming algorithms. We therefore need methods to approximate these algorithms by leveraging experience that is obtained through interaction with the environment. More precisely, the expected value of the state-value function (equation 2.7) has to be approximated from sampled data.

A sampled realisation of a random variable is an unbiased estimator of its expected value. We could therefore replace the expected value over the subsequent state and reward by the actual values, encountered through interaction with the environment. Unfortunately, the stochastic nature of the dynamics function gives rise to a high variance in this estimation and, therefore, results in distorted value functions and, ultimately, suboptimal policies. The obvious solution to high variance estimates is averaging over multiple sampled trajectories, as the variance of the estimation of the expected value falls with $1/\sqrt{N}$, where N is the number of samples. We may therefore approximate the state-value function of a state s as the average over the sampled discounted return from individual episodes, starting from the respective state:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\Big[\sum_{i=t+1}^{T} \gamma^{i-t-1} \cdot R_i \mid S_t = s\Big] \approx \frac{1}{n}\sum_{n=0}^{N}\sum_{i=t_n+1}^{T_n} \gamma^{i-t_n-1} \cdot r_{i,n}, \quad \text{with } s_{t_n,n} = s \;\forall n \in [1, N], \tag{2.26}$$

where the subscript $t_n,\, n$ denotes the t_n-th timestep of the n-th episode, and T_n denotes the length of the n-th episode. We here implicitly assume that the policy π is followed at all times. Note that a single episode can be used to estimate the value function for many states, as we can construct a trajectory from every state that is encountered during the episode to the terminal state. This class of algorithms was termed *Monte Carlo Methods* in Sutton and Barto, 2018, even though this terminology slightly clashes with the usual definition of Monte Carlo Methods as a general method of numerical problem solving through repeated random experiments.

According to the law of large numbers, the estimation converges to the true state-value function V^{π} in the limit $N \to \infty$ as long as the policy allows all states to be visited infinitely often. Of course, sampling an infinite number of trajectories is impractical. In most cases, however, it is sufficient to estimate the value function until it only changes marginally under the addition of new data. After approximate convergence of the state-value function, we can take a Policy Improvement step. This approach may be seen as the data-driven pendant to the Policy Iteration algorithm from the preceding section (see figure 2.4). Instead of estimating V^{π} until convergence before improving the policy, we can also use an algorithm similar to Value Iteration, that alternately collects a new episode of data to improve its value estimate according to equation 2.26 and then optimises its policy after every episode. Note however that this alternative thus far lacks a complete formal proof of convergence (Tsitsiklis, 2003).

In practice, RL methods do not keep a full history of episodes in order to apply equation 2.26.

Instead, value functions are usually updated incrementally by:

$$\hat{V}^{\pi}_{k+1}(s) = \hat{V}^{\pi}_{k} + \alpha \cdot [g_{t,k} - \hat{V}^{\pi}_{k}(s)], \text{ with } s_{t,k} = s, \tag{2.27}$$

where α is the *learning rate*, that defines the speed of the adaption. Of course, this is not an exact equivalent to equation 2.26, as it weighs the trajectories so that more recently encountered discounted returns contribute stronger to the current value estimate. While this can prohibit full convergence, it adds the advantages of being able to track value functions for non-stationary statistics and to require significantly less memory.

2.3.3. Temporal Difference Learning

A major drawback of estimating value functions with Monte Carlo Methods (equation 2.26) is that we have to wait until the very end of an episode to update our estimation of the value function. This also implies that the interaction of the MDP has to fall into separate episodes, or else the agent can never learn anything. We can solve this problem by reintroducing the concept of bootstrapping from the Bellman equations (equations 2.8 and 2.12) and replacing the expected values with unbiased estimators. Inserting the bootstrapped estimate of the state-value function for a given state into equation 2.27 yields:

$$\hat{V}^{\pi}_{k+1}(s) = \hat{V}^{\pi}_{k} + \alpha \cdot [r_t + \gamma \hat{V}^{\pi}_{k}(s_{t+1}) - \hat{V}^{\pi}_{k}(s)], \text{ with } s_t = s, \tag{2.28}$$

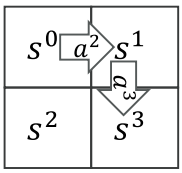
where we have dropped the index of the episode for convenience. The expression

$$r_t + \gamma \hat{V}^{\pi}_{k}(s_{t+1}) - \hat{V}^{\pi}_{k}(s_t) =: \delta_t \tag{2.29}$$

is called the TD-error, as it represents the error between the old estimate of the state-value function and the new, improved estimate, which incorporates the information of the experienced transition.

Applying equation 2.28 to every new, sampled transition s_t, r_{t+1}, s_{t+1} yields the *TD(0)* algorithm which is the "vanilla" version of *Temporal Difference Learning (TDL)*. For a fixed policy, the TD(0) algorithm has been proved to converge to the true state-value function, if the learning rate is sufficiently small and the policy allows reaching every state (Sutton, 1988).

In TDL, new data in the form of a reward propagates stepwise through the state-value function of different states as we employ equation 2.28. Importantly, this means that, in contrast to the Monte Carlo equivalent, a single pass over all encountered states may not fully incorporate the knowledge of the new data into the state-value function since the values that we bootstrap from may change when updating other state-values. To illustrate this, figure 2.5 shows how the example of figure 2.3 is learned by the Monte Carlo algorithm and the TD(0) algorithm. We initially guess all state values to be zero and choose the learning rate to be 1. Starting in the initial state s^0, we sample the trajectory: s^0, a^2, s^1, a^3, s^3 with the reward sequence $-1, 3$. In Monte Carlo estimation (equation 2.27), we would update the state-value function of all encountered states by the discounted return that followed the respective state, obtaining a new estimate of $\hat{V}^{\pi}(s^0) = 2$ and $\hat{V}^{\pi}(s^1) = 3$. The value function can be learned as soon as the entire trajectory is sampled. Note that repeatedly applying equation 2.27 does not change this estimate. In TDL, we apply equation 2.28 to all encountered states after every step in the environment. After the first step, going from s^0 to s^1, we obtain a new estimate of $\hat{V}^{\pi}(s^0) = -1$ (the bootstrapped value of



Trajectory

t	s_t	a_t	r_t
0	s^0	a^2	
1	s^1	a^3	-1
2	s^3		3

Monte Carlo Methods

t	$\hat{V}(s^0)$	$\hat{V}(s^1)$	$\hat{V}(s^2)$
0	0	0	0
1	0	0	0
2	2	3	0

Temporal Difference Learning

t	$\hat{V}(s^0)$	$\hat{V}(s^1)$	$\hat{V}(s^2)$
0	0	0	0
1	-1	0	0
2	-1	3	0

Figure 2.5.: Estimation of the state-value function in the grid-world MDP of figure 2.3 with Monte Carlo Methods respectively the TD(0) algorithm. In the initial timestep, there is no reward; and in the terminal state, no further action is executed. In Monte Carlo Methods, the update of the value function is executed at the end of the episode; in TDL, the values are updated after every timestep.

s^0 is 0, the obtained reward is -1). After the second step, we update $\hat{V}^{\pi}(s^1) = 3$ but leave the estimation of $s^0 = -1$. Note that subsequently applying equation 2.28 to the transition from s^0 to s^1 for a second time would lead us to the same estimate as the Monte Carlo version because we would incorporate the updated value of s^1 into our estimate of s^0. To conclude, Monte Carlo Methods thus yield a better estimate of the true discounted return, whereas Temporal Difference Learning (TDL) methods let us update the value function before ending the episode.

The preceding example illustrates that applying equation 2.28 every time we sample a new transition may not be enough. In particular, for a transition from state s to s', we may improve our estimate of the value of s' in subsequent timesteps and can thus reapply 2.28 to obtain a better estimate of the value of s (bootstrapping from s'). Oftentimes it makes therefore sense to save former transitions in a so-called *replay buffer* and revisit them later on (the idea of a replay buffer was introduced in Lin, 1992).

As we have seen, the difference of TD(0) and Monte Carlo Methods is the number of future rewards that are used to estimate the state-value function. The two methods can be generalised in the framework of *n-step bootstrapping*, where the estimate consists of n future rewards and a bootstrapped value of the state in timestep $t+n$. The TD-error in the case of n-step bootstrapping thus reads:

$$\delta_t = \sum_{i=1}^{n} \gamma^{i-1} r_{t+i} + \gamma^n \hat{V}^{\pi}(s_{t+n}) - \hat{V}^{\pi}(s_t). \tag{2.30}$$

Using n reward-steps for the update provides a trade-off between the good estimator of the discounted return of Monte Carlo Methods and the immediate feedback of TD(0). For a moderate n (usually between two and five) the n-step version of TDL has been observed to converge significantly faster than Monte Carlo Methods or TD(0).

In order to improve at a given task, a learning algorithm needs to leverage its knowledge of the value function towards choosing better actions. Unfortunately, if the dynamics function is not explicitly known, we cannot use equation 2.24 for the Policy Improvement step. It is therefore more common in RL to estimate the action-value function instead of the state-value function. Using the action-value function, we can easily execute a Policy Improvement step without any knowledge about the dynamics function by taking the action with the highest value for a given state:

$$\pi_{l+1}(a|s) = \arg\max_a \hat{Q}^{\pi_l}(s,a),\ \forall s \in \mathcal{S}. \tag{2.31}$$

Thus far, we have concentrated on the state-value function because it provides a more

straightforward introduction to the theory. However, as most modern RL algorithms work with action-values, we will now shift our focus more towards approximating action-value functions. In the following, several popular TDL algorithms for estimating the action-value function are introduced.

SARSA: On-Policy Temporal Difference Learning

When replacing the state-value function V in equation 2.28 with the action-value function Q, the TD-error results to $\delta_t = r_t + \gamma \hat{Q}^{\pi}_k(s_{t+1}, a') - \hat{Q}^{\pi}_k(s_t, a_t)$. Note that, when we want to update the action-value function of the state-action pair (s_t, a_t), the action a' is not yet defined. We thus need to select an action-value $\hat{Q}(s_{t+1}, a')$ to bootstrap from, by selecting an action a'.

A straightforward approach is to sample the action of the subsequent timestep a_{t+1} from the policy before updating the value function. In every timestep, the agent, that currently is in state s_t, takes a step in the environment, where the action a_t is sampled from its current policy $\pi(a_t|s_t)$. The environment returns the reward r_{t+1} and the subsequent state s_{t+1}. The agent can now sample the subsequent action a_{t+1} from $\pi(a_{t+1}|s_{t+1})$. We thus bootstrap from the action-value of the action that is actually executed by the agent. The update equation then reads:

$$\begin{aligned} \hat{Q}^{\pi}_{k+1}(s,a) &= \hat{Q}^{\pi}_k(s,a) + \alpha \cdot [r_t + \gamma \hat{Q}^{\pi}_k(s_{t+1}, a_{t+1}) - \hat{Q}^{\pi}_k(s,a)], \\ &\text{with } s_t = s,\ a_t = a. \end{aligned} \tag{2.32}$$

The quintuple $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$, that is needed for every learning step, gives rise to the name *SARSA*. Just as all previous algorithms, the SARSA algorithm alternates between Policy Evaluation and Policy Improvement. In the Policy Evaluation step, the experienced transition is inserted into equation 2.32 to lead to a better estimate of the action-value function. Then, the current policy is optimised through equation 2.31. Note that, since the only changed value is the one of state s_t, the improvement step is also carried out only in this state (compare figure 2.4).

An important observation is that the SARSA quintuple depends on the current policy as the subsequent action a_{t+1} is selected through it. All other elements are defined by the arguments of the function (a_t and s_t) and by the dynamics function (r_{t+1} and s_{t+1}). This dependence of the experience on the current policy constitutes a so-called *on-policy* algorithm. On-policy algorithms allow us to learn the value function of the policy that the agent currently follows, which intuitively may be the obvious choice. However, as we have discussed in section 2.1, in RL we are faced with the problem of carefully balancing exploring and exploiting actions. It may therefore be beneficial to be able to learn the optimal action-value function, while following a policy that takes some suboptimal actions in order to further explore its environment. To achieve this, the dependence of the update equation (eq. 2.32) on the current policy needs to be eliminated.

Q-Learning: Off-Policy Temporal Difference Learning

The Q-Learning algorithm lets us learn the optimal action-value function by assuming that subsequent actions are chosen optimally:

$$\begin{aligned} \hat{Q}^{\pi}_{k+1}(s,a) &= \hat{Q}^{\pi}_k(s,a) + \alpha \cdot [r_t + \gamma \max_{a'} \hat{Q}^{\pi}_k(s_{t+1}, a') - \hat{Q}^{\pi}_k(s,a)], \\ &\text{with } s_t = s,\ a_t = a. \end{aligned} \tag{2.33}$$

The update equation in Q-Learning is defined by the quadruple $(s_t, a_t, r_{t+1}, s_{t+1})$, which has no dependency on the policy that was followed at timepoint t. This allows us to follow a non-optimal policy while learning about the optimal action-value function. Q-Learning therefore is a so-called *off-policy* algorithm.

The most common strategy in Q-Learning is to follow a so-called *epsilon-greedy* policy. This policy takes the greedy action, which maximises the current estimate of the action-value function, most of the time, but with a probability of $\varepsilon \in [0, 1]$ it selects one of the available actions at random. We thus exploit our current knowledge of the MDP by taking optimal actions most of the time so that we may reach states that are hard to encounter when following a purely exploring policy. Additionally, we keep on exploring new actions that may yield higher discounted return than our estimate of the action-value function suggests. As discussed earlier, it can often be beneficial to keep the sampled transitions in a replay buffer and reapply the Policy Evaluation step later on. Another important advantage of off-policy algorithms is the option of reusing data that was collected under a former policy. This is in contrast to on-policy algorithms, where we may only learn from transitions that were experienced under the current policy. We will further explore this advantage when introducing Q-Learning with function approximation in section 2.5.

Double Q-Learning

Assuming the agent to take the action that maximises the action-value function in the next step comes with a problem: Since the estimate of the values of the subsequent state are noisy, taking the maximal value introduces a bias, so that the Q-Learning update tends to overestimate the true discounted return (Thrun and Schwartz, 1993). As an illustration, imagine that the true action-values of all actions in the subsequent state are 0 and our current estimate of the action-values is drawn from a Gaussian with zero-mean but non-zero variance. In this scenario, the maximum operation of the Q-Learning algorithm would tend to overestimate the values.

The *Double Q-Learning* algorithm overcomes this overestimation by decoupling the maximum-operation from the value-estimate. Instead of estimating a single action-value function, it separately learns two action-value functions $\hat{Q}$ and $\widetilde{Q}$ (with different initialisation), and then uses the value of one of the functions but maximises over the other. The two value functions are learned through:

$$\begin{aligned}
\hat{Q}^{\pi}_{k+1}(s,a) &= \hat{Q}^{\pi}_{k} + \alpha \cdot [r_t + \gamma \hat{Q}^{\pi}_{k}\big(s_{t+1}, \arg\max_{a'} \widetilde{Q}^{\pi}_{k}(s_{t+1}, a')\big) - \hat{Q}^{\pi}_{k}(s,a)], \\
\widetilde{Q}^{\pi}_{k+1}(s,a) &= \widetilde{Q}^{\pi}_{k} + \alpha \cdot [r_t + \gamma \widetilde{Q}^{\pi}_{k}\big(s_{t+1}, \arg\max_{a'} \hat{Q}^{\pi}_{k}(s_{t+1}, a')\big) - \widetilde{Q}^{\pi}_{k}(s,a)], \\
&\text{with } s_t = s,\ a_t = a.
\end{aligned} \tag{2.34}$$

Note that this algorithm doubles both the complexity of the update as well as the memory footprint of regular Q-Learning.

Summary of Tabular Methods

Figure 2.6 shows a unified framework of the discussed methods. For discrete state and action spaces, we can describe all possible trajectories of states and actions as a decision tree where nodes are states, edges are actions and leafs are the terminal states. The naive method of traversing the entire tree of possible states and actions, referred to as *exhaustive search*, was not discussed, as it is usually infeasible. Dynamic programming simplifies the exhaustive search by

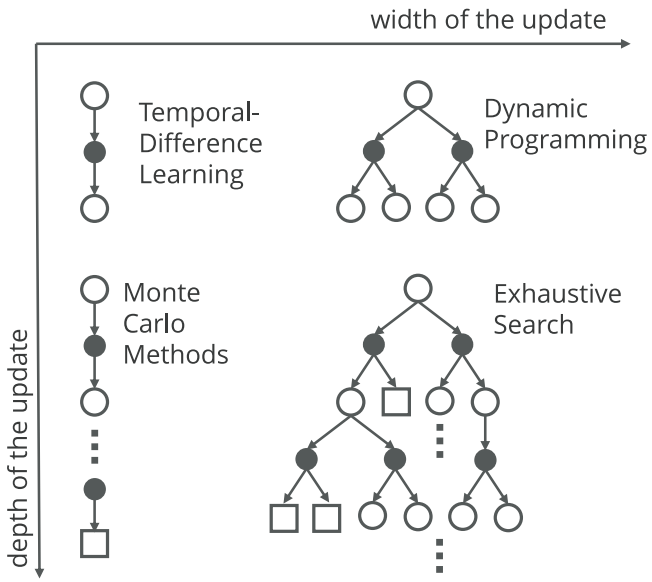


Figure 2.6.: Summary of tabular learning algorithms. Shows the discussed learning methods in a unified framework. White circles show states; black circles, actions and squares, terminal states. Adopted from Sutton and Barto, 2018

considering only one decision-step at a time. It averages over possible actions and subsequent states for the next step and thus considers the entire breadth of possibilities for a single decision in the tree. Monte Carlo methods adopt the opposite strategy and consider just a single path of the entire depth of the tree. Finally, Temporal Difference Learning considers the smallest possible unit of only one transition. Note that it is possible to construct an intermediate algorithm that may consider part of the width and depth of the tree but not the entire tree.

This concludes our treatment of the tabular case. The described principles and algorithms are the foundation of all modern RL methods and provide a crucial insight into the workings of more involved algorithms. We have not been able to truly give credit to the breadth or depth of the field and would strongly recommend reading the part on Tabular Learning in Sutton and Barto, 2018 for a deeper understanding of all areas of RL.

2.4. Function Approximation

Tabular learning provides a powerful framework for solving MDPs with a finite number of discrete states and actions. However, with large state and action spaces, the value functions and policies are very memory consuming and quickly become unfeasible to store on today's hardware. In the continuous case (e.g. if the state is the position and speed of a car) arises the need to divide the state and action spaces into a finite set of values. This discretisation is weird and limiting to the expressiveness of states and distinctiveness of actions.

Moreover, in many cases the representation of the state- and action-values through individual values may be counterproductive. In particular, if a tabular agent encounters a previously unseen state it has no notion on how to behave. However, having seen similar states, an agent should leverage its knowledge on how to navigate the similar situation towards deciding on the current action to take. This feature of interpolating the values of unseen states through the knowledge of similar states is called generalisation.

The identification of similar states requires the state space to admit some metric in order to

asses the similarity of two states. To do this we will consider individual states to be points in multidimensional continuous state spaces. Instead of a discrete value that uniquely identifies a state, we will thus denote states as real-valued vectors:

$$\boldsymbol{s} = (s^1, s^2, s^3, \dots s^N). \tag{2.35}$$

Note that we have reassigned the superscripts to denote the dimensions of the state space, instead of being an unique identifier of discrete states. For identifying similar states, we then require the state space to admit some spatial coherence. In particular, we may assume the value function to be a continuously differentiable manifold on the features of the state vector so that a marginal change in the state may only result in a small change in the value function. In the tabular example in figure 2.3, we could, for instance, use a state representation of one vertical and one horizontal coordinate. The unique states would then translate to the coordinates: $s^0 = (0, 0)$, $s^1 = (0, 1)$, $s^2 = (1, 0)$, $s^3 = (1, 1)$.

The use of parametric functions offers an alternative to saving individual values for all possible states and actions. For example, one might use a linear combination of the features of the state vector to represent the optimal state-value function of figure 2.3:

$$\hat{Q}^*(\boldsymbol{s}) = \theta_0 + \theta_1 \cdot x_1 + \theta_2 \cdot x_2, \text{ with } \boldsymbol{s} = (x_1, x_2), \tag{2.36}$$

where the coefficients θ_0, θ_1 and θ_2 are the parameters of the function that an ML algorithm tries to find. Notice that we here denote the features of the state space by x_1 and x_2 instead of s^1 and s^2, in order to not confuse them with the discrete identifiers for the unique states of the tabular example.

For parameters $\theta_0 = 1, \theta_1 = 1, \theta_2 = 1$, we can verify that the state-value function is represented exactly (except for the terminal state, which, per definition, has a value of 0). This should come as no surprise, as we replace the three state-values by a function with three parameters and, therefore, may represent arbitrary values. Our goal, of course, is to use functions with fewer parameters than the number of distinct states of the MDP, so that we may express the value function of an arbitrarily large state space, using a limited amount of memory. This means, however, that in general we may no longer exactly represent the corresponding, correct tabular values of all states. Furthermore, it may not be possible to adjust our value-estimate of one state without changing the ones of other states. Consequently, it may also be impossible to find the true optimal policy.

Function approximation in the Machine Learning framework is concerned with the study of how to select one particular function among a well-defined class of candidates, that most closely matches a number of data points. Oftentimes, this class is defined by a summation over different terms that are weighted by a set of parameters (in equation 2.36, for example, we sum over weighted linear terms). These parameters can be learned so that the goal of the algorithm is to find the set of parameters that yields the function that best describes the available data. The quality of a candidate is defined by a so-called *objective-function*, that assigns a numerical value to the pair of a function and a set of data points. For example, in the linear case, we often try to minimise the mean of squared distances between the data and the approximating function — called the *Mean Squared Error (MSE)*. Figure 2.7 shows an example of function approximation with a linear model. The left image shows a group of data points and a straight line that is the best fit to the data in an MSE sense (the line has a slope of 0.49 and an offset of 0.58). The right image shows a contour plot of the MSE for different offsets and slopes and marks the spot of

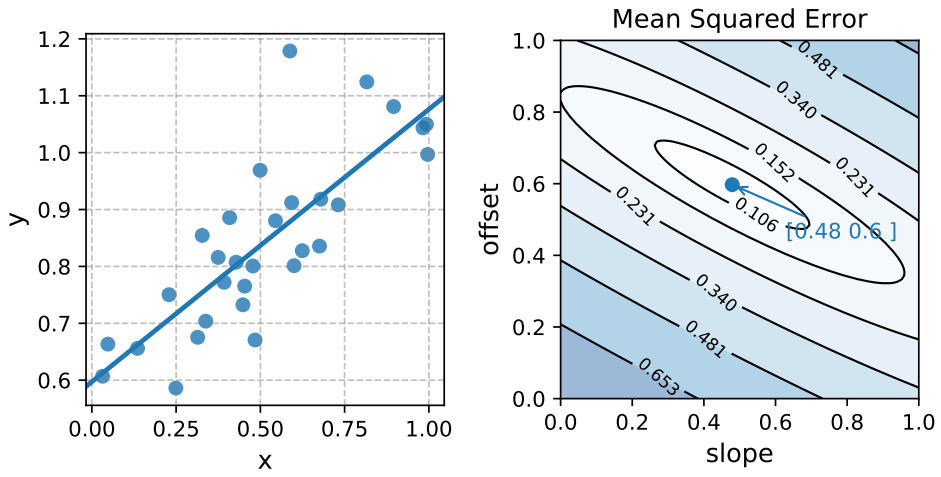


Figure 2.7.: Example of linear function approximation. Left: The dots depict several data points, which could be individual measurements of some physical entity. The line is the linear function that best describes the data in a MSE sense. Right: A contour plot of the MSE for different offsets and slopes of the linear function. The dot marks the best set of parameters, which correspond to the line in the left plot.

the lowest MSE.

The field of Machine Learning and function approximation is broad and will not be fully covered in this work. For a deep dive into most relevant algorithms, we strongly recommend Hastie et al., 2004 or Bishop, 2006 for a comprehensive Bayesian interpretation. In this section we will give a brief introduction to function approximation with Neural Networks (NNs) as they are the current weapon of choice in Machine Learning in general, and Reinforcement Learning in particular, and will also be made use of extensively in this work. For a thorough study of NNs, the interested reader is referred to Goodfellow et al., 2016.

2.4.1. Neural Network Architecture

Previously, we have introduced the idea of function approximation as a means to parameterise the state-value function. In this section, we will discuss Neural Networks (NNs) without explicitly assuming RL as an application domain. Through this, we hope to paint NNs as the broad and versatile framework that they are.

The Perceptron

The fundamental building block of Neural Networks is the *Perceptron*. Conceptually, it is loosely based on a simple model of biological neurons, which accounts for the word "Neural" in Neural Network. A Perceptron performs a nonlinear mapping from a vector of inputs to a numerical output. Similar to the linear model (equation 2.36), it computes the sum over the weighted entries of its input vector and some offset. The result of the summation is often called the *activation*. Subsequently, it applies some nonlinear operation to the outcome of the summation:

$$\begin{aligned} \hat{y} &= \sigma\Big(\theta_0 + \sum_{i=1}^{N} \theta_i \, x_i\Big) = \sigma(\boldsymbol{\theta} \cdot \boldsymbol{x}), \text{ where} \\ \boldsymbol{w} &= (\theta_0, \theta_1, \ldots, \theta_N), \; \boldsymbol{x} = (1, x_1, x_2, \ldots, x_N)^T. \end{aligned} \tag{2.37}$$

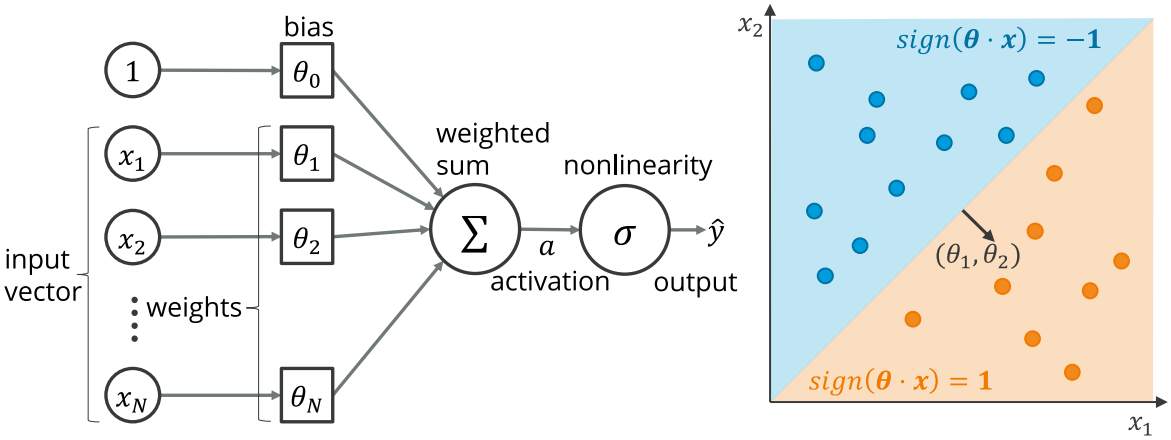


Figure 2.8.: Left: Schematic of a Perceptron. It computes a weighted sum of its inputs and applies some nonlinear activation function to the result. Right: Example of the operation of a Perceptron with a sign function as nonlinearity. A set of data points that lie in a two-dimensional space, each belonging to one of two classes (blue or orange). The Perceptron establishes a linear boundary between the two classes, which, in this simple example, can correctly classify all data points.

Here, $\boldsymbol{x}$ is the input vector, $\hat{y}$ is the computed output, $\boldsymbol{\theta}$ is the parameter vector and σ is some nonlinear function, called the *activation function*. The first element of the parameter vector is called the *bias*, as it is added to the sum irrespective of the input vector. The remaining elements are referred to as *weights*, as they weigh the influence that individual inputs have on the output. Figure 2.8 shows a schematic of a Perceptron (left) alongside with an example of the operation of a Perceptron (right). The Perceptron in the example on the right uses a sign function as nonlinearity, which could be regarded as being loosely based on the function of a biological neuron, that is activated when the intensity of the sum of the ingoing signals surpasses some threshold (Dayan and Abbott, 2005). Even though, in practice, the sign function is rarely ever used as a nonlinearity, this example gives a good intuition of the type of mappings that a Perceptron can perform. In the example, a set of data points is plotted in a two-dimensional space. Each of the data points belongs to one of two classes (blue or orange). The Perceptron maps every input (the coordinates of the data point) to a value of -1 or +1 (or 0, exactly on the boundary), which may be interpreted as the two different classes. The Perceptron divides the space by a linear boundary, where the weight-vector (without the bias) can be interpreted as the normal vector of the dividing line. In this simple example, a linear decision boundary between the two classes can correctly classify all given data points. Note that for different activation functions, the interpretation of the output may be different.

Deep Neural Networks

In most cases, a linear regression model (figure 2.7) or a linear decision boundary (figure 2.8) is not enough to approximate a complex function like the state-value function of a difficult RL problem. Multilayer Perceptrons (MLPs), also called Neural Networks (NNs), take a compositional approach and approximate complex functions through the combination of multiple Perceptrons. More precisely, in NNs, the input vector $\boldsymbol{x}$ is processed by N individual Perceptrons, where each has its own weights and bias. The N outputs of the Perceptrons can then be processed by a subsequent layer of Perceptrons and so on. The final layer of Perceptrons outputs the vector $\hat{\boldsymbol{y}}$. The intermediate layers are called *hidden layers*, and the intermediate output vectors are

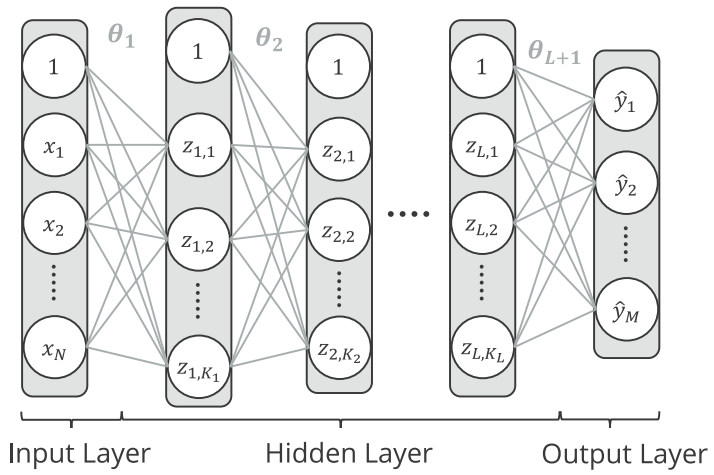


Figure 2.9.: A Multilayer Perceptron that consists of an input layer with N inputs, L hidden layers, with K_l, $l = 1, \ldots, L$ units respectively, and an output layer with M outputs. Each circle, except for the inputs, here represents the weighted sum and the nonlinear mapping operation of a single Perceptron.

denoted $\boldsymbol{z}$. An NN is considered to be "Deep" if it uses more than one hidden layer.

Figure 2.9 shows a schematic of an NN. Note that the signal in the depicted NN strictly flows from the input layer to the output layer. Notably, there exist other types of NN architectures that, for instance, allow for the signal to be convolved (LeCun et al., 1999), skip hidden layers (Szegedy et al., 2014) or save the signals from one timestep and use the saved state for the next (Hochreiter and Schmidhuber, 1997).

Here we will focus on the basic version, shown in figure 2.9. Every layer thus transforms its ingoing signal into an outgoing signal, which may have a different number of dimensions:

$$\begin{aligned} \boldsymbol{z_i}(\boldsymbol{z_{i-1}}) &= \sigma_i(\boldsymbol{\theta_i} \cdot \boldsymbol{z_{i-1}}),\ i = 1, \ldots, L+1, \text{ with} \\ \boldsymbol{\theta_i} &= \begin{pmatrix} \theta_{i,1,0} & \theta_{i,1,1} & \ldots & \theta_{i,1,K_i} \\ \theta_{i,2,0} & \theta_{i,2,1} & \ldots & \theta_{i,2,K_i} \\ \vdots & \vdots & \ddots & \vdots \\ \theta_{i,K_{i+1},0} & \theta_{i,K_{i+1},1} & \ldots & \theta_{i,K_{i+1},K_i} \end{pmatrix} \\ \boldsymbol{z_i} &= (1, z_{i,1}, z_{i,2}, \ldots, z_{i,K_i})^T, \text{ and } \boldsymbol{z_0} = \boldsymbol{x},\ \boldsymbol{z_{L+1}} = \hat{\boldsymbol{y}}. \end{aligned} \tag{2.38}$$

Here, the index i, j, k denotes the k-th input of the j-th Perceptron in the i-th layer and the function σ_i applies the nonlinear mapping to its input vector. $\boldsymbol{\theta_i}$ thus denotes the matrix of parameters of the i-th layer. The entire set of parameters of the model will be denoted θ, without making any explicit assumptions about its structure. Note that the architecture of an NN again borrows from biologic neurons; the weighted connections could be interpreted as synapses, that connect the axons of the neurons in the previous layer to the dendrites of the neurons in the subsequent one, and the summation and activation function as the soma of the neurons (Dayan and Abbott, 2005).

NNs turn out to be able to represent a versatile range of functions and are therefore well-suited to be used as a general function approximator for learning systems. Whereas many other function approximators require a cumbersome manual design of the features of the input space, the complexity and versatility of the mappings that an NN can perform, enables it to deal with raw and unstructured input data. In particular, NNs have been found to find structure in raw

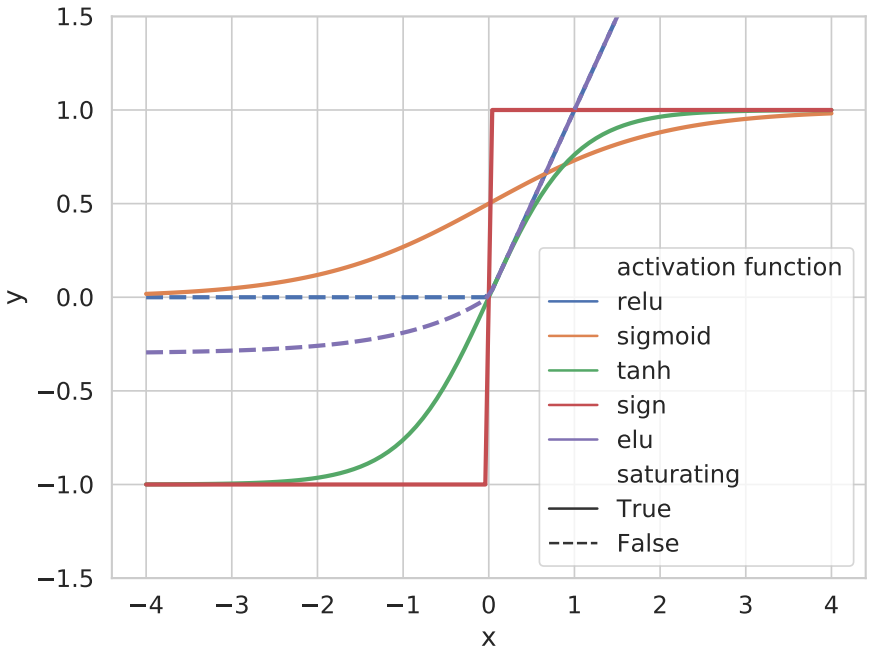


Figure 2.10.: Some common forms of the nonlinear activation function of NNs. An important distinction is the one between saturating and non-saturating activation functions.

sensory data, like the pixels of an image (He et al., 2015a) or the waveform of an audio signal (Oord et al., 2016).

Activation Functions

The activation function σ may generally be any nonlinear function. In most cases however, one of the functions depicted in figure 2.10 is used (sometimes with slight adaptions).

An important distinction is the one between saturating (e.g. sigmoid and tanh) and non-saturating (e.g. relu and elu) activation functions. Saturating activation functions converge to a fixed value for both increasing as well as decreasing inputs. They were the most common choice when Perceptrons were first introduced. However, further research revealed that the vanishing gradient of these functions can impair learning, as gradient-based learning methods may converge very slowly. This is especially true for NNs with many hidden layers. Nowadays, the general suggestion is, therefore, to use saturating activation functions only in cases where a finite output interval is explicitly required (e.g. if the output of the Perceptron is interpreted as a probability, the chosen activation function is usually a sigmoid as it produces values in the [0,1] interval). In this work, we will make use of the *tanh* and the *elu* function that are defined as:

$$\sigma_{elu}(x) = \begin{cases} x & \text{if } x > 0, \\ \alpha \cdot (e^x - 1) & \text{if } x \leq 0, \text{ with } \alpha \in (0, 1) \end{cases} \tag{2.39}$$

$$\sigma_{tanh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.40}$$

In an NN, we may generally define different activation functions for every Perceptron. The common practice is to use a non-saturating function for the hidden layers, whereas the activation function of the output layer may be a non-saturating function, a saturating function or even just the unit function. Finally, we may also employ an activation function that combines multiple

inputs, like the so-called *softmax* function:

$$\sigma_{softmax}(x_n) = \frac{e^{x_n}}{\sum_{i=1}^{N} e^{x_i}},\ n = 1, \ldots, N, \tag{2.41}$$

that transforms its N-dimensional input into an N-dimensional output that sums to one. The softmax function is therefore often used when the NN needs to output a categorical distribution like, for example, probabilities of an object in an image belonging to a set of classes (e.g. dog vs cat vs shark) or the probabilities of discrete actions that an RL agent might take.

2.4.2. Learning Neural Network Parameters

Now that we know how to build an NN and how to infer outputs from inputs, we have to discuss how to obtain a good set of parameters from data. The process of computing a set of parameters from a given dataset is called *training*. While there is no unique way of training an NN, the gradient-based approach that we will describe in this section is considered to be the most effective of currently-known methods and is responsible for many recent advances in different areas of Artificial Intelligence.

Loss Functions

In order to adopt a set of parameters so that our model improves at a certain task, we need to be able to evaluate how well or badly it performs for a specific parameter setting. Using some quantitative evaluation, the formal goal of a training algorithm is then to find the set of parameters that maximises a measure of goodness or minimises a measure of badness. In optimisation, this function is referred to as the *objective-function*. The *loss* is an objective-function that is lower for a better solution. In the example of figure 2.7, we used the MSE as loss function:

$$MSE(\boldsymbol{x}, \boldsymbol{y}) = \frac{1}{N} \sum_{i=1}^{N} \left(y_i - \hat{y}(\boldsymbol{x}_i)\right)^2, \tag{2.42}$$

where $\boldsymbol{x_i}$ and y_i are the input vector and the correct target value of the N respective data points of the so-called *training set*, respectively, and $\hat{y}(\boldsymbol{x_i})$ is the prediction of the model for a given input. The MSE thus is a measure of how well our model predicts the correct outputs of the data.

Depending on the operation that we want to learn, we may choose different loss functions. In the case of the MSE, our goal is to predict a single output value from a vector of inputs. Another common case is that we want to output a categorical probability distribution instead of a single value. Here, the most common loss function is the *categorical crossentropy*:

$$\mathcal{H}(\boldsymbol{x}, \boldsymbol{y}) = \frac{1}{N} \sum_{i=1}^{N} \boldsymbol{y_i}^T \cdot \log\left(\hat{\boldsymbol{y}}(\boldsymbol{x_i})\right), \tag{2.43}$$

where $\boldsymbol{y_i}$ denotes the distribution over the M different options that we want our network to predict (a distribution of a known fact is implemented by a probability of 1 for the correct answer) and $\hat{\boldsymbol{y}}(\boldsymbol{x_i})$ denotes the prediction of our model. Note that the crossentropy is, in fact, not a proper measure of how well our prediction matches the target distribution. A more natural choice would be to use the KL-Divergence as a loss function as it is commonly used to define the distance between two distributions. However, as we will see later on, we are actually interested

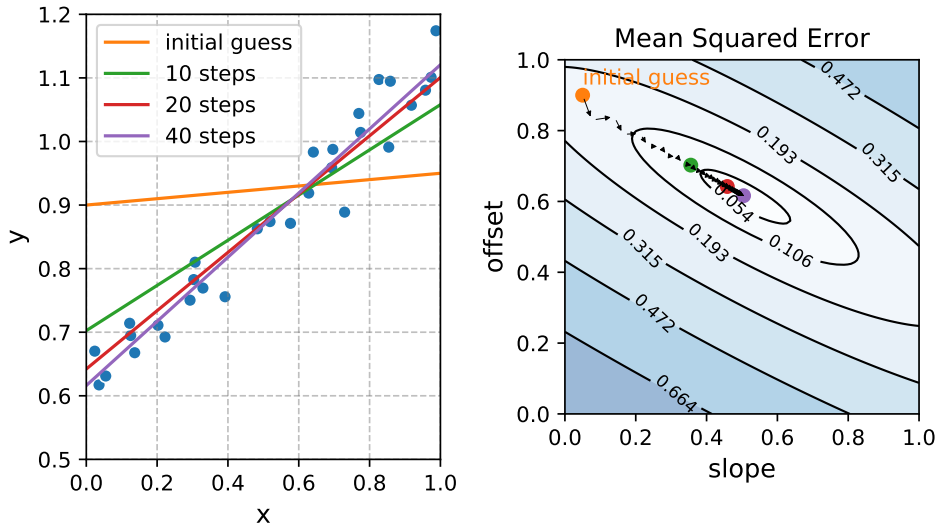


Figure 2.11.: Example of the gradient descent algorithm for a linear regression problem. Left: Blue dots show the individual data points; coloured lines show the predictions of the linear model after 0, 10, 20 and 40 steps of the gradient descent algorithm. Right: Contour plot of the MSE for different values of the slope and offset of the linear model. Coloured dots show the parameters of the lines in the left plot; arrows depict the computed gradients of the loss function with respect to the two parameters for every step of the gradient descent algorithm.

in the gradient of the loss function rather than in its value, which turns out to be the same for crossentropy and KL-Divergence.

There are many more types of loss functions commonly being applied for training NNs. In general, every function that describes our minimisation goal and that is differentiable w.r.t. the model parameters may be used.

Gradient Descent

Computing the gradient of the loss function w.r.t. one parameter of our model gives us a local approximation of how the loss function changes if we modify the parameter. Changing the respective parameter by a small amount in the opposite direction of the gradient should result in a model that yields a slightly lower loss function. In the gradient descent algorithm, we adapt all parameters according to their respective gradients:

$$\theta^{l+1} = \theta^l - \alpha \cdot \nabla_\theta J(\boldsymbol{x}, \boldsymbol{y}, \theta)|_{\theta=\theta^l}, \tag{2.44}$$

where $J(\boldsymbol{x}, \boldsymbol{y}, \theta)$ is a loss function for a set of data points $(\boldsymbol{x}, \boldsymbol{y})$, called the training set, and a set of parameters θ^l, that is obtained after l iterations of gradient descent. $\nabla_\theta J|_{\theta=\theta^l}$ denotes the vector of partial derivatives of the loss function w.r.t. all parameters, evaluated at $\theta = \theta^l$. The learning rate α defines the size of the step that is taken. The initial parameters θ^0 are usually some small, random numbers. Figure 2.11 shows how gradient descent finds a set of parameters for a linear model. The left plot shows a set of data points and four different linear models that are obtained after some steps of the gradient descent algorithm. The right plot shows the MSE for the different models and the trajectory of the parameters. The gradients, depicted by black

arrows in the right image, are computed by:

$$\nabla_\theta J(\boldsymbol{x}, \boldsymbol{y}, \theta) = \frac{\partial}{\partial \theta}\Big(\frac{1}{N}\sum_{i=1}^{N}\big(y_i - \hat{y_i}\big)^2\Big) = -\frac{2}{N}\sum_{i=1}^{N}\begin{pmatrix}1\\ x_i\end{pmatrix}\cdot\big(y_i - \theta_0 - \theta_1 \cdot x_i\big) \tag{2.45}$$

It is important to note that the optimisation problem is not always as easy to solve as in our example. In particular, the loss function usually does not have a convex shape with a single minimum at its centre but may have many local minima as well as flat regions — called plateaus or saddle points. In general, there can be no guarantee that the global minimum of a non-convex loss function is found by gradient descent. However, in Neural Networks this turns out not to be much of a problem since all local minima can be assumed to be somewhat close to the global minimum (Choromanska et al., 2014). Thus, even though we may not find the optimal solution that yields the minimal loss, we can at least be confident to have found a 'good' solution as soon as our algorithm has converged to any local minimum. Note however that it is not uncommon that the gradient descent algorithm gets stuck in saddle points.

To compute the gradient of the loss function, we have to apply our model to every single data point in our training set. While this is easy to do for a model with two parameters and 30 data points (like in figure 2.11), it takes a lot of computational power for complex models with many parameters and large training sets. Instead of computing the exact loss function, we therefore often approximate it by computing it only for a small, random subset of the entire training set. This algorithm, called Stochastic Gradient Descent (SGD) (Robbins and Monro, 1951), shows similar convergence as gradient descent and is usually the preferred choice for training NNs. Using a larger number of training samples to compute the loss results in a better approximation of the true loss; thus, we have to trade off the accuracy of the gradient against computational complexity.

It is important to note that there have been many approaches to improve the performance of gradient descent and SGD. Most notably, the addition of momentum and second-order approximations have been shown to improve convergence. The concept of momentum lets the optimiser take a step in a direction that is defined by a weighted combination of the computed gradient and the direction of the update from the last optimisation step. An optimiser with momentum can be imagined like a ball that runs down a hill. If the ball weighs next to nothing (the case of gradient descent without momentum), it easily gets stuck at plateau points. However, for a heavy ball, the momentum can take the ball over the plateau and prevents it from getting stuck in a flat region. The second notable addition to gradient descent is the utilisation of second-order approximations of the loss function, instead of only a linear approximation. This helps mitigating oscillations of the parameters and preventing divergence due to overly large, or very slow convergence due to overly small learning rates. In figure 2.11, for example, we can see that the initial steps of gradient descent are a lot larger than the latter ones as the MSE-surface gets flatter close to the optimum. Using a second order method, the optimiser may take larger steps in the flat region and could thus speed up convergence. The Adaptive Moment Estimation (ADAM) algorithm (Kingma and Ba, 2014) is one of the most commonly used optimisation algorithms that combines momentum and second-order approximation and will also be used in this work. We have to note, however, that we cannot generally recommend the usage of second-order optimisation algorithms due to recent concerns about their convergence properties (Wilson et al., 2017).

Backpropagation of Errors

For a Deep Neural Network, the gradient can be computed by the *Backpropagation of Errors* algorithm (Rumelhart et al., 1986). The name Backpropagation originates from the fact that, after having computed the loss in a forward step, we have to compute the gradient of the output layer first and then go backwards through the individual layers to compute the gradients. In the forward pass of the algorithm, we compute the output of our model and store all intermediate signals. In the backward pass, we compute the gradient of the loss function w.r.t. each of the weights and biases of the model through the application of the chain rule. First, we compute the gradient of the loss function w.r.t. the activation of the final layer:

$$\frac{\partial J}{\partial a_{K_{L+1},j}} = \frac{\partial J}{\partial z_{K_{L+1},j}} \cdot \frac{\partial z_{K_{L+1},j}}{\partial a_{K_{L+1},j}} = \frac{\partial J}{\partial \hat{y}_j} \cdot \frac{\partial \hat{y}_j}{\partial a_{K_{L+1},j}} = \frac{\partial J}{\partial \hat{y}_j} \cdot \frac{\partial \sigma_{K_{L+1}}}{\partial a_{K_{L+1},j}}, \tag{2.46}$$

where the activation $a_{i,j}$ is the weighted sum over the inputs of the j-th Perceptron of the i-th layer, and the output $z_{i,j}$ is obtained through applying the activation function to the activation. Note that the first term of the right-hand side is the derivative of the loss function w.r.t. the predicted output and the second term is the derivative of the activation function w.r.t. its j-th argument – of both terms we have an analytical form. The gradients of the loss function in the final layer can then be propagated backwards to compute the gradients in previous layers:

$$\begin{aligned} \frac{\partial J}{\partial a_{i,j}} &= \frac{\partial J}{\partial z_{i,j}} \cdot \frac{\partial z_{i,j}}{\partial a_{i,j}} = \frac{\partial \sigma_i}{\partial a_{i,j}} \cdot \sum_{j'=1}^{K_{i+1}} \frac{\partial J}{\partial a_{i+1,j'}} \cdot \frac{\partial a_{i+1,j'}}{\partial z_{i,j}} \\ &= \frac{\partial \sigma_i}{\partial a_{i,j}} \cdot \sum_{j'=1}^{K_{i+1}} \theta_{i+1,j',j} \cdot \frac{\partial J}{\partial a_{i+1,j'}}, \; i = L, L-1, \ldots, 1. \end{aligned} \tag{2.47}$$

Knowing the gradient of the activation of one layer $\boldsymbol{a}_{i+1}$, we can thus infer the gradients of the preceding layer and can compute the gradient of every layer by going backwards through the network.

Finally, we can compute the gradients of all parameters of our NN as the derivative of the respective activations:

$$\frac{\partial J}{\partial \theta_{i,j,k}} = \frac{\partial J}{\partial a_{i,j}} \cdot \frac{\partial a_{i,j}}{\partial \theta_{i,j,k}} = \frac{\partial J}{\partial a_{i,j}} \cdot z_{i-1,k}, \tag{2.48}$$

with $z_{i,0} = 1$, $z_{0,j} = x_j$ and $z_{L+1,j} = y_j$. Having computed all gradients, we can apply equation 2.44 to adapt the parameters. Note that for clarity of notation, we here only compute the gradient for a single training sample. To obtain the gradient for a larger training set, we simply have to average over the gradients of the individual samples.

Figure 2.12 shows a minimal example of the Backpropagation of Errors algorithm for an NN with a single input and a single output and one hidden layer with one hidden unit. Notice that in the forward pass, all calculations rely on signals from the respective previous layer while in the backward pass, all gradients depend on the gradients from the respective subsequent layer.

Initialisation

As a final remark in our introduction to Neural Networks, we want to stress the importance of good initialisation of parameters. In order to not have all Perceptrons in a layer learn the same function, we have to initialise the parameters of every Perceptron differently. This is referred

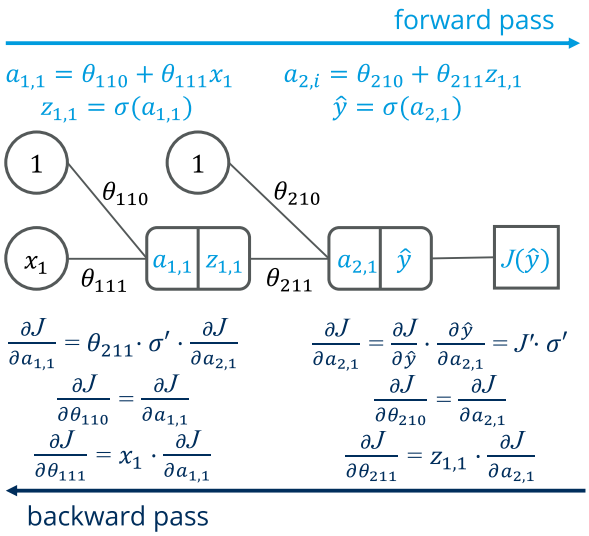


Figure 2.12.: A minimal example of the Backpropagation of Errors algorithm. In the forward pass, all signals are computed — beginning from the input and finishing at the loss function J. In the backward pass, the gradients of the loss function w.r.t. all parameters are computed — starting from the loss function and going backwards to the first layer.

to as *symmetry-breaking* and is usually done by choosing every parameter as a small random number. However, simply sampling every parameter from some fixed distribution turns out to result in poor convergence due to the problem of vanishing gradients. The random initialisation has to be carefully tuned to the number of outputs of the Perceptron preceding the weighted connection, the number of inputs of the Perceptron following the weighted connection and the used activation function. Popular initialisation schemes are the so-called Xavier Initialisation (Glorot and Bengio, 2010) or He Initialisation (He et al., 2015b). For example, for a ReLU activation function, He Initialisation will assign random values, drawn from a Gaussian with zero mean and a variance that is inversely proportional to the number of afferents:

$$var(w_{i,j,k}) = \frac{2}{K_{i-1}} \tag{2.49}$$

2.5. Deep Q-Learning

We can now replace the tabular action-value function of the Q-Learning algorithm by a Neural Network. This method — known as Deep Q-Learning (DQL) — is arguably the most widely known RL algorithm. Even though the idea of using a nonlinear function approximator as the action-value function has been around for a while, most approaches turned out to be unstable or divergent (Tsitsiklis et al., 1997). However, using two further adaptions, Mnih et al., 2015 managed to let a DQL agent successfully learn how to play a range of arcade games with raw pixels as its only input.

The first adaption of Mnih et al., 2015, was to use a replay buffer that saves previously encountered transitions (s, a, r, s'), as already mentioned in section 2.3.3. At every training step, the algorithm samples a random batch of previous transitions and executes a step of SGD, using the MSE of the TD-errors as its loss function. Intuitively, computing a gradient from the last N encountered steps will result in a biased estimate since the state can be assumed to show temporal correlations (for instance if our state is the speed and position of a vehicle, we can

expect both values to change relatively slowly). This is problematic since the adaption of the action-value for one state-action pair may also change all other action-values, and small changes in the action-values can significantly alter the policy. Biases in the gradient can, therefore, prevent convergence of the DQL algorithm. By sampling from a large replay buffer, the effects of strong temporal correlations of the state can be mitigated.

The second adaption deals with the correlation in the two action-values of the TD-error (the action-value of the current state and the action-value of the subsequent state). If two subsequent states, s_t and s_{t+1}, are very similar, the predicted action-values of the two states, used in the Bellman equation will also be correlated. With a biased TD-error, the computed gradients are biased, resulting in poor convergence. Mnih et al., 2015 propose to overcome this correlation by using two different NNs for the two action-values in the TD-error. Whereas $Q(s, a)$ is predicted by the model that is being learned, $Q(s', a')$ is predicted by the so-called *target network*. The target network is not learned through a gradient descent algorithm but periodically copies the parameters from the learned network (e.g. every 1000 steps of SGD) and is not altered in between updates. Alternatively, the parameters of the target network can be a low-pass filtered version of the ones from learned network. The loss function for the DQL algorithm is therefore approximated by:

$$J_Q \approx \frac{1}{N} \sum_{i=1}^{N} \left(r + \gamma \max_{a'} \hat{Q}^{\pi}_{target}(\boldsymbol{s}'_i, a') - \hat{Q}^{\pi}(\boldsymbol{s}_i, a_i) \right)^2, \tag{2.50}$$

where N is the number of sampled transitions, $\hat{Q}$ is the learned network and $\hat{Q}_{target}$ is the target network. With these two adaptions, the DQN algorithm reliably converges and has been successfully applied in many different domains. Note that, instead of evaluating the action-value function for every possible action, it is more common to use the NN to predict all action-values for a given state at once:

$$\hat{Q}(\boldsymbol{s}) = (\hat{Q}(\boldsymbol{s}, a_1), \hat{Q}(\boldsymbol{s}, a_2), \dots) \tag{2.51}$$

Following the paper of Mnih et al., 2015, many further additions, that improve convergence, have been proposed. In Hasselt et al., 2015, for example, Double Q-Learning (see section 2.3.3) is used to mitigate the maximisation bias. In Schaul et al., 2016, the transitions are sampled non-uniformly from the replay buffer, so that transitions that the NN predicts very badly are sampled more often than transitions that were already predicted accurately. In Bellemare et al., 2017, the NN is used to predict a distribution of action-values instead of their expected value. Many of the proposed adaptions are brought together in Hessel et al., 2017, resulting in an algorithm that significantly outperforms the original DQL algorithm.

Using a NN to predict action-values solves the problem of very large or continuous state spaces. However, since our model can only have a finite amount of outputs, we are still limited to a discrete set of actions that the agent might choose from. For problems that require us to have continuous actions, the DQL algorithm is therefore not suited.

2.6. Policy Gradient Methods

We have thus far considered RL algorithms that compute an estimation of how good it is to be in a certain state and to take a certain action. Following a policy then means to select actions that are expected to yield high discounted returns. These methods are called value-based, as they rely on the estimation of state- or action-values in the action-selection process. They address the

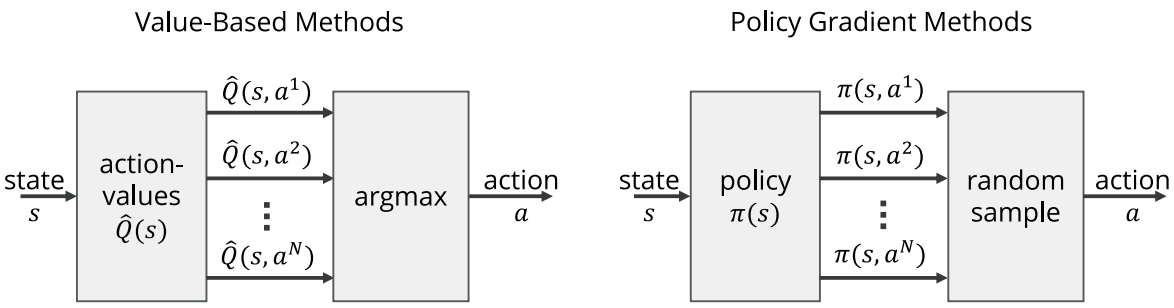


Figure 2.13.: Comparison of action selection for value-based and PG methods in the case of discrete actions. In value-based methods, the model computes the action-value for all available actions and chooses the one with the highest expected discounted return. In PG methods, the action is sampled from a categorical distribution, which is computed by the policy model.

question of how to behave in a certain situation indirectly, by answering the related question of what outcome to expect, depending on our behaviour. In many cases, however, we are actually not interested in the exact value function, raising the question if it is really necessary to estimate it.

Policy Gradient (PG) methods offer an alternative to value-based methods that approaches the problem more directly by letting a differentiable parameterised model (e.g. an NN) output the policy for a given state. Even though we may still want to learn a value function, as it can improve convergence (see section 2.6.2), we will not need to evaluate it for choosing an action. Directly learning a policy can be beneficial, as it is oftentimes easier to approximate a policy than a value function (Sutton and Barto, 2018). Furthermore, optimising a policy with gradient descent results in a policy that changes smoothly over the course of learning. In contrast, in value-based methods, a small change in the value function may result in a tremendous change in the policy due to the maximum operation. For this reason, PG methods provide favourable convergence guarantees (Sutton and Barto, 2018). Note, however, that in some domains learning a value function is actually easier than learning the policy so that value-based methods have an edge over PG methods (e.g. Simsek et al., 2016).

For discrete action spaces, the policy model predicts the probabilities of the individual actions so that the agent can then sample an action from the predicted distribution. Sampling from a policy distribution results in the nice property that the agent can learn to act stochastically. This is an advantage because the optimal policy in environments with high uncertainty may in fact be stochastic, and because a stochastic policy encourages further exploration. NNs for stochastic policies often use a softmax function (see section 2.4.1) in the last layer to output a categorical distribution over actions. We will denote the parameterised policy as a function $\pi_\theta(a|s)$ that maps from a state to a probability distribution over actions, where θ is the set of parameters of the model. Figure 2.13 shows the process of action selection for value-based methods and for PG methods in a domain of discrete actions.

Using a PG method, it is also straightforward to implement continuous action spaces by letting a model predict continuous values. For example, if the actions of the agent are the acceleration and steering angle of a car, the policy can directly output these two signals. In order to ensure exploration, it is common practice to add Gaussian noise with zero mean to the value that is given by the model. The output of the policy model $\pi_\theta(a|s)$ in the continuous case can, therefore, be interpreted as a multivariate Gaussian distribution from that a concrete action is sampled. Note, however, that the respective dimensions of the action are usually sampled individually,

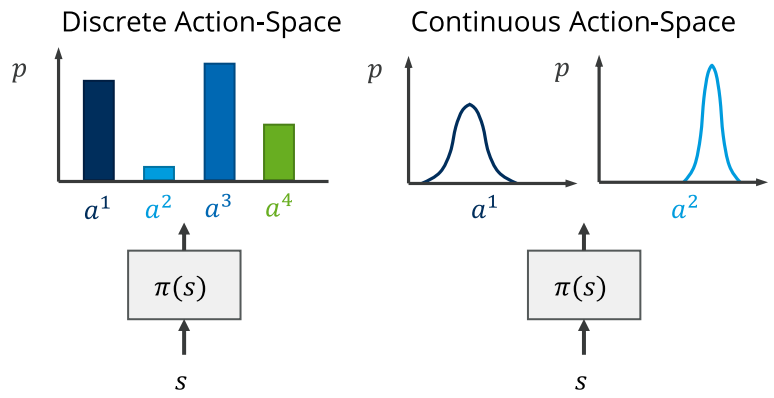


Figure 2.14.: Action distributions for discrete and continuous action-spaces. In the discrete case, the policy outputs a categorical distribution over the available actions. In the continuous case, it outputs a Gaussian distribution for every dimension of the continuous action-space.

meaning that the multivariate Gaussian is constrained to have a diagonal covariance matrix. Figure 2.14 shows the prediction of an action distribution for the discrete and for the continuous case. Note that the superscripts in the discrete case identify individual actions, whereas they denote the dimension of a multidimensional action-space for the continuous case.

The central idea of PG algorithms is that we can differentiate the expected discounted return V^{π_θ} w.r.t. the parameters θ of our model. The gradient is given by the policy gradient theorem:

$$\nabla_\theta J_\pi = \nabla_\theta \mathbb{E}\big[-V^{\pi_\theta}(S)\big] = \mathbb{E}_\pi\big[-Q^{\pi_\theta}(S,A)\cdot\nabla_\theta \log\big(\pi_\theta(A|S)\big)\big] \tag{2.52}$$

where Q^{π_θ} is the true action-value function of the policy π_θ. Note that, in order to maximise the state-value function, we need to minimise its negative. The proof of the policy gradient theorem is a bit lengthy but straightforward and shall not be replicated here. The interested reader is referred to Sutton and Barto, 2018 for a comprehensive proof under the assumption of discrete action spaces.

Just as in tabular learning, we again encounter the problem of not knowing the correct action-value function Q^π and thus having to estimate it. As we have learned in section 2.3, there exist several approaches towards approximating the value function through inference from data.

2.6.1. The REINFORCE Algorithm

In section 2.3.2, we used the discounted return of an experienced episode as an unbiased estimator of the value function. Doing the same for equation 2.52, yields the REINFORCE algorithm (Williams, 1992). The policy gradient for the REINFORCE algorithm can be computed as:

$$\nabla_\theta J_\pi \approx -g_t \cdot \nabla_\theta \log\big(\pi_\theta(a|s)\big), \text{ with } s = s_t,\ a = a_t, \tag{2.53}$$

where g_t is the discounted return that is obtained after timestep t. The parameters can therefore be updated by:

$$\theta^{k+1} = \theta^k + \alpha g_t \nabla_\theta log\big(\pi_\theta(a_t|s_t)\big)|_{\theta=\theta^k} = \theta^k + \alpha g_t \frac{\nabla_\theta \pi_\theta(a_t|s_t)|_{\theta=\theta^k}}{\pi_\theta(a_t|s_t)}, \tag{2.54}$$

where α is the learning rate. Note that, through averaging over several runs, the variance of the gradient estimate can be reduced.

The gradient update is the experienced discounted return – times the gradient of taking the action that was actually executed – divided by the probability of the action. Intuitively this makes a lot of sense: If the agent receives a large positive discounted return after taking some action, the probability of taking this action again will be increased; for a negative discounted return the probability will be decreased. Additionally, if the taken action is highly improbable under the current policy, the gradient will be even larger so that already probable actions do not get an advantage through being sampled more often. Note that, since the probabilities of actions must sum to one, increasing the probability of one action means decreasing the other ones. This also means that, even if no discounted returns are actually negative, the probability of 'bad' actions will be decreased over time because the gradient for good actions is larger than for bad ones.

Like Monte Carlo methods in tabular learning, the REINFORCE update shows relatively high variance, which can slow down convergence. A widely used measure to reduce variance is to subtract a so-called *baseline* $b(s)$ from the estimate, that leaves the gradient estimate unbiased:

$$\nabla_\theta J_\pi \approx -\big(g_t - b(s)\big) \cdot \nabla_\theta \log\big(\pi_\theta(a|s)\big), \text{ with } s = s_t,\ a = a_t. \tag{2.55}$$

The baseline must not depend on the action or it would bias the gradient. Intuitively, when two different actions both yield a high return, but one is slightly better than the other, the REINFORCE gradient for both actions would be very similar, and the policy would converge only slowly towards preferring the better action. Subtracting a baseline, that is close to the average discounted return of the two actions, makes the distinction between the two utilities much clearer. A baseline can thus reduce the variance of the gradient while leaving it unbiased. Typically, a baseline $b(s_t)$ that is correlated with g_t is chosen, as the magnitude of discounted returns generally depends on the state. A common choice for the baseline would be a moving average of the experienced discounted returns or the state-value function, that can be learned in parallel to the policy.

2.6.2. Actor-Critic Methods

As already discussed in section 2.3.3, the use of the full discounted return comes at the cost of having to wait until the end of an episode, before the algorithm can learn. The proposed solution in Temporal Difference Learning was to bootstrap the value function from the subsequent estimate. This, of course, means that we cannot learn only the policy but must learn the value function in parallel. Methods that use a bootstrapped estimate of the discounted return in order to learn a parameterised policy are called *Actor-Critic* methods. The actor (the policy) takes actions and is evaluated for the quality of its actions by the critic (the value function). Note that introducing the value function as a baseline to the REINFORCE algorithm does not qualify to be an Actor-Critic algorithm, as it still uses Monte-Carlo estimates of the discounted return.

Replacing the discounted return in equation 2.55 by the bootstrapped value and using the state-value function as baseline yields:

$$\nabla_\theta J_\pi \approx -\big(r_t + \gamma \hat{V}^\pi(s_{t+1}) - \hat{V}^\pi(s)\big) \cdot \nabla_\theta \log\big(\pi_\theta(a|s)\big), \text{ with } s = s_t,\ a = a_t. \tag{2.56}$$

Note that $\delta_t = r_t + \gamma \hat{V}^\pi(s_{t+1}) - \hat{V}^\pi(s_t)$ is the TD-error, which is also used to update the state-value function (compare equation 2.28). The term $r_t + \gamma \hat{V}^\pi(s_{t+1})$ under the condition

$s = s_t$, $a = a_t$ is an estimator of the action-value for the state-action pair, whereas $\hat{V}^{\pi}(s)$ is the state-value of the state. This difference is often called the *advantage*, as it represents the additional discounted return that we expect from taking the respective actions and following the policy π afterwards, instead of following the current policy all the time. An action that is superior to the one chosen under the current policy thus has a positive advantage; an inferior action has a negative advantage. The algorithm that uses 2.56 as the gradient estimate is therefore often called *Advantage Actor-Critic (A2C)* (Mnih et al., 2016).

It is important to note that the policy update of Policy Gradient (PG) methods (e.g. equation 2.54) depends on the current policy. REINFORCE as well as A2C are thus on-policy algorithms (see section 2.3.3), meaning that they need fresh data to learn from. This implicates that we cannot use transitions that were experienced under an old policy to train the current policy. In section 2.5, we have seen that the temporal correlation of the state can be problematic as it can bias the gradient. In DQL this problem was solved by sampling old transition from a replay buffer instead of using the latest experience. For REINFORCE and Actor-Critic methods, the same problem is encountered but cannot be solved by a replay buffer due to their on-policy nature. The original paper of the A2C algorithm (Mnih et al., 2016) proposed to solve the problem of correlated states by collecting experience from multiple environments, that all follow the current policy, in parallel. The resulting algorithm was termed *Asynchronous Advantage Actor-Critic (A3C)*. Note of course, that this option is only available for simulated MDPs.

2.6.3. Natural Gradients and Trust Regions

There are many more extensions that improve over the simple REINFORCE or the Actor-Critic algorithm. Possibly the most notable is a promising line of research that incorporates so-called trust-regions. In these methods, the policy is adapted according to the *natural gradient* (Amari, 1998; Kakade, 2001), meaning that steps of the optimisation algorithm consider the curvature of the policy that is defined by the parameters of the model. Through this, the policy can be constrained to change slowly. A slowly changing policy is important, as we cannot "trust" that a linear approximation of the value function around the current set of parameters will be accurate enough for a large change of the policy. While these problems technically apply to other domains of ML, they are especially pronounced in RL, as a large change in policy can strongly alter the experienced returns and can ultimately lead to divergence. In contrast, algorithms with a fixed training set can usually recover from a bad gradient step. This phenomenon of a policy suddenly decreasing in performance is referred to as *policy-breaking*. To mitigate this, it is, therefore, useful to take smaller steps in parameter space when small changes constitute a large change in the policy. Vice versa, if the policy is relatively insensible to the change in the parameters, it makes sense to take larger steps in order to speed up convergence.

A popular example is the *Trust Region Policy Optimisation (TRPO)* algorithm (Schulman et al., 2015), in that the policy is updated under a constraint on the KL-Divergence between the policy before and after the update. To extract the steepest gradient direction under the constraint, the curvature of the policy manifold on the parameters of the model is approximated by the Fisher-Information matrix. Having found an approximate gradient direction, the algorithm then performs a line-search to find a stepsize that meets the constraint. Note that this algorithm is largely considered to be too slow to be applied in DRL. In the *Proximal Policy Optimisation (PPO)* algorithm (Schulman et al., 2017), a simple linear approximation to the TRPO algorithm is used that works surprisingly well in practice. In the Actor-Critic using Knoecker-Factored Trust Region

(ACKTR) algorithm (Wu et al., 2017b), the ideas of trust-regions and second-order optimisation are combined to form a more stable version of SGD, and Nachum et al., 2017 proposes a trust-region algorithm that can deal with off-policy data. Under some constraints (that usually cannot be fully met), trust-region approaches can guarantee a monotonous improvement of the performance of the agent.

A full review of trust-region approaches is beyond the scope of this work. The interested reader is encouraged to read the cited publications for an introduction to the application of the natural gradient in RL and Amari, 2016 for a general introduction to learning from the standpoint of Riemannian manifolds.

2.7. Deterministic Policy Gradients

As a last step in our introduction to RL, we want to introduce the Deterministic Policy Gradient (DPG) algorithm. It was first introduced in Silver et al., 2014 and has since developed to be a standard algorithm in the RL toolbox for solving MDPs with continuous action-spaces. As the name suggests, DPG is a Policy Gradient algorithm (section 2.6); deterministic here means that the policy model does not output a probability distribution from that we may sample a concrete action, but is a deterministic mapping from the state to the action to take. To discriminate the deterministic from the stochastic policy, we will denote the them by $\mu(a|s)$ and $\pi(a|s)$ respectively. Note that DPG may be considered a special case of the general PG scenario in that the probability distribution is infinitely narrow, resulting in only one possible action to be sampled. In consequence, the expected value over the probabilistic variable A in equation 2.52 can be replaced by the deterministic action resulting in the *deterministic policy gradient theorem*:

$$\nabla_\theta J_\pi = \nabla_\theta \mathbb{E}\big[-Q^{\mu_\theta}\big(S, \mu_\theta(S)\big)\big] = \mathbb{E}\big[-\nabla_\theta \mu_\theta(S) \nabla_a Q^{\mu_\theta}\big(S, a\big)\big)|_{a=\mu_\theta(S)}\big]. \tag{2.57}$$

Note that, because of the deterministic actions, we can apply the chain rule.

As discussed earlier, it is generally infeasible to obtain the exact action-value function or its derivative. However, following the approach of Actor-Critic algorithms, we may replace the action-value function in 2.57 by a learned, differentiable model. We thus learn two different models: a policy model $\mu_\theta(s)$ that maps from states to actions and one value model $Q_\omega(s, a)$ that predicts the expected discounted return for specific states and actions, with the set of parameters θ and ω respectively. The policy can then be improved as to maximise the discounted return that the action-value function model approximates. When converged, the policy greedily selects those actions that are predicted to yield the highest discounted return. It is obvious that both models need to be carefully tuned in order to learn a good solution, as a good policy cannot be found without an accurate approximation of the action-value function, and without a good policy, the action-value function cannot learn about superior outcomes.

The DPG algorithm generalises DQL to the case of continuous action spaces by replacing the multiple outputs of the action-value function model (compare figure 2.13) by a model that predicts the value for a continuous action and then learning a policy that maximises it. Instead of the policy, we could also learn only the action-value function and then use some algorithm to find a maximum in the predicted values. However, this turns out to be far more computationally expensive. Figure 2.15 shows a comparison of architecture and training of different PG methods.

Using a policy that deterministically chooses an action for a given state comes with a problem

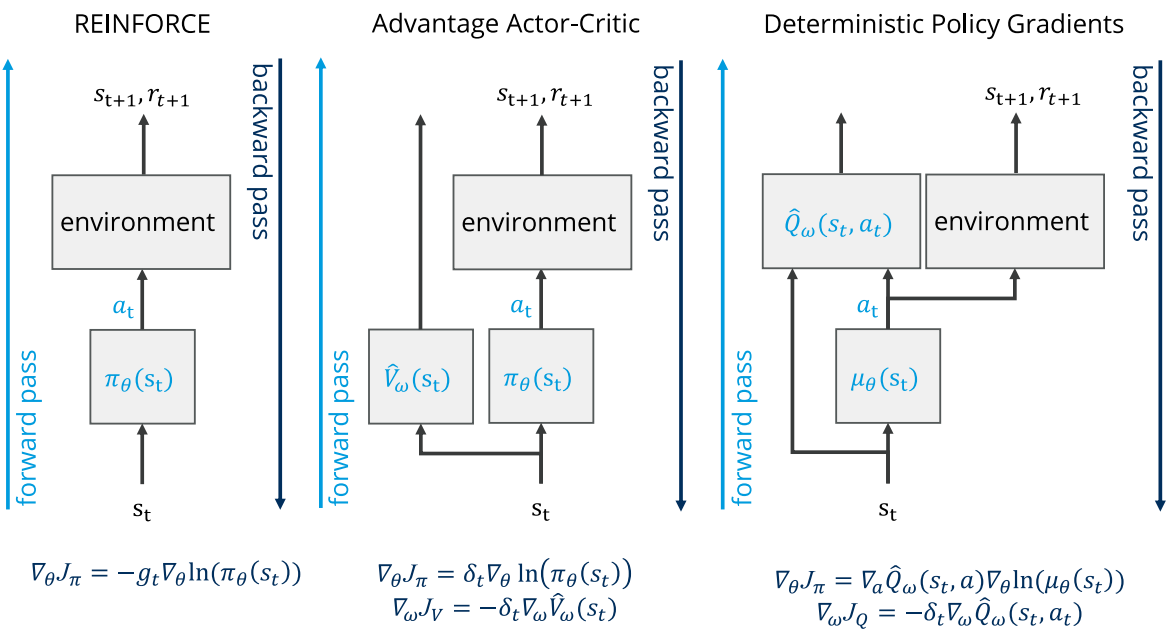


Figure 2.15.: Comparison of architecture and training of different Policy Gradient methods. All methods learn a policy function to select actions. Advantage Actor-Critic and DPG additionally learn a value function, where the former uses the value function to estimate the discounted return that is needed for the gradient estimate, while the latter uses the chain rule to differentiate the policy parameters w.r.t. the estimated value function.

of insufficient exploration. Without the introduction of some randomness in the policy, the algorithm can easily settle for some suboptimal action while leaving superior options untried. One solution to this problem is to execute some different, exploring policy while learning the deterministic policy in an off-policy fashion. To do so, we need the training of both models to be independent from the policy that generates the transitions. Since the policy gradient does not depend on the former policy, we simply have to choose some off-policy algorithm for learning the action-value function (for example Q-Learning).

2.7.1. Deep Deterministic Policy Gradients

While the original version of DPG was not intended to be used with large DNNs, the same research group later proposed some tweaks that made DPG usable with DNNs (Lillicrap et al., 2015). The most significant change was the use of target networks for bootstrapping the value function (compare section 2.5). A second adaption was the use of batch normalisation, in that all features of the state of a batch of sampled transitions are normalised to have unit mean and variance. The full DDPG algorithm thus reads:

$$\delta_t = r_{t+1} + \gamma \hat{Q}_{\omega'_t}\left(s_{t+1}, \mu_{\theta'}(s_{t+1})\right) - \hat{Q}_{\omega_t}(s_t, a_t) \tag{2.58}$$

$$\omega^{t+1} = \omega^t + \alpha_\omega \delta_t \nabla_\omega \hat{Q}_\omega(s_t, a_t)|_{\omega=\omega^t} \tag{2.59}$$

$$\theta^{t+1} = \theta^t + \alpha_\theta \nabla_\theta \mu_\theta(s_t)|_{\theta=\theta^t} \cdot \nabla_a \hat{Q}_{\omega_t}(s_t, a))|_{a=\mu_\theta(s_t)}, \tag{2.60}$$

where θ' and ω' denote the parameters of the target policy and the target action-value function, respectively. The two learning rates are denoted α_θ and α_ω.

2.7.2. The Reparameterisation Trick

As already mentioned, the policy that is followed is usually a noisy version of the deterministic policy in order to ensure sufficient exploration. We therefore need to introduce some stochastic nodes into our policy model. Ironically, this makes the policy of the Deterministic Policy Gradient algorithm non-deterministic.

Unfortunately, sampling from a policy distribution is problematic since we cannot differentiate the sampling operation w.r.t. the sufficient statistic and, therefore, cannot use the backpropagation algorithm to train the model. In order to avoid the differentiation "through" the stochastic nodes, the so-called *reparameterisation trick* is applied (Kingma and Welling, 2014; Rezende et al., 2014). Here, the stochastic nodes are reparameterised into deterministic nodes, that take as input the sufficient statistic and a random input with constant statistics. For example, a Gaussian distribution with a given mean and standard deviation can be reparameterised as a function of mean, standard deviation and a standard normal distribution:

$$\mathcal{N}(\mu, \sigma) = \mu + \sigma \cdot \mathcal{N}(0, 1), \tag{2.61}$$

where $\mathcal{N}(\mu, \sigma)$ denotes the normal distribution with mean μ and standard deviation σ. Note that the right hand side of equation 2.61 can be easily differentiated w.r.t. μ and σ and thus w.r.t. the parameters θ that deterministically define them. In DPG, the policy outputs the deterministic action μ and the variance of the stochastic exploration-policy is chosen to be some fixed value.

The DPG algorithm is mostly used for continuous action domains. For small discrete spaces, it is usually more efficient to use the DQL algorithm as it is easier to predict action-values for all available options and choose the best one. There are special cases, however, in that has been found to be beneficial to apply DPG to discrete action spaces. In Dulac-Arnold et al., 2015 for example, it is applied to select actions from very large discrete spaces. Instead of sampling the action from a categorical distribution, the discrete actions are embedded into a continuous space. The continuous policy outputs a continuous action and the discrete embedding that, according to some metric, is closest to the output is executed. In Mordatch and Abbeel, 2017, a population of agents use a continuous policy to navigate an environment and can communicate with each other through discrete symbols. The policy model, therefore, needs to output continuous as well as discrete signals.

Reparameterising a categorical distribution, as is needed in order to sample an action from a discrete space with DPG, is slightly more complicated than for the Gaussian policy. Maddison et al., 2016 and Jang et al., 2016 independently suggest to use a soft relaxation of the discrete distribution. The two groups called this distribution *Concrete distribution* and *Gumbel-Softmax distribution*, respectively; here, we will use the later one. For a number of n discrete actions, the model outputs n values y_k that sum to one:

$$y_k = \frac{\exp\big((\log \pi_k + G_k)/\lambda\big)}{\sum_{i=1}^{n} \exp\big((\log \pi_i + G_i)/\lambda\big)}, \text{ with } G_i \sim \text{Gumbel}(0, 1), \tag{2.62}$$

where π_k are the probabilities of the categorical distribution, produced by the policy model. The independent random samples of the Gumbel(0,1) distribution (Gumbel, 1954) can be produced by $\log(\log(u))$ where u is sampled from a uniform distribution between 0 and 1. Taking the argmax function $\arg\max_k(\log \pi_k + G_k)$, we obtain the discrete action. The action a_k is then, in fact, sampled with probability π_k. Note that in equation 2.62, we use the softmax function (compare equation 2.41). The softmax is here used as a continuous, differentiable approximation

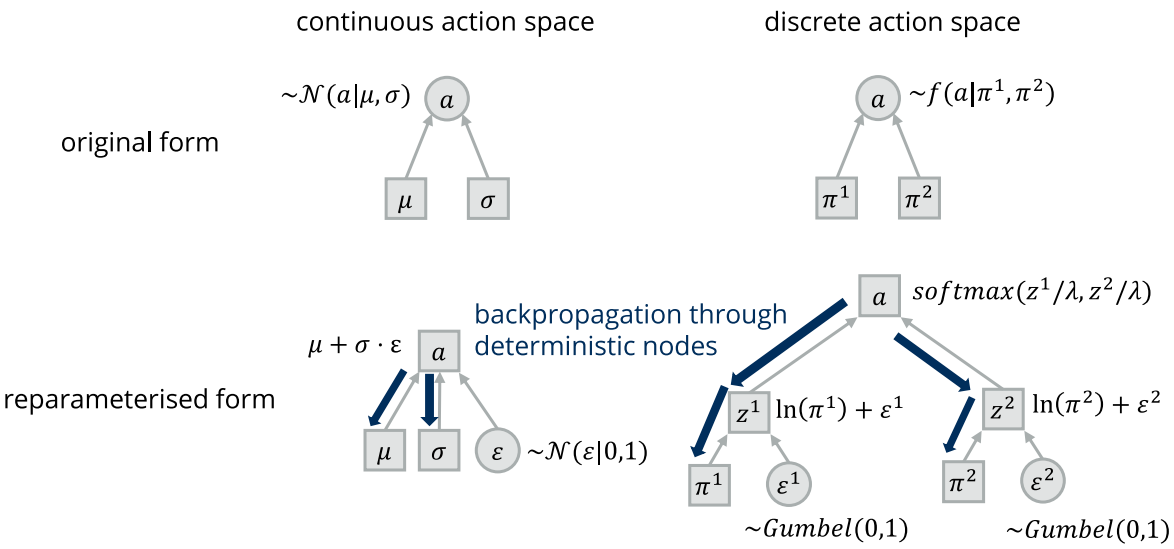


Figure 2.16.: The reparameterisation trick reformulates a sampling operation so that we may use Backpropagation through deterministic nodes. The upper row shows the original form of the sampling operation, and the lower row shows the reparameterised form for a Gaussian distribution on the left and a Categorical distribution on the right. Squares denote deterministic nodes and circles stochastic ones.

to the argmax function so that we may use the Backpropagation algorithm. The temperature parameter λ defines the closeness of the Gumbel-Softmax to the argmax. As the temperature approaches 0, the Gumbel-Softmax converges to the argmax, whereas at larger temperatures the Gumbel-Softmax becomes uniform.

Figure 2.16 shows the reparameterisation of a Gaussian (left) and a Categorical distribution (right). Squares denote deterministic nodes and circles stochastic ones. Note that through the reparameterisation, the stochastic parts are considered leaf nodes in the computation graph. We thus do not need to differentiate the sampling operation and can use Backpropagation to differentiate through a completely deterministic path. In the original form, on the other hand, the signal passes "through" the stochastic nodes, and we cannot compute the gradient.

Computing the gradient with the Gumbel-Softmax trick turns out to result in a biased estimate of the gradient. In later publications (Tucker et al., 2017; Grathwohl et al., 2017), better estimators were introduced that eliminate the bias at the cost of introducing additional control variates.

2.7.3. Further Improvements of DDPG

DDPG has been successfully applied to a range of continuous control tasks. However, the original version (Lillicrap et al., 2015) turns out to be brittle and sensitive to the setting of hyperparameters and is thus hard to use (Duan et al., 2016; Henderson et al., 2017). Since the original publication, many improvements have been suggested, some of which we will make use of in this work. Most of the modifications intent to improve the estimation of the action-value function and have already been used and proven beneficial in DQL.

D4PG

In Barth-Maron et al., 2018 the Distributed Distributional Deterministic Policy Gradient (D4PG) algorithm is introduced. It augments the action-value function to predict a distribution of values

instead of a single expected value (this is similar to Bellemare et al., 2017), which turns out to provide a better, more stable learning signal. The authors here choose to use a categorical distribution over a number of predefined bins and can thus use the categorical crossentropy (equation 2.43) as a loss function.

Furthermore, the training of the action-values is using n-step returns, which was suggested in Sutton, 1988. Using the next n rewards and bootstrapping from the state at timestep $t+n$ results in a better estimator of the discounted return and speeds up convergence of the action-value function. The TD-error thus reads:

$$\delta_t = \sum_{i=1}^{n} \gamma^{i-1} r_{t+i} + \gamma^n \hat{Q}\big(s_{t+n}, \mu(s_{t+n})\big) - \hat{Q}(s_t, a_t) \tag{2.63}$$

Note that n-step bootstrapping in the DDPG setting is technically verboten since the actions, executed during the n steps are chosen w.r.t. an old policy. This makes the learning algorithm on-policy and thus prohibits the utilisation of a replay buffer. However, in practice, it turns out that the better estimation of the discounted return outweighs the violation of the off-policy regime as long as n is chosen to be moderately small. In Barth-Maron et al., 2018, a value of n=5 shows to drastically improve convergence speed and stability of learning.

The utilisation of a prioritised replay buffer was suggested for DQL in Schaul et al., 2016 and improves convergence by sampling those transitions more often from the replay buffer, that are poorly predicted by the learned action-value function. In the D4PG algorithm, using a prioritised replay buffer led to a slight improvement in convergence. However, the small advantage was considered not to justify the added computational expense.

Finally, Barth-Maron et al., 2018 suggest to collect experience from several environment simulations in parallel (compare Horgan et al., 2018). The individual simulations can simply add the experienced transitions to a shared replay buffer, and an independent process can then sample from the buffer due to the off-policy nature of the Q-Learning algorithm. While this modification does nothing to the learning process of the algorithm itself, it can greatly improve the speed of convergence in terms of wall-clock time.

ADPG-R

Popov et al., 2017 propose in the Asynchronous DPG with Variable Replay Steps (ADPG-R) algorithm to not only use several instances to simulate the environment and collect experience but also to use several instances that sample from the replay buffer and apply the computed gradients to the shared parameters of the learned model. This version of distributed learning is a bit more involved than the former one but can further boost the speed of convergence.

Additionally, the paper suggests to augment the reward signal in environments with very sparse rewards to incorporate some intermediate goals and also to occasionally let the agent start in more advanced situations. For example, a robotic arm that needs to pick up a brick and move it to some destination would learn faster if it got a reward for picking up the brick instead of only getting rewarded for completing the entire task. The robotic arm could also occasionally start out with the brick already placed in its hand. Both modifications improve the convergence in environments in that rewards are sparse. In order to improve data efficiency, Popov et al., 2017 also propose to do several gradient steps for every step in the environment. This can also help the learning process, as it allows the learned model to converge on the available data before adding more data to the replay buffer.

TD3

Fujimoto et al., 2018 introduce the Twin-Delayed Deep Deterministic Policy Gradient (TD3) algorithm which proposes three adjustments to DDPG.

Firstly and most importantly, it suggests to use Double Q-Learning, which alleviates the maximisation bias of the Q-Learning algorithm (compare section 2.3.3 and 2.5). In Double Q-Learning, the overestimation due to the explicit maximum over the action-values was mitigated through decoupling the maximisation and the prediction of the action-value. The subsequent action (of the action-value to bootstrap from) is optimised for one model and then the value from a different model is used in the TD-error (see equation 2.34). In contrast, in DPG, the maximisation is implicit through the deterministic policy that tries to maximise the approximated value function. The Double Q-Learning algorithm therefore needs to be adapted, in order to be applied in the DPG setting. Fujimoto et al., 2018 propose to independently approximate two action-value function and bootstrap from the one that predicts smaller values. The TD-error thus results to:

$$\begin{aligned}\delta_t &= r_{t+1} + \gamma \min\big(\hat{Q}_{\omega_1}(s_{t+1}, \mu(s_{t+1})), \hat{Q}_{\omega_2}(s_{t+1}, \mu(s_{t+1}))\big) - \hat{Q}_{\omega_1}(s_t, a_t) \\ \delta'_t &= r_{t+1} + \gamma \min\big(\hat{Q}_{\omega_1}(s_{t+1}, \mu(s_{t+1})), \hat{Q}_{\omega_2}(s_{t+1}, \mu(s_{t+1}))\big) - \hat{Q}_{\omega_2}(s_t, a_t),\end{aligned} \tag{2.64}$$

where ω_1 and ω_2 are the parameters of the two models, and δ_t and δ'_t are the TD-errors, used to update the respective parameters. Even though, technically, minimising over the two overestimated functions does not yield an unbiased estimator of the true action-value function, in practice, this adaptation shows to be able to mitigate the bias and improve convergence.

Secondly, the parameters of the policy model are updated less frequently than those of the action-value function. For example, we could only use the gradients of every other batch to update the policy. This can stabilise convergence, as a well approximated action-value function is essential for the policy to improve.

Finally, the paper suggests adding truncated, zero-mean Gaussian noise to the bootstrapped value function, which results in a smoother approximation of the action-value function, that generalises better to previously unseen states.

2.7.4. Soft Actor-Critic

Haarnoja et al., 2018a propose the Soft Actor-Critic (SAC) algorithm that uses the same Double Q-Learning approach as TD3. The SAC algorithm learns a stochastic instead of a deterministic policy and is thus technically not a DPG algorithm. However, even though it takes a different approach, the two algorithms end up being very similar.

In order to encourage sufficient exploration of the stochastic policy, the SAC algorithm augments the reward function by an entropy term that rewards randomness in the action selection. In addition to the action-value function and the policy, the state-value function is learned; it is then used in the update equation of the action-value function to increase the stability of learning. The state-value function can be learned by tracking the action-value function without using the action as an input. The loss functions of the action- and state-value function thus read:

$$J_V = \mathbb{E}_\pi\big[\big(\hat{V}_\psi(S_t) - \hat{Q}_\omega(S_t, A_t) + \log \pi_\theta(A_t|S_t)\big)^2\big] \tag{2.65}$$

$$J_Q = \mathbb{E}_\pi\big[\big(\hat{Q}_\omega(S_t, A_t) - R_{t+1} - \gamma \hat{V}_\psi(S_{t+1})\big)^2\big],, \tag{2.66}$$

where ψ, ω and θ are the parameters of the state-value function, the action-value function and the policy, respectively. The action-value function is learned as in previous methods, with the exception that it bootstraps from the state-value function instead of bootstrapping from itself in a later state. This stabilises convergence as, for a stochastic policy, the state-value function is a better approximator of the value of state s_{t+1} than the action-value function of a sampled action. The state-value function is learned to track the action-value function without the knowledge of the action to take. It is thus an approximator of the expected value of the action-value function with actions drawn from the stochastic policy. Minimising the entropy term $\log \pi_\theta(A_t|S_t)$ levels out the action-probabilities so that the learned policy may be as random as possible while achieving high rewards. Importantly, the actions in equations 2.65 and 2.66 are selected by the stochastic policy, which is why a gradient descent algorithm has to average over the state- and the action-space; in DPG the actions are deterministic and gradient descent only averages over the state-space. For this reason, DPG algorithms are usually considered to need fewer data to converge to a solution.

The policy selects those actions with a high predicted action-value with high probability and those with low action-values with smaller probability. It can be distilled by reducing the KL Divergence between the policy-distribution and the softmax distribution of the action-values:

$$\begin{aligned} J_\pi &= \mathbb{E}_\pi\Big[D_{KL}\Big(\pi_\theta(A|S)\Big|\Big|\frac{exp(\hat{Q}_\omega(S,A))}{Z_\omega(S)}\Big)\Big] \\ &= \mathbb{E}_\pi\Big[-\mathcal{H}\big(\pi_\theta(a|S)\big) + \log Z_\omega(S) - \hat{Q}_\omega(A,S)\Big] \end{aligned} \tag{2.67}$$

where $\mathcal{H}\big(\pi_\theta(a|S)\big)$ denotes the entropy of the action-distribution, and $D_{KL}(X||Y)$ denotes the KL-Divergence of the distributions X and Y. The strategy of the SAC algorithm is thus to assign exponentially higher probabilities to those actions that have a higher predicted utility. As this distribution is not necessarily realisable by the parameterised policy, it is projected onto the policy so that the two are as close as possible.

Through the softmax, the policy distribution depends on the scale of the predicted action-values. This means that an agent that receives a reward of one for every successful step in the environment will behave differently than an agent that receives a reward of 10. This dependence on the scale of the rewards can be eliminated by a rescaling of the reward signal: $r' = \beta r$. Haarnoja et al., 2018a state the scaling factor β to be the only parameter in the SAC algorithm that has to be tuned carefully in order for the algorithm to converge. In a later paper (Haarnoja et al., 2018b), the algorithm was enhanced to automatically adapt the scaling parameter in order for the policy-distribution to admit some desired level of entropy.

Note that for continuous action-spaces, computing the partition function $Z_\omega(s)$ (which is needed so that the action-probabilities sum to one) is actually infeasible. However, as it does not depend on the policy parameters, it does not appear in the gradient:

$$\nabla_\theta J_\pi = \nabla_\theta \mathbb{E}_\pi\big[\mathcal{H}\big(\pi_\theta(a|S)\big) - \hat{Q}_\omega(S,A)\big]. \tag{2.68}$$

Comparing the resulting gradient to the deterministic policy gradient (equation 2.57), we observe that, even though the conceptual ideas of the two algorithms are very different, they result to be very similar. Of course, due to the different natures of the two policies, the policy-gradient of DPG and SAC differ in some points. The obvious difference is the entropy term in equation 2.68. Intuitively, DPG optimises the policy to deterministically select the action that has the highest value; the entropy term in SAC ensures that the policy will stay stochastic and only favours

those actions that yield higher discounted returns by assigning them a higher probability. The second, less obvious difference is that the action in the policy gradient is deterministic in DPG and stochastic in SAC. The former, therefore, often converges faster than the latter because it does not have to average over the action-space.

We have discussed many different algorithms of Reinforcement Learning. In general, we would suggest selecting algorithms that are as complicated as needed. In particular, when the state- and action-space naturally fall into discrete options and are sufficiently small, we would suggest applying Temporal Difference Learning algorithms like SARSA or Q-Learning (section 2.3.3), as they have stronger convergence guarantees than methods that use function approximation. If the state-space is continuous or too large and shows some spatial coherence that can be leveraged to make better decisions, Deep Q-Learning (section 2.5) or an Actor-Critic method (section 2.6.2) would be the preferred choice. Finally, if the action-space is continuous, SAC and TD3 may be considered the state-of-the-art.

This concludes our treatment of the Reinforcement Learning framework. We have tried to introduce those concepts that are made use of in this thesis in as much detail as needed, without getting lost in the specifics. It is important to note that the described methods only represent a small slice of a much broader framework, and many fundamental concepts were not introduced here. In the next chapter, we will take a close look at the specifics of traffic control and will thereafter try to apply the RL framework to it.

3. Road Traffic Control

Mobility and transportation have played an important role throughout all human history, and their relevance in modern society is ever-increasing. Until the early 20^{th} century, private transportation means were only available to a privileged few, and intricate traffic regulation was avoidable due to the scarcity and limited velocity of privately owned vehicles. Decreasing production cost and increasing availability of motorised vehicles — in particular through the start of mass production of the Ford Model T in 1913 (McShane, 1999) — alongside a general rise in income and standard of living in industrialised nations, led to increasing transportation demands and left the number of vehicles soaring to ever-new heights throughout the second half of the last century (Papageorgiou, 2004). An increasing number of traffic participants using a common traffic infrastructure without sophisticated regulation, naturally, induces congestion and accidents. These problems led to many advancements in traffic infrastructure and legislation that largely mitigated the problem of diminished safety in road traffic and, for the most part, made the utilisation of private vehicles a relatively low-risk experience.

Modern road traffic is a complex system comprised of many autonomous entities that show a versatile range of different behaviours and all act to pursue their individual goals. In large parts, traffic flows are guided by passive regulations that predefine a framework of how traffic participants should behave in order to provide safe travels for everyone. Apart from universally applicable legal restrictions (like driving on the right side of the road or granting right of way to vehicles on the right hand road of an intersection), traffic signs, road surface markings and other static measures provide a means to tailor the regulation in a particular road section to the respective requirements and, once implemented, passively govern the traffic system. However, passive regulations are often not sufficient to ensure the safety and efficiency of the traffic system and have to be enhanced by actuated measures that actively regulate the traffic flow in a road network. This is particularly true in crowded cities, where traffic volume is high, and congestion poses a threat to safety and causes high economic costs due to commuter delay.

There exist many different measures that can be implemented to actively guide traffic flows. The most important and impactful is the implementation of traffic lights (Papageorgiou, 2004), which we will focus on in this work. Other measures include adaptive speed limits (Al-Dweik et al., 2017), barriers, reversible lanes that can switch direction of travel (Wu et al., 2009) and ramp metering (Papageorgiou and Kotsialos, 2002). Traffic lights have been observed to not only decrease the risk of accidents but also, if applied correctly, to be able to increase the throughput of the traffic network significantly. In this chapter, we will discuss the control of road traffic

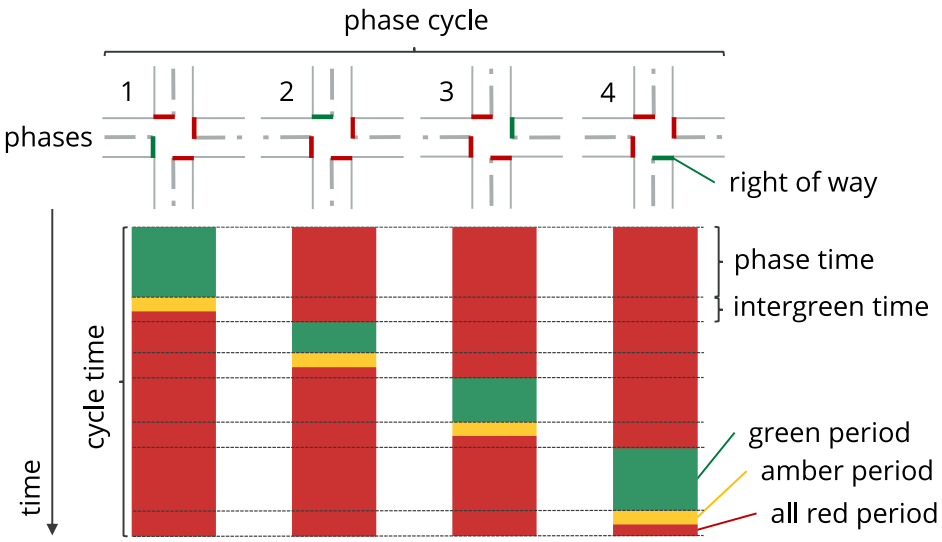


Figure 3.1.: A traffic light phase cycle of an intersection with four different phases. All phases consist of only compatible streams. The first and last phase have a higher split than the other two. Each green period needs to be followed by an intergreen period, that can be further subdivided into the amber and the all red period.

with traffic lights. We will first explain the common terminology in urban traffic control and the problem of congestion. Then, we outline what are the factors that make traffic control hard and explain some of the existing control methods that are implemented in major cities around the world. Subsequently, we will give a brief overview of the simulation of traffic scenarios, which not only serves as the evaluation of new traffic strategies but is an essential prerequisite for the application of many traditional control schemes. Finally, we will discuss the emerging possibility of fast, reliable Vehicle to Infrastructure (V2I) communication. This chapter is partly based on Papageorgiou, 2004.

3.1. Traffic Lights

Of particular interest in traffic networks are *intersections* (also called *junctions*) — at that two or more routes interfere — as they are focal points for both safety and efficiency. Traffic lights are used to control traffic flow at intersections or pedestrian crossings by sequentially granting the *right of way* to different *streams*. In this work, we will focus on intersections that only consist of streams of motorised vehicles and neglect the need to account for pedestrians, cyclists or other non-motorised means of transportation. A stream is defined as a trajectory of a group of vehicles that go from an entry point to an exit point of the intersection. In a so-called *phase* or *stage*, a subset of the possible streams is simultaneously allowed the right of way. A traffic light iterates between allowing the right of way to different phases in a *phase cycle*. The set of phases and the respective streams that each phase comprises of is referred to as the *phase scheme*, the time that every phase is allocated within the phase cycle is called its *phase time* and the sum of all phase times is the *cycle time*. The relative phase times of the different phases are called the *phase split*. Figure 3.1 shows a phase cycle, consisting of four different phases. Note that the first and last phase have a higher split than the other two. The depicted cycle gives the right of way to only one of the afferent streets at a time, which naturally results in compatible streams. In this work we will generally assume right hand traffic.

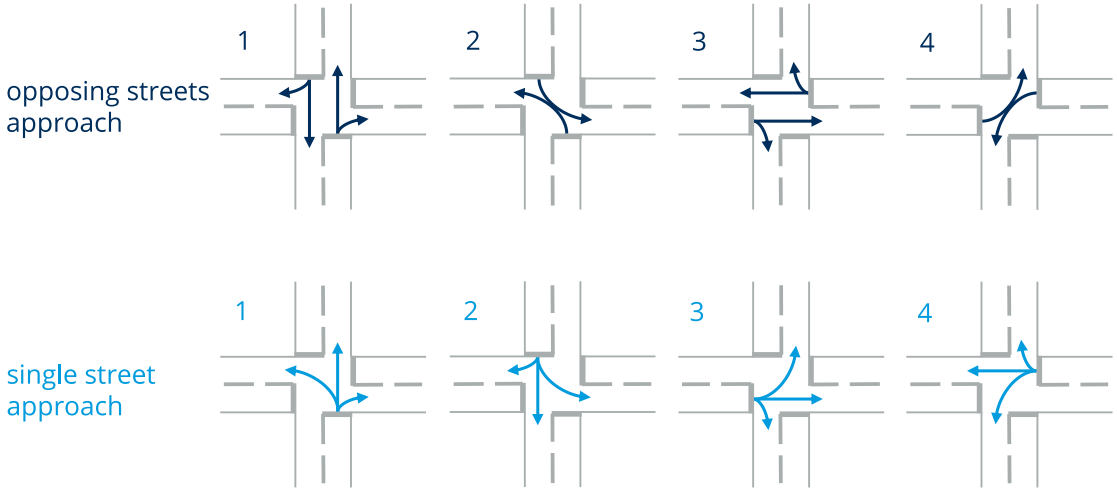


Figure 3.2.: Two popular phase schemes. In the opposing streets approach, the traffic light grants the right of way to two opposing streets at a time and allows straights and right-turns in one phase and left-turns in another phase. In the single street approach, right of way is granted to all streams, that originate from one particular street.

Individual streams that form a phase are called *compatible* if they can all cross the intersection without interfering with each other, and a set of streams that are non-compatible is called *antagonistic*. Of course, it is generally preferable if phases consist of only compatible streams, as antagonistic streams that simultaneously have the right of way tend to increase the risk of accidents. In between the phases, *intergreen times* are needed to avoid interference of antagonistic streams and to ensure safety. Intergreen times can be further subdivided into an *amber period*, in that vehicles of the phase that is losing the right of way have time to break or enter the intersection if they are already too close to come to a halt before the traffic light, and an *all red period*, to allow all vehicles and pedestrians to clear the junction. Importantly, whereas the phase times can be adapted to match current traffic demands, the intergreen times are subject to the intersection's geometry and, for safety reasons, may not be altered. Switching the phase, therefore, is always linked to a loss of throughput.

In most cases, a traffic light is required to grant the right of way to every stream at least once in its phase cycle. To improve throughput and reduce latency of every stream, most approaches allocate as many compatible streams as possible to each phase. Figure 3.2 shows two popular, fully compatible phase schemes which we will here call *opposing streets approach* and *single street approach*, respectively. Numbers denote the position of the shown phase within the phase cycle; arrows show the streams that make up each phase. In the opposing streets approach, right of way is always granted to a pair of opposing streams. In one phase, the right-turn and the straight streams are given right of way, and in another phase, the two opposing left-turn streams can cross the intersection. In the single street approach, right of way is granted to all flows that originate from one afferent street at once. Depending on the particular structure of an intersection, many more phase schemes may be employed. In areas with limited traffic volumes, it may be beneficial to compose phases of several antagonistic streams and prioritise the interfering ones. In particular, a phase including all flows that originate from two opposing streets is not an uncommon sight. In this case, the left-turn stream is of lesser priority and has to grant the right of way to the other streams. Another common choice is always to allow right-turns as long as no other stream is disturbed (a green arrow).

Oftentimes, a single traffic light cannot be viewed as an isolated system but is a part of a

larger network. Choosing a phase scheme and a phase split for every traffic light in a system individually, thus, may not lead to efficient utilisation of the traffic network so that individual traffic lights may need to be coordinated by a so-called *offset*. The offset defines when the phase cycle of a particular traffic light is started with respect to some reference timepoint. In particular, several traffic lights alongside a road can be assigned increasing offsets, that correspond to the average travel time between them, to create a so-called *green wave* in that a stream of vehicles can traverse the entire street without having to stop.

3.2. Traffic Congestion

The *demand* is the average rate of vehicles of a stream that arrive at an intersection. The *saturation flow* is the average rate of vehicles of a stream, that can cross the intersection under the current phase scheme and split. In *undersaturated traffic conditions* — when the demand is lower than the saturation flow — queues that build up during the red phase of a stream can dissolve during the green phase. In *oversaturated traffic conditions*, queues keep building up, leading to ever-increasing levels of congestion in the traffic network.

In most cases, traffic volumes vary over time so that congestion that builds up during a time, can dissolve later on. In an idealised traffic model, arriving vehicles queue up during high demand periods and, after some delay, cross the intersection. The throughput at the intersection thus increases with rising demand until the saturation flow is met, and the throughput stagnates. In this case, higher demand can thus result in higher delays of individual vehicles but will not decrease the rate at which vehicles cross the intersection. Unfortunately, in reality, rising congestion levels lead to a degraded use of the available infrastructure and thus decreasing throughput. The mutual impediment of vehicles in congested traffic then leads to even faster formation of more congestion. Through this feedback-loop, congestion reinforces itself, leading to ever-decreasing throughput and increasing delays. In the worst case, a *gridlock* situation arises in that intersections are blocked by vehicles and the throughput plummets down to near-zero.

The direct effects of increased congestion are, alongside heightened risk of accidents, increased emissions and pollution, rising noise levels as well as stressed commuters (Papageorgiou, 2004). According to recent studies, congestion in the European Union costs approximately 1% of its annual GDP (European Comission, 2017), and commuters in crowded cities spend up to 200 hours per year, stuck in traffic (Inrix, 2018). Due to these unfavourable dynamics and effects, minimising and dissolving congestion as fast as possible is of tremendous economic, social and environmental interest. Increasing throughput in a 'brute force' approach of continuous expansion of road infrastructure is expensive and often limited by existing infrastructure or other social or environmental factors. A more efficient utilisation of existing road infrastructure thus is of particular interest as it provides a cost-efficient measure to mitigate congestion, increase throughput and decrease delays.

The original introduction of the electric traffic light in 1914 in Cleveland (McShane, 1999) and its widespread integration in the first half of the 20^{th} century, was driven by safety concerns. However, it was soon realised that, if carefully tuned, traffic lights can significantly increase the throughput of the traffic network whilst keeping risk-levels low. Approaches towards designing an intelligent traffic control system by choosing sensible parameters (phase scheme, phase split, cycle time and offset) for all installed traffic lights were first investigated in the 1930s (Gartner et al., 2001) and have been an area of active research ever-since. The long, ongoing development of advancements in the field is reminiscent of the inherent difficulty of the traffic control problem.

3.3. What Makes Traffic Control Hard

Finding the optimal parameter configuration for a set of traffic lights turns out to be difficult for a number of reasons:

- Maybe the most obvious problem is the high dimensionality of the parameter space; jointly choosing a phase scheme, a phase split, a cycle time and an offset for all junctions in a large area admits a combinatorial number of discrete parameter configurations alongside a high-dimensional continuous parameter space, which is notoriously hard to optimise.
- Furthermore, it is often unclear, which quantities we actually want to optimise. While it is rather obvious to maximise throughput and minimise delay, these two quantities turn out to negatively influence one another. Moreover, there may be other factors to consider, that cannot be simply rendered into a numeric indicator. For example, we may want to take into account social factors like fairness or environmental factors like air pollution.
- Another aspect that makes traffic systems hard to optimise is the presence of unpredictable behaviours and disturbances like accidents, blockage due to illegal parking, passing ambulance or police cars, and many more.
- Moreover, in many cases the current state of the traffic network is hard to determine due to sparsity of sensors and noisy measurements.
- Finally, traffic light control is subject to tight real-time constraints which render impractical many slow optimisation algorithms of the traffic policy, conditioned on the current traffic state.

Theses insurmountable difficulties render an optimal solution infeasible for more than a single intersection and force control strategies to introduce several limiting simplifications and heuristics that generally may at least lead to a good solution, if not the optimal one (Papageorgiou, 2004).

3.4. Traditional Control Methods

After many years of research, there exists a wide variety of different control strategies, implemented in many cities around the world. Because of various factors like the heterogeneity of traffic situations or the availability of communication infrastructure, a centralised traffic management system and expert know-how, there does not exist a single unified solution to the traffic control problem. In fact, even within most cities, traffic control does not follow a homogeneous strategy but consist of many individual, heterogeneous solutions, each considering only a single or a small number of intersections. It is beyond the scope of this work to give credit to the entire breadth of these control strategies. In this section, we will try to give a coarse overview of different approaches and briefly explain some of the more popular algorithms.

Most existing control strategies can roughly be categorised alongside two axes (Gartner et al., 2001; Papageorgiou, 2004):

Responsiveness *Fixed-time* strategies are based on traffic counts that are collected offline. Oftentimes, signalling is then selected to match the historical statistics and take into account various temporal patterns like rush-hours in the morning and late afternoon,

decreased traffic volume on weekends or commuter traffic, that leads to high volumes entering a city in the morning and leaving it in the afternoon. Problematically, because fixed-time strategies cannot adapt to current traffic conditions, they are only suited to deal with under-saturated traffic conditions. *Traffic-responsive* strategies utilise real-time information of the traffic system, which is typically collected by inductive loop sensors in the streets but may also include data sources like video surveillance or weather reports. From the knowledge about the current traffic state, a controller can compute appropriate signalling.

Coordination *Isolated* strategies consider only a single traffic light. For these strategies, it is often possible to find an optimal solution to the traffic control problem. However, since the traffic light is only one part of a larger system, this solution will probably not be optimal when evaluated on a network scale. *Coordinated* strategies consider a system of many intersections, whose individual behaviours strongly interact. Jointly computing a configuration for many intersections can lead to significantly better solutions. However, the sheer size of the optimisation problem often renders finding the optimal solution infeasible.

In the following, we will further discuss the advantages of the different control strategies and introduce some of the most widely used algorithms.

3.4.1. Isolated Fixed-Time Control

Isolated control is the simplest of all approaches as it neglects both the need to cooperate with other intersections as well as to adapt to current traffic. Since no current measurements have to be taken into account, strategies can be computed offline from historical data and are thus not subject to real-time constraints. Furthermore, due to the lack of coordination between intersections, the offset parameter does not have to be tuned. In the so-called *stage-based* strategies, the phase scheme is predefined so that only the phase split and the cycle length have to be computed. The continuous nature of the optimisation problem allows us to solve it with methods of linear programming. Popular stage-based strategies are, for example, *SIGSET* (Allsop, 1971), that minimises total intersection delay or *SIGCAP* (Allsop, 1976) that maximises the intersections capacity. Considering the phase scheme as part of the optimisation problem as in Improta and Cantarella, 1984 results in a much more demanding optimisation problem as it includes both continuous as well as discrete elements but can lead to better strategies.

3.4.2. Coordinated Fixed-Time Control

Isolated control strategies can only lead to locally optimal behaviour, which will often lead to inefficient utilisation of the road infrastructure on a network scale. In particular in crowded areas, where efficient control is of particular importance, isolated strategies therefore often fail to mitigate the formation of congestion effectively. In contrast, coordinated behaviour across multiple junctions can lead to significantly better utilisation of the infrastructure. However, as always, more sophisticated strategies requires a more intricate optimisation process. Two famous examples are the *MAXBAND* (Little, 1966) and the *Traffic Network Study Tool (TRANSYT)* algorithm (Robertson, 1969).

MAXBAND was designed to address traffic settings of many intersections along a two-way arterial road. It computes offsets for all traffic lights as to maximise the number of vehicles

that can traverse the arterial without ever stopping (a green wave). This can be formulated and solved by mixed integer linear programming. Later on, some extensions to MAXBAND were introduced. In the SAFEBAND algorithm, over-speeding is discouraged by switching the next light of the arterial just in time to let the vehicles pass that did respect the respective speed limits, and in the MULTIBAND (Stamatiadis and Gartner, 1996) algorithm the individual bandwidths of the respective connecting roads between the intersections are taken into account.

TRANSYT is the most broadly known and most frequently implemented signal control strategy and is therefore often used as a reference-frame to assess the performance of more advanced methods (Papageorgiou, 2004). It tries to globally minimise waiting times and the number of stops within the traffic network, by using a mix of heuristic search algorithms like hill-climbing and genetic algorithms. It can also include more advanced performance measures like fuel consumption. To optimise the parameters, TRANSYT employs a macroscopic network model to simulate the traffic flow and then iteratively adapts splits, offsets and cycle times of all traffic lights in small steps until a local minimum is found. Since its introduction in the 1969, TRANSYT has been further developed (the current version is TRANSYT 15.5) to, for example, include more sophisticated traffic models, optimise the parameters more efficiently or to better incorporate real-world data. Note that, as with all fixed-time strategies, the optimisation process can be computed offline and is thus not bound by any real-time constraints.

3.4.3. Isolated Responsive Control

Fixed-time strategies derive reoccurring patterns from historical data and presume that future traffic demand will follow these patterns. Of course, this is a crude oversimplification. Even if daily or weekly patterns are taken into account, traffic volumes show strong variance and more often than not, the true demand will differ from the one considered by the fixed-time controller. Furthermore, due to changing commuter patterns and infrastructure changes — like a new mall or closed roads due to building sites — mean values of the traffic statistics may shift over a longer time period. These inaccuracies lead to under-performing strategies, increased delays and decreased throughput, even if the fixed-time algorithm converges to the optimal parameter setting (which it rarely ever does). Considering the current traffic state through information from sensory input thus has the potential to find more efficient traffic strategies that can adapt to changing requirements on both a short- and a long-term scale.

In most cases, sensory input of responsive control strategies is limited to inductive loop sensors that are located in all approaching lanes of the intersection. As no coordination with other traffic lights is required, the phase times can be altered arbitrarily. In quiet areas with low traffic volumes, the right of way often is given by default to the main road and is only granted to lower priority roads, if a vehicle approaches the intersection on that road. In areas with higher traffic volume, more sophisticated algorithms are needed. For example, the *Microprocessor Optimised Vehicle Actuation (MOVA)* algorithm (Vincent and Young, 1986) uses a method developed in Miller, 1963, in that the controller decides in discrete timesteps whether or not it is time to proceed to the next stage. This decision is made through an estimation of the net time gains and losses of vehicles in afferent streets based on the current measurements of the induction loop sensors. For undersaturated traffic, MOVA aims to disperse all queues that have built up at the afferent roads. In a congested state, it switches to a strategy that maximises the throughput. In contrast to the fixed-time regime, the calculations of a responsive strategy are executed during runtime and are thus subject to tight real-time constraints.

3.4.4. Coordinated Responsive Control

Coordinated responsive control can be considered the most powerful but also the most demanding of the strategies that have been discussed so far. Deriving a joint signalling plan during runtime is no trivial task and is only feasible when employing some simplifying assumptions.

The Split Cycle Offset Optimisation Technique (SCOOT) (Hunt and Robertson, 1982) was originally developed by the Transport and Road Research Laboratory in the UK and has been further advanced ever since. In different studies it showed to reduce average delay by around 15-30%, compared to Fixed-Time Strategies like TRANSYT (e.g. Kergaye et al., 2008). SCOOT is heavily adapted throughout large parts of the UK and is also employed in other major cities around the world like Madrid, Bangkok, Beijing and Toronto. In particular, it is known for automating 6.000 junctions in 4.500 installations throughout London, and it was famously applied to mitigate congestion in the 'Olympic Route Network' during the 2012 Olympics. SCOOT uses data from loop detectors, located around 100-300 m from the intersection, to estimate car arrival profiles. It then adapts splits, offsets and cycle time to minimise waiting times and the number of stops throughout the network. The parameters are adapted in small, incremental steps by a hill-climbing algorithm that runs on a central server. To facilitate coordination, the cycle time is equal for all intersections. Due to the strong similarities, SCOOT is often called the traffic-responsive version of TRANSYT (Papageorgiou, 2004). Due to the long distance between intersections and respective loop sensors, congested traffic conditions are recognised rather late. When SCOOT registers long queues, it switches to a regime that prioritises the diffusion of congestion over the original goal of minimisation of travel time and stops.

The formerly introduced traffic control strategies all aim to optimise splits, offsets and cycle times of the controlled intersections (phase schemes are often predefined). Furthermore, there exist a variety of more advanced strategies that, given a predefined phase scheme, try to find optimal lengths for the current phases. These algorithms rely on sophisticated traffic models to optimise some performance index. Popular examples are PRODYN (Henry et al., 1983) and CRONOS (Boillot, 1994) that solve the optimisation problem by dynamic programming methods. Realistic traffic models include discrete variables to reflect the impact of red/green phases on traffic flow (Papageorgiou, 2004), which leads to an exponential complexity of the optimisation problem and thus renders an optimal solution for more than a few intersections infeasible. So-called *store-and-forward* approaches employ simplified, continuous traffic models to trade off the accuracy of the solution against the complexity of the problem.

Figure 3.3 depicts the formerly discussed, traditional control strategies alongside the axes of coordination and responsiveness.

3.4.5. Drawbacks of Traditional Traffic Control Strategies

All discussed control strategies show some individual strengths and weaknesses. As with most optimisation approaches, a more accurate solution generally introduces additional complexity. For isolated control strategies, the optimal control strategy can often be found, whereas, for coordinated control of many intersections, the optimal solution quickly becomes unfeasible. However, on a network scale, the coordinated approaches often find better solutions. Responsive control strategies can adapt to current traffic situations but add the burden of real-time constraints and increased complexity of the optimisation problem, which can thus only be met with additional computational power. Coordinated responsive control can, therefore, only be applied to relatively small traffic networks. The common practice is to optimise small subsets of

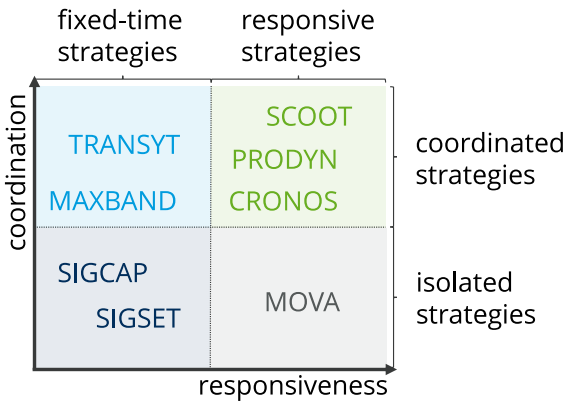


Figure 3.3.: Overview of the here-presented traditional control strategies and a categorisation alongside the coordination- and the responsiveness-axis.

the entire traffic network independently, which generally leads to suboptimal behaviour on a network scale.

A problem of most of the discussed methods is their dependence on some traffic model that is used to optimise the configuration. While a more sophisticated model can lead to better solutions, it is oftentimes also more complex to optimise, again demanding a trade-off between accuracy and complexity. In particular, the utilisation of discrete variables in the model puts a severe strain on the scalability of the control strategy. Furthermore, all simulations have to introduce some simplifications and are thus prone to biases, introduced by the poor quality of the traffic model. Finally, most traditional approaches are limited to relatively slow changes in their policy and are thus often unable to adapt to rapidly changing demands or unexpected incidents.

3.5. Traffic Simulation

Traffic simulations are an essential measure for evaluating the efficiency of traffic networks. It provides cheap and safe means to investigate the impact of changes to road infrastructure, traffic light signalling or traffic legislation on the dynamics of the traffic system. Furthermore, as we have seen in the preceding section, the optimisation of signalling behaviours generally requires a model of the traffic network in that the optimisation algorithm can freely change the configuration of all intersections and simulate the possible impact of applied changes on some objective function, like the throughput of the traffic network. Based on historical arrival profiles (fixed-time strategies) or current measurements from loop sensors (responsive strategies), the algorithm can then compute optimal or near-optimal configurations that are subsequently applied in the real traffic system.

A significant amount of effort has been put into the simulation of urban traffic networks, spawning a wide range of methods and software packages. For any project that researches traffic systems, the choice of a simulation approach and a software implementation is an essential one as it can have a strong influence on the outcome of experiments. Approaches can be roughly divided into two categories (Kotusevski and Hawick, 2009):

- Microscopic approaches view traffic as individual vehicles that navigate the traffic network and are simulated individually. Each vehicle behaves according to some local condition of the traffic network (like the phase of a traffic light or the position and speed of the

vehicle in front) and some individual parameters (like a destination or personal driving preferences).

- Macroscopic approaches consider traffic from a more abstract, statistical standpoint. Instead of considering individual vehicles, streams are described as vehicle densities that contract at intersections and disperse in free flow. The movement of vehicles in a traffic network can then be described in terms of fluid dynamics with vehicle densities instead of fluid densities and traffic lights instead of valves.

As one can imagine, individually modelling each autonomous vehicle and pedestrian is a lot more computationally expensive than the statistical view of macroscopic models. This problem is especially pronounced for large-scale simulations of many intersections. Furthermore, macroscopic models are usually easier to optimise as they model vehicle flows as continuous variables instead of discrete units. On the other hand, the more realistic setting of individual entities can model real-world traffic more precisely and thus leads to more accurate predictions. For small traffic networks of only a few intersections, that require detailed simulation, microscopic simulators should be used. In large-scale simulations of entire cities, the exact behaviour of individual vehicles is often not important, which is why a macroscopic simulator could be preferable in these situations.

In the following, the simulation software which will be used throughout this work will be briefly introduced.

3.5.1. SUMO

Simulation of Urban MObility (SUMO) is an open-source traffic simulator that is developed by the German Aerospace Center (DLR) (Behrisch et al., 2011). It can simulate a versatile range of traffic scenarios that consist of roads, intersections, traffic lights, inductive loop sensors and much more. The scenario is inhabited by different classes of autonomous units such as cars, trucks, motorcycles, emergency vehicles, public transport and pedestrians. Traffic networks can be generated either by manually defining roads and intersections or by using *netconvert*, a program that generates SUMO-networks from a range of common formats like Open Street Map (OSM) files. The simulated networks can be visualised in a graphical user-interface using OpenGL.

SUMO is a purely microscopic simulator, meaning that every vehicle is simulated as an individual entity. Note that, being a microscopic simulator, SUMO is well suited for small-scale simulations but shows poor performance when used to simulate entire cities. The traffic network is simulated in discrete, equidistant timesteps (the default timestep is one second) in that vehicles move according to some model of driving-behaviour. The simulation is space-continuous, and a vehicle's position is described by the lane that it is on and the distance to the beginning of this lane. Note that the positioning of vehicles on one of a discrete set of lanes assumes highly ordered traffic that may be found in developed nations, where cars are the predominant vehicle-class. However, in many developing countries, where traffic mostly consists of motorised rickshaws and motorcycles that show a more continuous lateral motion pattern, the assumption of lane-based traffic may be highly inaccurate.

A vehicle is identified by an ID, a departure time and its route through the network. Furthermore, it can be described in more detail. The departure and arrival properties can be further specified for example by defining desired lanes on that the simulation should be entered or left. A vehicle can be assigned a type, that has certain physical properties like the length, that defines the space that it occupies on its lane, or the weight, which changes its acceleration and

deceleration properties. Other variables change the vehicle's appearance in the graphical user interface. Furthermore, a pollutant and noise emissions class can be assigned (Behrisch et al., 2011).

The driving behaviour is defined by a longitudinal controller, the so-called car-following model, and a lateral controller, the so-called lane-changing model. The car-following model is used to accelerate and decelerate on a lane. Most controllers intent to drive as fast as possible and allowed, while maintaining a safe distance to the vehicle in front. The lane-changing model defines when to switch lanes. Lane changes are necessary when the currently occupied lane does not allow the desired turn. Furthermore, switching the lane can be beneficial if vehicles on another lane move faster or other queues are shorter. The default car-following model is an adapted version of the one introduced in Krauss, 1998 and the used lane-changing model was developed in Krajzewicz, 2010.

At the beginning of a simulation, a set of vehicles is placed at predefined locations. Additional vehicles can be generated during the simulation, according to a so-called Origin-Destination-Matrix that defines the number of vehicles that traverse the network from a specified origin to a specified destination within a certain timeframe. The exact trajectory from the origin to the destination can then be inferred by standard routing algorithms. Alternatively, vehicles can enter the network at specified points and then randomly turn at every intersection until an outgoing lane is reached.

The Traffic Control Interface (TraCI) (Wegener et al., 2008) allows the user to interact with a running simulation via a socket connection. It is thus possible to change almost all parameters of the SUMO simulation during runtime so that we may externally control cars, traffic lights, speed limits, available turning options and many more.

3.5.2. Flow

Flow is an open-source Python framework that is created and developed in the Mobile Sensing Lab at the University of California, Berkeley (Wu et al., 2017a). It provides a framework to study the application of Deep Reinforcement Learning to various traffic settings, including traffic light control and driving behaviour of autonomous vehicles. It can be used with the two popular microscopic traffic simulators SUMO and AIMSUN (here we will only use SUMO). Flow provides a lightweight interface that frames the time-discrete traffic simulation as an MDP. To initialise, reset and advance the simulated environment, it utilises the API of the popular OpenAI Gym framework (Brockman et al., 2016), which is used to benchmark RL algorithms on a broad range of open-source environments. Apart from the environment, Flow also includes two popular libraries of RL algorithms, Berkeley RLL's rllab (Duan et al., 2016) and RISELabs's RLlib (Liang et al., 2017), which feature state-of-the-art algorithms like DDPG (section 2.7.1) or TRPO (section 2.6.3). It thus implements the entire agent-environment interaction loop (see figure 2.1), including the training of a policy and therefore strongly facilitates the investigation of RL algorithms in traffic control scenarios.

Flow includes a range of predefined environments like the 'Ring-Road' environment (Wu et al., 2017a), that can be used to study the effect of autonomous vehicles on the stability of traffic flow on a circular road; or the 'Merge' environment (Vinitsky et al., 2018) in that autonomous vehicles are trained to speed up the traffic at an on-ramp merge. In addition to the included examples, users can create their own environments. The design of individual traffic environments in Flow is separated into a so-called 'scenario' and an 'environment' class. The scenario defines parts of

the system that are static elements of the traffic simulation like roads, intersections, entrance and exit routes of the network, inflows, traffic lights, loop detectors as well as the lateral and longitudinal controllers of vehicles. The environment class acts as an interface for RL algorithms and thus needs to define the observation- and action-space as well as the reward function of the MDP.

Vehicles that are not controlled by an RL agent can employ a range of different longitudinal and lateral controllers. Flow includes all of SUMO's default controllers and also facilitates the implementation of new ones. Non-RL-controlled traffic lights employ a fixed phase scheme, phase split and duration and may also include information that is collected from a loop sensor that is placed at a specified distance before the traffic light.

In discrete timesteps, the RL agent controls the simulation by choosing actions, that are conditioned on the current observation, respectively the history of previous actions and observations. Flow then translates the chosen actions of the agent to TraCI commands to apply them to the SUMO simulation and advances the simulation by the defined time interval. Note that through TraCI's plethora of possibilities to influence a running SUMO simulation, the action space of an RL agent can also include a wide range of different control measures. Finally, the obtained reward as well as the subsequent observation are computed and returned to the RL algorithm. At the end of an episode, the simulation is reset to its initial state.

3.6. Vehicle to Infrastructure Communication

The term Vehicle to Infrastructure (V2I) communication describes the wireless, bidirectional exchange of information between individual vehicles and the traffic infrastructure. While mostly being merely a vision of innovative minds in both academia and industry and only being implemented in small-scale pilot projects, widespread adoption is assumed to be around the corner and expected to impact almost all aspects of modern road traffic (Arena and Pau, 2019). On the one hand, the infrastructure can inform individual drivers about the traffic situation in the road ahead to alleviate the risk of accidents, or it could advise about a reasonable speed to catch the next green light just in time and thus increase traffic flow. On the other hand, detailed information about the state of individual vehicles could enable the infrastructure to make better control decisions, such as the smarter control of traffic lights and speed limits to increase safety and efficiency or to automatically charge the drivers account for parking fees and highway tolls. V2I is only a part of the larger Vehicle to X (V2X) framework that also includes direct vehicle to vehicle (V2V) communication which can enable cooperative behaviour, such as platooning among a group of several automated vehicles.

An important distinction in smart traffic applications that are enabled through V2I and V2V is the one between safety-relevant applications, like adaptive traffic light control, and non-safety-relevant applications, like the collection of parking fees. Safety-relevant applications tend to regulate certain aspects of the traffic system in real-time and thus have to adhere to tight schedules. Apart from suitable hard- and software, those applications therefore demand low-latency communication interfaces that enable the transmission of state information within a timeframe of only a few milliseconds. To this end, different communication standards may offer a low-latency and high-throughput interface, and new standards are needed for the safe operation of V2X applications. In particular, the IEEE 802.11p standard has explicitly been introduced for wireless access in vehicular environments (Arena and Pau, 2019) and the European Union reserves a 30 MHz frequency band at 5,9 GHz for the communication of Intelligent

Transportation System (ITS) (Shi and Sung, 2014). The short transmission range of the IEEE 802.11 standard, however, demands a dense net of devices to form an ad-hoc mesh network to transmit information over larger distances. Cellular technologies like UMTS and LTE offer an alternative to short-range transmission. Especially the emerging 5G standard is expected to show a latency as low as one millisecond as well as staggering throughput of up to 10 Gbit/s for a plethora of up to 100 billion independent devices (Peramandai Govindasamy, 2015; Salvatori, 2016) and is thus well equipped to enable intricate real-time control decisions in V2X scenarios.

The widespread deployment of fast internet and the installation of suitable communication hardware in vehicles and infrastructure is only one side of the application of ITS. Once able to share real-time information among the local traffic infrastructure and nearby vehicles, we face the question of how to make use of the vast amounts of data and how to translate the unstructured information into concrete control decisions. Data sources, as well as control domains, can be manifold. As already mentioned, the traffic system may control traffic lights, speed limits, speed and route suggestions to individual drivers, acceleration and steering-angles of autonomous vehicles and more. In the decision process, the system may consider position and velocity of all vehicles as well as other features of individual vehicles such as maximal acceleration and deceleration, physical proportions, emissions and noise levels, or even vital parameters of the driver and other occupants, such as fatigue or stress. Furthermore, many other factors may be taken into account, such as weather conditions or data from surveillance cameras that may track pedestrians and cyclists. As one can imagine, optimising such a complex system is not a trivial task, especially when considering the tight real-time constraints of the control problem. A unified, optimal solution is clearly infeasible, and the problem needs to be decomposed into smaller, manageable sub-problems.

This concludes our treatment of traffic systems in general and traditional traffic control specifically. In the next chapter, we will combine the two preceding ones (Reinforcement Learning and traffic light control) to develop an intelligent system that learns to control traffic lights with a Neural Network as the controller.

4. Deep Reinforcement Learning for Urban Traffic Light Control

In chapter 2, we discussed the Reinforcement Learning framework which is used to learn a controller — called the policy — through interaction with an environment that is framed as a Markov Decision Process. Subsequently, in chapter 3, we introduced the traffic control problem and explained why more intelligent traffic light control strategies could cause ample economic, environmental and social benefits, but also why the optimisation of traffic control policies is no trivial problem and asks for new, elaborate optimisation algorithms. In this chapter, we will connect the two fields and develop an attempt to learn the intelligent control of traffic lights through the application of a DRL algorithm. First, we will analyse the potential strengths of traffic control policies and their advantages over traditional traffic control strategies. We also discuss some expected obstacles that might make it difficult to learn an intelligent control strategy. Subsequently, we will present related work and analyse several recent approaches of RL for traffic control. We will then review the approach that is used in this work: First, we discuss the traffic control MDP and explain observation- and action-spaces, as well as the used reward signal. Then, the entire RL architecture is described, including the employed RL algorithm and NN function-approximators. Finally, we will briefly analyse the option of deploying an RL controller in a real-world traffic system.

4.1. Advantages of RL for Traffic Light Control

RL and especially DRL control strategies show promising characteristics that may enable them to better manage complex traffic systems than traditional approaches, and allow them to cope with the vast amounts of traffic data that are made available through emerging V2I communication. These anticipated advantages include:

Ability to handle large state-spaces – Whereas for other control algorithms it is often unclear how to deal with large amounts of unstructured input-data, the ability of NNs to structure data and to detect reoccurring patterns enables a DRL algorithm to effortlessly handle large state-spaces. A DRL traffic controller could thus be well suited to handle the large data-streams that individual vehicles may transmit via a V2I communication interface.

No traffic model is needed – Traditional traffic control strategies often employ some model of the traffic network that is used to optimise phase schemes, splits and offsets (see section 3.3). Inaccuracies and simplifying assumptions are inherent to every traffic model and can lead to suboptimal solutions and erroneous predictions. RL methods — at least the ones used in this work — are model-free, meaning that no explicit model of the traffic system is required to learn a good policy. These algorithms could, therefore, eliminate a major source of errors of traditional systems. Note, however, that we will often still use a model to obtain simulated experiences as collecting experience in the real world might be inefficient and sometimes even dangerous.

Ability to quickly adapt to changing traffic conditions – Most traditional traffic control strategies gradually adapt phase splits and offsets of a set of intersections and are thus slow to adjust to changing traffic conditions. RL controllers, on the other hand, may utilise an action-space that allows the adaption to changing conditions within seconds. Action-spaces may very well be designed to let the agent replicate the behaviour of a traditional approach but also to quickly adapt to changing conditions in case of an unforeseen event.

No iterative optimisation in the inference pass – Responsive strategies (such as SCOOT) rely on iterative optimisation of the control strategy at runtime, leveraging some simulated model of the traffic system. As control systems are subject to tight real-time constraints, the traffic controller may need to execute a control decision before the optimisation algorithm has converged. An NN does not apply any iterative optimisation and has a relatively fast inference pass. DRL controllers are therefore able to output a definite control action within a comparably slow timeframe and satisfy the real-time constraints of the traffic system without the need to compromise the quality of the executed control decisions.

Better scalability – The complexity of coordinated traditional control strategies (e.g. TRANSYT) often grows exponentially with the number of considered intersections and renders many approaches infeasible for anything more than a small handful of coordinated traffic lights. DRL strategies certainly also have their limits when scaling up to the coordinated control of many intersections. However, these methods tend to scale a lot better with the number of considered junctions (see section 4.5). Furthermore, a DRL agent can easily be enhanced to a Multi-Agent Reinforcement Learning (MARL) setting in that several independent controllers learn to navigate their respective part of the traffic infrastructure and to cooperate through communication among the agents, breaking up the control problem into smaller, manageable sub-problems.

4.2. Challenges of RL for Traffic Light Control

Even though RL methods show great promise and have been successfully applied towards solving various difficult control problems in simulated environments (e.g. Mnih et al., 2015; Duan et al., 2016), the instances of it being applied in real-world and safety-relevant systems are limited. This is due to some general problems in RL that are hard to overcome. For the application of an RL controller to the traffic control problem in particular, prominent pitfalls are:

No optimality claims or convergence guarantees – A general problem of DRL algorithms is that there can rarely be made any claims on the optimality of a found solution. The unsatisfying practice is mostly that RL systems are trained until their performance no

longer increases. Even though the resulting policy is often able to outperform other state-of-the-art controllers, there usually is no notion of whether or not the performance might increase even more. It is generally possible that a different learning algorithm or a different architecture of the used NNs may result in a better policy. Even worse, for a specific algorithm and configuration, different runs may yield different results and a particular learned policy might severely underperform. The reason for this is partly that DRL algorithms initialise the parameters of the used NNs by some random values and that the resulting model strongly depends on these initial values. An RL traffic engineer would therefore be faced with the question if a trained system is good enough or should be further improved, without having a clear notion of what a good solution might be and how hard it is to obtain a better one.

No hard performance guarantees – NN-policies are notoriously hard to analyse and understand. In so-called adversarial attacks, it is often shown that a small, unusual change in the input signal of a NN can lead to a completely different output signal (e.g. Kurakin et al., 2016; Fawaz et al., 2019). It is therefore difficult to make any definite claims on the behaviour and the safety of a learned policy, as a small change in the input signal may lead to unexpected, radically different behaviour. Such unexpected behaviour may have disastrous results in safety-relevant systems such as traffic lights.

Poor sample-efficiency and risky exploration – Most RL algorithms suffer from poor sample-efficiency, meaning that it may take many sampled interactions before a good policy is found. For example, the AlphaGo Zero algorithm, that learned to play the game of Go through self-play, was trained in 29 million games during 40 days and still kept getting better (Silver et al., 2017). In simulated environments, poor sample-efficiency is mostly unproblematic; however, when learning from real-world experience, training times of learning-based approaches may be prohibitively long. In the traffic control setting, for example, traffic authorities could probably not afford to let a poorly performing policy control traffic during rush-hours. Moreover, the trial-and-error learning approach and an unconverged traffic policy at the beginning of training may cause poor performance that can lead to inefficient and even dangerous traffic situations. Note that issues due to unsafe exploration are an active area of research, and many publications propose approaches towards safe exploratory actions (e.g. Lütjens et al., 2018; Koller et al., 2018).

Combinatorial action-spaces are hard to handle – A common choice of the action-space of an RL traffic light would let an agent choose the next of a discrete set of phases that a traffic light will display. If a traffic light admits N_i different phase options, jointly selecting the phases of M intersections will result in $N_{all} := \prod_{i=1}^{M} N_i$ different options. If ,for example, we would train a DQL algorithm (see section 2.5) to predict all N_{all} action-values, the NN would need N_{all} output nodes. This combinatorial scaling of the action-space puts a strain on the number of intersections that we may optimise jointly.

Slow convergence for broad action-spaces – An RL algorithm may employ different action-spaces that provide different degrees of freedom. For example, the agent could make small adjustments to phase splits and offsets and thus create an action-space that results in similar strategies as traditional approaches such as SCOOT. However, this choice of an action-space is limiting to the kinds of strategies that the DRL agent can implement. A broader set of possible behaviours could, for instance, be implemented by letting the

agent choose the appropriate phase at every timestep (e.g. every second). This is a more flexible formulation as the latter approach could very well replicate all behaviours of the former but not vice versa. On the other hand, the first approach might converge faster because it is easier to optimise. In particular, the exploratory actions of the latter approach could result in very short phase times and predominant intergreen times, as a constant phase would require the agent to take the same action many times (e.g. if we want to implement one minute of a particular phase, the policy would need to sample this action 60 times in a row, and any exploratory action would force the traffic light to go through the amber and the all red period). Exploratory action in the first approach, on the other hand, would only cause a small change in the phase split of the intersection. The approach with the broader action-space could therefore take considerably longer to find a good policy.

Delayed rewards – Most MDPs do not give direct feedback on the taken actions but reward the agent for good actions after a significant time delay. The agent is consequently faced with the challenge to figure out which actions led to high rewards and which did not. Changing the phase of a traffic light lets it go through the intergreen phase, commanding all vehicles to break and halt. When the new phase is shown, vehicles may take a while to reach full speed. Common reward-functions, such as the average velocity or the average trip-time, thus may be strongly decreased for an intermediate timeframe after the agent selects a new phase, even when the decision was appropriate. Furthermore, some actions may have little effect on the environment. For example, executed actions during the intergreen time may be ignored, or a policy that controls the phase split may have no effect on what is happening when the split of the phase that is altered is not currently shown. This is in contrast to control problems with more immediate feedback (e.g. in Atari games like Pong, the up action will result in an instantaneous movement of the paddle). The agent may therefore struggle to learn which are the outcomes of its actions.

Some of the discussed challenges might be alleviated through a good choice of the used RL algorithm or a suitable action-space. Others are inherent to the general RL framework and are thus hard to avoid. In the following, we will explain the methods that are used in this work, including the RL algorithm and the full MDP of the traffic system, and explain how our design choices may influence the relevance of the aforementioned advantages and challenges of traffic light control with RL.

4.3. Related Work

Reinforcement Learning is not a new idea and research towards the application to real-world problems has a long and ongoing history. First approaches of applying RL algorithms in the domain of adaptive traffic signal control date back to the 1990s (e.g. Mikami and Kakazu, 1994; Thorpe, 1998). In the 2010s, the advent of deep learning algorithms led to an ongoing increase of research efforts in ML in general and RL in particular (Henderson et al., 2017), spawning a plethora of research articles that utilise NNs to learn value-functions and policies. Approaches to apply the RL framework to the traffic control problem can therefore be divided into two categories: approaches relying on classical RL, such as tabular methods or linear function approximation, and those utilising modern DRL. Table 4.1 shows an excerpt of publications of both classical and modern approaches.

	MDP			RL		
	observations	actions	rewards	algorithm	DRL	MARL
Richter et al., 2006	current phase, current phase duration, detector information & information about incoming flows from neighbouring intersections	next phase to be activated; in 16 timesteps each phase needs to be activated at least once	throughput of the intersection (local)	Natural Policy Gradient	✗	✓
Shoufeng et al., 2008	total delay at the intersection	phase times for a predefined phase scheme	total delay at the intersection (global)	Q-Learning	✗	✗
Arel et al., 2010	delay for every afferent lane, relative to the average delay for local and neighbouring intersections	next phase to be activated	relative reduction in cumulative delay at the intersection (local)	Q-Learning	✗	✓
El-Tantawy et al., 2013	current phase, time of current phase & maximum queue lengths associated with each phase	next phase to be activated	reduction in cumulative delay of all vehicles (local)	Q-Learning	✗	✓
Prabuchandran et al., 2015	time since last activation for each phase & queue lengths of afferent lanes	next phase to be activated	combination of negative average delay of vehicles and negative red-time for each phase (local)	Q-Learning	✗	✓
Mannion et al., 2016	current phase, time of current phase & maximum queue lengths associated with each phase	stay with current phase or advance to the next	comparison of 3 reward functions, including queue lengths & waiting times (local)	Q-Learning	✗	✓
Van der Pol and Oliehoek, 2016	binary matrix of the positions of vehicles on the lanes & current phase information	next phase to be activated	waiting times, delays, teleports, emergency stops (indicate crashes or jams) & phase switches (local)	DQL	✓	✓
Casas, 2017	average velocity of vehicles for every afferent lane	phase split of a predefined phase scheme; 80% of cycle time is allocated by the agent, 20% is fixed	velocities of each lane (global)	DDPG	✓	✗
Mousavi et al., 2017	overview images of the intersection for the last four timesteps	next phase to be activated	reduction in cumulative delay of all vehicles (only 1 intersection)	DQL & PG	✓	✗
Liu et al., 2017	binary matrix of vehicle positions, matrix of velocities & current phases of local and neighbouring intersections	next phase to be activated	quadratic waiting time; loosely relates to driver patience (global)	DQL	✓	✓
Zhang et al., 2018	number of vehicles and distance of nearest vehicle for all afferent lanes, current phase and phase time, indicator of amber period & daytime	keep the current traffic light phase, or to switch to the next	velocity relative to its maximal value (global)	DQL	✓	✓

Table 4.1.: Previous Reinforcement Learning approaches for adaptive traffic signal control.

Classic RL leverages tabular representations or simple approximators of the action-value function to find actions that promise high reward. These methods include algorithms like SARSA and Q-Learning (see section 2.3.3), with Q-Learning being the most popular option for the traffic control problem (e.g. Richter et al., 2006; Salkham et al., 2008; El-Tantawy et al., 2013; Prabuchandran et al., 2015; Mannion et al., 2016). El-Tantawy et al., 2014 gives a review of popular publications. DRL algorithms are able to encompass large or even continuous observation- and action-spaces. For example, Casas, 2017 learns a centralised controller that simultaneously controls a large number of traffic lights, leveraging a large amount of traffic data. Other publications use camera images of the traffic scenario (Mousavi et al., 2017) or spatial matrices of positions and velocities (Van der Pol and Oliehoek, 2016; Liu et al., 2017) as observation, leveraging the power of Convolutional Neural Networks (CNNs) to deal with large, spatially correlated input data such as images. Apart from the learning algorithm, the presented methods differ in the nature of the observations, actions and rewards.

The observation representation defines the basis on which an agent can make decisions. Many works feature some high-level description of the traffic state like queue lengths or vehicle counts for all afferent roads of an intersection (e.g. Thorpe, 1998; Wiering, 2000; Oliveira Boschetti et al., 2006; Salkham et al., 2008; Mannion et al., 2016; Zhang et al., 2018) or the vehicle delay at an intersection (e.g. Shoufeng et al., 2008; Arel et al., 2010; Medina and Benekohal, 2012). Other publications like Richter et al., 2006 argue that this information is rarely available in a real-world traffic system and therefore operate on the partial traffic information that can be inferred from inductive loop detectors. Furthermore, depending on the action-space, most approaches feature some information about the current observation of the traffic lights, like the current phase and phase times (e.g. Richter et al., 2006; El-Tantawy et al., 2013; Mannion et al., 2016; Van der Pol and Oliehoek, 2016), or the time that a specific phase has not been activated (e.g. Prabuchandran et al., 2015). Some works also feature other data like average velocities of vehicles in afferent lanes (Casas, 2017) or the current daytime (Zhang et al., 2018).

Many publications consider a fixed phase scheme and let the agent decide upon the respective length of each phase time, e.g. by deciding in every timestep whether to advance to the next phase or to stay with the old one (e.g. Thorpe, 1998; Oliveira Boschetti et al., 2006; Shoufeng et al., 2008; Mannion et al., 2016). A different, broader approach lets the agent select the next phase from a predetermined set and thus allows for variable phase sequences (e.g. Richter et al., 2006; Salkham et al., 2008; Arel et al., 2010; El-Tantawy et al., 2013; Prabuchandran et al., 2015). The generality of the latter approach is, however, often weakened by long intervals in between actions (e.g. 5 seconds in Richter et al., 2006 or 20 seconds in Arel et al., 2010). Another distinguishing factor is the number of distinct phases that the agent may choose from (e.g. 2 in Mousavi et al., 2017 or 4 in Liu et al., 2017).

The reward function defines what the agent tries to achieve; therefore, it embodies the characteristics that we want the traffic system to display, which can range from efficiency (e.g. a low average trip time) to social or environmental aspects (e.g. low noise levels or CO_2 emissions). Most publications try to optimise measures like the throughput (e.g. Richter et al., 2006), the average delay (e.g. Shoufeng et al., 2008; Arel et al., 2010; El-Tantawy et al., 2013), queue lengths (e.g. Mannion et al., 2016) or average velocity (e.g. Casas, 2017; Zhang et al., 2018). Other, not so common measures include the average quadratic waiting time, which loosely represents a drivers patience (Liu et al., 2017), or a penalty for teleports and emergency stops of vehicles, which indicate crashes and traffic jams in SUMO (Van der Pol and Oliehoek, 2016). Some of these measures are strongly related (e.g. an increased average delay will result in decreased average

velocity), whereas others might not correlate or may even be antagonistic (e.g. CO_2 emissions and average velocity). Some publications therefore try to optimise a weighted combination of several measures (e.g. Prabuchandran et al., 2015; Van der Pol and Oliehoek, 2016), and thus require the agent to find a trade-off between several antagonistic factors.

The poor scalability of classical methods (especially tabular methods) forces many works to limit their experiments to the control of only a single intersection (e.g. Thorpe, 1998). To avoid scalability issues, many approaches instead tackle the control problem by means of Multi-Agent Reinforcement Learning (MARL). MARL considers a number of agents that can each control a small part of an environment. In a traffic light environment, one agent often controls only a single intersection, mostly based on some local observation of the traffic state at the the respective intersection and at neighbouring intersections (e.g. Richter et al., 2006; Arel et al., 2010; Liu et al., 2017). In a simple setting, an agent learns to control his intersection without being aware of other agents (e.g. Richter et al., 2006; Arel et al., 2010). Reward functions of the individual agents can either be based on local measures (e.g. Arel et al., 2010; El-Tantawy et al., 2013; Van der Pol and Oliehoek, 2016) or the agents can share one global function, which encourages cooperation (e.g. Liu et al., 2017; Zhang et al., 2018). As control decisions of individual agents can strongly influence the reward of others, a better solution may be found through explicit coordination of the agents. This can be achieved by means of coordination-graphs and game-theoretic algorithms such as max-plus (Kuyer et al., 2008; Bakker et al., 2010; Medina and Benekohal, 2012). The max-plus algorithm is used to choose a joint action by iterative negotiation between the agents.

To the best of our knowledge, there has so far been no real-world implementation of a full RL system for traffic light control. This can be explained by the poor sample efficiency and safety issues (see section 4.2). Experiments strongly differ in the realism of the simulation. An important factor in the realism of simulations is the used traffic simulator (see section 3.5). Macroscopic simulators (e.g. in Wiering, 2000; Richter et al., 2006) do not provide the same realism as microscopic simulators (e.g. in Casas, 2017; Mousavi et al., 2017) but are superior in terms of simulation time. Whereas most models rely on generic road networks such as arterial roads (e.g. Medina and Benekohal, 2012) or regular grids (e.g. Richter et al., 2006; Liu et al., 2017) as an environment, some publications consider real-world examples such as small districts of cities like Bangalore (Prabuchandran et al., 2015), Toronto (El-Tantawy et al., 2013), Dublin (Salkham et al., 2008) or Barcelona (Casas, 2017). Unfortunately, many publications are unclear about their employed assumptions for the traffic model. Simplifications like vehicles that only drive straight and never turn (Mousavi et al., 2017) are not uncommon and are often not clearly stated. The lack of a common benchmark makes the comparison of different approaches almost impossible. A notable exception provides Vinitsky et al., 2018 that, among other environments that focus more on autonomous vehicles, proposes a benchmark grid-environment (often called Manhattan) and allows the comparison of RL traffic light controllers. Vehicles in this environment always follow the road that they are on and never turn. Roads have only one lane and traffic lights therefore only need two distinct phases (East-West green and North-South green). When leaving the simulation at the end of the respective street, they immediately reappear at the beginning of the road, resulting in a constant number of vehicles on every road.

The diversity of environments in different publications results in a lack of comparability. Many works therefore study how their approach compares to traditional traffic control methods (see section 3.4); however, as most traditional methods do not have an open-source implementation, many authors implement their own algorithm that may or may not correspond to the actual control methods, they try to compete with (e.g. Salkham et al., 2008; Prabuchandran et al., 2015;

Mousavi et al., 2017). Most of these works report to significantly outperform traditional control methods in a range of experiments. Other publications compare the results of two RL algorithms in their own environment (e.g. Richter et al., 2006; Liu et al., 2017; Casas, 2017), or even show the improvement of their methods over a completely random policy (e.g. Casas, 2017). Despite the lack of comparability and the questionable realism of implementations of traditional control methods, authors generally consent on the great potential of RL control methods and superior performance compared to other approaches.

4.4. A Traffic Light Control MDP

Markov Decision Processes (MDPs) formalise environments in that an agent can infer some observation of the state of the system and execute actions according to its policy to obtain a scalar reward (see section 2.2). In order to characterise the traffic MDP that will be used in this work, we therefore have to define observation- and action-spaces as well as a reward function.

In this thesis we will not explicitly address the issues of partial observability. The policy π and the action a_t at time t thus only depend on the current observation o_t and not on the history h_t of previous observations and actions (compare to the general POMDP setting in section 2.2):

$$a_t \sim \pi(o_t). \tag{4.1}$$

Instead, we will try to implement the information of previous observations and action implicitly into the current observation.

4.4.1. Observations

The observation-space of the agent may generally consist of everything that the agent can measure and is thus defined by the implemented sensory hardware of the traffic system. As formerly discussed, this thesis investigates the benefits of rich information about the current state of the traffic system, that can be obtained through emerging V2I communication interfaces (see section 3.6). We thus need to compare the performance of the policies of two different agents: One having very limited access to the current traffic state — we will call this the *solitary agent* — and one that features information of the current state of individual vehicles — the *communicative agent*. This distinction can be implemented through two different observation-spaces.

The Solitary Agent

We will here assume that the agent can always access all signals that are internal to the controlled traffic lights. In particular, a traffic light should always be able to infer which phase it is currently showing. As our traffic controller centrally defines the actions of all controlled traffic lights, we have to assume an existing communication infrastructure that connects all controlled traffic lights to a central server.

In order to be fed to an NN, the available traffic information has to be encoded as a fixed-length, real-valued vector. Figure 4.1 shows a time trajectory of the observation-vector for a solitary agent that controls a single intersection with two different phases. The current phase of every intersection can be described by a one-hot *phase vector* s^{phase} (e.g., if the phase scheme consists of four different options as in figure 3.2, the vector is four-dimensional). As this vector can

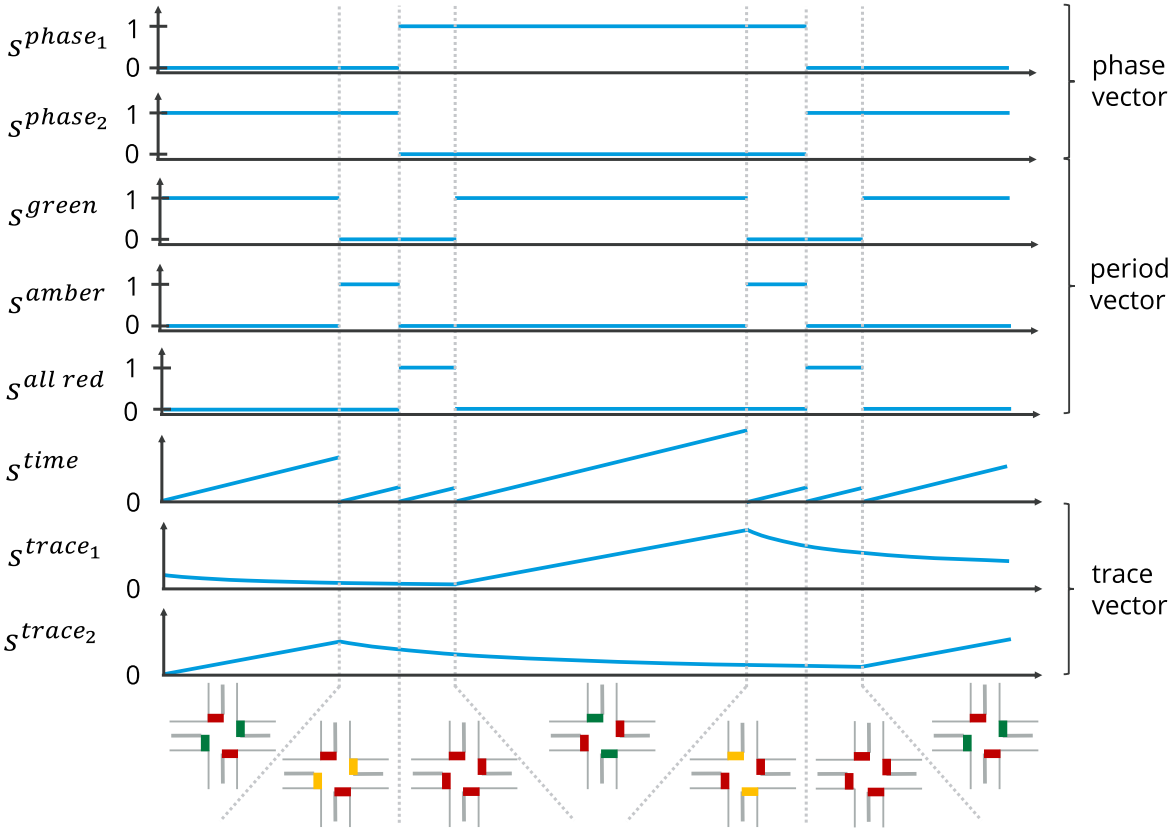


Figure 4.1.: Example trajectory of the observation-vector of the solitary agent for a single intersection with only two phases. The phase and the period vector encode the current phase of the intersection; the time component shows the passed time since the last change; the trace vector gives some notion about the recent history of the phases.

not encode whether the traffic light currently shows the green-phase or is in an intergreen period (see figure 3.1), a second, three-dimensional one-hot *period vector* shows if the traffic light currently is in the green period (s^{green}), the amber period (s^{amber}) or the all red period ($s^{all\ red}$). During the amber period, the phase vector encodes the phase that is currently showing the amber signal, and during the all red phase, it encodes the next scheduled green-phase. The state also contains a temporal element s^{time} that denotes the time since the last change in the traffic period. It thus provides explicit timing information which, for example, may help to properly time two adjacent intersections to create a green wave.

In order to have some notion of the history of former phases, the observation features a *trace vector* s^{trace} that is implemented as a leaky integrator of the respective phase signal:

$$C \cdot \frac{ds^{trace_i}}{dt} = -\frac{s^{trace_i}}{R} + s^{phase_i} \cdot s^{green}, \tag{4.2}$$

where i identifies the respective phase and R and C are parameters of the integrator that define the shape of the signal. As the trace increases for an active phase and slowly decays for an inactive phase, it conveys some information of the amount of time that the respective phase was recently shown. Other publications choose to include the elapsed time since a particular phase has been shown (e.g. Prabuchandran et al., 2015), but we found the trace to be more informative as it conveys information about how long the last green phase might have lasted. Including both measures did not improve the results of our experiments.

Note that, even though the history is partly encoded in the traces, it would certainly be possible to infer more information when considering the agent's entire trajectory of observations and actions. The Markov assumption (see section 2.2) is therefore not fulfilled. In a more sophisticated approach, we could, for example, consider implementing the full history by using a Recurrent Neural Network (RNN), which is often used to infer the hidden state for time-series data in Hidden Markov Models.

Table 4.2 shows a summary of the observation-space of the solitary agent and calculates its dimensionality, where M denotes the number of controlled intersections, and N_i denotes the number of possible phases of intersection i.

Feature	Description	Dimensions
phase	the current phase of all traffic lights	$\sum_{i=1}^{M} N_i$
period	the current period of all traffic lights	$3M$
time	the time passed since the last change in period	M
trace	shows how much each phase was shown lately	$\sum_{i=1}^{M} N_i$
		$2\sum_{i=1}^{M} N_i + 4M$

Table 4.2.: Summary of the observation-space of the solitary agent and a calculation of its dimensionality. M here denotes the number of intersections that the agent controls and N_i is the number of distinct phases that can be shown at the intersection i.

The Communicative Agent

In order to model the availability of a V2I interface, via that vehicles can send state-information to the traffic infrastructure, we enhance the observation-space by information about the position and velocity of individual vehicles. The position can here be described by the road that the car is on, the position along that road and the lane that it is currently occupying. The velocity is the rate of change of the position along the current road.

As we need a fixed-size vector as an input to the NN, it is not clear how to handle the varying number of vehicles that navigate the traffic scenario. A possible approach would be to transform the knowledge about individual vehicles into some statistics of fixed length, such as queue size on particular lanes (as in El-Tantawy and Abdulhai, 2010) or vehicle densities in particular road sections. However, this approach could be limiting to the expressiveness of the observation. Another approach would be to introduce some NN architecture that can handle the variable length of observation, such as a Convolutional Neural Network (CNN) which can reduce a variable-length vector to a fixed-length through Pooling operations (as in Young et al., 2017). Here we will take a simplistic approach in that the traffic system can communicate with up to K vehicles on every afferent road of every intersection. Figure 4.2 shows this for a single intersection and $K = 2$, where blue vehicles are observed by the traffic infrastructure and grey ones are not. If more than K vehicles are on a road, only the K vehicles that are closest to the intersection will be explicitly observed. Vice versa, if fewer than K vehicles are on a road, the missing entries in the observation-vector are filled with zeros. In order to, at least, partly account for the not explicitly observed vehicles, the observation-vector also features the number of vehicles on a particular road and their average velocity for each of the roads in the simulation.

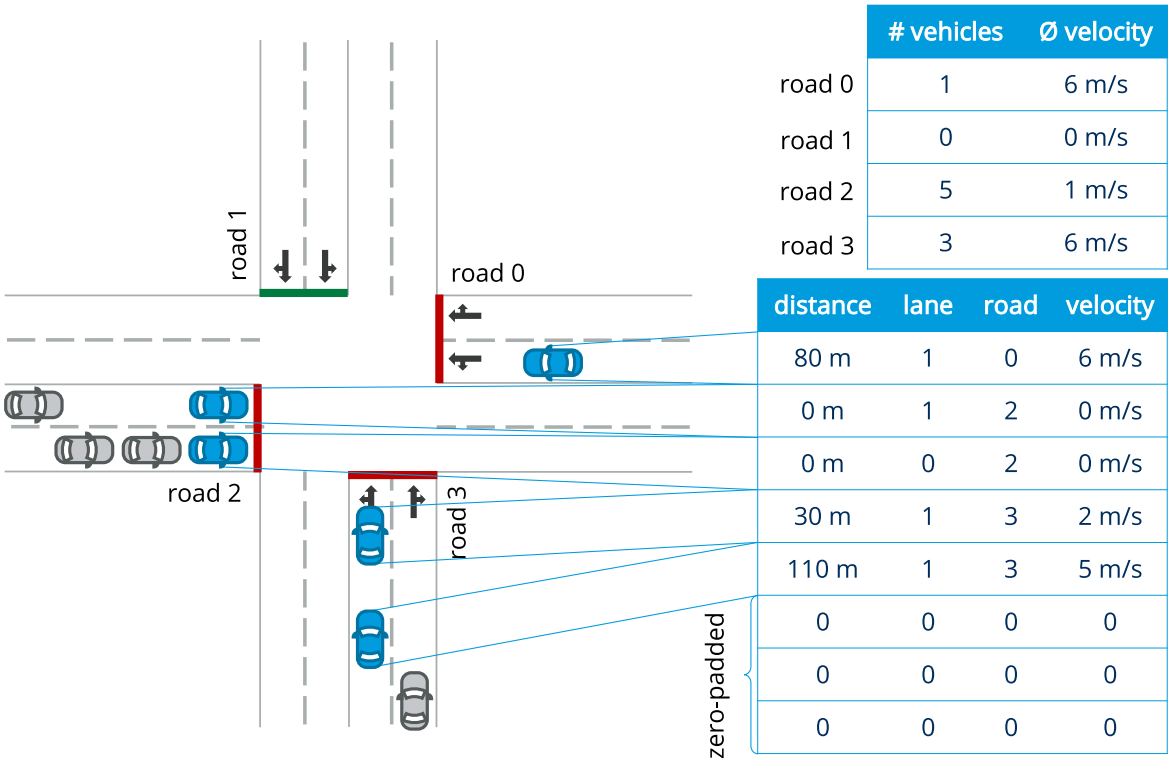


Figure 4.2.: Observation-space of the communicative agent. Here, the first two vehicles, shown in blue, on every road are considered in the agent's observation-space. If fewer than two vehicles are on a road, the missing vector elements are filled with zeros. To at least partly account for the remaining vehicles, the observation-space also features the number of vehicles and the average velocity for each of the roads.

The transmitted information is deliberately limited to the position and velocity, which can always be known to the vehicle, even though other measures could eventually enable more informed control decisions. In particular, route information or the next turn intention might strongly benefit the performance of a RL agent. However, these measures are internal to the driver and are not always known to the vehicle. There are many other features that a vehicle could transmit, but we do not consider here. It could, for example, infer some statistical data such as the expected route (e.g. if the current route matches the typical trajectory from the driver's workplace to his/her home, the vehicle could anticipate the next turn) or the driver's temperament. In an early approach, we used an observation-space that consisted only of the number of vehicles on every lane, the average velocity of vehicles on every lane and the distance between the closest vehicle and the intersection for every lane. Even though the agent was still able to learn a reasonable control policy, the final performance was significantly worse than the one of the formerly described agent. Another approach, that only featured the positions and velocities of the observed vehicles but not the statistical data of each of the roads, also led to decreased performance, especially for high demand scenarios.

Note that we do not explicitly model the V2I communication but assume the knowledge about individual vehicles to be available to the agent at all times. In a more sophisticated approach, we could model the communication with a conventional simulator. However, since the latency of V2I communication (on the order of only one millisecond for 5G) is significantly shorter than the timestep of our traffic MDP (one second), we would expect the effects of a more realistic communication protocol on the results of our experiments to be negligible. For controllers that

need faster feedback (such as for collective breaking behaviour in platoons), this assumption should be reconsidered.

Within our simulation, the current traffic phases, alongside the position, velocity and the route of each vehicle, exhaustively describe the state of the traffic system. Even though the communicative agent might still obtain additional knowledge by considering its entire history, the current observation captures the true state of the POMDP significantly better than in the solitary case. The Markov assumption may therefore be considered to be approximately satisfied.

Table 4.3 shows a summary of the observation-space of the solitary agent and calculates its dimensionality, where M denotes the number of intersections that the agent controls, N_i is the number of distinct phases that can be shown at the intersection i, K is the number of observed vehicles per road and L_i is the number of afferent roads, leading up to intersection i.

Feature	**Description**	**Dimensions**
phase	the current phase of all traffic lights	$\sum_{i=1}^{M} N_i$
period	the current period of all traffic lights	$3M$
time	the time passed since the last change in period	M
trace	shows how much each phase was shown lately	$\sum_{i=1}^{M} N_i$
position	position along the current road of all observed vehicles	$K \sum_{i=1}^{M} L_i$
lane	occupied lane of the current road of all observed vehicles	$K \sum_{i=1}^{M} L_i$
velocity	velocity along the current road of all observed vehicles	$K \sum_{i=1}^{M} L_i$
road	the current road of all observed vehicles	$K \sum_{i=1}^{M} L_i$
road velocities	average velocity for every road in the simulation	E
road occupancy	number of vehicles for every road in the simulation	E
		$\sum_{i=1}^{M} 2N_i + 4KL_i + 4M + 2E$

Table 4.3.: Summary of the observation-space of the communicative agent and a calculation of its dimensionality. M here denotes the number of intersections that the agent controls, N_i is the number of distinct phases that can be shown at the intersection i, K is the number of observed vehicles per road L_i is the number of afferent roads, leading up to intersection i and E is the number of interconnecting roads in the simulation.

Note that neither of the agents uses measurements of inductive loop detectors, even though especially the solitary agent might significantly increase its performance through the utilisation of this partial traffic information. The reason for that is that the deployment of loop detectors again results in unclear design decisions. In particular, the distance to the intersection and the number of installed detectors per lane can strongly influence the resulting policy. Furthermore, in all but one of our experiments, the traffic demand does not vary over the course of the experiment, which strongly alleviates the need for loop detectors. The agent thus implicitly incorporates information on the arrival statistics of vehicles. Another way to look at the experimental setting would, therefore, be that some sensors (e.g. loop detectors) measure the traffic load and the traffic server selects the RL agent that matches this particular demand.

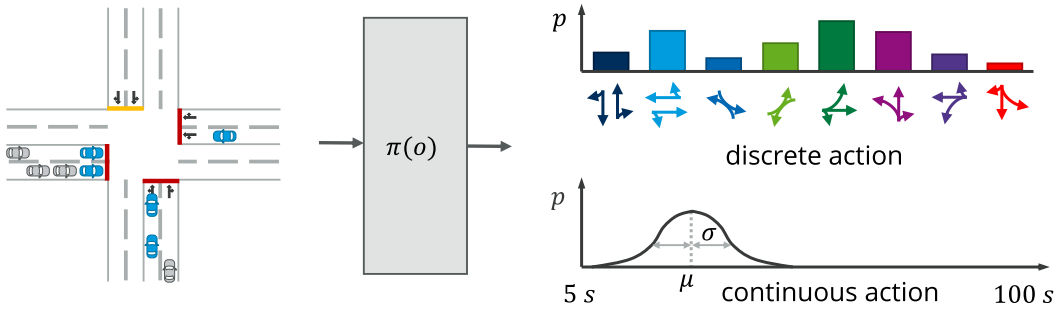


Figure 4.3.: Action-space of the agent for a single intersection. The policy outputs a discrete distribution over the available phase options and a Gaussian distribution over the duration of the active phase. For eight phase options, the resulting action-space is ten-dimensional (mean and standard deviation of the Gaussian and eight discrete phase probabilities). For more than one traffic light, the dimensionality of the action-space grows linearly.

4.4.2. Control Actions

As already discussed in section 4.2, the design of the action-space of a traffic agent is no trivial task as it strongly influences the speed of convergence of the RL algorithm as well as the diversity of behaviours that the agent may exhibit. Some publications limit the action-space to control only the phase split (e.g. Casas, 2017). Such a narrow framing of the control problem can result in fast convergence, but it is unclear if the limited range of behaviours that this policy can encompass may navigate the traffic as precisely as a more general policy. Another approach decides in fixed time-intervals, which phase to display next (e.g. Richter et al., 2006). The problem with this action-space is that exploratory actions immediately let the intersection undergo an intergreen period. Most works alleviate this problem by using a relatively long time interval in between actions, again limiting the range of different behaviours that the policy can exhibit, or by neglecting intergreen periods in the simulation.

We here try to formulate the action-space to be able to implement a fairly wide variety of different strategies. For every controlled intersection, our policy outputs:

- A discrete distribution over the **next phase** options. The available phases consist of all options of the opposing streets approach and the single street approach (see figure 3.2), resulting in eight different phases. All phase options consist of only compatible streams, which ensures safety but slightly constrains the range of possible signalling strategies.
- A continuous distribution over the full **duration of the active phase**. The phase duration can vary between 5 seconds and 100 seconds. The lower limit here ensures that at least one of the waiting vehicles may pass the intersection during each phase. The upper limit constrains the time that a phase is usually shown, avoiding unnecessary green-idling (granting the right of way to approaches without any vehicles waiting). Note however that the agent may keep on choosing the current phase (the discrete action) to further increase the phase time.

Figure 4.3 visualises the action space. This action-space can exhibit a wide variety of different behaviours and also avoids the described problem of exploration, by individually letting the agent learn to control the phase and the phase duration. Note that this framing of the traffic control problem requires an agent that can simultaneously handle discrete and continuous action dimensions. To the best of our knowledge, we are the first ones to use this action-space

to control traffic and among the firsts to consider mixed discrete and continuous action-spaces.

We here let the agent choose from a fixed set of phases that consists of all options of the opposing streets approach and the single street approach (see figure 3.2). An alternative approach would be to choose every signal individually, resulting in an even broader policy. However, allowing antagonistic streams to simultaneously obtain the right of way may result in accidents as a consequence of exploratory actions. In contrast, the limitation to phases that give simultaneous right of way only to compatible streams strongly alleviates the risks of exploration.

In an early approach, we also tried to decide the next phase in every decision step. However, in our experiment, this led to slower convergence than the aforementioned approach and inferior final performances. We therefore decided to abandon this approach.

Switching the phase lets the traffic light undergo the intergreen period, first showing an amber signal and then the all red signal. The length of intergreen periods is designed to ensure safety and cannot be altered by the agent. In the case that the agent switches between phases that contain some identical streams, those streams do not have to undergo an intergreen period (e.g. if it switches from the northern single street to the north-south opposing streets left turn, the northern left turn can simply retain its right of way).

4.4.3. Rewards

Finally, we have to define a reward signal that translates our qualitative goals for the traffic system into a quantitative measure that RL algorithms can optimise. In games and other generic environments, the design of the reward-function is straightforward as the goals are clearly defined. In the real world, however, goals can be ambiguous, and choosing a performance measure is less intuitive than one might imagine. For example, while in most arcade games a reward signal is readily available in form of the game-score (Mnih et al., 2015), an autonomous car agent needs to handle a plethora of different goals; some of which are not easily quantified, and need to implement complex concepts such as morality (e.g. the infamous dilemma of deciding whether to avoid a playing child, that suddenly enters the road, by steering into the opposing traffic) or fairness (e.g. the question of how much diminished individual performance a vehicle should accept in order to increase the systems gross performance).

That being said, an exhaustive analysis of the goals of traffic systems, including a discussion of factors like morality, fairness, safety, efficiency and many more, is beyond the scope of this work. Most RL approaches, as well as most traditional control strategies, concentrate on only one performance measure, such as velocity or delay (see section 4.3). Some approaches also combine several performance measures by using a weighted combination of them (e.g. Khamis et al., 2012).

For the most part, we will here use only the average velocity of all vehicles as a reward function. We will constrain all rewards to values between zero and one. The velocity is therefore divided by the respective speed limits. In a later experiment, we will take a closer look at the reward function and consider multiple objectives through rewarding the agent by a weighted combination of:

- the average velocity of all vehicles divided by the respective speed limit
- the flow rate (proportion of moving vehicles)
- the total CO2 emissions in the controlled traffic network; normalised between 0 and 1
- the average quadratic proportion of time spent waiting within the last 100 seconds, which loosely corresponds to the average driver patience (Liu et al., 2017).

Even though these measures might be strongly correlated (e.g. the flow rate is the average over the binarised velocity), we hope for increased performance in at least some of the performance measures, through the application of a combined reward function.

Note that the presence of a particular reward signal requires the availability of corresponding sensory infrastructure. In particular, our employed rewards would need to be measured by the individual vehicles and transmitted to a central server that runs the RL algorithm. In the setting of the communicative agent, this does not pose a problem since the reward can easily be transmitted together with the observation. For the solitary agent, however, the availability of the reward to the RL algorithm is unrealistic. Even though we might infer the average velocity through measurements from inductive loop sensors, the resulting reward would be an inaccurate, time-delayed version of the actual velocity, which might slow down convergence. Nevertheless, for this work, we will assume the full reward to be available in both the communicative as well as the solitary agent. This could, for instance, correspond to a setting in that the solitary RL agent is trained in simulation and, after convergence, is deployed to a real traffic system. The agent then cannot keep on learning after deployment. The communicative agent, on the other hand, could simply learn a policy from scratch, using real-world experience or it could be trained first in a simulation and then keep on learning after deployment.

4.5. Agent 4D7

Having discussed the traffic control MDP, we will now explain the algorithm that is used to learn a traffic light control policy in order to optimise the obtained rewards. As has become apparent in chapter 2, the space of available algorithms is large and versatile. In this work, as the title suggests, we intend to use a modern Deep Reinforcement Learning approach. The predominantly used algorithm throughout DRL literature in general, and approaches towards intelligent traffic infrastructure control in particular, is the DQL algorithm (e.g. Van der Pol and Oliehoek, 2016; Mousavi et al., 2017; Liu et al., 2017; Zhang et al., 2018).

As discussed in section 4.4.2, we deal with an action-space that consists of a number of discrete phase options and a continuous phase time for every controlled intersection. The DQL algorithm can only deal with continuous action-spaces by casting them into a finite number of discrete values. This not only constrains the expressiveness of the actions but may also slow convergence. Furthermore, jointly choosing the executed actions of several intersections admits a combinatorial number of distinct options to choose from. The number of predicted action-values and so the number of output nodes of the value-function thus grows exponentially with the number of controlled intersections. The DQL algorithm is therefore not well suited to learn the control of the MDP presented in section 4.4.

Apart from being able to handle the partly continuous, partly discrete, combinatorial action-space, the employed algorithm is required to be able to make use of off-policy data (see section 2.3). Off-policy algorithms have the advantage of being capable of learning from fewer environment-interactions than on-policy algorithms as they can reuse older transitions from a replay buffer. In a real-world traffic scenario, this is of crucial importance since experience cannot be collected faster than real-time and the RL agent is required to show reasonable behaviour as soon as possible. For simulated environments, good sample-efficiency is still important. In particular, the utilisation of microscopic traffic models (see section 3.5) makes the collection of experience relatively slow. Even in a simulated environment, algorithms that need a large number of environment-interactions therefore may take significantly longer than more sample-efficient

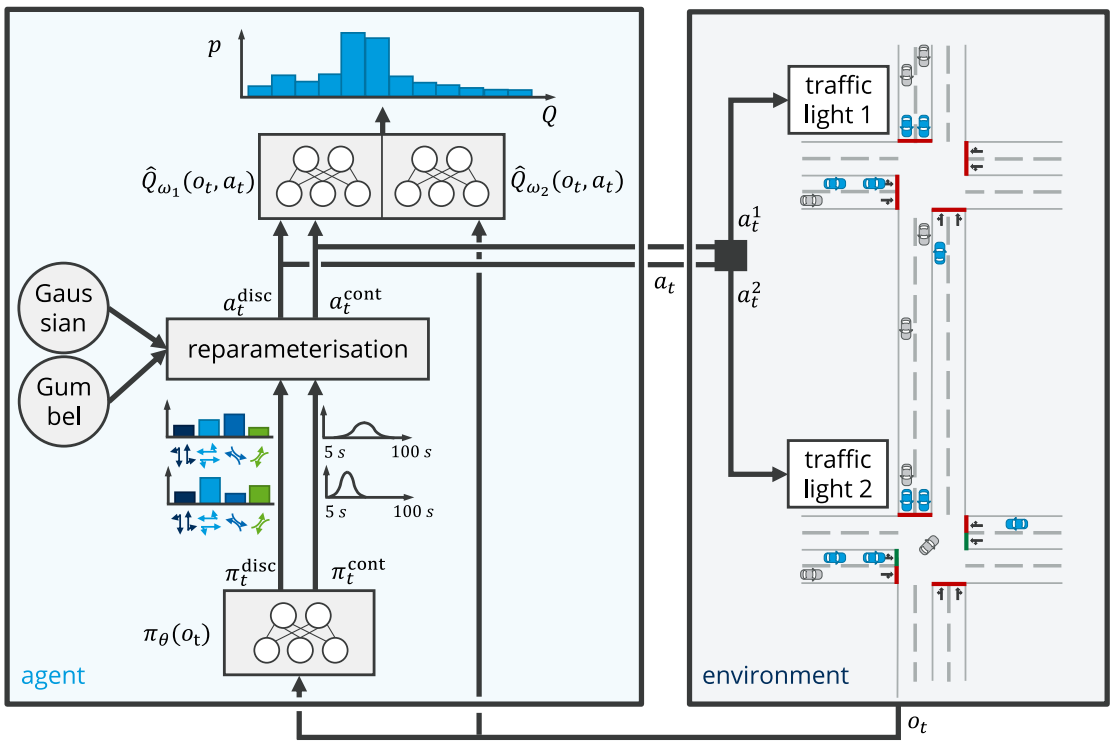


Figure 4.4.: Full agent-environment interaction loop for a traffic setting of two connected intersections with four distinct phase options each. The policy network computes a probability distribution over actions, based on the agent's observation. In the reparameterisation-block, a concrete action is sampled, which is then executed in the environment. The sampled action and the observation are used by two action-value functions to predict two probability distributions over action-values. The predicted action-value function can then be used to optimise the policy. The two action-value functions can be learned with the DQL algorithm.

ones. The necessity of an off-policy algorithm rules out many candidates such as A3C and PPO.

In the following, we will explain the algorithm that is used in this work. In addition, we also explain the variants that we tried out but found to show inferior performance in our experiments.

4.5.1. Architecture

The RL algorithm that we will use throughout this work is based on DDPG (see section 2.7.1), but adopts many ideas of the presented improvements (see section 2.7.3), as well as of the SAC algorithm (see section 2.7.4). The only publication that we are aware of that is using DDPG for traffic infrastructure control is Casas, 2017, which uses the continuous policy to centrally control the phase splits of several intersections. Figure 4.4 shows the full agent-environment interaction loop for a traffic setting of two connected intersections with four distinct phase options each (note that actually in all our experiments we will use eight different phase options as depicted in figure 4.3).

Actor

In each timestep, the agent obtains an observation of the environment and outputs an action that consists of the next phase and the duration of the current phase for all controlled intersections (see section 4.4.2). The policy $\pi_\theta(o_t)$ is implemented as a feed-forward NN (see section 2.4.1) with

the parameter set θ. In every timestep, it outputs a discrete distribution over phase options (the probability of choosing each phase) and a continuous distribution over the phase duration (the mean and standard deviation of a Gaussian distribution; the standard deviation is constrained to a reasonable interval). Note that the number of output dimension of the policy grows linearly with the number of controlled intersections (the number of outputs for intersections with eight phase options each is $10 \cdot M$ where M is the number of controlled intersections). In order to sample a discrete and a continuous action for each intersection, we need to apply the Gumbel-Softmax trick and the Gaussian reparameterisation trick, respectively (see section 2.7.2). This approach is similar to the one presented in Mordatch and Abbeel, 2017. Recall that directly sampling from the respective distributions would prevent gradient-based optimisation algorithms like Backpropagation (see section 2.4.2) to work properly. We thus obtain an action-vector that features a one-hot vector which selects one discrete phase option for each intersection and a continuous phase duration for each intersection. The sampled action can then be applied to the traffic lights in the environment. Note that we do not assume the actions to be selected deterministically (as in DDPG) but follow the approach of SAC of a stochastic policy.

Our algorithm can freely choose the distribution of both the discrete and the continuous actions. In order to not prematurely converge to a policy with very low entropy, we add a negative entropy term to the cost function of the policy. The policy is therefore encouraged to employ a high degree of randomness, which leads to more exploratory actions and thus, faster convergence. Another way to look at this is that the policy is trained to imitate the softmax distribution over the values of the action-value function (see section 2.7.4). In Haarnoja et al., 2018b, where this idea was proposed, the authors claimed that the scale of the rewards is the only parameter of their algorithm that needs to be carefully tuned. This makes a lot of sense since the softmax over only small values results in a relatively uniform distribution, whereas the softmax over values that differ strongly from another is close to deterministic. We here choose to scale the entropy term instead of the rewards since we need our reward function to admit clear upper and lower bounds (because we use a categorical action-value distribution as we will see in the next section). As our experiments showed that the appropriate entropy scaling factors of the discrete and of the continuous actions do not necessarily coincide, we introduce two additional parameters ε_{disc} and ε_{cont} to scale the two respective entropies. The loss function for learning the policy-NN therefore amounts to:

$$J_\pi = -\varepsilon_{disc} \cdot \mathcal{H}_{disc}\big(\pi_\theta(o)\big) - \varepsilon_{cont} \cdot \mathcal{H}_{cont}\big(\pi_\theta(o)\big) - \hat{Q}_\omega(a \sim \pi_\theta(o, o), \tag{4.3}$$

where $\mathcal{H}_{disc}$ denotes the sum over the respective Shannon entropies of each of the categorical distributions over discrete actions and $\mathcal{H}_{cont}$ denotes the sum over the differential entropies of each of the continuous Gaussian distributions.

The two entropy parameters turn out to require intricate, iterative tuning. A value that is too low results in premature convergence to an almost deterministic policy, which no longer explores other strategies. A high value results in highly random policies, which can rarely match the final performance of solutions with lower entropy. The selected values for ε_{disc} and ε_{cont} furthermore depend on the respective experiment. Another option is to continuously anneal the two values over the course of learning instead of carefully choosing a fixed value. An initially high value ensures appropriate exploration at the beginning of learning, and the gradual decay towards smaller values lets the policy slowly reduce the amount of exploration and start to exploit what has already been learned.

Critic

The central idea to DDPG is to learn an action-value function that maps from an observation and an action to the action-value, and to use the gradient of this estimation to optimise the policy (see section 2.7.1 for more detail). In order to learn the value-function and ultimately the policy, the sampled action together with the observation is thus fed to another feed-forward NN that approximates the action-value function. In fact, instead of using a vector of the action and the raw observation, we preprocess the observation by a third NN, in order to obtain a more high-level representation (see figure 4.5). We denote the parameters of the action-value NN, including the preprocessing of the observation by ω. The value-NN then outputs a distribution over the action-value, as suggested in Barth-Maron et al., 2018.

The action-value distribution is implemented as a categorical distribution that predicts the probability of the action-value to fall in a particular, discrete bin. We therefore have to define a minimal and maximal value of the action-value distribution as well as a number of discrete bins. The used loss function is the Kl-Divergence of the old action-value distribution and the new distribution, which is obtained by projecting the obtained reward on the predicted distribution of the subsequent timestep. This can be understood as the distributional equivalent of minimising the Mean Squared TD-error. Projecting the obtained reward on the bootstrapped distribution can be done by simply moving every probability mass of the categorical distribution by the reward:

$$mass(Y^i) = \sum_j mass(\hat{Q}^j) \text{ if } (r + \gamma \hat{Q}^j = Y^i), \tag{4.4}$$

where $\hat{Q}^j$ denotes the value of the j^{th} bin of the categorical distribution, $mass(\hat{Q}^j)$ denotes the probability mass of the j^{th} bin and Y^i denotes the i^{th} bin of the projected distribution. Note that, in most cases, the projected value will not fall exactly on the value of a bin so that the probability mass has to be distributed accordingly (e.g. if the projection $r + \gamma \hat{Q}^j$ ends up exactly between the values of two bins, both bins should get half of the mass). Furthermore, the outermost bins will be accredited all mass that would be projected outside the boundaries of the categorical distribution.

Barth-Maron et al., 2018 also proposed to use n-step bootstrapping for learning the action-value function. Even though this makes the algorithm on-policy and therefore technically prohibits the use of a replay buffer, we observed that for small $n \leq 5$, n-step bootstrapping strongly improved convergence speed. As suggested in Mnih et al., 2015, we use target networks for the policy- as well as the value network (see section 2.5). This helps to mitigate the bias due to the correlation between the two action-values in the Bellman equation (equation 2.12). We also adopt the idea of Fujimoto et al., 2018, to use two, separately learned action-value functions and use the one that predicts the lower action-value in the Bellman update. This helps to overcome the maximisation bias and thus stabilises learning (see section 2.7.3). To update the parameters, we always use the predicted action-value function of the same NN, as was suggested in Fujimoto et al., 2018. However, using the smaller action-value function to update the policy parameters, as in Haarnoja et al., 2018a, turned out to work equally well. The loss of the action-value function results to:

$$\begin{aligned} J_Q &= \min_{\omega=\omega_1,\omega_2} D_{KL}\Big(Y \,||\, \hat{Q}_\omega(o_t, a_t)\Big), \\ \text{where } Y &= \Gamma\Big(\sum_{n=0}^{N-1} \gamma^n r_{i+n},\ \hat{Q}_{\omega'}(o_{t+N}, a \sim \pi_{\theta'}(o_{t+N}))\Big), \end{aligned} \tag{4.5}$$

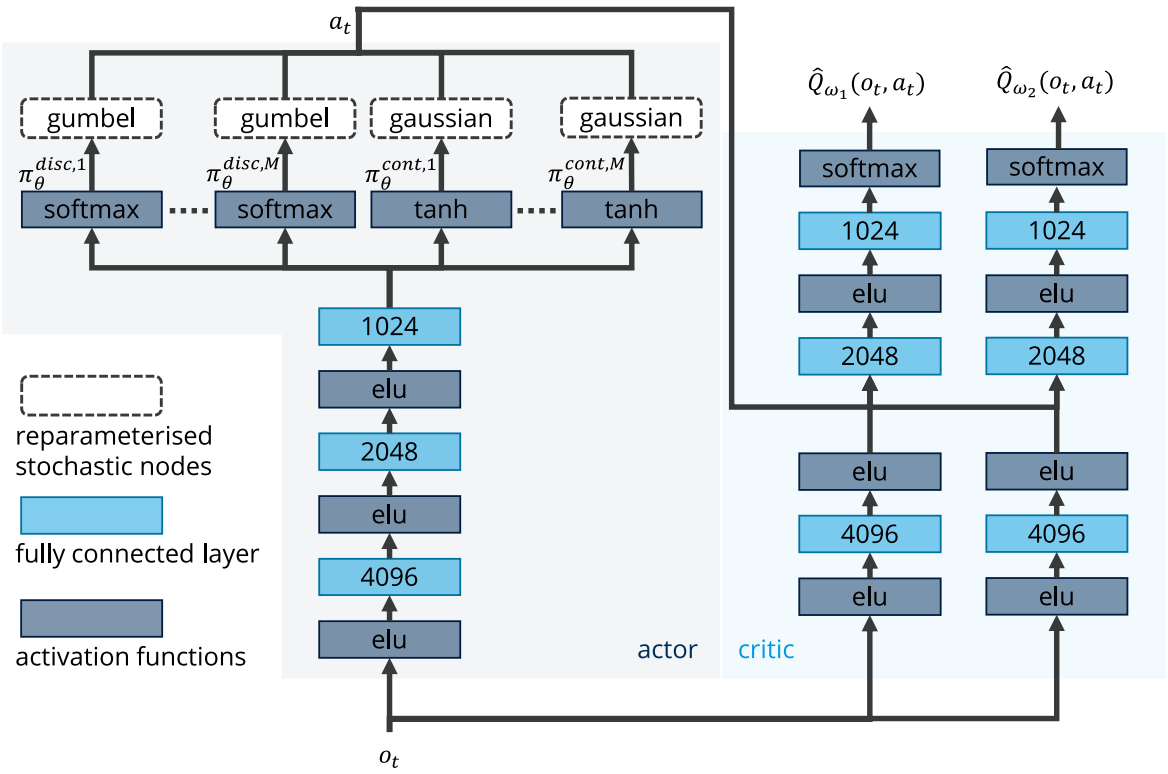


Figure 4.5.: Neural Network architecture of the learning algorithm. The policy network consists of three hidden layers with 4096, 2048 and 1024 nodes, respectively. Each of the two action-value function uses one layer of 4096 nodes for preprocessing the observation and another two layers of 2048 and 1024 nodes to compute the action-values. Note that we use two copies of the depicted network: one that is learned and the target network.

where ω_1 and ω_2 denote the parameters of the two action-value functions, ω' denotes the parameters of the respective target network, θ' denotes the parameters of the target policy network, Γ denotes the projection operation (equation 4.4) and $\min_{\omega=\omega_1,\omega_2}$ selects the distribution with the smaller mean value.

Being a microscopic simulator (see section 3.5), SUMO can be fairly slow when simulating the traffic network, particularly if many vehicles are considered. In order to speed up learning in terms of wall-clock time, we use a distributed sampling approach that collects experience from various environments in parallel (these can be simulated on different CPU cores). Other publications such as Barth-Maron et al., 2018 also take this distributed approach. Note that, other than Popov et al., 2017, we use multiple workers for the collection of experience but only one worker to compute gradients and optimise the parameters of the policy and the action-value function. Figure 4.5 depicts the used NN architecture, showing the number of neurons of each fully-connected layer (light blue) and all used activation functions (dark blue). Note that we do not show the target networks (the target networks are a full copy of the depicted networks).

Apart from the discussed adaptions to the original DDPG algorithm, we also tried out other reported improvements from section 2.7.3. The idea of Popov et al., 2017, to perform several steps of gradient descent for every interaction with the environment did only result in slower convergence in our experiments. The attempt to use a prioritised replay buffer, as in Barth-Maron et al., 2018, to sample transitions that are not yet well approximated by the action-value function approximator did not result in faster convergence and sometimes even led to diminished

performance. Adding zero-mean Gaussian noise to the bootstrapped action-values as in Fujimoto et al., 2018 did not show any effect in our experiments, and neither did the idea to update the policy parameters less frequently than the parameters of the action-value function (also proposed in Fujimoto et al., 2018).

Following the unspoken naming convention in the RL field of simply denoting some of the used concepts so that the first letters result in a more or less meaningful word (such as Actor-Critic using Knoecker-Factored Trust Region $\rightarrow$ ACKTR — pronounced actor — or Distributed Distributional Deterministic Policy Gradient $\rightarrow$ D4PG), we will name this algorithm 'Deep Distributed Distributional Discrete-Continuous Entropy Regularised N-Step Policy Gradients'. Successively shortening this unwieldy name we reach at DDDDCERNSPG $\rightarrow$ 4DCERNSPG $\rightarrow$ 4D7 (accounting for 4 D's and 7 other letters). Appendix A shows the full Agent4D7 algorithm.

4.5.2. Learning and Optimisation

We optimise all sets of parameters θ, ω_1 and ω_2 with an ADAM optimiser (see section 2.4.2). Due to recent concerns on the ability of ADAM to reliably find a good solution (Wilson et al., 2017), we also experimented with other optimisers such as SGD with momentum and the trust-region approach of ACKTR (Wu et al., 2017b). However, neither of these approaches showed superior final performance in our experiments and both converged significantly slower than the one using ADAM. All parameters are initialised using Kaiming initialisation (He et al., 2015b). The policy and action-value function NNs are trained with the two learning rates α_π and α_Q, respectively.

Finding an adequate learning rate for the optimiser of the policy turns out to be tricky. Low learning rates result in slow convergence whereas high ones show quick progress in the beginning but sometimes lead to a sudden, steep decrease in performance of the policy (often referred to as policy-breaking). This different learning behaviour was partly due to different rates of change of the discrete policy, depending on how well our action-value-NN approximates the true action-value function. In order to avoid this we implemented an adaptive learning rate that uses a simple proportional controller to keep the KL-Divergence between the policy before and the one after the update of the parameters θ on a predefined level. More precisely, we used the symmetric $D_2(A, B)$ metric that is the sum of the two KL-Divergences $D_{KL}(A||B)$ and $D_{KL}(B||A)$, where A and B are the two distributions. The learning rate can only be adapted within reasonable bounds. Note that the second-order approximation of the ADAM optimiser, combined with the adaption of the learning rate to reach a desired level of change in the policy, may be considered a very coarse approximation to a trust-region approach (see section 2.6.3). For optimising the action-value function, we simply use a fixed learning rate.

Appendix B.1 shows all parameter values of the learning algorithm that we used during our experiments.

4.6. Real-World RL Traffic Control

As a final remark in this chapter, we want to comment on the possibility of deploying the RL traffic controller in a real-world traffic network.

Due to the safety problems discussed in section 4.2, there are little to no instances of an RL agent being left in charge of the control of any safety-relevant system. The control of traffic could, however, very well be among the first domains of widespread deployment of RL agents. This is because the safety of the traffic system can be enforced easily, by constraining traffic lights to

admit simultaneous right of way only to compatible streams and to employ appropriate amber- and all red periods. Safety concerns in the control of traffic lights are therefore less pronounced.

Unfortunately, these safety constraints can not ensure efficient operation of the traffic system. In fact, the buildup of high congestion through an inefficient traffic policies could ultimately result in unsafe situations and accidents. In addition, traffic legislators could be unwilling to give up a working system when the replacing system would need a significant amount of time before it can efficiently operate the traffic network. These problems may be partially mitigated through the option of pre-learning a traffic agent in a simulated environment that closely resembles the true traffic scenario. After convergence in simulation, the system could be deployed to the real world, where it keeps on learning to further adapt to the traffic network and to mitigate possible inefficient behaviours that were learned because of inaccuracies or simplifications in the traffic simulation. An inefficient, initial exploratory phase of the system could thus be avoided.

It is important to note that the continuous adaption of a real-world RL traffic control system is only realistic for the communicating agent. Through the availability of rich knowledge on the state of individual vehicles, a meaningful reward function can be designed. The solitary agent, on the other hand, would have to rely on approximate rewards (e.g. from inductive loop sensors) which might not be able to give sufficiently rich feedback to learn a good policy. The availability of a reward function can thus be seen as one of the biggest advantages of V2I communication to RL approaches to the traffic control problem.

5. Experiments and Results

In this chapter, we want to investigate the behaviour and performance of our DRL approach to traffic light control (presented in chapter 4). To that end, we developed a set of experimental setups in a simulated environment in that the traffic signalling can be influenced by the RL agent. In each of these environments, we let an agent learn a traffic control policy, describe its emergent behaviour and asses its performance. Due to the already widely reported superiority of RL approaches over traditional control methods (see section 4.3), we do not attempt to exhaustively compare our algorithm against existing control methods (described in section 3.4). Instead, we will focus on the comparison of performance and emergent behaviours of agents that employ the two different observation-spaces (see section 4.4.1), embodying the availability of a V2I communication interface, or the lack thereof. Through that, we hope to showcase the great potential of V2I communication to improve the performance of modern traffic systems and the power of DRL approaches to handle the large, emerging observation-spaces. An exception to this is the first experimental scenario in that we compare the policies of the DRL agents against the optimal fixed cycle strategy.

We will first describe the elements of the simulation environment that are common to all experiments. Subsequently, we will give a detailed description of each experiment, including the specifics of the respective traffic network and the feature that we hope to showcase. A figure of the road infrastructure of every scenario can be found in the appendix. Following the description of each experimental setup, we will present and analyse the obtained results. Each conducted experiment increases the complexity of the former one. Starting from a single controlled intersection, we will work our way up to more complex settings that require the coordination of multiple intersections. Finally, we will briefly discuss the convergence properties of the learning algorithm.

5.1. Simulation Setup

The conducted experiments are all based on a common traffic framework. The implementation of our algorithm is written in Python 3.6 (Van Rossum and Drake Jr, 1995). We use SUMO (see section 3.5.1) as a microscopic simulation software for the traffic system and Flow (see section 3.5.2) as a Python interface that implements the traffic MDP, described in section 4.4. For the implementation of all Neural Networks (see section 4.5), we use PyTorch (Paszke et al., 2017). Other frameworks that were heavily made use of include NumPy (Walt et al., 2011), pandas

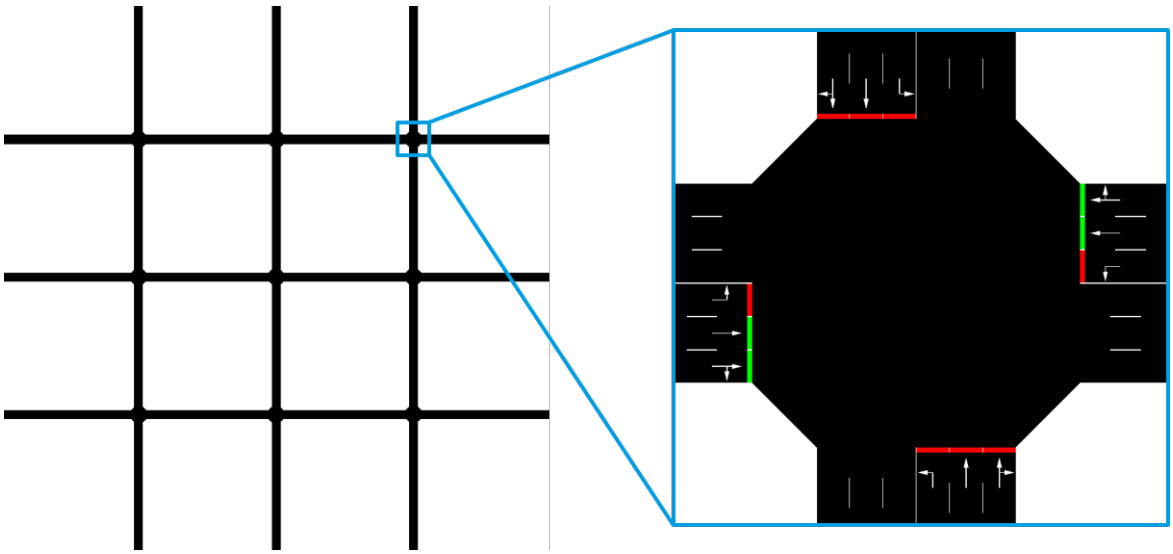


Figure 5.1.: The traffic network that we use in our simulations. Most experiments use a regular grid of intersections (not necessarily the depicted number) with interconnecting roads of six lanes each. Vehicles enter the network on one road and aim to leave it at another; at each intersection vehicles can go straight or turn left or right to reach their destination. Each intersection is controlled by a traffic light that can show eight distinct phases. Taken from SUMO's GUI.

(Mckinney, 2010), ptan (Lapan, 2018), IPython (Perez and Granger, 2007) and matplotlib (Hunter, 2007).

The traffic network in our simulations consists of a regular, two-dimensional $n \times m$ grid of intersections as depicted in figure 5.1. The dimensions of the grid vary from one experiment to the other. Each interconnecting lane has the same length and speed limit. Vehicles can be situated on one of three lanes per road and direction (six lanes per road), where the leftmost lane allows only left turns, the middle lane allows only straights and the rightmost lane allows both right turns and straights (see figure 5.1). Each intersection is actuated by a system of traffic lights that can show either of the eight phases depicted in figure 3.2, where each phase comprises of only compatible streams. Note that all streams that originate from the same lane (here this only concerns the rightmost lane of every road) always have simultaneous right of way; therefore, streams do not get blocked because another stream, using the same lane, does not have the right of way (for example, if in some phase we allow right turns but prohibit straights at the rightmost lane, a vehicle that intends to go straight would block the road for all subsequent vehicles that want to go right). For the last experiment (section 5.5), we will abandon this generic scenario and instead simulate the road network of the l'Antigua Esquerra de l'Eixample neighbourhood in Barcelona.

In each timestep, the RL agent can control the next phase as well as the phase time (within reasonable boundaries) of each intersection, as described in section 4.4.2. The selection of a new phase lets the intersection go through an amber and an all red period of predefined lengths, before the new phase can be shown. The duration of the amber period here approximately matches the legal requirements. In order to prevent accidents, which force us to reset the simulation, we here use all red periods that are slightly longer than usual.

Vehicles can enter or leave the traffic network at either of the outermost roads. The generation of new vehicles is done by one Poisson process for each combination of entry- and exit points. The total demand in terms of vehicles per hour can be controlled, as well as some biases of the

statistics of origin and destination such as the proportion of vehicles that leave the simulation on the same road that they entered it or the ratio of vertical to horizontal traffic. The lane on which a new vehicle is spawned is chosen randomly and the initial velocity of all vehicles is predefined. Note that the assumption of Poisson processes for vehicle generation is a debatable one. In a Poisson process, the generation of each vehicle is statistically independent of other vehicles. It generates every sequence of n vehicles in a fixed time-interval with the same probability, while the number of generated vehicles n is Poisson distributed (Dayan and Abbott, 2005). If the surroundings of the considered traffic systems can be assumed to consist of long streets with no traffic lights, vehicles might be dispersed enough to justify the assumption of Poisson processes. However, in the (maybe more realistic) scenario of the considered system being surrounded by other actuated intersections, the generation of individual vehicles should be considered to be correlated to account for the coordinated inlet of dense streams of vehicles at nearby intersections.

To route each vehicle through the network, we use SUMO's default routing protocol (see section 3.5.1), which employs a shortest path algorithm that takes into account the length and speed limit of the connecting roads as well as their current occupancy. Vehicles are routed once upon entering the simulation but do not adapt their initial route later on. Note that adaptive route selection might not be the standard in today's traffic network but could very well be the norm in future scenarios that feature V2X communications. The lane-changing behaviour is also governed by SUMO's default controller. Vehicles change lanes to reach the required lane for their next turn and can also adapt to the current traffic by choosing the lane with the shortest queue or the highest velocity. If a vehicle is not on the correct lane for its next turn but cannot change the lane because another vehicle is blocking it, both vehicles can adapt their velocities to allow the former to change lane. This also includes that vehicles may open up a spot in a long queue to let another vehicle enter the lane. The car-following model uses the Intelligent Driver Model (IDM) introduced in Treiber et al., 2000. The IDM dynamics are additionally perturbed by Gaussian acceleration noise as proposed in Wu et al., 2017a. This results in non-deterministic driving behaviour which more closely matches the real world than a fixed acceleration value.

Real-world traffic systems need to work in continuous time. In theory, the traffic networks should therefore be simulated continuously and the collected experience should not fall into individual episodes. However, a poor policy at the beginning of the learning process can result in strong congestion and deadlocks, which are hard to resolve. Framing the traffic simulation as an infinite horizon MDP therefore could slow or even prevent the agent from learning an efficient policy. For our experiments, the environment is therefore simulated for a predefined duration and is reset to its initial state at the end of each episode. Appendix B.2 shows the values of all parameters of the traffic environment.

In each experiment, one or several DRL models are trained to control traffic lights. As discussed in section 4.2, it is often difficult to judge whether a DRL algorithm has converged to the final solution or if it might further improve at its task. Furthermore, independant runs of the learning algorithm might yield different solutions, depending on the hyperparameter setting and the initialisation of the NN parameters. For the here-presented results, each learning process was run until the performance of the respective policy did no longer change for a significant amount of time. To mitigate the risk of having found an inferior solution, we ran every learning process several times, with a varying set of hyperparameters.

The raw data of all figures as well as videos of all experiments are included on the CD.

5.2. Single Intersection

In our first experimental setting, we will treat the easiest possible case of only a single controlled intersection. Figure C.1 in appendix C.1 shows the road infrastructure.

We will here investigate the performance of the learning traffic agent for a wide spectrum of different demand settings. The lowest considered demand of 200 vehicles per hour (vehs/h) can be considered to represent quiet traffic conditions during the early afternoon or on weekends, whereas the highest demand of 3000 vehs/h represents strongly oversaturated traffic conditions that might, for example, correspond to a rush hour in a dense city centre. All routes in this scenario have the same demand, meaning that every incoming vehicle turns left, turns right or goes straight with the same probability and the expected value of generated vehicles is equal for every ingoing lane. In addition to a comparison of the performance of the two agents (the solitary and the communicative agent), in this setting, we will also investigate how well the proposed RL approach compares to a fixed-cycle strategy.

5.2.1. Fixed-Cycle Strategy

If we assume a fixed phase scheme for the simple case of a single intersection, it is relatively easy to find a near-optimal signalling strategy. We will therefore compare the RL approach to a fixed-cycle strategy, which could be found by a traditional strategy such as MOVA (see section 3.4). Due to the regular inflow statistics on all lanes (the inflow statistics are completely symmetric for all ingoing lanes), it is obvious that a fixed-cycle controller should give the same phase time to each of the phases. A traffic controller therefore only has to find an appropriate phase time.

Results

Figure 5.2 shows the measured average velocities of vehicles for different phase times (ranging from 5 to 100 seconds), different demands (ranging from 200 to 3000 vehs/h) and for the two different phase schemes of figure 3.2. Each datapoint is the mean of 100 hours of simulated time which, for our simulation timestep of one second, corresponds to 360,000 datapoints. The depicted 95% confidence intervals are therefore barely visible. Note that we can only define phase times that are multiples of the simulation timestep.

Different demand scenarios are depicted in different colours. For each demand scenario, the average velocity first increases with the phase times, as a higher number of queued vehicles can pass the intersection in a single green period. The average velocity sharply increases every time that an additional vehicle can cross the intersection, followed by a slight decrease due to the fact that green times increase without additional vehicles passing the intersection (this leads to the zigzag form of the graphs). Each demand scenario shows an individual maximum of the average velocity. These maxima are reached when the number of vehicles that can cross the intersection in each phase coincides with the average number of vehicles that queue up at the respective approach during its red phase. A higher demand naturally results in a longer optimal phase time. Even longer phase times (than the one of maximal average velocity) result in green idling, during that the traffic light of an approach continues to show a green light even though no further vehicles are queued. After the respective maximum, the average velocity therefore decreases smoothly for each of the demand scenarios.

Solid lines show the results for the single street approach and doted ones for the opposing streets approach. For scenarios of lower demand, the performance of the two approaches

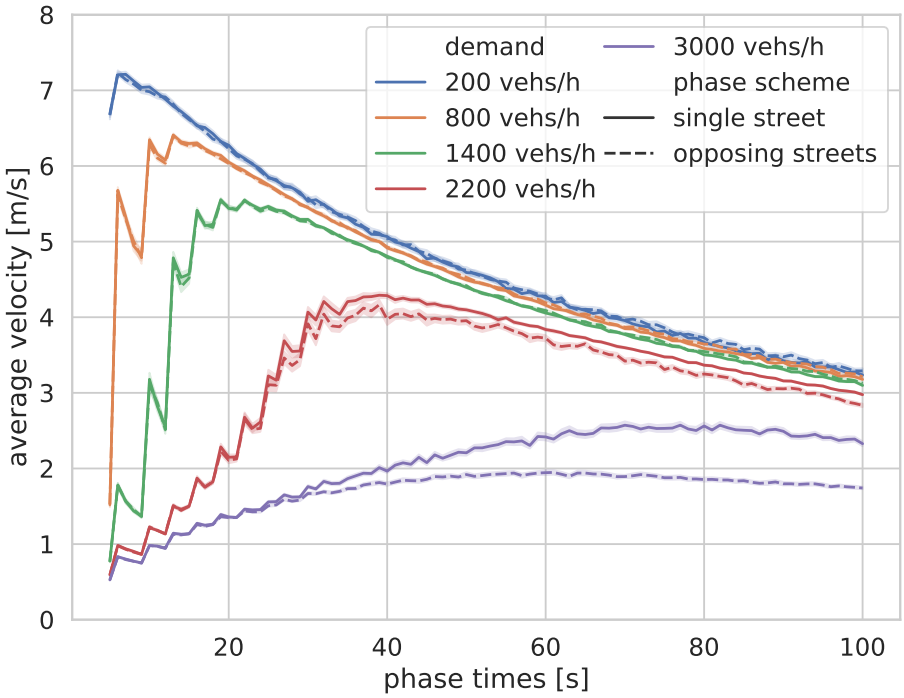


Figure 5.2.: Experimental results of a fixed-cycle strategy that controls the signalling at a single intersection. Shows the average velocities of vehicles for different phase times, demands and phase schemes. For higher demands, longer phase times yield better results. For low demands, both phase schemes show comparable results, whereas for higher demands, the single street approach outperforms the opposing streets approach.

appears to be identical. This makes sense since, in both approaches, all streets are given the same green time within the full cycle. For higher demands, it appears that the single street approach outperforms the opposing street approach. The reason for this is that in the opposing streets approach, some vehicles may be on the wrong lane for their next turn and thus block a lane that has the right of way, while waiting to be allowed to switch lanes (for example if a vehicle wants to turn left but is on the middle lane it will block the middle lane until it can switch to the left lane). It may be debatable whether this behaviour is realistic or if drivers would simply choose another route when they cannot switch lanes.

In the following experiments, we will use the phase times that resulted in the best performance and compare the results to a policy that is derived by a DRL agent. Note that it is unclear whether the fixed-cycle controller corresponds to a solution that may be found by an isolated or a responsive strategy. On the one hand, a responsive strategy might leverage the knowledge of individual vehicles that cross the inductive loop detectors to adapt its phase times appropriately. On the other hand, most traditional control strategies use traffic counts only to estimate arrival statistics but do not respond to the presence of individual vehicles. As the arrival statistics are constant in our experiments, one could argue that the resulting strategy incorporates this knowledge implicitly and therefore finds a solution that is comparable to the one that a responsive traffic controller may employ.

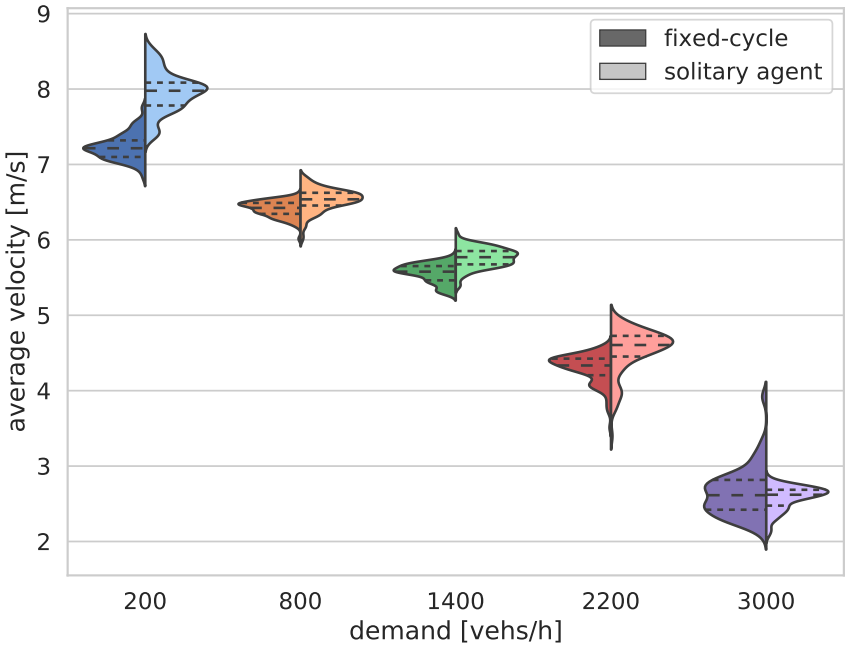


Figure 5.3.: Comparison of the distributions of average velocities of vehicles of the solitary agent and the fixed-cycle approach for different demands in the single intersection scenario. For all demand scenarios, the solitary agent appears to find a solution that performs on par with the fixed-cycle approach.

5.2.2. DRL: Solitary Agent

In the next step, we want to compare the results for the fixed-cycle control strategy to a solution that is obtained by our DRL agent. The goal of this experiment is to show that our agent can find a policy that performs on par with the fixed-cycle solution while using the same information. We here use the solitary agent (see section 4.4.1) as it has no access to information about individual vehicles. Since the fixed-cycle strategy of the previous section can be considered to be near-optimal, we do not expect the solitary agent to be able to outperform it.

Results

Figure 5.3 compares the distribution of average velocities of the best fixed-cycle strategy from the previous experiment to a solution found by the DRL agent. For each demand, the left, darker distribution corresponds to the best result from figure 5.2 (shown in the same colours). The right, lighter distribution shows the results of our DRL approach, using the solitary agent. Each distribution is approximated using 100 data points, where each data point corresponds to the average velocity of vehicles, measured over the timespan of one hour (each distribution contains data of 100 hours of simulated time). The dotted lines in the distributions show the respective medians and the upper and lower quartiles. Note that we scaled all distributions to the same height for better visibility. The spaces under the curves are therefore not equal.

Most pairs of distributions are relatively similar, which leads us to conclude that, at least for this simple example, our DRL approach is able to find a solution that is on par with a traditional control strategy. In the case of the lowest demand (200 vehs/h), the DRL solution, in fact, strongly outperforms the fixed-cycle strategy. We believe that this is mostly due to the DRL agent's ability to employ a sequence of varying phase lengths. For example, if during the red period of a stream,

an average of 2.5 vehicles queue up at the approach, it may be beneficial to iterate between two phase-times: one allowing two vehicles to cross and one allowing three vehicles to cross. In this example, the fixed-cycle strategy would need to select the phase time at which three vehicles can pass, resulting in some green idling, or else vehicles would keep on queuing up. This advantage is more pronounced for low demands.

For the case of very low demand (200 vehs/h), the resulting policy non-deterministically switches between the eight different phase options while using very short phase times. Every new phase tends to grant the right of way to flows that have gone longest without it. Since queues build up slowly, activating all streams in the shortest time possible appears to be the best strategy. For the slightly higher demand (800 vehs/h), the policy converged towards using only the single street approach in some trails, and ended up using only the opposing streets approach in other trials. The phase times remain relatively short, but the sequence of phases appears more deterministic than in the low demand case. This shows that both approaches (single street and opposing streets) result in similar rewards, but mixing the two tends to yield inferior performance. For higher demands (1400 to 3000 vehs/h), the resulting policy always ends up deterministically iterating through the phase options of the single street approach. This makes a lot of sense since the opposing streets approach has shown to be inferior for the higher demand setting (see figure 5.2). Phase times increase with higher demands to reduce intergreen times while avoiding green idling.

Particularly interesting is the emergent behaviour for the case of 3000 vehs/h, as two fundamentally different policies appear to result in the same average reward. For some trails, the agent periodically grants the right of way to all phases of the single street approach. For other trails, it ended up completely blocking one of the afferent roads. In the blocked road, queues build up until the entry point so that no further vehicles can enter the road network. On the one hand, this leads to a fixed number of standing vehicles, which negatively influence the average velocity. On the other hand, blocking one of the entry points reduces the demand from 3000 vehs/h to merely 2250 vehs/h, allowing the agent to keep the queues at the remaining three roads relatively short. It is interesting to see that these two policies result in very similar rewards, making them equally good in the eyes of the agent, whereas a human observer would probably condemn the latter strategy for being unfair. Even though this behaviour is exploiting a shortcoming of our simulation and would not be possible in the real world, it shows that we have to design the reward function with care as the agent might find a way to gain high rewards by behaving in a way that we may not have anticipated and that might not be desirable. We will take a closer look on different reward functions in section 5.4.

5.2.3. DRL: Communicative Agent

As a last experiment in the single intersection scenario, we want to compare the performance of the two different agents (see section 4.4.1) that represent the availability of information about individual vehicles through V2I communication, or the lack thereof. In addition to the results of the preceding section, we thus let our communicative agent learn the control of the traffic system. The agent here can incorporate the knowledge of up to 30 vehicles per incoming road (120 vehicles per intersection).

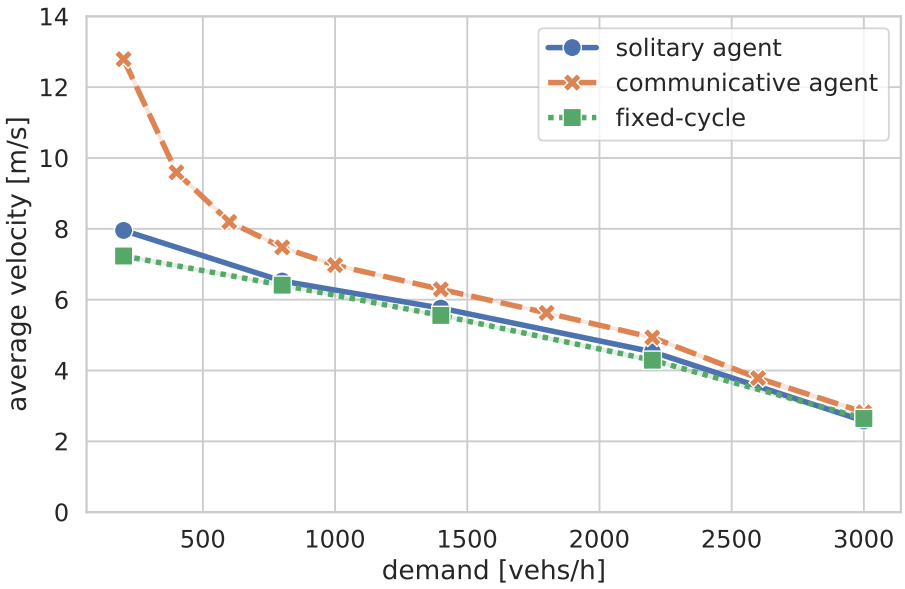


Figure 5.4.: Comparison the average velocities of vehicles for the communicative agent, the solitary agent and the fixed-cycle strategy for different demands in the single intersection scenario. The communicative agent reliably outperforms the other approaches. The advantage of V2I is especially pronounced for lower demands.

Results

Figure 5.4 shows the average velocities of vehicles for different demands. Results are shown for the fixed cycle approach (see section 5.2.1), the solitary agent (see section 5.2.2) and the communicative agent. Markers denote the measured average velocities, whereas lines merely connect the markers. Each data point shows the average velocity of 100 hours of data (corresponds to 360,000 data points). The depicted 95 % confidence intervals are therefore barely visible. Note that the fixed cycle approach and the solitary agent each require independent optimisation for each of the different demands. Every data point of the solitary agent therefore is obtained with a different, independently trained policy. The communicative agent, on the other hand, simply uses one model for all different demand settings. Whereas it is trained on the same demands as the other agent (200, 800, 1400, 2200 and 3000 vehs/h), we chose to also evaluate the learned model for other demands (400, 600, 1000, 1800 and 2600 vehs/h) and found it to perform equally well. This shows that the communicative agent is able to efficiently navigate different demands without explicitly having to estimate traffic statistics. In a real-world setting, where demands are not steady and hard to quantify accurately, the communicative agent might therefore outperform the other approaches by an even larger margin, due to its ability to successfully manage a wide range of different demands with a single model. In section 5.3, we will further investigate the communicative agent's ability to adapt to changing demands.

Table 5.1 shows the exact and relative values of the average velocities of all three approaches for different demands. Comparing the results of the three different approaches, we again notice that the fixed cycle approach and the solitary agent show comparable performance (see section 5.2.1). The communicating agent, on the other hand, shows superior performance for all demands. For the low demand setting, the communicative agent significantly outperforms the other approaches, whereas, for higher demands, the advantage of communication becomes negligibly small. The strong advantage for the low demand case results from the strategy of the

demand	fixed-cycle	solitary agent	communicative agent
$200\ vehs/h$	$7.23\ \frac{m}{s}\ (100\ \%)$	$7.95\ \frac{m}{s}\ (110\ \%)$	$12.79\ \frac{m}{s}\ (177\ \%)$
$800\ vehs/h$	$6.41\ \frac{m}{s}\ (100\ \%)$	$6.53\ \frac{m}{s}\ (102\ \%)$	$7.48\ \frac{m}{s}\ (117\ \%)$
$1400\ vehs/h$	$5.55\ \frac{m}{s}\ (100\ \%)$	$5.76\ \frac{m}{s}\ (104\ \%)$	$6.29\ \frac{m}{s}\ (113\ \%)$
$2200\ vehs/h$	$4.29\ \frac{m}{s}\ (100\ \%)$	$4.53\ \frac{m}{s}\ (106\ \%)$	$4.93\ \frac{m}{s}\ (115\ \%)$
$3000\ vehs/h$	$2.64\ \frac{m}{s}\ (100\ \%)$	$2.57\ \frac{m}{s}\ (97\ \%)$	$2.82\ \frac{m}{s}\ (107\ \%)$

Table 5.1.: Experimental results for the two agents in the single intersection scenario. Shows the exact and relative values of the average velocities for all three approaches for different demands.

communicative agent to individually grant the right of way to approaching vehicles. Furthermore, it can decide to keep a phase when it observes another vehicle that is quickly approaching the intersection. In contrast, the solitary agent blindly activates phases that have not been shown for a long time, often granting the right of way to streams with no queued vehicles or forcing a fast vehicle to break just before the intersection. In settings of higher demand, queues build up quickly so that switching to a phase that has not been shown for a while almost certainly results in the reduction of a queue. However, the communicative agent still manages to retain a slight edge over the other approaches by ending a phase when most lanes of that phase have been cleared.

5.3. Arterial Road

In this section, we want to investigate the effect of V2I communication in a traffic setting that requires the coordination of multiple intersections. We here consider an arterial traffic scenario that consists of a chain of four connected intersections, forming one long, horizontal road which connects to four shorter, vertical roads. Figure C.3 in appendix C.2 shows the road infrastructure. The horizontal road in this scenario is considered to be a heavily utilised main road, whereas the vertical roads are supposed to represent side roads with smaller traffic demand. We therefore need to change the spawn rates of the Poisson processes to account for increased traffic demand on the main road and decreased traffic demand on the side roads. To do this, we increase the rate of vehicles being spawned on routes that originate on the main road or end on the main road by a factor of five and the rate of vehicles being spawned on the route that both begins and ends on the main road by a factor of 25. Consequently, many vehicles traverse the entire main road, a moderate amount of vehicles enter the simulation on the main road but leave it on a side road or vice versa, and relatively few vehicles both enter and leave the simulation on a side road. Note that we have to normalise the resulting spawn rates to meet the desired demand.

5.3.1. Steady Demand

For this experiment, we compare the two agents for different demand scenarios, ranging from low demand (200 vehs/h) to moderately high demand (2000 vehs/h). As the number of intersections is larger than in the previous experiment, resulting in a larger observation-space, we here reduced the number of observed vehicles of the communicative agent from 30 to 10 (40 per intersection).

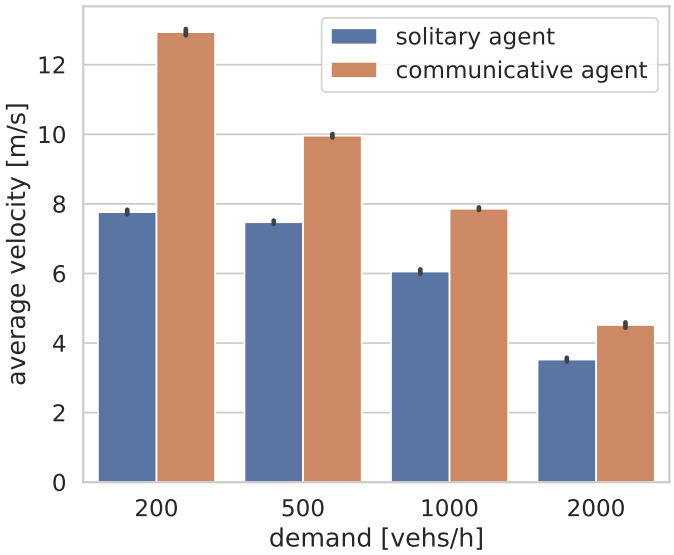


Figure 5.5.: Comparison of the average velocities of vehicles for the communicative- and the solitary agent for different demands in the arterial scenario. The communicative agent surpasses its solitary counterpart for all demand settings.

Results

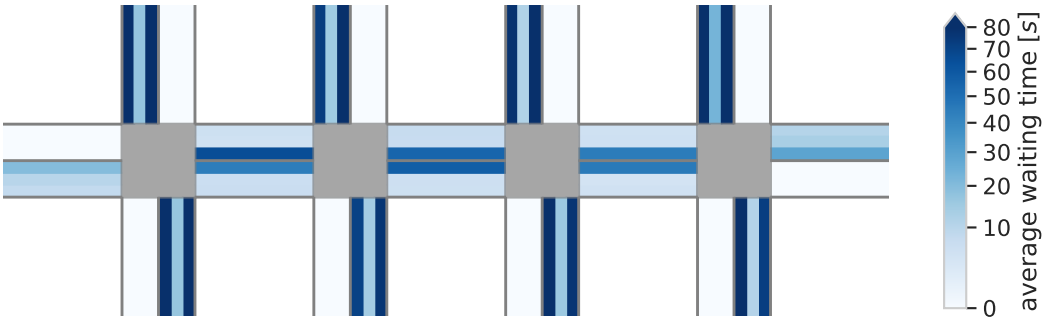
Figure 5.5 compares the measured average velocities for the two agents and different demand scenarios. Each of the bars is obtained by averaging over 100 hours of simulated time. The small candlewicks depict the 95 % confidence intervals. Note that, just as in the previous experiment, four different solitary agents were trained — one for each of the four demand scenarios. In contrast, only one communicative agent was trained in an environment with varying demand and later evaluated in a fixed-demand scenario.

As in the previous experiment, we observe that the communicative agent reliably outperforms the solitary agent by a significant margin, where the difference is especially prominent for scenarios of very low demand. Table 5.2 shows the obtained average velocities of the two agents and their relative values.

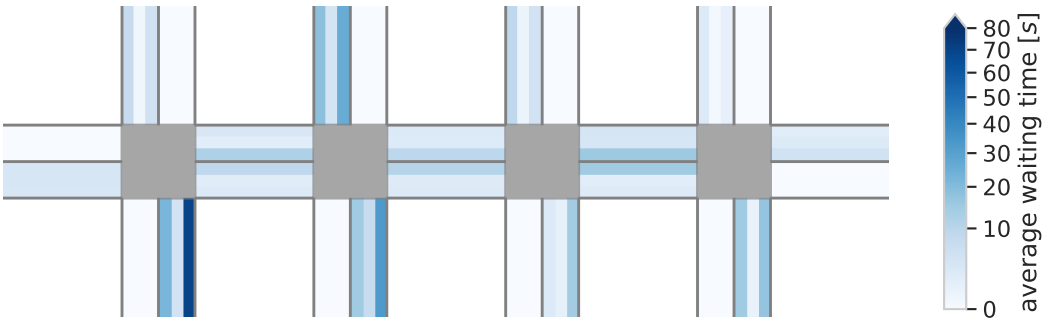
demand	solitary agent	communicative agent	relative value
$200\ vehs/h$	$7.76\ \frac{m}{s}$	$12.94\ \frac{m}{s}$	$167\ \%$
$500\ vehs/h$	$7.47\ \frac{m}{s}$	$9.96\ \frac{m}{s}$	$133\ \%$
$1000\ vehs/h$	$6.06\ \frac{m}{s}$	$7.86\ \frac{m}{s}$	$130\ \%$
$2000\ vehs/h$	$3.53\ \frac{m}{s}$	$4.52\ \frac{m}{s}$	$128\ \%$

Table 5.2.: Experimental results for the two agents in the arterial scenario. Shows the exact and relative values of the average velocities for the two different agents for different demands.

The performance of the solitary agent barely changes when increasing the demand from 200 to 500 vehs/h, whereas the performance of the communicative agent sharply decreases. For higher demands, the relative performance of the two agents is relatively stable. In the 200 vehs/h scenario, the substantial advantage of the communicative agent stems from its ability to react to individual vehicles by granting the right of way to the respective streams just in time for the



(a) Solitary agent.



(b) Communicative agent.

Figure 5.6.: Average waiting times for every lane in the arterial scenario for the two different agents and a demand of 200 vehs/h. The communicative agent strongly reduces waiting times of vehicles that enter or leave the main road.

vehicles to cross the intersection without having to wait. In scenarios of higher demands, the communicative agent still outperforms the solitary agent through a more intelligent allocation of green times, based on the number of queued vehicles. However, its advantage decreases for higher demands.

In contrast to the previous experiment, the inflows here are not symmetric. The learned policy may therefore also be asymmetric, and the traffic situation in different parts of the simulation may vary. To investigate these differences, in figure 5.6 we show the average time that a vehicle spends waiting (at a velocity of less than $1\ \frac{m}{s}$) on each lane of the traffic infrastructure in the 200 vehs/h demand scenario for the solitary agent and the communicative agent. Importantly, we did not explicitly optimise the waiting times but may still use it as a means to evaluate the behaviour of the two agents, as it strongly correlates with the average velocities. When comparing the waiting times on individual lanes, we have to bear in mind that the traffic volumes on the respective lanes strongly differ. For example, a long waiting time on a lane which is barely ever used can be acceptable, while an intermediate waiting time on a heavily used lane can be very problematic in the eyes of an agent that optimises average velocity.

Naturally, vehicles do not have to wait on any of the outgoing lanes in our simulation since we do not consider them to be controlled by traffic lights and vehicles can leave the simulation at any velocity. Note, of course, that this isolated view of our traffic environment is a simplification of a real-world traffic scenario, where inflow and outflow is strongly influenced by the surrounding infrastructure.

The solitary agent (figure 5.6a) grants most of the green time to the straight and right-turn

streams of the main road, resulting in relatively short waiting times on the main road. It creates green waves so that plateaus of vehicles can traverse the main road without having to stop. In contrast, streams that enter or leave the main road tend to have to wait a considerable amount of time at a red light (except for the right turns, leaving the main road). This makes a lot of sense since the heavily utilised main road has a stronger impact on the average velocity (which is what the agent tries to optimise) of the system than the quiet side roads.

Comparing the two figures (5.6a and 5.6b), we observe that the communicative agent further reduces the average waiting times on the main road by a few seconds. However, the more noticeable change concerns the waiting times for entering or leaving the main road, which are reduced by up to 90 %. The ability to observe individual vehicles here allows the communicative agent to let vehicles enter or leave the main road, without interrupting the traffic flow. We conclude that the advantage of the communicative agent is especially prominent if the scenario features streams with very low arrival rates, where it can respond efficiently to individual vehicles. Note that not all lanes show the same decline in average waiting times. In particular, the rightmost lane of the southern approach of the leftmost intersection appears to benefit very little from the knowledge of the communicative agent. This shows that the solution that is found by the DRL agent not necessarily is the optimal one. It is possible that the agent would have learned to successfully control all approaches if we would not have ended the training process.

Figure C.4 in appendix C.2 shows the waiting times of each lane for the communicative agent in the 500- and the 1000 vehs/h scenario. For higher demand scenarios, the average waiting times increase strongly on the lanes that are leaving or entering the main road, whereas the waiting times on the main road itself are kept low. This shows that the agent prioritises the traffic on the main road to deal with the increased traffic volume.

5.3.2. Sudden Inflow

In this experiment, we want to investigate the effect of unsteady traffic demands. More precisely, we want to see how the two agents handle a sudden, unexpected increase in traffic volume. We simulate the arterial scenario for a total time of two hours, where the first 30 minutes we use a moderate demand of 1000 vehs/h. We then increase the demand for 30 minutes to 2000 vehs/h and finally reduce it back to 1000 vehs/h for another hour. This experiment could, for example, correspond to the heavily increased traffic volume after an important sports event.

Results

Figure 5.7 shows the average velocities of vehicles over the course of the experiment. The depicted plots show the average values for 100 trails (in total, 200 hours of simulated time for each agent). To further reduce the noise in the graphs, we smoothed them with a Gaussian kernel. The depicted bounds represent the 95 % confidence intervals. We here use the same learned communicative agent as in the previous experiment. For the solitary agent, we use the best of the four trained models from the last experiment (the model that was trained in the 1000 vehs/h scenario performed best).

As we might expect, the ability of the communicative agent to observe the sudden increase in traffic volume allows it to reliably outperform the solitary agent. After the demand returns to the moderate level (1000 vehs/h), the communicative agent recovers quickly and returns to its former performance after approximately 20 minutes. The solitary agent, on the other hand, recovers slowly and does not return to its former performance within the remaining hour of

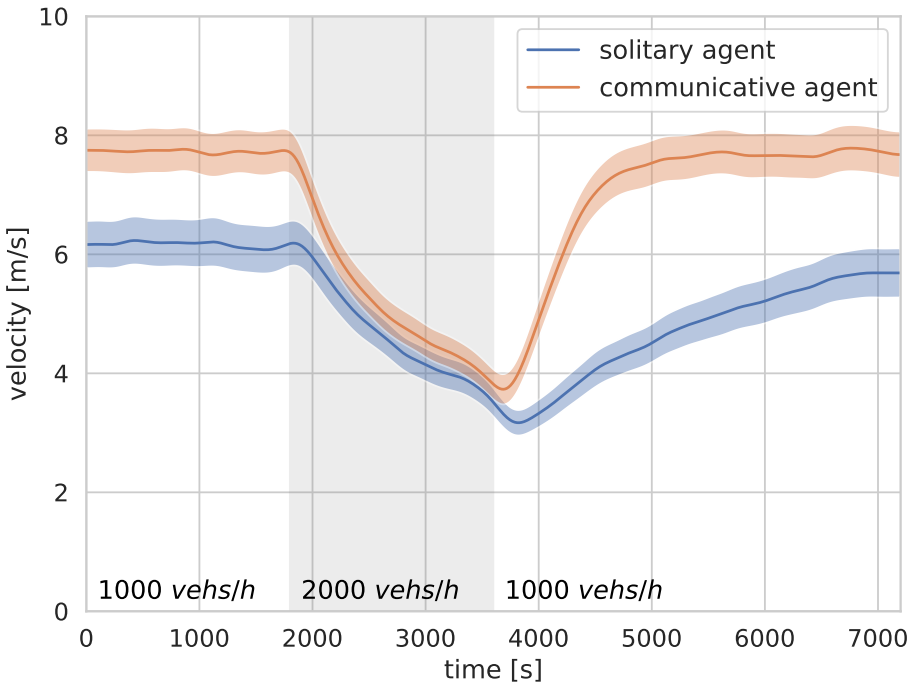


Figure 5.7.: Average velocities of vehicles for a scenario in that the demand is suddenly doubled for 30 minutes and then reduced back to its former value. The communicative agent reliably outperforms the solitary one and recovers significantly faster from the unexpected inflow.

the simulation. This result shows, maybe unsurprisingly, that the communicative agent has a significant advantage over the solitary agent when demands are unsteady. Note, however, that in this particular setting, the solitary agent could probably benefit greatly from loop-detector data.

5.4. Grid

Having scaled up the traffic infrastructure horizontally (from a single intersection to a sequence of intersections), we now also increase the size of the infrastructure in the vertical dimension. In this scenario we will use a grid of 3×3 intersections as depicted in figure C.6 in appendix C.3.

5.4.1. Destination Bias

In contrast to the previous experiments, we here do not vary the demand but consider a constant, moderate traffic inflow of a 1000 vehs/h. Instead, we will alter the origin-destination matrix to investigate the effect of different statistics of the routes, on which vehicles traverse the network. For the single intersection experiments, we assumed that the inflow at each of the ingoing roads is equal and that vehicles are uniformly routed to any of the outgoing roads (except of course the one that they entered on). In the arterial experiments, vehicles were more likely to enter and leave the simulation on the busy main road than on the side roads. In this scenario, we keep the uniform inflows of the single intersection scenario but change the destination probabilities as to make vehicles more likely to stay on the road on that they enter the traffic network. We will call this alteration in the destination statistics the 'destination bias'. A larger destination

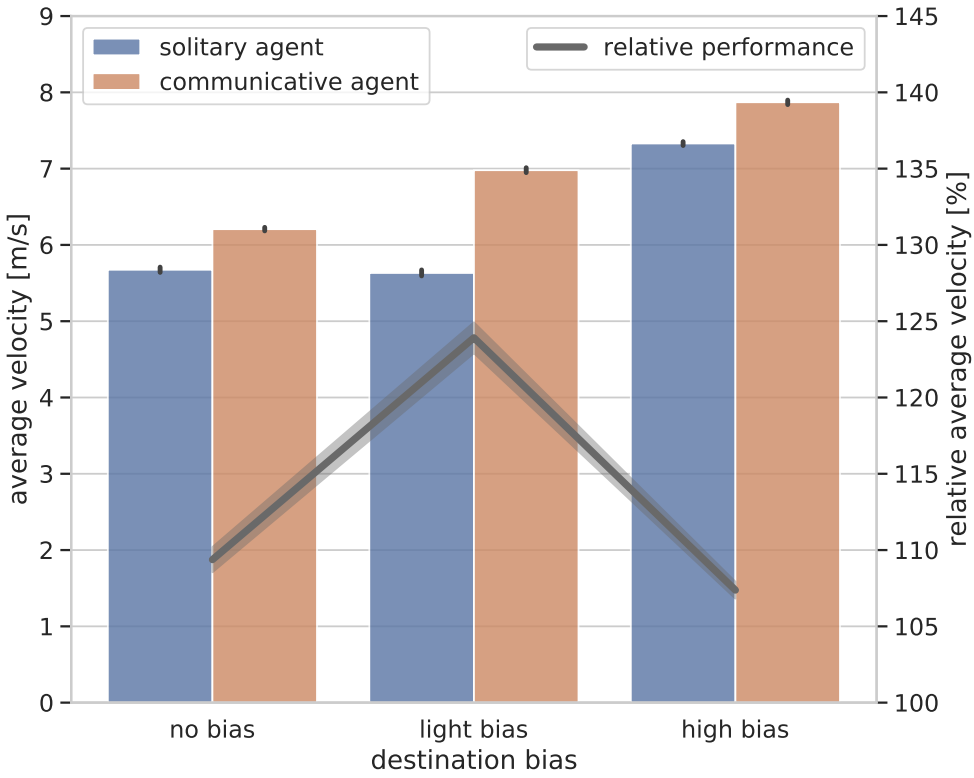


Figure 5.8.: Average velocities of vehicles for different settings of the destination bias. The bars depict the absolute results of the two agents and the line shows their relative performance. The benefit of V2I communication is strongest for the light bias setting.

bias increases the number of vehicles that traverse the entire grid traffic network without ever turning.

Results

Figure 5.8 shows the average velocities for the solitary and the communicative agent and for different strengths of the destination bias. In addition, it shows the relative performances of the two agents. The 'no bias' setting here assumes uniform destination statistics. At each intersection, vehicles take left turns, right turns and go straight with relatively high probabilities. In the 'light bias' setting, an increased amount of vehicles stay on the same road and never turn left or right. We here let approximately 70 % of vehicles stay on the road they entered on, and 30 % are routed uniformly to the other outgoing roads. At each intersection, the number of vehicles that turn left or right is decreased, whereas the number of vehicles that go straight is increased. Finally, in the 'high bias' settings, almost all vehicles traverse the simulation without ever turning. Left and right turns are therefore virtually never used. Note that the light bias setting is arguably the most realistic one if we consider the grid scenario to be part of a bigger traffic network.

As in previous experiments, we again observe that the communicative agent consistently outperforms the solitary agent. More interesting than the absolute performance of the individual agents, is here their relative performance. The advantage of the communicative agent over its solitary counterpart is significantly higher for the light bias setting than for the other two settings. In the light bias setting, the relatively rare event of a vehicle wanting to take a left turn at an intersection can be handled more efficiently by the communicative agent through its knowledge

about individual vehicles. In the no bias setting, the frequency of arriving vehicles that want to turn left is significantly higher, and in the high bias setting it is significantly lower. For higher arrival rates at a stream, the probability of vehicles queuing up during the red phase of that stream increases. The risk of granting the right of way to a stream at which no vehicles are waiting therefore decreases. The advantage of the communicative agent hence shrinks when all streams at an intersection have high arrival rates.

We conclude that the benefit of V2I communication is particularly prominent if the traffic scenario includes streams with low arrival probabilities. In case of a vehicle from a low probability stream arriving at an intersection, the communicative agent can allow it to pass the intersection 'on-demand'. Therefore, the communicative agent can simultaneously reduce waiting times of individual vehicles and prevent the loss of throughput due to phases being activated unnecessarily. On the other hand, streams with high arrival probabilities are well described by their statistics, and the communicative agent can improve only slightly over its solitary counterpart, by choosing more adequate phase times.

5.4.2. Composite Reward Functions

After having thoroughly investigated the influence of different traffic settings, we now want to examine the effect of different reward functions on the performance of the DRL traffic control system. As described in section 4.4.3, we will compare the results of using only the average velocity of vehicles as reward (like we have done until now) and using a composite reward function. The composite reward here comprises of: the average velocity of vehicles, the flow rate, the CO_2 emissions (the SUMO emissions model assumes Euro-4 gasoline standards) and the average quadratic amount of time that was spent waiting during the last 100 seconds (which, according to Liu et al., 2017, loosely corresponds to driver patience). The four parts of the reward function are weighted equally (all measures are normalised between 0 and 0.25). Note of course that we want to maximise the average velocity and flow rate and minimise the CO_2 emissions and stress levels. The reward function therefore increases for lower CO_2 and stress levels. The performance of both systems will be assessed, based on the resulting values of all of the four elements of the composite reward function.

Results

Figure 5.9 shows the average velocities, the flow rates, the CO_2 emissions and the average stress of drivers for the two reward functions. For each reward function, we show histograms of the obtained data (the diagonal of the figure) and a projection of the data points on each of the pairwise planes. We also use linear regression to show the respective pairwise correlations of the four performance indicators. Each of the 100 data points here is the mean value of one hour of simulated time. Table 5.3, shows the mean values of the four performance measures for the two agents as well as their relative values.

The agent that is optimising the composite reward function performs on par or outperforms the other agent in terms of all of the four measures. For the composite reward function, the agent appears to find a control strategy that leads to fewer full stops of vehicles than for the velocity reward function. This results in a slightly higher flow rate, a significantly lower average stress and slightly lower CO_2 emissions. Interestingly, the composite reward even leads to a slightly increased average velocity. Note that, since there is no guarantee for the DRL agent to find the optimal policy, it is possible for an agent that optimises the composite reward to find a

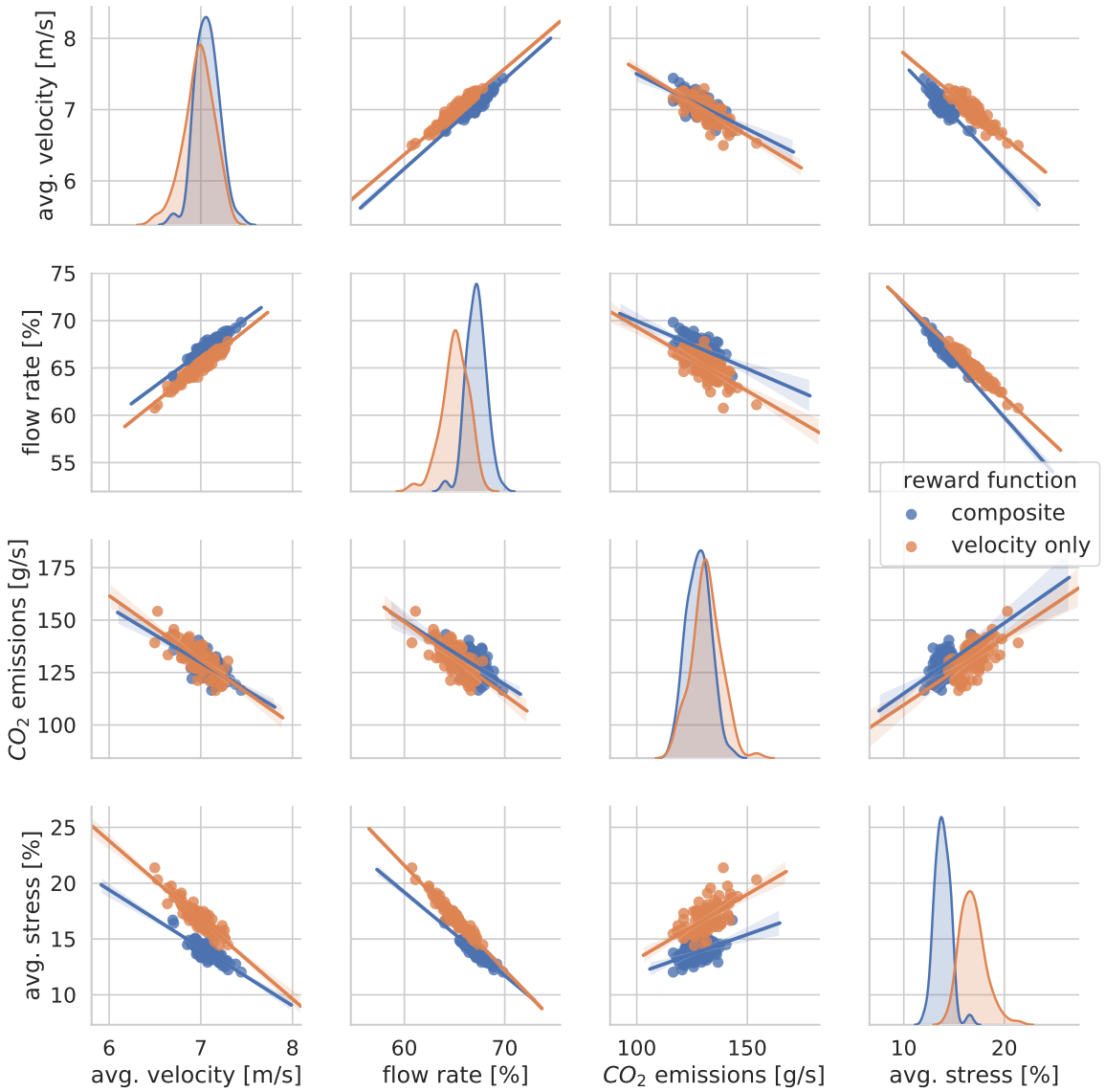


Figure 5.9.: Comparison of the average velocities, the flow rates, the CO_2 emissions and the average stress of drivers for two different reward functions. The figure shows both the distributions of each performance measure as well as their pairwise correlation. One reward function features only the average velocity (as in all previous experiments). For the other function, the agent optimises a uniformly weighted sum of the displayed variables. The composite reward results in a policy that equals or exceeds the result of the velocity reward in terms of all performance measures.

solution with higher average velocity than if it were to optimise the velocity exclusively. However, the difference in average velocity for the two different reward functions is negligibly small so that we would consider the two policies to result in equal average velocities. In a further experiment, reported in figure C.9 in appendix C.4, in which we repeated the reward function experiment for a different traffic scenario, the resulting policy for the composite reward slightly improves the flow rate, CO_2 emissions and average stress, but resulted in slightly lower average velocities. We conclude that our DRL approach can simultaneously optimise a wide variety of different reward functions, and to successfully trade-off a variety of — possibly contradicting — objectives.

Analysing the leftmost column of figure 5.9, reveals that the use of a composite reward function results in a slight decorrelation of the performance measures (manifesting in steeper

reward function	avg. velocity	flow rate	CO_2 emissions	avg. stress
composite	$7.07\ \frac{m}{s}$	$67.14\ \%$	$127.99\ \frac{g}{s}$	$13.86\ \%$
velocity only	$6.98\ \frac{m}{s}$	$65.06\ \%$	$131.52\ \frac{g}{s}$	$16.85\ \%$
relative value	$101\ \%$	$103\ \%$	$97\ \%$	$82\ \%$

Table 5.3.: Absolute and relative results of the two different reward functions in the grid scenario. The optimisation of the composite reward results in similar or improved performance in terms of all of the reported performance measures.

slopes of the linear regression models for the velocity reward function). It is remarkable that the velocity reward function yields similar results in all four performance measures (except maybe the average stress), despite not explicitly optimising the latter three. The reason for that, clearly is the strong correlation of the respective measures, as shown in the non-diagonal elements in figure 5.9. The agent can thus improve in terms of all performance indicators, despite maximising only one of them. We would therefore argue that the use of composite reward functions, which have been utilised in many publications (see section 4.3), is not necessary if all parts of the reward function show strong positive correlations. It would be interesting to combine antagonistic reward functions, forcing the agent to trade off different objectives. Unfortunately, we did not find any meaningful antagonistic rewards.

In figure C.7 in appendix C.3, we show bar diagrams of the mean value of the respective performance measures, as well as the confidence intervals. We also include the average time that a vehicle takes for traversing the traffic network from its origin to its destination and the average time that a vehicle spends waiting while traversing the network. These two additional measures make interesting performance indicators but are not well suited to be used as a reward function. The composite reward function results in a slight decrease in the average waiting time and trip time because of a strong correlation with the flow rate and the average stress.

5.5. L'Antiga Esquerra de l'Eixample

In our final experiment, we want to apply the DRL traffic control algorithm to a more realistic traffic network. As described in section 3.5.1, SUMO comes with an application that allows us to import road networks from OSM data. We here choose a part of the 'L'Antiga Esquerra de l'Eixample' neighbourhood, in the centre of Barcelona, as an example scenario. Figure C.8 in appendix C.4 shows the location of the neighbourhood on a city map and the extracted road network. The zone is characterised by a grid pattern of straight roads, most of which allowing only one direction of travel. It consists mostly of residential buildings or smaller offices, with the exception of the large 'Hospital Clínic' hospital on the west end of our road network.

Every intersection in the road network is controlled by a traffic light system. As all streets in our simulation only allow for one direction of travel, phase schemes mostly consist of just two different phases. We could, therefore, use a DRL agent that only controls the phase times of the traffic lights. However, for the purpose of generality, we choose to use the action-space, described in section 4.4.2 but limit the available phase options to the ones provided by the OSM data. Since the physical dimensions of intersections are shorter and allowed velocities are smaller than in the previous experiments, we reduce the duration of the amber period to 4 seconds and the all-red period to 2 seconds. Note that we do not explicitly account for

pedestrians or cyclist traffic.

The dense population and central location of the area give rise to busy commuter traffic, especially in the morning and afternoon. We here choose to run our experiments for three different demand levels, ranging from a moderate 1000 vehs/h to a heavy 3000 vehs/h. We empirically find these values to be on a realistic scale. Vehicles are spawned with equal probability on every ingoing lane of the road network so that wider streets have higher inflow. Similarly, the destination of every new vehicle is drawn from a distribution with probabilities proportional to the number of lanes of the outgoing road. The limitation to only a small patch of the road network gives rise to some unnatural routes and traffic volumes. For example, a vehicle may need to drive in a U-shape, if its destination is the neighbouring parallel street of its origin. In this case, most drivers would, of course, take a route that never enters our simulated road network. However, since we are more concerned with the ability of our agents to handle any traffic situation than with the realism of the simulation, we choose to allow these somewhat unnatural routes.

Note that all aspects of the experimental setting are designed to provide a plug-and-play architecture that is unspecific to the particular road network. We could, therefore, easily interchange the road network.

The previous experiment showed that our DRL method is capable of optimising an aggregated reward function. However, we struggled to design an aggregated reward function that accurately quantifies our qualitative objectives. Figure C.9 in appendix C.4 compares the results for the two different reward functions used in the previous experiment. Even though the composite reward resulted in a policy that outperformed the velocity only reward's policy in three of the four performance measures, visual inspection revealed that the latter policy better matched our qualitative objectives. We therefore choose to optimise only the velocity but assess the system's performance in terms of the average velocity, the flow rate, the CO_2 emissions, the average stress levels, the average trip time and the average waiting time of vehicles.

Results

Figure 5.9 shows the average velocities, the flow rates, the CO_2 emissions, the average stress of drivers, the average trip time and the average time spent waiting at intersections for the two agents in the L'Antigua Esquerra de l'Eixample setting. Each bar shows the respective mean value of 100 hours of simulated time.

The communicative agent significantly outperforms the solitary agent in terms of all six measures for all demand scenarios (note that, for the average velocity and the flow rate, a higher value is desirable whereas, for CO_2 emissions, average stress, average trip time and average waiting time, a lower value is better). As in previous experiments, the benefit of V2I communication slightly decreases as demand increases; however, the difference appears to be minute. For both agents, the policy creates green waves for vehicles on busy routes while letting vehicles enter from less important routes in between waves (note that, due to our origin-destination distributions, the busiest routes do not necessarily coincide with the actual main roads). Visual inspection of the simulated environment shows that many vehicles traverse the network without ever stopping. The availability of information about individual vehicles enables the agent to choose more accurate phase times and to better handle streams with low arrival rates. Table 5.4 shows the exact values of all measures as well as their relative values.

For all three demand scenarios, the communicative agent manages to allocate green time more efficiently and to reduce waiting times by approximately 50 %. The lower waiting times

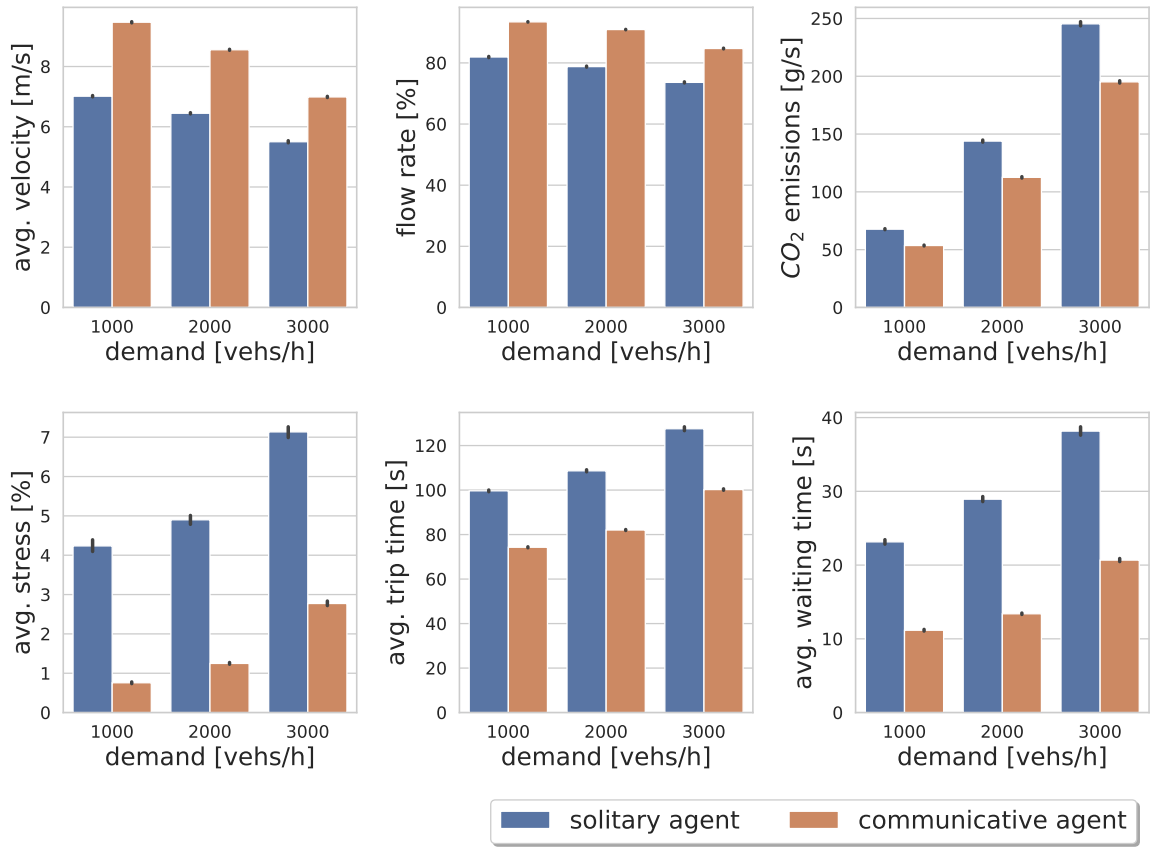


Figure 5.10.: Comparison of the average velocities, the flow rates, the CO_2 emissions and the average stress of drivers ,the average trip time and the average time spent waiting at intersections for the two agents in the L'Antigua Esquerra de l'Eixample setting. The communicative agent consistently outperforms the solitary agent in terms of all six performance indicators.

result in a higher average velocity and flow rate as well as reduced stress levels. Comparing the reduction in trip times ($25.36\ s$ for 1000 vehs/h, $26.59\ s$ for 2000 vehs/h and $27.33\ s$ for 3000 vehs/h) and in waiting times ($11.99\ s$ for 1000 vehs/h, $15.53\ s$ for 2000 vehs/h and $17.49\ s$ for 3000 vehs/h), we observe that they do not coincide. This shows that the communicative agent not only decreases the waiting time per stop but also the total number of stops, reducing the need of decelerating and accelerating among vehicles and resulting in more fluent traffic. The reduced acceleration among vehicles, alongside the diminished time that vehicles spend on the road, result in 20 % reduction of CO_2 emissions, mitigating the environmental repercussions of congestion.

5.6. Convergence

Finally, we briefly want to review the convergence properties of the discussed experiments. In contrast to many previous works, we chose not to show the learning phase of the DRL system and, instead, focused on the final performance of the agent. As we consider our agents to be trained in simulation and not in the real world, the transient phase of learning is of little importance.

As mentioned earlier, the term 'final performance' has to be used carefully, since it is hard to

demand	agent	avg. velocity	flow rate	CO_2 emissions	avg. stress	avg. trip time	avg. waiting time
1000	solitary	$7.01\ \frac{m}{s}$	$81.93\ \%$	$67.70\ \frac{g}{s}$	$4.23\ \%$	$99.65\ s$	$23.15\ s$
	communicative	$9.47\ \frac{m}{s}$	$93.41\ \%$	$53.58\ \frac{g}{s}$	$0.76\ \%$	$74.29\ s$	$11.16\ s$
	relative value	$135\ \%$	$114\ \%$	$79\ \%$	$18\ \%$	$75\ \%$	$48\ \%$
2000	solitary	$6.45\ \frac{m}{s}$	$78.79\ \%$	$143.91\ \frac{g}{s}$	$4.90\ \%$	$108.64\ s$	$28.93\ s$
	communicative	$8.56\ \frac{m}{s}$	$90.91\ \%$	$112.56\ \frac{g}{s}$	$1.25\ \%$	$82.05\ s$	$13.40\ s$
	relative value	$133\ \%$	$115\ \%$	$78\ \%$	$26\ \%$	$76\ \%$	$46\ \%$
3000	solitary	$5.50\ \frac{m}{s}$	$73.63\ \%$	$245.33\ \frac{g}{s}$	$7.13\ \%$	$127.55\ s$	$38.16\ s$
	communicative	$6.99\ \frac{m}{s}$	$84.71\ \%$	$195.14\ \frac{g}{s}$	$2.77\ \%$	$100.22\ s$	$20.67\ s$
	relative value	$127\ \%$	$115\ \%$	$80\ \%$	$39\ \%$	$79\ \%$	$54\ \%$

Table 5.4.: Absolute and relative results of the two different agents in the l'Antigua Esquerra de l'Eixample scenario.

determine whether the policy has converged or if the performance may increase further during additional training. We here trained all agents until the total undiscounted reward per episode as well as the two loss functions of the action-value function and the policy plateaued.

In general, the initial phase of training showed stagnating or even decreasing performances, as the policy tried to maximise a very poorly approximated action-value function. As soon as the NN better approximates its epitome, the performance starts to improve. Following the Pareto principle, the performance first improves very quickly and then gradually reduces convergence speed. Figure C.2 in appendix C.1 shows the development of the average velocity for 10^6 training steps for the solitary agent in the single intersection scenario. Since one training step corresponds to one second of simulated time, a real-world agent would take about 11.5 days to converge.

6. Summary

In this chapter, we will briefly summarise the methods that were developed in this work and the outcomes of our experimental study thereof. Subsequently, we will outline the findings and conclusions of our investigation. Finally, we will give a short outlook on further investigations that might be interesting in the context of this work but were not explored here.

We developed a Deep Reinforcement Learning (DRL) agent that learns the control of multiple traffic lights in a simulated traffic network. To that end, we framed the traffic control problem as a Markov Decision Process (MDP) in that the agent can observe its environment, take regulating actions and is evaluated in terms of a numerical reward signal (see section 4.4). In every timestep, the agent selects a phase time and a subsequent phase, where available phase options are selected as to guarantee safety. We extended the popular DDPG algorithm to introduce a method that can cope with the discrete-continuous action-domain of the traffic control problem and the combinatorial complexity of the centralised control of multiple intersections (see section 4.5). This approach is orthogonal to many previous works, which avoid large action-spaces by independently optimising the signalling at single intersections (see section 4.3).

The principal goal of this work was to study the influence of Vehicle to Infrastructure (V2I) communication on the efficacy of traffic control algorithms in mitigating traffic congestion and delay and to investigate the capacity of the DRL framework to leverage the resulting, large amounts of data in order to make better control decisions. To do so, we developed two different agents: one that makes decisions based solely on timing — the so-called solitary agent — and one that additionally incorporates the position and velocity of nearby vehicles — the communicative agent. The solitary agent here symbolises the setting of traditional control paradigms, whereas the communicative agent embodies the availability of V2I communication.

In a series of experiments, we showed that the availability of V2I communication enables the communicative agent to consistently outperform its solitary counterpart. We implemented four different scenarios: the atomic setting of only a single controlled intersection (see section 5.2), a long arterial road that is crossed by four shorter roads (see section 5.3), a regular grid of three by three perpendicular roads (see section 5.4) and a part of the L'Antiga Esquerra de l'Eixample neighbourhood in Barcelona (see section 5.5).

For the single controlled intersection, we compared the two agents for a range of different traffic volumes (see figure 5.4). For low volumes, the availability of information about nearby vehicles enabled the communicative agent to grant the right of way to individual approaching vehicles and thus to significantly exceed the performance of the setting without V2I communi-

cation (for the lowest tested traffic volume, the average velocity of vehicles was increased by 77 %). For larger volumes, the advantage of the communicative agent gradually decreased. As it is straightforward to find the optimal fixed-cycle signalling strategy in this simple setting, we also verified that the solitary agent is able to match the performance of the optimal fixed-cycle strategy.

In further experiments, we confirmed that the advantage of the communicative agent also holds for settings that require the coordination of traffic lights at multiple intersections. In the arterial and the grid scenario, it demonstrated to improve the system's efficacy by better coordination, more accurate phase times and more intelligently managing the inflow from side roads (see figure 5.5, 5.6 and 5.8). The benefit of V2I communication turns out to be particularly strong when the agent has to manage streams with low arrival rates, which are not well described by statistics. In particular, the 'blindness' of the solitary agent often lets it grant the right of way to empty approaches, while vehicles on busy approaches have to stop and wait for their green phase. In contrast, the communicative agent can better manage lesser travelled routes by granting the right of way only when it is required. For highly frequented routes, the communicative agent can at least partly preserve its advantage by choosing more accurate phase times and by better coordinating the intersections. Therefore, its benefit is especially strong for low traffic volumes and decreases for higher volumes. For example, in the arterial setting we observed a 67% increase in average velocity for the lowest tested volume and approximately 30 % increase for higher volumes.

In the arterial scenario, the communicating agent showed to handle fluctuating traffic volumes more efficiently and to recover significantly faster from a sudden increase in demand than the solitary agent (see figure 5.7). Furthermore, the communicating agent was trained with varying traffic volumes to learn a single policy for all demand settings. The solitary agent, on the other hand, needs an individual policy for every level of traffic demand in the different scenarios. This ability to seamlessly handle a wide range of different demands puts the communicative agent at a significant advantage in more realistic settings in which the current traffic volume is unknown.

The choice of a reward function is crucial as it ultimately defines the behaviour of the learned agent. A particularly appealing feature of DRL methods is its potential to maximise composite reward functions that consist of many different performance measures. This ability enables us to implement a wide variety of diverse objectives that we require a traffic controller to accomplish. In contrast, traditional traffic control approaches are often tailored to optimise one specific variable such as throughput or delay. We here investigated the effects of employing such an aggregated reward function, consisting of four distinct performance measures. The DRL agent showed to be able to jointly optimise all four components (see figure 5.9, C.7 and C.9). Note that, in a real-world scenario, the availability of V2I communication a necessary condition for the application of DRL methods as it is needed to transmit the reward function to the learning system.

In the L'Antigua Esquerra de l'Eixample setting, we demonstrated that our system can be used to optimise real-world traffic scenarios without much effort. The traffic scenario is generated from Open Street Map (OSM) data and the same methods could therefore quickly be applied to other neighbourhoods, enabling rapid prototyping of DRL traffic control policies. As in other examples, the availability of V2I communication showed to enable the traffic system to manage traffic more efficiently (see figure 5.10). For all tested demands, we observed an approximately 30 % higher average velocity, approximately 20 % lower CO_2 emissions and approximately 50 % less waiting time at traffic lights.

6.1. Conclusions

1. The availability of Vehicle to Infrastructure (V2I) communication clearly has the potential to improve the efficacy of traffic systems, alleviating the economic burden and environmental repercussions of traffic congestion.

2. The Deep Reinforcement Learning (DRL) framework in general, and the here-developed methods in particular, appear to be fully capable of leveraging the additional information that is made available through the communication interface, to take better control decisions and, therefore, to mitigate congestion and decrease delays.

3. Due to their model-free nature, DRL methods allow the optimisation of traffic control policies without requiring us to accurately model the traffic scenario. This not only alleviates the burden on traffic engineers but also mitigates the inefficiencies of the control strategy due to approximations and limiting assumptions of the traffic model.

4. The availability of V2I communication enables the DRL agent to seamlessly handle a wide range of different traffic volumes, as well as fluctuating traffic volumes. This puts it at an advantage when handling more realistic traffic scenarios, in which the volume is unknown.

5. The benefit of communication is especially strong when arrival rates of some of the controlled approaches are low, and the agent can grant the right of way to individual vehicles. For higher traffic volumes, the communicative agent can still manages to significantly outperform the solitary agent; however, its benefit is slightly dampened.

6. The use of composite reward functions enables the DRL system to optimise the diverse objectives of the traffic control problem jointly. Using V2I communication, vehicles can transmit a plethora of measures that we could optimise.

7. Due to safety issues, DRL methods are rarely ever used in real-world control systems. However, traffic control action-spaces can easily be designed to prevent unsafe situations. Intelligent traffic light control therefore has great potential to be among the first real-world applications of DRL methods.

6.2. Outlook

The complexity of the traffic control problem and the sheer size of the design space of DRL methods moves an exhaustive review of all aspects of the matter beyond the scope of this work. In this section, we want to hint towards interesting research directions that we consider to require further investigation:

Tangible goals and reward functions – We have shown that the DRL system can learn to optimise both a simple and an aggregated reward function. However, designing an appropriate reward function might be just as challenging as optimising one. Deriving a tangible reward signal from the abstract goals that we want our traffic system to achieve is no trivial task, and inadequate rewards may result in inefficient or even harmful policies. Further research, therefore, needs to identify important subgoals of the traffic system and appropriately quantify and weigh them.

Influence of different observation spaces – For our experiments, we tried out different representations of the observation-space. Even though, technically, all representations featured the same level of information, the resulting performance of the DRL agent varied strongly. We are not entirely sure what makes the observation-space described in section 4.4.1 better than other approaches or if a different structure of the observational data may have further improved our results. Additional investigation therefore is needed, to better understand how data can be structured for a Neural Network (NN) to make sense of it, or, alternatively, what NN architecture would be better suited to deal with our data representation.

Explicit treatment of partial observability – We have started our work by describing the Partially Observable Markov Decision Process (POMDP) (see section 2.2) in that an agent may consider its entire history of past observations and actions. Our agent however, bases its decisions only on the most recent observation. Even though we partly integrate the agent's past by implementing trace- and timing variables in the observation, additional information could certainly be gained by using its entire history. It would be interesting to experiment with models that can implement a series of past actions and observations. In particular, replacing the feed-forward NN of our model with a recurrent NN might further improve the performance of the agent.

Multi-agent learning – The here-developed DRL approach focuses on dealing with the scalability issues of other approaches like Deep Q-Learning (DQL), and centrally learns a joint control policy for several traffic lights. However, this approach certainly has its limits and cannot be scaled to control the traffic system of entire cities. Furthermore, large spatial distances between intersections strongly alleviate the need for intricate coordination between their respective traffic light systems. The widespread application of such methods would, therefore, require a divide-and-conquer approach in that several agents are learned, each operating a part of the entire traffic system. To navigate the boundaries between different agents successfully, some coordination technique may be necessary. Investigating the interaction and developing coordination methods of multiple agents in a multi-agent system would be crucial for scaling up the applicability of our methods to larger traffic networks.

More elaborate traffic simulations – The real-world performance of a policy that is learned in a simulated environment can only be as good as the simulation is realistic. Simulations need to make some simplifying assumptions and thus inevitably diverge from the real world. We designed our traffic environment, having in mind its realism; however, many aspects of it clearly do not correspond to its epitome. In particular, the incorporation of other entities such as pedestrians, cyclists and motorcyclists as well as a more realistic behaviour of drivers might result in fundamentally different policies.

Real-world deployment – Being able to manage traffic in a simulated environment, of course, is worth very little if our methods fail to handle traffic in the real world. Maybe the most interesting further investigation would therefore be to deploy a DRL agent in a real traffic system. Due to safety issues, such an experiment should only be conducted under the close supervision of traffic authorities, and extensive measures should be taken to protect all traffic participants from harm.

Two-way V2I communication – Emerging V2I communication interfaces allow the bilateral

exchange of information between vehicles and the traffic infrastructure. In this work, we used information of individual vehicles to select adequate control actions of traffic lights. In a next step, one could allow the traffic agent to send messages to individual vehicles. For example, sending velocity suggestions to approaching vehicles could enable the traffic agent to actively manage traffic streams and improve traffic flow.

5G network simulator – In this work, we integrate a V2I communication interface by simply assuming a vehicle's information to be available to the traffic agent. An interesting addition to the simulation would be to model the communication channel explicitly.

Bibliography

Allsop, R. E. (1971). “Sigset: A Computer Program for Calculating Traffic Signal Settings”. In: *Traffic Engineering & Control* 2.13, pp. 58–60.

Allsop, R. E. (1976). “SIGCAP: A computer programm for assesing the traffic capacity of signal-controlled road junctions”. In: *Traffic Engineering & Control* 17.819, pp. 338–341.

Amari, S.-I. (1998). “Natural Gradient Works Efficiently in Learning”. In: *Neural Computation* 10.2, pp. 251–276.

Amari, S.-I. (2016). *Information Geometry and Its Applications*. Springer Publishing.

Arel, I., C. Liu, T. Urbanik, and A. G. Kohls (2010). “Reinforcement learning-based multi-agent system for network traffic signal control”. In: *IET Intelligent Transport Systems* 4.2, pp. 128–135.

Arena, F. and G. Pau (2019). “An Overview of Vehicular Communications”. In: *Future Internet* 11, p. 27.

Bakker, B., S. Whiteson, L. Kester, and and Groen Frans C A (2010). “Traffic Light Control by Multiagent Reinforcement Learning Systems”. In: *Interactive Collaborative Information Systems*. Springer Publishing, pp. 475–510.

Barth-Maron, G., M. W. Hoffman, D. Budden, W. Dabney, D. Horgan, D. TB, A. Muldal, N. Heess, and T. P. Lillicrap (2018). “Distributed Distributional Deterministic Policy Gradients”. In: *arXiv e-prints* arXiv:1804.08617.

Behrisch, M., L. Bieker, J. Erdmann, and D. Krajzewicz (2011). “SUMO – Simulation of Urban MObility: An Overview”. In: *Proceedings of SIMUL 2011, The 3rd International Conference on Advances in System Simulation*, pp. 63–68.

Bellemare, M. G., W. Dabney, and R. Munos (2017). “A Distributional Perspective on Reinforcement Learning”. In: *arXiv e-prints* arXiv:1707.06887.

Bellman, R. (1958). “Dynamic programming and stochastic control processes”. In: *Information and Control* 1.3, pp. 228–239.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer Publishing.

Boillot, F. (1994). “Evaluation of the Real-Time Urban Traffic Control Algorithm CRONOS: First Phase”. In: *IFAC Proceedings Volumes* 27.12, pp. 585–590.

Brockman, G., V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba (2016). “OpenAI Gym”. In: *arXiv e-prints* arXiv:1606.01540.

Casas, N. (2017). “Deep Deterministic Policy Gradient for Urban Traffic Light Control”. In: *arXiv e-prints* arXiv:1703.09035.

Choromanska, A., M. Henaff, M. Mathieu, G. B. Arous, and Y. LeCun (2014). “The Loss Surface of Multilayer Networks”. In: *arXiv e-prints* arXiv:1412.0233.

Dayan, P. and L. F. Abbott (2005). *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*. The MIT Press.

Duan, Y., X. Chen, R. Houthooft, J. Schulman, and P. Abbeel (2016). “Benchmarking Deep Reinforcement Learning for Continuous Control”. In: *arXiv e-prints* arXiv:1604.06778.

Dulac-Arnold, G., R. Evans, P. Sunehag, and B. Coppin (2015). “Reinforcement Learning in Large Discrete Action Spaces”. In: *arXiv e-prints* arXiv:1512.07679.

Al-Dweik, A. J., M. Mayhew, R. Muresan, S. M. Ali, and A. Shami (2017). “Using Technology to Make Roads Safer: Adaptive Speed Limits for an Intelligent Transportation System”. In: *IEEE Vehicular Technology Magazine* 12.1, pp. 39–47.

European Comission (2017). “European Urban Mobility - Policy Context”. Brussels, Belgium.

Fawaz, H. I., G. Forestier, J. Weber, L. Idoumghar, and P.-A. Muller (2019). “Adversarial Attacks on Deep Neural Networks for Time Series Classification”. In: *arXiv e-prints* arXiv:1903.07054.

Fujimoto, S., H. van Hoof, and D. Meger (2018). “Addressing Function Approximation Error in Actor-Critic Methods”. In: *arXiv e-prints* arXiv:1802.09477.

Gartner, N., C. J. Messer, and A. K. Rathi (2001). *Traffic flow theory: A state-of-the-art report*.

Gil, J., E. Tobari, M. Lemlij, A. Rose, and A. Penn (2009). “The Differentiating Behaviour of Shoppers: Clustering of individual movement traces in a supermarket”. In: *Proceedings of the 7th International Space Syntax Symposium*.

Glorot, X. and Y. Bengio (2010). “Understanding the difficulty of training deep feedforward neural networks”. In: *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics*, pp. 249–256.

Goodfellow, I., Y. Bengio, and A. Courville (2016). *Deep Learning*. MIT Press.

Grathwohl, W., D. Choi, Y. Wu, G. Roeder, and D. K. Duvenaud (2017). “Backpropagation through the Void: Optimizing control variates for black-box gradient estimation”. In: *arXiv e-prints* arXiv:1711.00123.

Gumbel, E. J. (1954). *Statistical theory of extreme values and some practical applications: a series of lectures*. Applied mathematics series. U. S. Govt. Print. Office.

Haarnoja, T., A. Zhou, P. Abbeel, and S. Levine (2018a). “Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor”. In: *arXiv e-prints* arXiv:1801.01290.

Haarnoja, T., A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine (2018b). “Soft Actor-Critic Algorithms and Applications”. In: *arXiv e-prints* arXiv:1812.05905.

Hasselt, H. van, A. Guez, and D. Silver (2015). “Deep Reinforcement Learning with Double Q-learning”. In: *arXiv e-prints* arXiv:1509.06461.

Hastie, T., R. Tibshirani, J. Friedman, and J. Franklin (2004). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Vol. 27. Springer Publishing.

He, K., X. Zhang, S. Ren, and J. Sun (2015a). “Deep Residual Learning for Image Recognition”. In: *arXiv e-prints* arXiv:1512.03385.

He, K., X. Zhang, S. Ren, and J. Sun (2015b). "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: *arXiv e-prints* arXiv:1502.01852.

Henderson, P., R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger (2017). "Deep Reinforcement Learning that Matters". In: *arXiv e-prints* arXiv:1709.06560.

Henry, J., J. Farges, and J. Tuffal (1983). "The Prodyn Real Time Traffic Algorithm". In: *IFAC Proceedings Volumes* 16.4, pp. 305–310.

Hessel, M., J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. G. Azar, and D. Silver (2017). "Rainbow: Combining Improvements in Deep Reinforcement Learning". In: *arXiv 1409.4842* arXiv:1710.02298.

Hochreiter, S. and J. Schmidhuber (1997). "Long Short-Term Memory". In: *Neural Computation* 9.8, pp. 1735–1780.

Horgan, D., J. Quan, D. Budden, G. Barth-Maron, M. Hessel, H. van Hasselt, and D. Silver (2018). "Distributed Prioritized Experience Replay". In: *arXiv e-prints* arXiv:1803.00933.

Hunt, P. B. and D. I. Robertson (1982). "The SCOOT on-line traffic signal optimisation technique". In: *Traffic Engineering & Control* 23.4, pp. 190–192.

Hunter, J. D. (2007). "Matplotlib: A 2D Graphics Environment". In: *Computing in Science & Engineering* 9.3, pp. 90–95.

Improta, G. and G. Cantarella (1984). "Control system design for an individual signalized junction". In: *Transportation Research Part B: Methodological* 18.2, pp. 147–167.

Inrix (2018). "Inrix Global Traffic Scoreboard".

Jang, E., S. Gu, and B. Poole (2016). "Categorical Reparameterization with Gumbel-Softmax". In: *arXiv e-prints* arXiv:1611.01144.

Kakade, S. (2001). "A Natural Policy Gradient". In: *Advances in Neural Information Processing Systems*. Vol. 14, pp. 1531–1538.

Kergaye, C., A. Stevanovic, and P. Martin (2008). "An Evaluation of SCOOT and SCATS through Microsimulation". In: *Proceedings of the 10th International Conference on Applications of Advanced Technologies in Transportation*, pp. 1166–1180.

Khamis, M., W. Gomaa, and H. El-Shishiny (2012). "Multi-Objective Traffic Light Control System based on Bayesian Probability Interpretation". In: *IEEE Conference on Intelligent Transportation Systems*, pp. 995–1000.

Kingma, D. P. and J. Ba (2014). "Adam: A Method for Stochastic Optimization". In: *Proceedings of the 3rd International Conference on Learning Representations*.

Kingma, D. P. and M. Welling (2014). "Auto-Encoding Variational Bayes". In: *Proceedings of the 2nd International Conference on Learning Representations*.

Koller, T., F. Berkenkamp, M. Turchetta, and A. Krause (2018). "Learning-based Model Predictive Control for Safe Exploration and Reinforcement Learning". In: *arXiv 1409.4842* arXiv:1803.08287.

Kotusevski, G. and K. Hawick (2009). "A Review of Traffic Simulation Software". In: *Research Letters in the Information and Mathematical Sciences* 13, pp. 35–54.

Krajzewicz, D. (2010). "Traffic Simulation with SUMO – Simulation of Urban Mobility". In: *Fundamentals of Traffic Simulation*. Springer Publishing, pp. 269–293.

Krauss, S. (1998). “Microscopic modeling of traffic flow: investigation of collision free vehicle dynamics”. PhD thesis. DLR Deutsches Zentrum fuer Luft- und Raumfahrt; Koeln Univ. (Germany). Mathematisch-Naturwissenschaftliche Fakultaet.

Kurakin, A., I. J. Goodfellow, and S. Bengio (2016). “Adversarial examples in the physical world”. In: *arXiv e-prints* arXiv:1607.02533.

Kuyer, L., S. Whiteson, B. Bakker, and N. Vlassis (2008). “Multiagent Reinforcement Learning for Urban Traffic Control Using Coordination Graphs”. In: *Machine Learning and Knowledge Discovery in Databases*. Springer Publishing, pp. 656–671.

Lapan, M. (2018). *Deep Reinforcement Learning Hands-On: Apply Modern RL Methods, with Deep Q-Networks, Value Iteration, Policy Gradients, TRPO, AlphaGo Zero and More*. Expert insight. Packt Publishing.

LeCun, Y., P. Haffner, L. Bottou, and Y. Bengio (1999). “Object Recognition with Gradient-Based Learning”. In: *Shape, Contour and Grouping in Computer Vision*. Springer Publishing, p. 319.

Liang, E., R. Liaw, R. Nishihara, P. Moritz, R. Fox, J. Gonzalez, K. Goldberg, and I. Stoica (2017). “Ray RLLib: A Composable and Scalable Reinforcement Learning Library”. In: *Proceedings of the 31st Conference on Neural Information Processing Systems*.

Lillicrap, T. P., J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra (2015). “Continuous control with deep reinforcement learning”. In: *arXiv e-prints* arXiv:1509.02971.

Lin, L.-J. (1992). “Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching”. In: *Machine Learning* 8.3-4, pp. 293–321.

Little, J. D. C. (1966). “The Synchronization of Traffic Signals by Mixed-Integer Linear Programming”. In: *Operations Research* 14.4, pp. 568–594.

Liu, M., J. Deng, X. Ming, Xianbo Zhang, and W. Wang (2017). “Cooperative Deep Reinforcement Learning for Traffic Signal Control”. In: *Proceedings of the 23rd ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*.

Lütjens, B., M. Everett, and J. P. How (2018). “Safe Reinforcement Learning with Model Uncertainty Estimates”. In: *arXiv e-prints* arXiv:1810.08700.

Maddison, C. J., A. Mnih, and Y. W. Teh (2016). “The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables”. In: *arXiv e-prints* arXiv:1611.00712.

Mannion, P., J. Duggan, and E. Howley (2016). “An Experimental Review of Reinforcement Learning Algorithms for Adaptive Traffic Signal Control”. In: *Autonomic Road Transport Support Systems*, pp. 47–66.

Mckinney, W. (2010). “Data Structures for Statistical Computing in Python”. In: *Proceedings of the 9th Python in Science Conference*, pp. 51–56.

McShane, C. (1999). “The Origins and Globalization of Traffic Control Signals”. In: *Journal of Urban History* 25, pp. 379–404.

Medina, J. C. and R. F. Benekohal (2012). “Traffic signal control using reinforcement learning and the max-plus algorithm as a coordinating strategy”. In: *Proceedings of the 15th International IEEE Conference on Intelligent Transportation Systems*, pp. 596–601.

Mikami, S. and Y. Kakazu (1994). “Genetic reinforcement learning for cooperative traffic signal control”. In: *Proceedings of the 1st IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, pp. 223–228.

Miller, A. J. (1963). "A Computer control system for traffic networks / Alan J. Miller". In: *Proceedings of the 2nd International Symposium on the Theory of Road Traffic Flow*, pp. 200–220.

Mnih, V., A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu (2016). "Asynchronous Methods for Deep Reinforcement Learning". In: *arXiv e-prints* arXiv:1602.01783.

Mnih, V., K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis (2015). "Human-level control through deep reinforcement learning". In: *Nature* 518, p. 529.

Mordatch, I. and P. Abbeel (2017). "Emergence of Grounded Compositional Language in Multi-Agent Populations". In: *arXiv e-prints* arXiv:1703.04908.

Mousavi, S. S., M. Schukat, and E. Howley (2017). "Traffic light control using deep policy-gradient and value-function-based reinforcement learning". In: *IET Intelligent Transport Systems* 11.7, pp. 417–423.

Nachum, O., M. Norouzi, K. Xu, and D. Schuurmans (2017). "Trust-PCL: An Off-Policy Trust Region Method for Continuous Control". In: *arXiv e-prints* arXiv:1707.01891.

Oliveira Boschetti, D. de, A. Bazzan, B. da Silva, E. Basso, and L. Nunes (2006). "Reinforcement Learning based Control of Traffic Lights in Non-stationary Environments: A Case Study in a Microscopic Simulator." In: *Proceedings of the 4th European Workshop on Multi-Agent Systems*.

Oord, A. van den, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. W. Senior, and K. Kavukcuoglu (2016). "WaveNet: A Generative Model for Raw Audio". In: *arXiv e-prints* arXiv:1609.03499.

Papageorgiou, M. and A. Kotsialos (2002). "Freeway ramp metering: an overview". In: *IEEE Transactions on Intelligent Transportation Systems* 3.4, pp. 271–281.

Papageorgiou, M. (2004). "Overview of Road Traffic Control Strategies". In: *IFAC Proceedings Volumes* 37.19, pp. 29–40.

Paszke, A., S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer (2017). "Automatic Differentiation in PyTorch". In: *Proceedings of the 31st Conference on Neural Information Processing Systems*.

Peramandai Govindasamy, K. (2015). "A Comparative Study on 4G and 5G Technology for Wireless Applications". In: *IOSR Journal of Electronics and Communication Engineering* 10.6, pp. 67–72.

Perez, F. and B. E. Granger (2007). "IPython: A System for Interactive Scientific Computing". In: *Computing in Science & Engineering* 9.3, pp. 21–29.

Popov, I., N. Heess, T. P. Lillicrap, R. Hafner, G. Barth-Maron, M. Vecerik, T. Lampe, Y. Tassa, T. Erez, and M. A. Riedmiller (2017). "Data-efficient Deep Reinforcement Learning for Dexterous Manipulation". In: *arXiv e-prints* arXiv:1704.03073.

Prabuchandran, K. J., H. K. A. N, and S. Bhatnagar (2015). "Decentralized learning for traffic signal control". In: *Proceedings of the 7th International Conference on Communication Systems and Networks*, pp. 1–6.

Rezende, D. J., S. Mohamed, and D. Wierstra (2014). "Stochastic Backpropagation and Approximate Inference in Deep Generative Models". In: *Proceedings of the 31st International Conference on Machine Learning*. Vol. 32. 2, pp. 1278–1286.

Richter, S., D. Aberdeen, and J. Yu (2006). "Natural Actor-critic for Road Traffic Optimisation". In: *Proceedings of the 19th International Conference on Neural Information Processing Systems*, pp. 1169–1176.

Robbins, H. and S. Monro (1951). "A Stochastic Approximation Method". In: *The Annals of Mathematical Statistics* 22.3, pp. 400–407.

Robertson, I. (1969). "TRANSYT method for area traffic control". In: *Traffic Engineering & Control* 10, pp. 276–281.

Rumelhart, D. E., G. E. Hinton, and R. J. Williams (1986). "Learning representations by back-propagating errors". In: *Nature* 323.6088, pp. 533–536.

Salkham, A., R. Cunningham, A. Garg, and V. Cahill (2008). "A Collaborative Reinforcement Learning Approach to Urban Traffic Control Optimization". In: *International Conference on Web Intelligence and Intelligent Agent Technology*. Vol. 2, pp. 560–566.

Salvatori, E. (2016). "5G und Car-to-X Schlüsseltechniken für den autonomen Straßenverkehr". In: *ATZelektronik* 11, pp. 28–33.

Schaul, T., J. Quan, I. Antonoglou, and D. Silver (2016). "Prioritized Experience Replay". In: *arXiv e-prints* arXiv:1511.05952.

Schulman, J., S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel (2015). "Trust Region Policy Optimization". In: *arXiv e-prints* arXiv:1502.05477.

Schulman, J., F. Wolski, P. Dhariwal, A. Radford, and O. Klimov (2017). "Proximal Policy Optimization Algorithms". In: *arXiv e-prints* arXiv:1707.06347.

Shi, L. and K. W. Sung (2014). "Spectrum Requirement for Vehicle-to-Vehicle Communication for Traffic Safety". In: *IEEE 79th Vehicular Technology Conference*, pp. 1–5.

Shoufeng, L., L. Ximin, and D. Shiqiang (2008). "Q-Learning for Adaptive Traffic Signal Control Based on Delay Minimization Strategy". In: *IEEE International Conference on Networking, Sensing and Control*, pp. 687–691.

Silver, D., A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis (2016). "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529, p. 484.

Silver, D., G. Lever, N. Heess, T. Degris, D. Wierstra, and M. A. Riedmiller (2014). "Deterministic Policy Gradient Algorithms". In: *Proceedings of the 31st International Conference on International Conference on Machine Learning*, pp. 387–395.

Silver, D., J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis (2017). "Mastering the game of Go without human knowledge". In: *Nature* 550, p. 354.

Simsek, O., S. Algorta, and A. Kothiyal (2016). "Why Most Decisions Are Easy in Tetris—And Perhaps in Other Sequential Decision Problems, As Well". In: *Proceedings of the 33rd International Conference on Machine Learning*. Vol. 48, pp. 1757–1765.

Stamatiadis, C. and N. Gartner (1996). "MULTIBAND-96: A Program for Variable-Bandwidth Progression Optimization of Multiarterial Traffic Networks". In: *Transportation Research Record Journal of the Transportation Research Board* 1554, pp. 9–17.

Sutton, R. S. and A. G. Barto (2018). *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning series. MIT Press.

Sutton, R. S. (1988). "Learning to predict by the methods of temporal differences". In: *Machine Learning* 3, pp. 9–44.

Szegedy, C., W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich (2014). "Going Deeper with Convolutions". In: *arXiv e-prints* arXiv:1409.4842.

El-Tantawy, S., B. Abdulhai, and H. Abdelgawad (2013). "Multiagent Reinforcement Learning for Integrated Network of Adaptive Traffic Signal Controllers (MARLIN-ATSC): Methodology and Large-Scale Application on Downtown Toronto". In: *IEEE Transactions on Intelligent Transportation Systems* 14.3, pp. 1140–1150.

El-Tantawy, S. and B. Abdulhai (2010). "An agent-based learning towards decentralized and coordinated traffic signal control". In: *IEEE Conference on Intelligent Transportation Systems*, pp. 665–670.

El-Tantawy, S., B. Abdulhai, and H. Abdelgawad (2014). "Design of Reinforcement Learning Parameters for Seamless Application of Adaptive Traffic Signal Control". In: *Journal of Intelligent Transportation Systems* 18.3, pp. 227–245.

Thorpe, T. L. (1998). *Vehicle Traffic Light Control Using SARSA*. Master's Thesis. Computer Science Department, Colorado State University.

Thrun, S. and A. Schwartz (1993). "Issues in Using Function Approximation for Reinforcement Learning". In: *Proceedings of the 1993 Connectionist Models Summer School*.

Treiber, M., A. Hennecke, and D. Helbing (2000). "Congested Traffic States in Empirical Observations and Microscopic Simulations". In: *Physical Review E* 62, pp. 1805–1824.

Tsitsiklis, J. N. (2003). "On the Convergence of Optimistic Policy Iteration". In: *Journal of Machine Learning Research* 3, pp. 59–72.

Tsitsiklis, J., B. Van Roy, I. of Technology. Laboratory for Information, and M. Decision Systems (1997). "An Analysis of Temporal-Difference Learning with Function Approximation". In: *IEEE Transactions on Automatic Control* 42.

Tucker, G., A. Mnih, C. J. Maddison, and J. Sohl-Dickstein (2017). "REBAR: Low-variance, unbiased gradient estimates for discrete latent variable models". In: *arXiv e-prints* arXiv:1703.07370.

Van der Pol, E. and F. A. Oliehoek (2016). "Coordinated Deep Reinforcement Learners for Traffic Light Control". In: *Workshop on Learning, Inference and Control of Multi-Agent Systems*.

Van Rossum, G. and F. L. Drake Jr (1995). *Python tutorial*. Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands.

Vincent, R. A. and C. P. Young (1986). "SELF-OPTIMIZING TRAFFIC SIGNAL CONTROL USING MICROPROCESSORS: THE TRRL "MOVA" STRATEGY FOR ISOLATED INTERSECTIONS". In: *Proceedings of the 2nd international conference on road traffic control* 260, pp. 102–105.

Vinitsky, E., A. Kreidieh, L. L. Flem, N. Kheterpal, K. Jang, C. Wu, F. Wu, R. Liaw, E. Liang, and A. M. Bayen (2018). "Benchmarks for reinforcement learning in mixed-autonomy traffic". In: *Proceedings of the 2nd Conference on Robot Learning*. Vol. 87, pp. 399–409.

Walt, S. v. d., S. C. Colbert, and G. Varoquaux (2011). "The NumPy Array: A Structure for Efficient Numerical Computation". In: *Computing in Science & Engineering* 13.2, pp. 22–30.

Wegener, A., M. Piorkowski, M. Raya, H. Hellbrück, S. Fischer, and J.-P. Hubaux (2008). "TraCI: An Interface for Coupling Road Traffic and Network Simulators". In: *Proceedings of the 11th Communications and Networking Simulation Symposium*. Vol. 155-163.

Wiering, M. (2000). "Multi-Agent Reinforcement Learning for Traffic Light Control." In: *Proceedings of the 7th International Conference on Machine Learning*, pp. 1151–1158.

Williams, R. J. (1992). "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine Learning* 8.3, pp. 229–256.

Wilson, A. C., R. Roelofs, M. Stern, N. Srebro, and B. Recht (2017). "The Marginal Value of Adaptive Gradient Methods in Machine Learning". In: *Advances in Neural Information Processing Systems 30*, pp. 4148–4158.

Wu, C., A. Kreidieh, K. Parvate, E. Vinitsky, and A. M. Bayen (2017a). "Flow: Architecture and Benchmarking for Reinforcement Learning in Traffic Control". In: *arXiv e-prints* arXiv:1710.05465.

Wu, J., H. Sun, Z. Y. Gao, and H. Z. Zhang (2009). "Reversible lane-based traffic network optimization with an advanced traveller information system". In: *Engineering Optimization* 41, pp. 87–97.

Wu, Y., E. Mansimov, S. Liao, R. B. Grosse, and J. Ba (2017b). "Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation". In: *arXiv e-prints* arXiv:1708.05144.

Young, T., D. Hazarika, S. Poria, and E. Cambria (2017). "Recent Trends in Deep Learning Based Natural Language Processing". In: *arXiv e-prints* arXiv:1708.02709.

Zhang, R., A. Ishikawa, W. Wang, B. Striner, and O. Tonguz (2018). "Intelligent Traffic Signal Control: Using Reinforcement Learning with Partial Detection". In: *arXiv e-prints* arXiv:1807.01628.

Appendices

A. Agent4D7 Algorithm

Algorithm 1: Agent4D7

Input: Batch size M; discount factor γ; replay buffer size R; initial environment steps B; actor learning rate α_π; critic learning rate α_Q; n-step bootstrapping steps N; discrete entropy scaling factor ε_{disc}; continuous entropy scaling factor ε_{cont}; target network update interval T_{target}; parameters of the categorical action-value distribution: lower bound Q_{min}, upper bound Q_{max} and number of discrete bins L.

Initialise network weights $(\theta, \omega_1, \omega_2)$ using Kaiming initialisation.

Initialise target network weights $(\theta', \omega_1', \omega_2') \leftarrow (\theta, \omega_1, \omega_2)$.

Initialise replay buffer $\mathcal{B}$.

Launch K agents and environments.

while *True* **do**

  if $len(\mathcal{B}) \geq B$ **then**

    Sample minibatch of M transitions of length N from the replay buffer.

    Sample for each transition $a'_{i+N} \sim \pi_{\theta'}(o_{i+N})$.

    Compute $\omega'_{min} = \arg\min_{\omega'_1, \omega'_2} \Big(mean\big(\hat{Q}_{\omega'_1}(o_{i+N}, a'_{i+N})\big), mean\big(\hat{Q}_{\omega'_2}(o_{i+N}, a'_{i+N})\big) \Big)$.

    Construct the target distributions $Y_i = \big(\sum_{n=0}^{N-1} \gamma^n r_{i+n}\big) + \gamma^N \hat{Q}_{\omega'_{min}}(o_{i+N}, a'_{i+N})$

      Note that the target distribution Y_i is constructed by projecting the discounted rewards on the bootstrapped distribution. This is done by moving the probability of each of the discrete bins of the categorical distribution according to the Bellman equation (see equation 4.4).

    Compute actor and critic updates:

$$\theta \leftarrow \theta - \alpha_\pi \frac{1}{M} \sum_{i=1}^{M} \nabla_\theta \Big(- \hat{Q}_{\omega_1}(o_i, a'_i \sim \pi_\theta(o_i)) - \varepsilon_{disc} \mathcal{H}_{disc}(\pi_\theta(o_i)) - \varepsilon_{cont} \mathcal{H}_{cont}(\pi_\theta(o_i)) \Big)$$

$$\omega_1 \leftarrow \omega_1 - \alpha_Q \frac{1}{M} \sum_{i=1}^{M} \nabla_{\omega_1} D_{KL}\Big(Y_i \,||\, \hat{Q}_{\omega_1}(o_i, a_i)\Big)$$

$$\omega_2 \leftarrow \omega_2 - \alpha_Q \frac{1}{M} \sum_{i=1}^{M} \nabla_{\omega_2} D_{KL}\Big(Y_i \,||\, \hat{Q}_{\omega_2}(o_i, a_i)\Big)$$

    if t *mod* $T_{target} = 0$ **then**

      Copy parameters to target networks $(\theta', \omega_1', \omega_2') \leftarrow (\theta, \omega_1, \omega_2)$.

    end

    (Optional) Adapt α_π so that the D_2 metric of the new and the old policy matches the desired value.

  end

end

Agent

while *True* **do**

  Sample action from policy $a \sim \pi_\theta(o)$ and observe reward r and new observation o'.

  Store transition (o, a, r, o') in replay buffer and delete old transitions if $len(\mathcal{B}) > R$.

end

B. Parameter Values

B.1. Agent4D7 Parameters

Category	Parameters
training	• discount factor: $\gamma = 0.99$ • batch size: $M = 256$ • actor learning rate: $\alpha_\pi = 10^{-3}$ • critic learning rate: $\alpha_Q = 10^{-3}$
replay buffer	• buffer size: $R = 10^6$ • initial transitions before learning starts: $B = 10^4$
n-step bootstrapping	• steps: $N = 5$
entropy regularisation	• discrete scaling factor: $\varepsilon_{disc} = 0.5 \rightarrow 0.01$ (annealed) • continuous scaling factor: $\varepsilon_{cont} = 0.01 \rightarrow 0.001$ (annealed)
distributional Q-value	• lower bound: $Q_{min} = 0$ • upper bound: $Q_{max} = 100$ • number of bins: $L = 100$
target networks	• update interval: $T_{target} = 1000$
reparameterisation	• gumbel temperature: $G = 2/3$ • minimal standard deviation: $\sigma_{min} = 0.005$ • maximal standard deviation: $\sigma_{max} = 0.5$
learning rate adaption	• target D_2 metric: $D_{target} = 0.005$ • parameter of proportional controller: $P = 1$ • minimal learning rate: $\alpha_{min} = 10^{-6}$ • maximal learning rate: $\alpha_{max} = 10^{-2}$
distributed experience	• number of simulated environments: $K = 3$

Table B.1.: Used parameter values of the Agent4D7 algorithm.

B.2. Traffic Environment Parameters

Parameter	Value
road length	$300\ m$
speed limit	$20\ \frac{m}{s}$
number of lanes per road and direction	3
minimal phase time	$5\ s$
maximal phase time	$100\ s$
number of distinct phase options	8
length of amber period	$5\ s$ ($4\ s$)
length of all red period	$7\ s$ ($2\ s$)
number of observed vehicles through V2I	$10/30$
horizon (length of episode)	1 hour/2 hours
length of simulation timestep	$1\ s$
initial vehicle velocity	$5\ \frac{m}{s}$

Table B.2.: Used parameter values of the traffic environment. Numbers in brackets show differing parameters of the l'Antiga Esquerra de l'Eixample scenario.

C. Additional Figures

C.1. Single Intersection

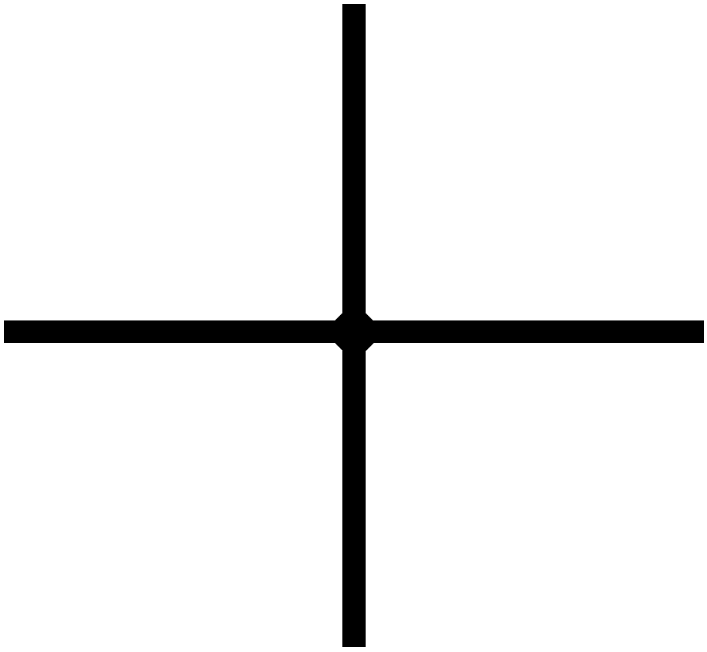


Figure C.1.: Traffic network of the single intersection scenario.

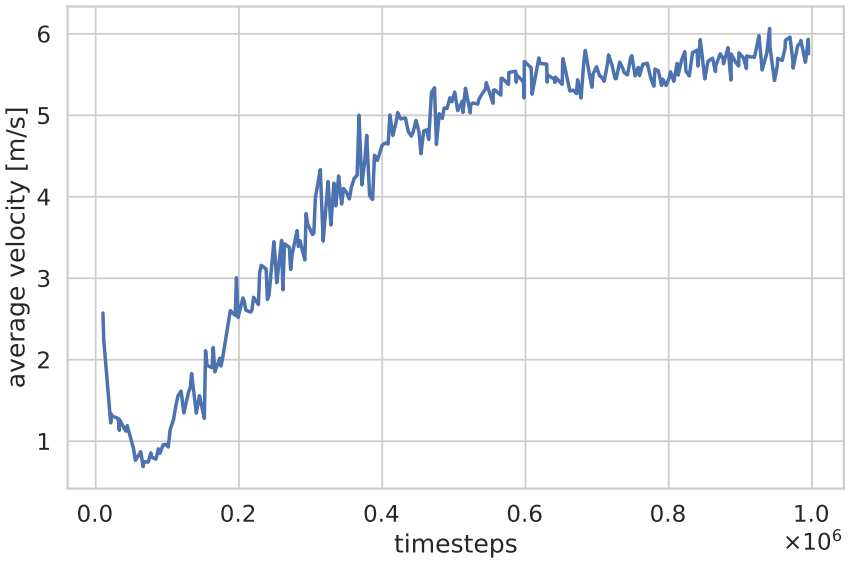


Figure C.2.: The learning curve for the solitary agent in the single intersection scenario for 10^6 training steps. After an initial decrease in performance due to an unconverged action-value function, the average velocity starts to increase and slowly converges to its final performance.

C.2. Arterial Road

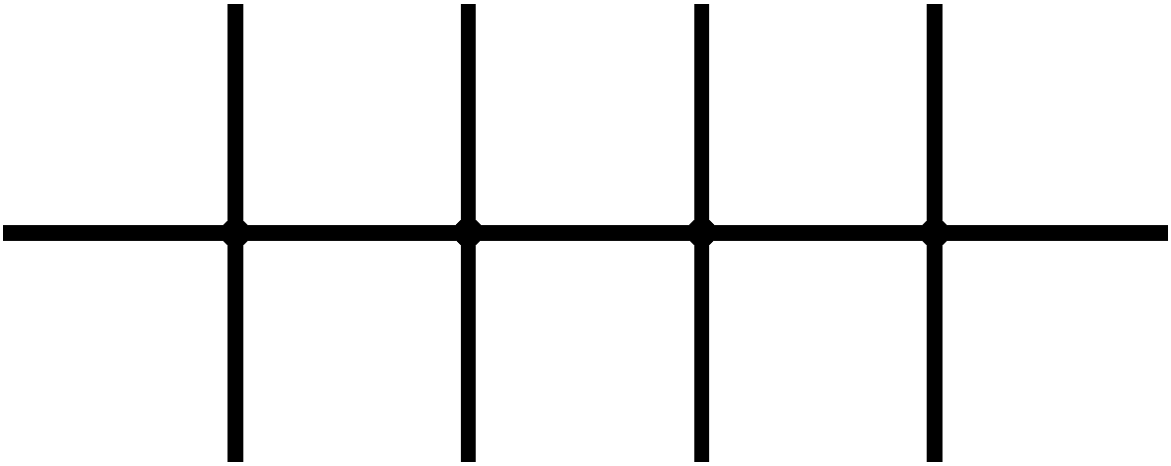
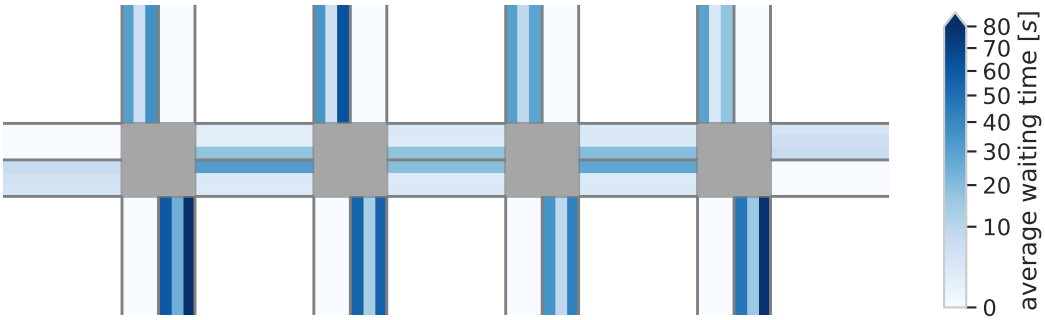
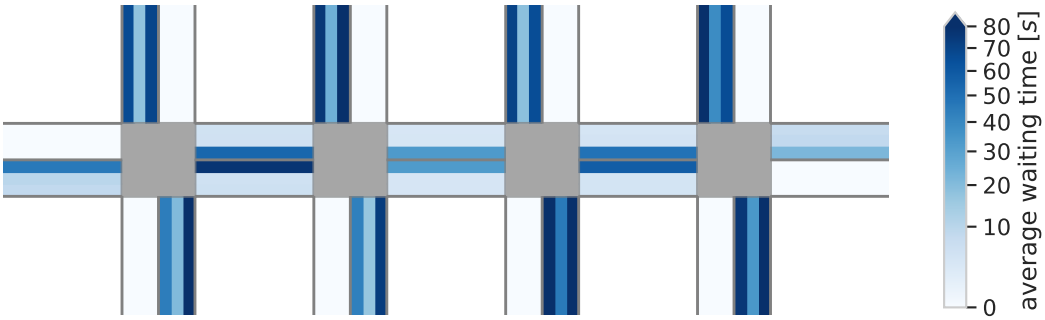


Figure C.3.: Traffic network of the arterial scenario. The horizontal main road is strongly utilised, wheras the vertical side roads are relatively quiet.

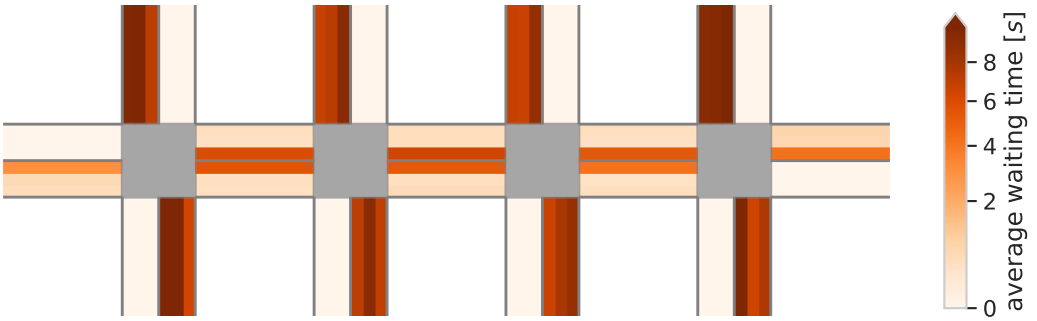


(a) 500 vehs/h.

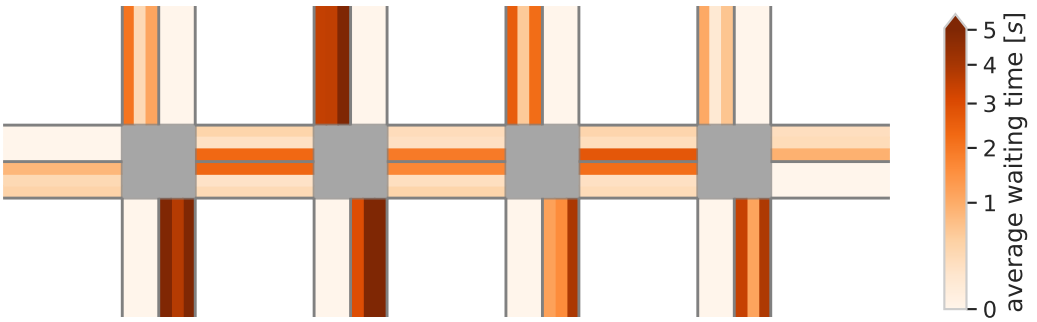


(b) 1000 vehs/h.

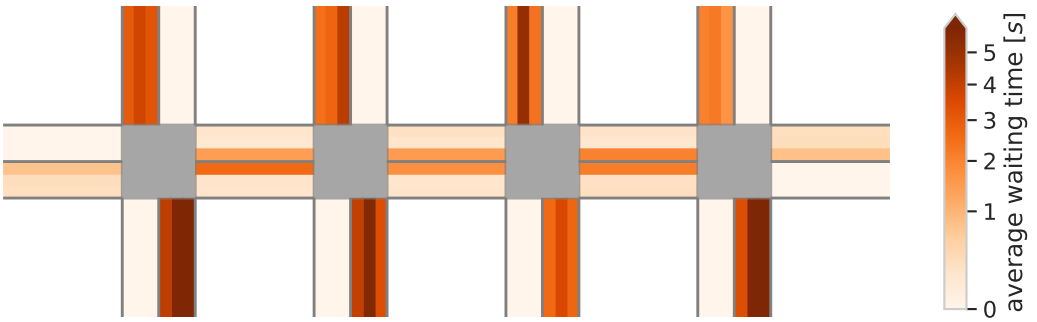
Figure C.4.: Average waiting times for every lane in the arterial scenario for the communicative agent for two different demand scenarios. For higher demands, vehicles on the side roads need to wait significantly longer for a green signal, whereas the waiting times on the main road change only marginally.



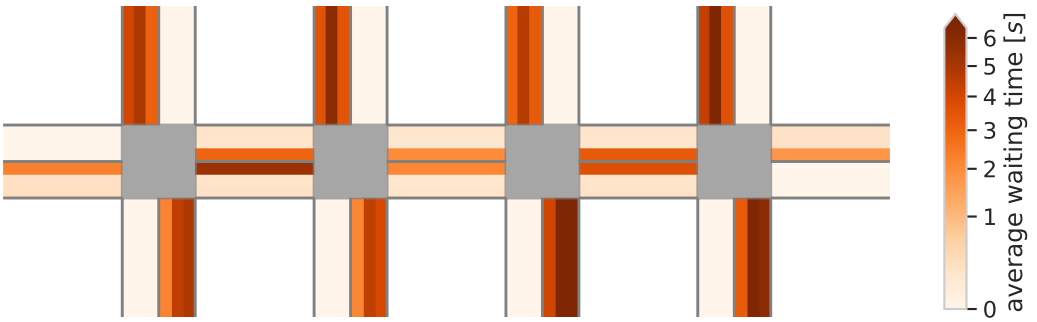
(a) Solitary agent for 200 vehs/h (figure 5.6a).



(b) Communicative agent for 200 vehs/h (figure 5.6b).



(c) Communicative agent for 500 vehs/h (figure C.4a).



(d) Communicative agent for 1000 vehs/h (figure C.4b).

Figure C.5.: Width of the 95% confidence intervals for the respective plots of average waiting time.

C.3. Grid

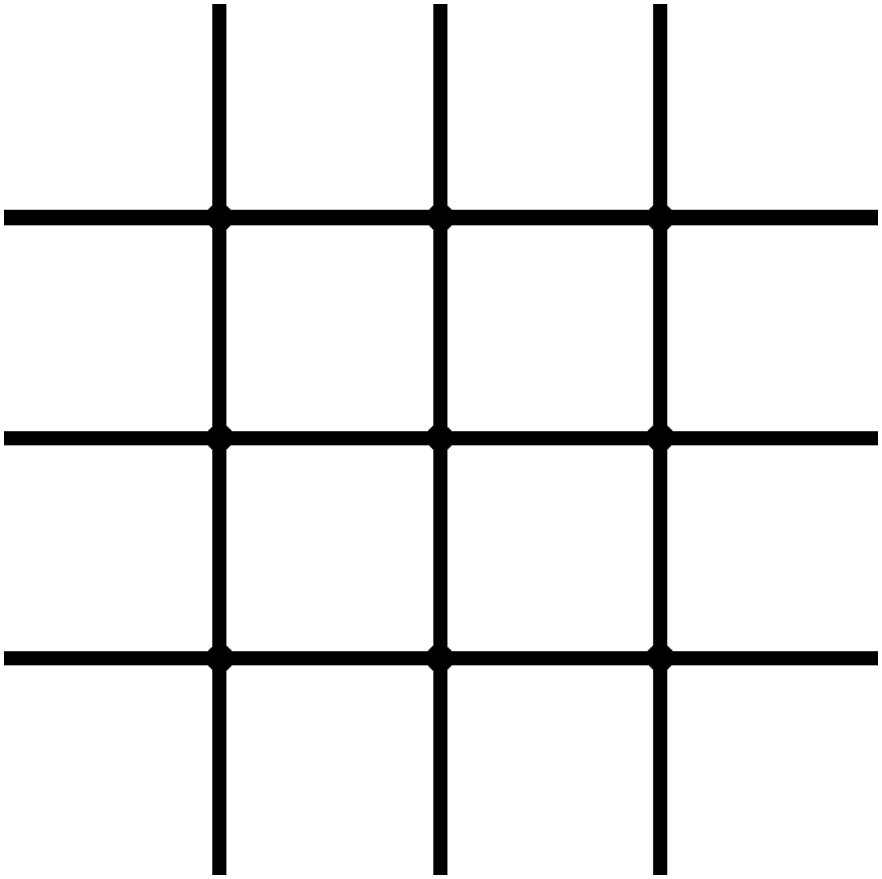


Figure C.6.: Traffic network of the grid scenario.

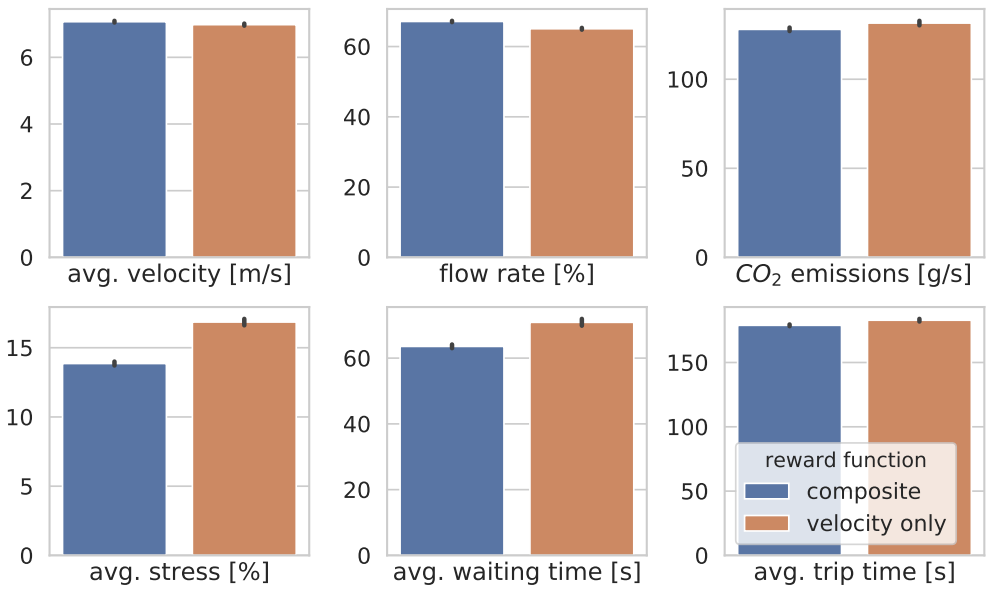
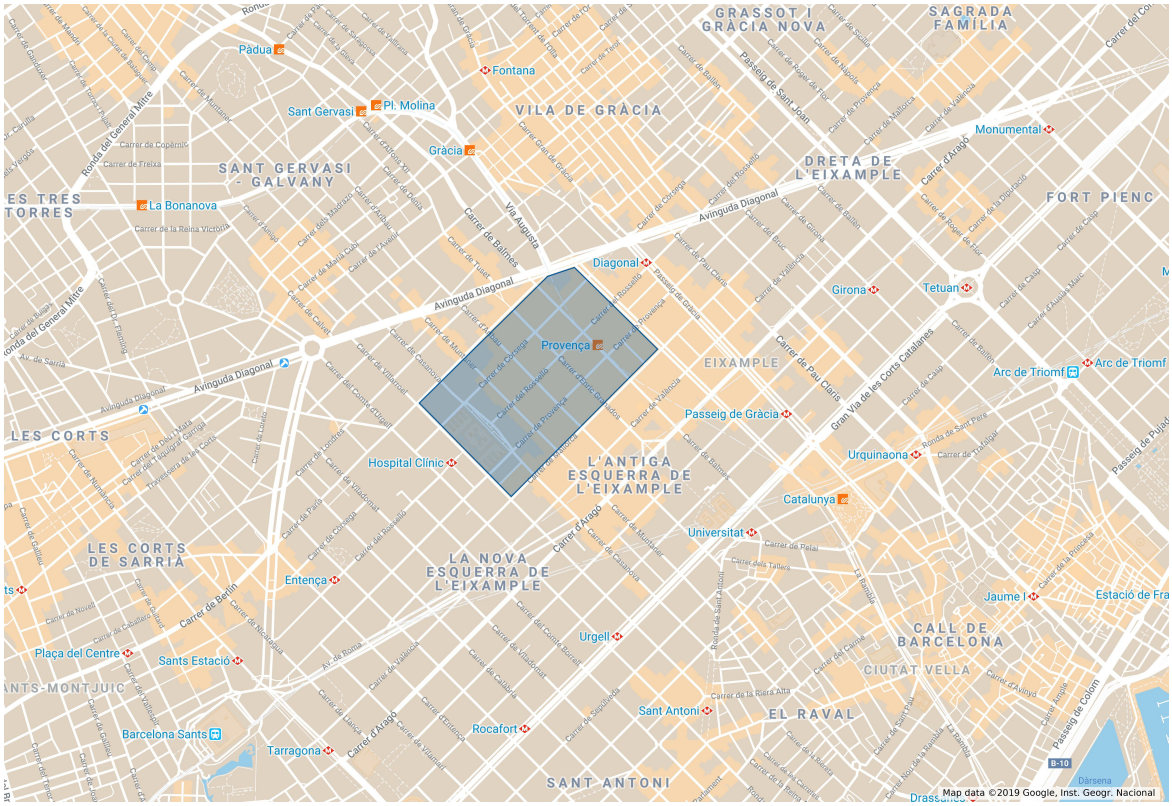
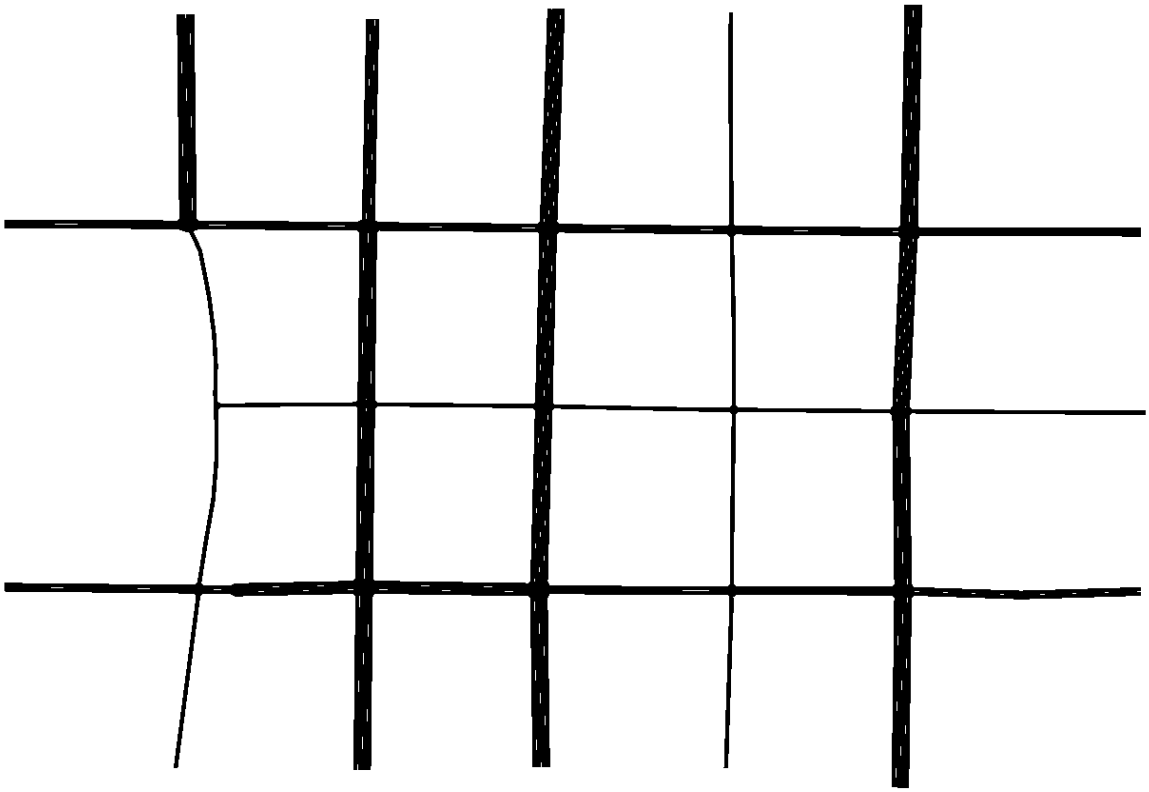


Figure C.7.: Comparison of the results for the two different reward functions. In addition to the four elements of the composite reward function, the figure also shows the average time that each vehicle spends waiting at an intersection during its trip and the average time that vehicles need to traverse the entire network.

C.4. L'Antiga Esquerra de l'Eixample



(a) Map of Barcelona. Taken from https://www.google.com/maps.



(b) The generated SUMO network. Note that we neglect some minor streets, which are not controlled by traffic lights as their incorporation led to accidents at the uncontrolled intersections.

Figure C.8.: Traffic network of the L'Antiga Esquerra de l'Eixample scenario.

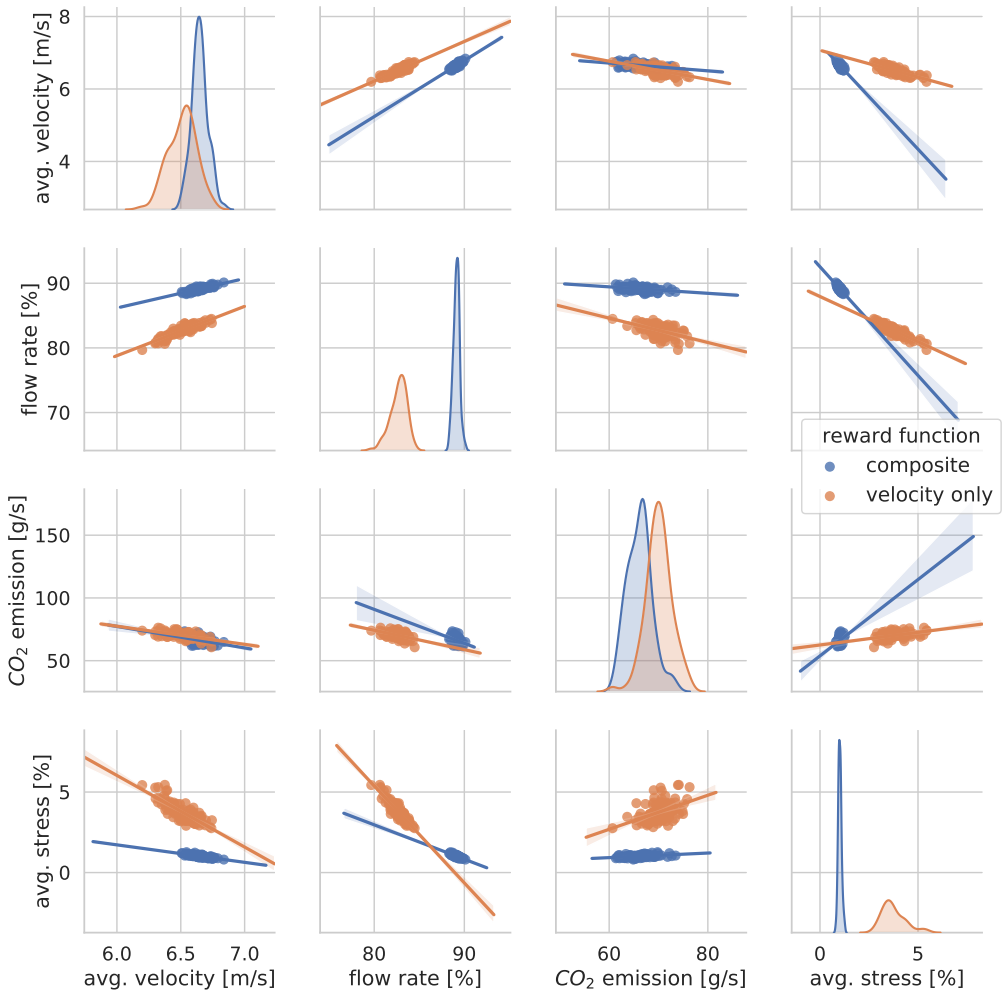


Figure C.9.: Comparison of the results for the two different reward functions. In addition to the four elements of the composite reward function, the figure also shows the average time that each vehicle spends waiting at an intersection during its trip and the average time that vehicles need to traverse the entire network.