DEEP REINFORCEMENT LEARNING FOR TRAFFIC SIGNAL CONTROL

DAVID SANWALD
341204

Decisions Under Uncertainty

Technische Universität Berlin
Fakultät V
Institut für Land- und Seeverkehr
Verkehrssystemplanung und Verkehrstelematik

David Sanwald: *Deep Reinforcement Learning for Traffic Signal Control,*
Decisions Under Uncertainty, © July 2018

REFEREE:
Professor Doktor Kai Nagel
SUPERVISOR:
Theresa Thunig
LOCATION: Berlin

TIME FRAME: July 2018

ABSTRACT

This thesis investigates the use of reinforcement learning methodology for traffic control. It focusses on the use of reinforcement learning to operate traffic lights on signalized intersections. It seeks to explain the current state of the field of applied reinforcement learning for traffic control. Furthermore, it attempts to demonstrate how the training process of reinforcement learning algorithms could be approached to move beyond the black-box view, that is still common in recent reinforcement learning traffic control publications. In the first part, this thesis introduces the necessary theoretical background to discuss the field's relevant publications in the second part. After that, the implementation of a traffic signal control algorithm demonstrates how to diagnose the training process of a traffic signal controller. The thesis shows how applied reinforcement learning differs from general reinforcement learning and why the application to traffic signal control is especially challenging.

ZUSAMMENFASSUNG

Diese Arbeit untersucht den Einsatz von Methoden des Reinforcement Learning für die Verkehrssteuerung. Der Fokus liegt auf dem Einsatz von Reinforcement Learning zur Steuerung von Ampelanlagen. Die Arbeit gibt einen Überblick über den aktuellen Stand von Reinforcement Learning für die Verkehrssteuerung. Darüber hinaus werden an einer Modellimplementation verschiedene Methoden demonstriert, um den Trainingsprozess von Agenten zu überwachen, zu diagnostizieren und zu verstehen und somit Einsichten zu gewinnend, die über die verbreiteten Black-Box Sichtweisen hinausgehen. Im ersten Teil dieser Arbeit wird der notwendige theoretische Hintergrund vorgestellt, um im zweiten Teil die relevanten Publikationen zu diskutieren. Dem folgend zeigt die Implementierung eines Ampelsteuerungsalgorithmus, wie der Trainingsprozess eines Ampelsteuergerätes zu diagnostizieren ist. Die Ergebnisse der Arbeit veranschaulichen, wie sich angewandtes Reinforcement Learning von der Grundlagenforschung in diesem Gebiet unterscheidet und warum die Anwendung zur Ampelsteuerung bis heute weitestgehend unerschlossen ist.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

ADAM	Adaptive Mement Estimation
ANN	artificial neural network
ATSC	adaptive traffic signal control
CNN	convolutional neural network
CPU	central processing unit
DDQN	Double Deep Q-Learning
DP	dynamic programming
DQN	Deep Q-Learning
GPU	graphics processing unit
MDP	Markov decision process
MP	Markov process
MRP	Markov reward process
MSE	minimal squared error
RL	reinforcement learning
RLTSC	reinforcement learning traffic signal control
SGD	stochastic gradient descent
TD	temporal difference
TSC	traffic signal control

SYMBOLS

Notation	Description
a	action.
α	step size parameter for updates.
D_t	set of collected experience tuples at time t.
ϵ	exploration parameter, probability to select random action.
e_t	experience tuple at time t consisting of s_t, a_t, r_t, s_{t+1}.
γ	discount factor for future rewards.
$\mathcal{L}(\theta)$	parametrized loss function for value function approximation.
π^*	optimal policy, behavior that maximizes expected return.
π	policy, descibing behavior.
$Q^*(s,a)$	true value of taking action a in state s and then following policy π.
$Q^\pi(s,a)$	value of taking action a in state s.
r	reward.
s	state.
θ	parameter of value function approximation.
$Q(s,a;\theta)$	parametrized approximation of the value function.
$\tilde{Q}(s,a;\theta^-)$	parametrized approximation of the target value function.
θ^-	parameter of the target function approximation.
$V^\pi(s)$	value of being in state s under policy π.

Part I

FUNDAMENTALS

1 INTRODUCTION

Traffic control is not a new problem but the issue presents itself with increasing urgency:

Traffic volume is subject to ongoing fluctuations, some periodic, many chaotic and impossible to predict. However the streets it flows trough are rigid and unable to adapt their capacity. Therefore, we need solutions to direct and control traffic in a way so that it is able to adapt to changing traffic conditions and volumes. The same can be said about the field of RL, that has existed for years but only recently gained the attention of a broader audience. Much of its recent rise in popularity goes back to the publication of "Human-level control through deep reinforcement learning" by Mnih et al. [17]. This publication has directly inspired multiple studies that attempt to apply the findings of Mnih et al. [17] to traffic signal control (TSC).

This thesis is concerned with three major issues regarding reinforcement learning traffic signal control (RLTSC):

STATUS QUO The first emphasis lies on the investigation of the current state of RLTSC. Until now the application of RL to traffic signal control has been limited to computer simulations. Other approches to adaptive TSC are currently tested in pilot studies on real intersections.

FUTURE For this reason it is important to ask how far away RLTSC is from being able to contribute to the solution of the traffic control problem. In addition, it is highly relevant to identify the obstacles that RLTSC has to overcome in order to do so. It is important to investigate to what extent recent breakthroughs in pure RL can contribute to the progress of RLTSC.

METHOLOGY The third goal of this thesis is to explore the general training process of RLTSC agents and to investigate similarities with and differences to usual benchmark problems commonly used in pure RL.

The thesis chooses to approach the issues described above by synthesizing a practical approach by means of the necessary theoretical background.

Chapter 2 introduces the necessary background to account for the fact that RLTSC is still closely tied to the current state of pure RL.

Chapter 3 the current state of RLTSC and how it is affected by developments coming from pure RL.

Chapter 4 chooses a practical approach to complement the theory centered approach of the previous chapter. It implements a traffic signal control agent to explore and demonstrate practical aspects of RLTSC. This provides the foundation for the examination of the current state of RLTSC and how it is affected by developments coming from pure RL.

The synthesis of both approaches finally allows to discuss the current state of RLTSC as well as its possible future.

2 THEORY OF RL

This chapter is divided into two parts: The first part provides the necessary background on how a variety of concrete problems can be abstracted into a common formulation, which is the core of reinforcement learning. The second part describes how algorithms can learn to solve those problems.

2.1 FORMULATION OF RL PROBLEMS

The application of RL methods to TSC requires a sound understanding of RL in general. It is important that RL is not a particular class of algorithms but a formal framework used to describe a wide range of different problems and to expose their common features [19, p. 4].

RL is about decisions and their long-term consequences. It is concerned with the problem of how to decide in the face of uncertainty. Therefore, RL formulations do not only deal with the description of decisions about learning but also with the reasoning about those decision sequences and their outcomes over time.

2.1.1 *Describing Actions and Consequences*

This section describes the interface between agent and environment, the two mandatory components of a reinforcement learning system. Subsequently, it is illustrated how the Markov decision process (MDP) specifies the interaction via that interface.

2.1.1.1 *Agent Environment Interface*

The acting entity, named the agent, interacts with its surrounding, the environment. The agent decides upon actions, which constitute the first element of the agent environment interface. Those actions are able to alter the state of the containing environment. After each action the agent receives the respective state signal and therefore knows about the immediate outcome of a chosen action. It is important to note that the reward signal is always scalar in contrast to the state signal, which can be potentially high dimensional [20, p. 52]. Hence, the reward signal provides purely evaluative feedback. Furthermore, it only reflects the immediate value of its particular state and does not contain any information about long-term implications of a specific action. For reasons of convenience the RL problem is described in discrete time. This allows to use discrete sums and simple probabilities

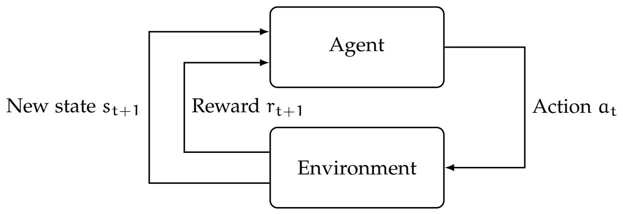


Figure 2.1: Agent evironment interface [20].

instead of integrals and probability densities. If necessary, it is possible to generalize to continuous time. This sequence is illustrated in Figure 2.1.

2.1.1.2 *Markov Decision Process*

While it is necessary to determine how the agent interfaces with its environment, this is not enough to describe the learning *process*. Learning is an optimization, which happens over a sequence of actions and the resulting state/reward signal feedback. The MDP serves as the fundamental framework to describe agent environment interaction over time. It is an extension to the Markov process (MP) that adds the notions of reward and action [24, p. 10]. An MDP consists of the quadruple $\langle \mathcal{S}, \mathcal{P}, \mathcal{R}, \mathcal{A} \rangle$, where

- $\mathcal{S}$ is a set of states,
- $\mathcal{P}$ denotes the state transition matrix defining the probabilities of some possible next state s' given any state s, $\mathcal{P}^a_{ss'} = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a]$,
- $\mathcal{R}^a_s = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$ the reward, extends the MP to the Markov reward process (MRP),
- $\mathcal{A}$ is a set of actions.

This MDP can sometimes be amended with a discount factor $\gamma \in [0, 1]$ for the reward. Furthermore, the sequence of states has to be memoryless. Thus, for all states of an MDP

$$\mathbb{P}[S_{t+1} | S_1, \dots, S_t] = \mathbb{P}[S_{t+1} | S_t] \tag{2.1}$$

has to be true [5, p. 234]. Put differently: *Given the present, the future is independent of the past.*

The memorylessness implies that the information each state signal contains is sufficient to decide optimally in this particular state [24, p. 11].

2.1.1.3 *Reward Signal*

Choosing the action which maximizes the expected next reward does not always lead to useful behavior. For example, a chess player sometimes has to pass on the opportunity to take her opponent's pawn because this opens up the chance to take a much higher valued piece a few turns later. So it is necessary to formulate a goal using the reward signal. The reward hypothesis states that this is possible for every goal:

> That all of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward) [19, p. 42].

2.1.2 *Reasoning About Actions and Consequences in the Future*

The previously introduced MDP makes it possible to describe sequences of actions, the resulting states and their rewards. To analyze sequential decisionmaking beyond immediate outcomes one has to be able to express the agent's behavior and the desirability of specific states. The necessary requirement will be introduced now.

POLICY The probability distribution over all actions given a particular state is called *policy*. Thus

$$\pi(a|s) = \mathbb{P}\left[a|s\right] \tag{2.2}$$

is the probability of taking action a in state s. It determines the agent's behavior at a given time [20, p. 6].

VALUE FUNCTIONS The policy, which maps states to actions, has to be combined with the optimization of the expected reward [24, p. 15]. This is achieved by expressing the goodness of a state by a value. The value of being in a particular state s given policy π is given by the *state-value function*

$$V^{\pi}(s) = \mathbb{E}[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots |s]\,, \tag{2.3}$$

where $\gamma \in [0, 1]$ discounts rewards that lie further in the future. The state value represents an estimation of how desirable it is to be in this state s following policy π regarding the optimization of the expected return. Further the *action-value function*

$$Q^{\pi}(s, a) = \mathbb{E}[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots |s, a]\,, \tag{2.4}$$

which is the value of a particular state s and some action a, represents the desirability of taking action a in this specific state s, when following policy π after taking this action. So these value functions link the concept of policy to the expectation of returned reward, optimal behavior seeks to optimize.

Value functions decompose recursively into the Bellman-Equations. The decomposition of the Q-function

$$Q^{\pi}(s, a) = \mathbb{E}[r + \gamma Q^{\pi}(s', a')|s, a] \tag{2.5}$$

shows that the value can be expressed as the sum of the immediate return and the Q-value of the next state.

2.2 SOLUTION OF RL PROBLEMS

RL problems vary widely in many aspects but there are several obstacles that appear in all of them:

DELAYED REWARDS RL is often confronted with the trade-off between immediate gratification and possible long-term rewards.

LARGE STATE SPACES In large or continuous state spaces RL is confronted with problems where agents never get to visit the same state twice.

THE NEED FOR GENERALIZATION Closely related to the prior point is the need for generalization. Agents need the ability to approximate never visited state spaces by approximating them by combining previous experiences.

EXPLORATION-EXPLOITATION DILEMMA Most of the time the agent has to decide whether to exploit existing knit owledge or to sacrifice immediate reward and to explore its environment.

Temporal difference (TD) methods, like Q-learning, which will be addressed next, deal with the first issue in a specific way. Generalization and large state spaces will be adressed in paragraph 2.2.2.

2.2.1 *Learning for Small State Spaces*

Having established how the value of a state or the value of taking an action in a particular state can be expressed, these concepts make it possible to understand how an optimal behavior can be learned. The optimal value function

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) = Q^{\pi^*}(s,a) \tag{2.6}$$

expresses the maximum value that can be earned following a certain policy. So it is obvious that knowing Q^* is always equal to knowing the optimal policy because acting optimally can be expressed as

$$\pi^*(s) = \operatorname*{argmax}_{a} Q^*(s,a)\,, \tag{2.7}$$

which means nothing more than always taking the action corresponding to the highest Q-value.

The optimal value function decomposes in the same manner as the value function and yields

$$Q^*(s,a) = \mathbb{E}_{s'}[r + \gamma \max_{a'} Q^*(s',a')|s,a]\,, \tag{2.8}$$

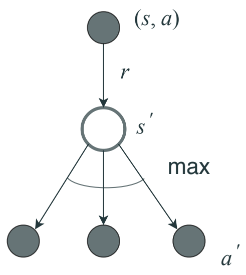


Figure 2.2: The backup step of Q-learning.

Algorithm 1 Q-learning algorithm

```
Initialize $Q(s, a)$ arbitrarily
Initialize $S$
repeat
    Choose $A$ from $S$ using policy derived from $Q$
    Take action $A$ observe $R$, $S'$
    Choose $A'$ from $S'$ using policy derived from $Q$
    $Q(S, A) \leftarrow Q(SA) + \alpha[R + \gamma \max_a Q(S', a) - Q(S A)]$
    $S \leftarrow S'$
until $S$ is terminal
```

the Bellman optimality equation.

The recursive decomposition of the value function is directly exploited by the Q-learning algorithm, which updates the estimated Q-function [20] towards a target term that follows directly from the Bellman decomposition. The Q-learning update can be written as

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \underbrace{[\underbrace{R_{t+1} + \gamma \max_a Q(s_{t+1}, a)}_{\text{target}} - \underbrace{Q(S_t, A_t)}_{\text{prediction}})]}_{\text{TD-Error}}, \tag{2.9}$$

where α represents the update's step size.

The target term $R_{t+1} + \gamma \max_a Q(s_{t+1}, a)$ contains the $\max_a$ over all possible actions, which can be different from the action the agent actually chose. This renders Q-learning to be off because the learned policy can be different from the followed policy.

Figure 2.2 shows the so-called backup diagram, which visualizes the telescopic nature of Q-learning.

The algorithmic notation in equation 1 follows directly from the Q-learning update step in equation 2.9.

2.2.2 *Learning for Large State Spaces*

Methods like the just described tabular Q-learning represent the value function as a table, where each state-action pair is separately stored and only updated on visit. The number of action values is the cartesian product of the set of all states with their associated actions. Because of that, tabular representations may work well for model problems like gridworlds that only have a small number of states, however, many desirable applications of RL have humungous combinatorial state spaces. The amount of memory required to store the tabular representation of such Q-functions might be the first issue that comes to mind. Yet the impossibility to visit all state action pairs a sufficient number of times to converge to the true value function is even more problematic. Tabular methods only update the currently visited state, regardless of the similarity of other state spaces.

To learn sucessfully from experience in large state spaces the agent has to be able to generalize. This means that it has to use previous experience to estimate a new, previously unseen state [19, p. 58].

2.2.2.1 *Naive Value Function Approximation*

The replacement of a tabular representation of a value function by an artificial neural network (ANN) is as straightforward as it is obvious. The Q-function is nothing more than a function mapping from state action pairs to a scalar value. The universal function approximation theorem states that feed-forward ANNs are capable of representing any continuous function by function composition if their activation function is non-linear. This property has been shown for the first time in the special case of sigmoid activation functions by Cybenko [3]. If the value function is approximated with the output of an ANN, the value function is parametrized by the network's weights, θ, the approximation is

$$Q(s, a, \theta) \approx Q(s, a)\,, \tag{2.10}$$

The loss function, here the expected mean-squared TD error, becomes a function of the network's weights as well:

$$\mathcal{L}(\theta) = \mathbb{E}\left[\left(r + \gamma \max_{a'} Q(s', a', \theta) - Q(s, a, \theta)\right)^2\right]\,. \tag{2.11}$$

The differentiation with respect to those weights leads to the following Q-learning gradient:

$$\frac{\partial \mathcal{L}(\theta)}{\partial \theta} = \mathbb{E}\left[\left(r + \gamma \max_{a'} Q(s', a', \theta) - Q(s, a, \theta)\right)\frac{\partial Q(s, a, \theta)}{\partial \theta}\right]\,. \tag{2.12}$$

The objective function can be optimized using stochastic gradient descent (SGD) or one of its many variants.

2.2.2.2 *Convergence Issues*

Applying the described, naive variant of Q-learning using an ANN to approximate the value function will more often than not fail. It will rarely converge to a stable value function that leads to a predictable policy. The reason for this is what [20, p. 215] call the Deadly Triad. This term denotes the stability issues that are inevitable if all of the following three elements occur:

BOOTSTRAPPING Updating estimates using targets, which are themselves then again estimates, is one of the key properties of temporal difference learning. Without bootstrapping dynamic programming would not be applicable and thus TD methods would lose their computational efficiency.

OFF-POLICY LEARNING Finally, off-policy learning, which learns from samples, which are generated by a policy different from the learned policy, makes it possible to explore by following a random policy while learning a different target policy. Not only does this mean that off-policy methods often converge to better policies, it also allows to learn from states, that depend on actions outside the learned target policy.

FUNCTION APPROXIMATION To scale learning to applications beyond small model problems, function approximation is, as just explained, mandatory. Furthermore, generalization constitutes one of the most exciting stepping stones to human-like artificial intelligence.

2.2.2.3 *Methods to Stabilize for Off-Policy Learning*

Only recently it became possible to stabilize the combination of Q-learning with function approximation using ANNs. Mnih et al. [17] developed two novel modifications to Q-learning, which stabilized the learning process enough to harness the capabilities of a convolutional network to deal with high dimensional structured input, like image data as a function approximator for Q-learning.

This made it possible to train an agent that achieved results comparable to the scores of a human player on 29 out of 49 games. It is impoartant to note, that just raw pixel input was fed as state signal to the agent and that there was just one agent trained on all games, so only one network had to approximate the action-value function of all played games.

EXPERIENCE REPLAY Mnih et al. [17] present Experience Replay. Instead of instantly using the tuple $e_t = (s_t, a_t, r_t, s_{t+1})$, it gets stored in a buffer, the replay memory. The agent learns at each time-step t from a dataset $D_t = \{e_1, \dots, e_t\}$ from which random sample batches for the update, $(s, a, r, s') \sim U(D)$, are drawn.

To update the action value function at each step a small batch of saved transition-tuples is sampled from the buffer and used to update the network weights, so the mean squared error on which we perform SGD with respect to the network weights becomes

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,a,r,s')\sim D}\left[\left(r + \gamma \max_{a'} Q(s', a', \theta) - Q(s, a, \theta)\right)^2\right]. \quad (2.13)$$

Experience Replay has three major advantages:

SAMPLE EFFICIENCY If after each step a sample is immediately used to perform only one update, the sampling efficiency is very low, Experience Replay allows to reuse samples for multiple weight updates and thus enables better sample efficiency.

SAMPLE DECORRELATION Consecutive samples are highly correlated, so learning efficiency is greatly reduced. Drawing random samples from the replay memory reduces the correlation because updates are now performed on samples from non-sequential steps.

DECOUPLING The simplistic approach to function approximation, described in paragraph 2.2.2.1, leads to updates, where the targets, $r + \gamma \max_{a'} Q(s', a', \theta)$ that update the weights also depend on the very same weights. This causes feedback loops between the forward pass to generate the targets and the backward pass to update the weights.

FIXED TARGET NETWORK The use of an additional target network with fixed weights θ^- to compute the Q-learning targets introduces temporal buffering between weight updates and target generation. Periodically, every n steps, the weights of the online network are copied to the target network. The dampening effect of the temporal lag between target and online network reduces the danger of feedback loops and has a stabilizing effect on the learning process. Updating the value function after each time step changes the policy, which again affects the distribution of the next training samples. This can lead to feedback loops, diverging value functions and complete destabilization of the learning process. Experience Replay smoothes the distribution of samples that update the policy and dampens the effect of policy updates on the distribution of learning samples.

FURTHER IMPROVEMENTS To increase the efficiency another modification to common Q-learning is used. Instead of passing every state-action pair separately into the network to get back the corresponding Q-value, it is possible to pass only the state, while the network outputs all Q-values. The modification depicted in Figure 2.3 allows estimating the Q-values of all actions in a particular state with only a single forward pass through the network.

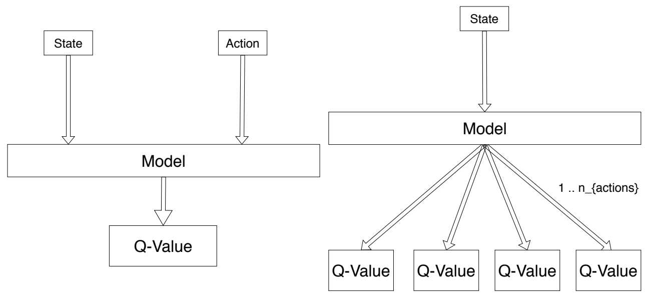


Figure 2.3: Modification to the forward pass to increase efficiency.

THE FINAL DQN ALGORITHM Algorithm 2 shows the sequence of steps executed by the DQN agent.

Figure 2.4 illustrates how the individual elements interact during the training of a Deep Q-Learning (DQN) agent.

DDQN TO REMOVE UPWARD BIAS Q-learning algorithms suffer from an upward bias in their Q-function estimation caused by the max operator in the update step. A slight modification of the update step, where the online network is used to select the action of the Q-value targets is able to reduce the upward bias. This increases the learning stability in almost all benchmarks [21].

The update step of Double Deep Q-Learning (DDQN) can be written as

$$Q(s,a) \leftarrow r + \gamma Q(s', \underset{a}{\operatorname{argmax}}\, Q(s',a), \theta^-)\,. \tag{2.14}$$

The original Q-learning [8] required a second value function compared to normal Q-learning. DDQN is able to use the target network of DQN to remove the upward bias. Thus, the increase of time and space complexity compared to basic DQN is minimal.

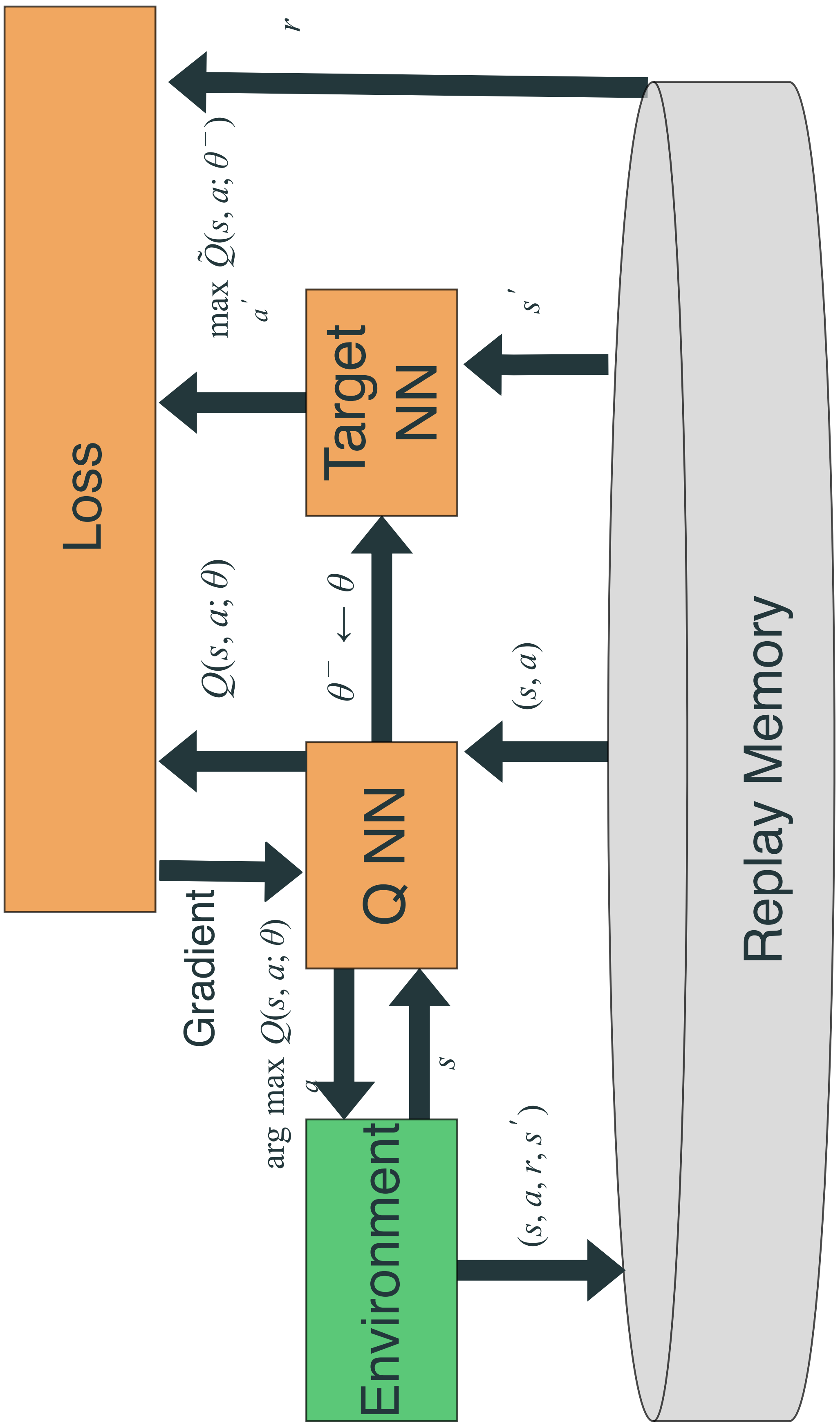

Loss
Target NN
Q NN
Environment
Replay Memory
Gradient
r
$\max_{a'} \tilde{Q}(s, a; \theta^{-})$
$Q(s, a; \theta)$
$\theta^{-} \leftarrow \theta$
s'
(s, a)
$\arg\max_{a} Q(s, a; \theta)$
s
(s, a, r, s')

Figure 2.4: Components of complete DQN architecture.

Algorithm 2 Pseudocode for DQN step sequence.

```
initialize replay memory $D$ with size $N$
initialize action-value function $Q(s, a)$ with random weights $\theta$
initialize target action-value function $\tilde{Q}(s, a)$ with random weights $\theta^- = \theta$
**for** episode=1, M **do**
    observe start state $s_1$
    **for** t=1,T **do**
        with probability $\varepsilon$ select random action $a_t$
        otherwise select $a_t = \operatorname{argmax}_a Q(s_t, a_t, \theta)$
        take action $a_t$, observe new state $s_{t+1}$ and reward $r_t$
        store transition $(s_t, a_t, r_t, s_{t+1})$ in replay memory $D$
        get random batch $(s_j, a_j, r_;, s_{j+1})$ from replay memory $D$
        $y_i = \begin{cases} r_j & \text{if terminal} \\ r_j + \gamma \max_{a'} \tilde{Q}(s_{t+1}, a', \theta^-) & \text{otherwise} \end{cases}$
        execute gradient descent on $(y_i - Q(s_j, a_j, \theta))^2$ with respect to network weights $\theta$
        every C steps copy network weights so target network so that $\tilde{Q} = Q$
    **end for**
**end for**
```

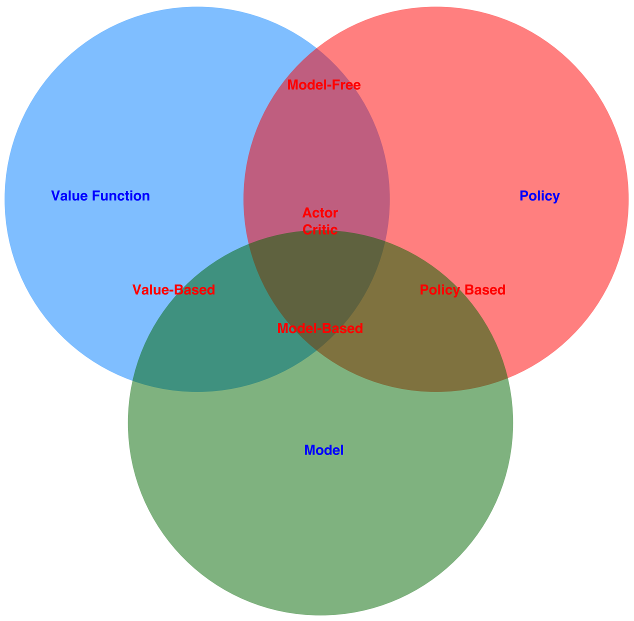


Figure 2.5: Major building blocks RL [20].

2.2.3 *Alternatives to Value-Based Methods*

This thesis focusses on value-based methods. However, it is important to know about alternative reinforcement learning methods and their relation to the value-based methods that are at the center of this thesis. As Figure 2.5 shows, the different areas of RL overlap. In addition, hybrid methods have recently become increasingly popular. Therefore, this chapter closes by giving an overview of the complete landscape of RL methology.

VALUE-BASED METHODS As previously discussed, the value function represents the expected value for a given fixed policy. It is also possible to optimize this value function and find the function with the highest value for some policy. So learning takes place in the value space instead of the policy space.

POLICY-BASED METHODS If the policy determines which action the agent takes in a particular state, improving the policy improves the optimality of the agent's behavior. So policy-based methods directly update the policy from experience.

MODEL-BASED METHODS The last learning approach learns a model of the actual environment. If the agent learns the actual transition probabilities between states for all possible actions from experience, or at least has some estimate of them, the agent can evaluate different policies by considering possible future situations before they actually happen [20]. Thus, it can find a better policy by *simulating* different policies. This approach depends very much on the problem domain.

Part II

APPLICATION

3 STATE OF RL TRAFFIC SIGNAL CONTROL

The greater part of this chapter will not mention traffic signal control at all. Why is that?

While the field of RL has continuously evolved over the last decades, its achievements were limited to small model problems like gridworlds. Only recently RL produced results outside of those small model problems.

The idea to apply RL to adaptive traffic signal control (ATSC) has been explored in plenty of publications over the last decades. Although most authors claim to have had promising results, this thesis could not find examples of studies with successful applications outside of simulations.

So it is not unreasonable to believe that RLTSC inherited a large part of its problems from genera RL. While pure RL struggled to overcome its limitations, how could it be possible for RLTSC to overcome those limitations?

RL is still a young field and its current state is shaped by achievements after Mnih et al. [17]. For the application of RL to ATSC the current state of general RL is much more relevant as it might be for the application of a more mature field. RLTSC is currently not ready to be applied outside of simulations. Therefore, it is necessary to estimate the future potential of RLTSC. Thus, an understanding of general RL, its direction, its strengths and weaknesses becomes even more important.

Therefore, the following chapter begins with a short overview over RLTSC but highlights a group of recent publications, all of which share a similar architecture based on the work of Mnih et al. [17]. Then a detailed discussion of the publication that introduced DQN gives necessary background to understand the current state of applied and general RL. Finally, the importance of state is explained and the impact of DQN and the use of convolutional neural networks (CNNs) on the design of the state signal is shown.

3.1 OVERVIEW OF RL TRAFFIC SIGNAL CONTROL

Abdulhai, Pringle, and Karakoulas [1] presented a Q-learning agent for an isolated intersection and their experiments had 'ecouraging' results. Because controlling a single isolated intersection is far less relevant to real-world TSC [16] many authors also extended their efforts to multi-agent approaches, the earliest dating back to 2000 [23]. Mannion, Duggan, and Howley [16] published a comprehensive survey of the most important multi-agent approaches to date. Every publication in the field of RLTSC claims to have had at least promising results. The following chapter tries to understand why nearly twenty years of publications about RL applied to ATSC did not produce any study or experimental real-world application outside of various simulations. To find possible reasons for the state of RLTSC, first, the current state of the art of value-based off-policy learning is reviewed to identify intentions and methodology. After that, the chapter shows how those findings enabled novel methods for representing the state of an intersection.

Q-learning was introduced by Watkins [22] and the interest in RL lead to several explanatory surveys by Keerthi and Ravindran [10] and Kaelbling, Littman, and Moore [9]. But despite continuous theoretical progress, it has not been possible to apply RL to real-world problems.

3.2 DEEP REINFORCEMENT LEARNING

The authors developed a novel approach that allowed to use CNNs to approximate the value function of an RL agent. CNNs have already been used successfully in supervised learning settings. The publication of DQN is especially relevant in the context of this thesis for several reasons:

1. The publication reached a wider audience and directly inspired multiple authors to apply its findings to TSC.
2. It was the first publication that yielded promising results on domains beyond low dimension model problems.
3. The publication on the front page of a highly competitive journal like Nature suggests that it could be a good benchmark for the status quo of cutting edge research on RL.

There are three areas, where the publication contains important insights for RLTSC:

1. The sucess of DQN is based on recent technological progress.
2. The authors emphasized the generality of their algorithm as opposed to its absolute performance.
3. The authors chose a simple, deterministic environment to produce unambivalent, transparent results.

3.2.1 *Relevance and Impact of DQN*

The authors of Mnih et al. [17] developed a single algorithm that could learn to succeed in a wide range of tasks. Their algorithm combined Q-learning, a well-known, off-policy TD-learning method, with a CNN for function approximation.

ANNs have already been successful in supervised settings, but their use as function approximator for off-policy RL has always been highly unstable. Through a combination of small modifications to the standard procedure, the authors brought the updates to the value function closer to a supervised offline setting and stabilized the learning process, while retaining most of the desirable qualities of online learning.

The effectivity of CNNs made it now possible to train the agent using high dimensional state representations and to successfully approximate the value function of a large number of different tasks with one single set of network weights.

The authors trained the agent on 49 classic arcade games of the Atari 2600 platform. Because the games employed a wide range of scoring functions, the score updates were clipped outside a range of -1 and $+1$ to represent the reward signal. The state representation consisted of 84×84 pixel screen captures stacked by fours to account for partial observability. One agent with a single set of network weights was trained on all 49 games. In 29 of them it reached at least 70% of the points of a human test player.

3.2.2 *Three Key Insights*

The level of visibility and impact of the publication makes it valid to extrapolate some main characteristics which can be considered exemplary for significant parts of recent research in the field of RL.

3.2.2.1 *Recent Technology and DQN*

In recent years ANNs became very successful. They performed increasingly well on a wide variety of tasks. Despite some improvements of their architecture, the main factor for their comeback from near irrelevancy lies in the availability of cheaper graphics processing units with more and more cores. Since their training is highly parallelizable neural networks have profited more than other methods from those technical advances. Their online training capabilities and their increasing effectivity on larger numbers of samples also played an essential role in their recent popularity.

Function approximation in combination with off-policy TD could not be stabilized. Therefore, not only ANNs but also many related recent technological achievements have been unavailable for RL. The sudden effectivity of Q-learning has to be understood as the result of making it possible to profit from already well-established developments in other fields. The sudden capability to train with high dimensional state representations was also a requirement for the next point.

3.2.2.2 *Generality*

The authors explicitly mention generality as the most significant quality of RL algorithms. This high prioritization of generality has a direct influence on the structure and methodology of their work. Their decision to clip the reward to be between -1 and $+1$ is an excellent example of this influence: Many of the games are very different in the scale of their scores. The authors could have normalized the score of each game individually. Another possibility could have been to adapt the agents learning rate from game to game. Instead, the authors accepted that clipping the rewards made it impossible to differentiate rewards outside the clipping range. It is very likely that this had a significant negative impact on the agent's final performance. Yet the authors chose the solution which preserved the most generality and introduced the least amount of prior knowledge to the algorithm.

The use of feature vectors was unavoidable for two reasons:

1. For a long time, it was impossible to stabilize function approximation with many effective models like artificial neural networks.
2. The computing power was not enough to allow effective training on a large number of samples. For that reason, it was necessary to rely on feature vectors to improve sample efficiency.

It is important to bear in mind that the construction of feature vectors out of higher dimensional states only reduces the absolute amount of information. If the original states cannot be expressed inside the Markov framework, this also applies to all of their feature representations. In contrast, the construction of features can render principally learnable states impossible to learn.

Therefore, it is justified to see the dependence on feature vectors as an artificial limitation to the generality of RL algorithms. It is reasonable to believe that RL is most effective where flexibility and generality are more important than pure performance.

3.2.2.3 *Simplicity and Transparency*

The current state of RL demands, that the learning process cannot be treated as a black box. The chosen arcade games expose several properties, which facilitate the analysis of RL methodology beyond the comparison of final scores.

- The individual games are episodic, which makes it easier to assess the performance of the algorithm.
- Even pure visual observation of the agent's behavior provides information on the current state of learning

- Most environments make it possible to gain meaningful intuition about important properties like memorylessness or partial observability
- To make results comparable it is not enough to initialize the generation of pseudorandom numbers using a seed state. In environments where actions can trigger cascading behavior, the performance of algorithms has much higher variance. Most games avoid cascading behavior and states from which the player cannot recover by design.
- The environments are different in their requirements profile. While some have a sparse reward structure, others require more long-term planning or different strategies in very similar states.

3.3 POST DQN STATE REPRESENTATIONS

Recent technical developments, as well as current scientific achievements, had their most significant impact on state representation in RL. But the following section aims to explain how the availability of CNNS changed the way RLTSC is able to design the state signal. The next section shows an approach to state representation common to all recent DQN-based publications [18, 6, 7, 14].

RL and even more so RL applications for TSC did not provide any substantial progress for a long time. So it is reasonable to investigate the significance of recent developments for traffic signal control applications. It is useful to recap that RL's central abstraction is the Markov Decision Process and that recent research and technical progress directly impact the state element of the MDP's tuple while it does not have any direct implications for the reward signal. For many of RL's underlying mathematical abstractions, convergence proofs or error bounds are available. But it remains challenging to map analytical results to less ideal real-world environments. In many cases, it is difficult to decide if states or their representations violate assumptions like the Markov property.

Furthermore, proof of convergence if the number of visits for each state approaches infinity does not imply it is possible to learn enough to be used within any practical time frame. On the other hand, practical scenarios can violate some necessary assumption for convergence guarantees to some extent and still perform well enough for some applications.

Lower dimensional feature representations of high dimensional direct state input have much lower computational costs. It is also possible to provide information, that is considered necessary in an explicit form. This can improve sample efficiency. If the task is to balance an upright pole, the extracted angle of the pole might yield faster results than an image representation of the whole pole.

Under the assumption that the original state input does not violate the Markov property the only hard requirement is that the feature representation also preserves the Markov property. For instance, consider an object with a constant velocity. Its position depends on its position at $t-1$, only given its current position at point in time t. The Markov property can be recovered explicitly by including the velocity, or implicitly by describing the state as a sequence of two sequential positions.

3.3.1 *Representations of Signalized Intersections*

It is intuitive that the position of road users is the core information to control an intersection in an efficient way. This means that the state signal has to contain the position or some representation derived from it. So the different blocks an of a state signal fall in only three categories:

1. properties containing information about the position of road users
2. properties that contain information about the change of position of road users
3. properties that are a combination of both

If some property P of the state signal is not constant over time in such way that

$$\mathbb{P}[P_{t+1}|P_1,\ldots,P_t] \neq \mathbb{P}[P_{t+1}|P_t], \tag{3.1}$$

the state has to contain derivatives of properties or sequences of those properties over time.

3.3.1.1 *State Signals of Traffic Signal Phases*

Into which category does the traffic signal state fall? The velocity of the road users at the intersection is, in the simplest case at least its magnitude, apparently also a function of the traffic light state. The probability of the next phase state depends primarily on the agent's actions but, in more realistic cases, also on the current position in a cycle. Therefore it is common practice to include the time elapsed since the last signal change. The representation of the signal's actual phase differs from other properties of a signalized intersection:

The phase of a traffic light is already low dimensional, explicit and irreducible. Because of that, the increased availability of computational resources and recent advancement in function approximation did not have much impact on its state representation.

Some authors decided to encode the different phases numerically. It is not uncommon to map n different phases to $1, 2, \ldots, n$ numeric values. This approach raises some questions which usually are not addressed. The chosen feature encoding also depends on the choice of the model. Traffic light phases are not numerical features. They are not additive, yet they have a temporal ordering. If temporal ordering means it is possible to understand the traffic light phase in the context of an MDP as an ordinal variable remains unclear. In supervised learning applications it is not unusual to determine the most encoding by experiment, which is not as straightforward in unsupervised settings.

Few publications choose not to imply any ordering and interpret the phase as purely nominal. For the reasons stated above this seems to be a more sound approach. While there are many different methodologies, one-hot encoding is the simplest and most common approach: For every phase of the traffic signal a dummy feature is introduced. This dummy feature is 1, if the encoded original feature falls into another category, or 1 if it falls into the feature's category. The light states of a four-phased intersection can be encoded as follows:

$$P_1, \dots, P_4 = \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix} \tag{3.2}$$

3.3.2 *Intersection State*

Due to the lack of alternatives, queue length-based state definitions have been by far the most popular choice to represent the traffic state of intersections [16]. The following chapter shows the advantages of this recent approach of state representation, which gives the necessary background to evaluate the results of the implementation introduced in the next chapters, which does not rely on the use of a CNN.

3.3.2.1 *Matrix Representation of Intersection State*

To represent the vehicles only by the queue lengths compresses the intersection state to a single scalar number for each approach. Each approach is assigned to a specific number. Therefore, the number also contains some metric information. Since all vehicles in a queue are standing still, it contains movement information as well.

The use of CNNs makes it possible to retain the dimension of the state because the input does not have to be flattened and the intersection can be discretized into squares and represented as a matrix containing zeros for empty squares or ones if a particular square is occupied by a vehicle.

Because the state shape now retains an approximate representation of the actual state's metric, the Markov property can be preserved by stacking a second matrix containing the speed of the vehicle that is

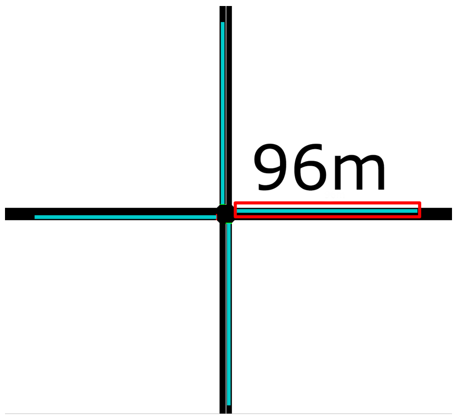


Figure 3.1: Intersection used to demonstrate the state signal construction.

1	1	1	0	1	0	0	0	0	0	0	0
0.1	0.7	0	0	0.6	0	0	0	0	0	0	0

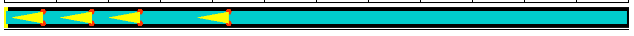


Figure 3.2: Speed and normalized position row encoding vehicle state.

occupying a particular square on top of the position matrix. Reducing the velocity, which is a vectorial property to a scalar given by its speed, is possible because the direction of a vehicle is a function of its lane.

3.3.2.2 *Advantages and Disadvantages of Matrix State Representations*

Using a matrix that preserves the intersection's metric property in its shape has a couple of advantages:

- continuous representation of vehicle speed
- inherent inclusion of all vehicles
- more local updates due to larger network sizes
- improved ability to discriminate between different states
- elegant way to achieve more Markov state signals

But there are also several significant difficulties. Some of them are listed below. Therefore, the implementation in the second half of this thesis does not rely on the use of a CNN:

- Figure 3.3 shows that the discretization is problematic if the distribution of vehicle lengths is not homogenous.
- The states are only Markov in first order approximation. More exact approximations would need higher derivatives.
- Real use cases would require a large number of sensors, drone-mounted cameras or similar technologies.

1	1	1	1	1	1	0	0	0	0	0	0
0	0.7	0.6	0.1	0.5	0	0	0	0	0	0	0

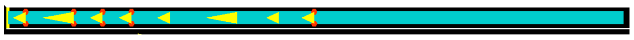


Figure 3.3: Position and normalized speed matrix with inhomogeneous vehicle lengths

- The training of CNNs is computationally expensive compared to fully connected ANNs.

4 RLTSC IN PRACTICE

This chapter starts by reviewing the elements of the MDP in the context of traffic signal control. It then turns to a practical implementation of a RL algorithm applied to traffic light control at an intersection. This implementation is used to conduct an experiment that serves two main purposes:

- The data produced by logging several properties over the training process should provide clear evidence for the agent's learning ability even without the use of CNNs.
- The implementation should demonstrate examples of how the training process can be approached beyond the common black-box view.

The simplified model of a traffic control problem sacrifices realism but provides clearer, less ambiguous results.

4.1 MDP FORMULATIONS OF TRAFFIC CONTROL DOMAINS

Applying RL is at least as much about finding a suitable formulation of the problem as it is about solving it.

The framework to formalize learning sequences is the MDP, therefore, examining its elements and finding appropriate representations for its defining elements is a good entry point to approach the application of RL.

For improved clarity the most minimal case that contains the characteristic properties of traffic signal control is considered: A single signalized intersection controlled by one agent.

Because multi-agent learning can be applied to a network of several coordinated intersections, it gives the illusion of being more relevant for actual applications.

Unfortunately, the outcomes of multi-agent approaches are so far from being mature enough to test them in real-world applications, that single agent learning is hardly less practical in real-world scenarios.

Instead, it is much harder to reason about multi-agent experiments, as game theoretic considerations add to the many moving parts even simple single agent TSC consists of.

4.1.1 *Environment*

To provide the dynamics of state transitions, which result from the agent's actions, an appropriate environment must be found or created.

Using simulations instead of actual physical environments is common in RL. It is often accepted that the simulated dynamics differ from the real-world counterpart they simulate because using a simulated environment has significant advantages:

- improved cost efficiency
- increased learning speed by running faster than the simulated environment, thus providing more samples in less time
- no safety concerns
- being able to use data, that would not be available outside a simulation, which leads to greater flexibility for construction of state or reward signals

For the evaluation of TSC schemes that are much closer to real-world applications, the realism of the chosen scenario is a crucial factor for the validity of the experiment's outcomes.

For training and developing RL algorithms, other properties of the environment become more important.

- **performance** the frequency in which the simulation can compute state transitions should be as high as possible so that the agent can be trained on many samples in a short amount of time
- **flexiblity of execution** because some learning architectures require graphics processing unit (GPU) acceleration to be practical, it might be necessary to run the experiments on computing instances provided by Amazon Web Services and likes, so it is highly desirable if the simulation has few runtime dependencies, can easily be containerized or is already available as container image.
- **interoperability** to reproduce results or to be able to benchmark and compare learning agents from different sources interoperability with different languages or other ways which allow to adapt agents engineered with different technologies is an important property
- **rich output** to be as flexible as possible the simulation should provide as many metrics, state representations and statistics as possible to construct state signals, reward functions and evaluate agents

4.1.2 *State Signal Construction*

For a long time extracting the right features out of a high dimensional state signal has been crucial to the successful application of RL.

Constructing the right features often required not only many iterations and test runs but also some domain knowledge from the engineer or researcher.

A significant disadvantage of relying too much on the construction of appropriate features was that the generality of the developed solution suffered to a great extent. Furthermore, extracting features out of the actual state representation introduces a lot of prior knowledge to the solution of the problem. Therefore, the probability of finding novel solutions that are superior to the non-RL algorithms is significantly lowered by manually extracting features from a higher dimensional state representation.

In the last year, affordable graphics processors with an increasing number of cores have become available. Highly parallelizable simple matrix computations as they are required to train neural networks have greatly benefited from those technological advancements. Furthermore, neural network architectures like convolutional networks, which share weights between local regions, have been developed. Those architectures can deal with high dimensional inputs like image data while preserving their spatial structure.

This also leads to a fairly new way to represent the state, which is heavily inspired by the architecture applied by Mnih et al. [17].

In this new architecture, the area of the intersection is discretized by dividing it into little cells. The state signal then consists of a position matrix of elements which are set to zero if the corresponding cells are empty and one if the cell is occupied. This matrix alone would not fulfill the definition of being Markov because without knowing about the velocities of the elements represented by the non-zero entries of the position matrix it is not possible to compute the probabilities of the next state given only the present state. To solve this problem one of two approaches has been used:

Stacking multiple successor states to one state representation allows reconstructing vehicle velocities. This is also the solution chosen by Mnih et al. [17].

The second solution stacks the position matrix and a value matrix resulting from the same discretization explicitly containing vehicle velocities.

Although the fraction of those 'spatially aware', high dimensional state signals can be expected to further increase shortly, lower dimensional feature vectors are still relevant. Features of earlier works consisted of a vector that contained the queue length on all four approaches of an intersection in combination with the elapsed time of the current phase [1]. Chin et al. [2] map the actual queue length of each intersection to one of four different queue length levels which results, in combination with the four phases of the simulated intersection, in just 256 possible states.

4.1.3 *Reward Function Design*

At first, constructing the reward signal could appear to be much simpler than finding a state representation. The reward is just a scalar value, in contrast to the state signal which can be high dimensional or a very sophisticatedly engineered feature vector. It is crucial for the successful application of RL that the policy which maximizes the reward over time is identical or at least close to the actual solution of the problem [20, p. 42].

It might be tempting to design the reward function by encoding prior assumptions into the reward function. One might think it could be advantageous to provide the agent guidance and lead him to possible intermediate behavior, which might be helpful to achieve the final overall goal.

Augmenting the reward function with those supplemental rewards is something that has to be avoided no matter what. Contaminating the reward function with those guesses could lead to the agent trying to optimize its policy to maximize those supplemental rewards so that the agent might not reach the overall goal at all.

If prior knowledge has to be provided to the agent, it should be done by initializing the value function to incorporate prior assumptions [20, p. 384].

The most commonly used reward functions for single agent scenarios contain the queue lengths at the current time step or the summed delays since the last time step. This reward definition is proportional to the summed queue lengths. Most researchers have found it necessary to punish longer waiting times, otherwise agents converged to a policy where very long queues formed because the agents did not discriminate between situations with equal length queues and situations involving very short and long queues but with the same overall queue length. The most straightforward solution to this problem is to use the sum of the squared queue length [1]. A less elegant approach was used by Chin et al. [2], where a penalty factor has been assigned to each phase that increased with the green time assigned to this phase.

It is arguable if those modifications to the reward function to achieve certain behavior are a case of augmenting the reward signal to introduce guesses of desirable behavior. Even though those modifications might prevent some unwanted behavior, this behavior could indicate that queue length-based rewards might not express the actual goal behavior.

4.2 PRACTICAL IMPLEMENTATION

This section describes the implementation of the example domain as well as the actual reinforcement learning agent. Most importantly it lists the main parameters of the conducted experiments.

4.2.1 *MDP Construction*

The implementation requires to address each part of the MDP. So it is necessary to find:

- reward
- state
- actions
- state transition probabilities

Those elements also guide the process of understanding and dissecting the problem. The next section presents important implementation details for them.

STATE REPRESENTATION The state signal consists of two parts, one decoding information about road users and one decoding information about the current signalized state. Both parts are concatenated to a vector of nine elements.

$$s(t) = s_{\text{signal}} + s_{\text{vehicles}} \tag{4.1}$$

Signal State The four phases of the signal are decoded via the method of effective encoding. This avoids implying numerical properties that do not apply to the traffic signal. The phases and their encoding are presented in table 4.1.

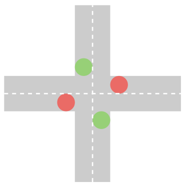
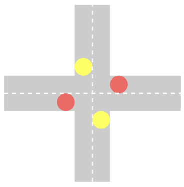
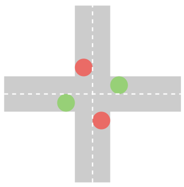
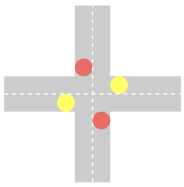
To improve the state signals compliance with the Markov assumption, the encoded current phase is concatenated with the number of seconds elapsed since the last signal change.

Road User State For each of the two approaches, the state signal extracts three properties inside a radius of 96 m around the intersection center. Those properties are:

1. the number of waiting vehicles on the approach
2. the number of all vehicles on the approach
3. the average speed of all vehicles in t

This compound information encodes not only information about the position but also some information about its derivative with respect to time. Thus, it preserves some of the state signal's Markov property.

Table 4.1: Overview of all possible traffic light states.

phase index	diagram	string	state signal	min. duration/s
1		grgr	$\begin{bmatrix}0\\1\\0\\0\end{bmatrix}$	5
2		yryr	$\begin{bmatrix}0\\0\\1\\0\end{bmatrix}$	2
3		rgrg	$\begin{bmatrix}0\\0\\0\\1\end{bmatrix}$	5
4		ryry	$\begin{bmatrix}-1\\-1\\-1\\-1\end{bmatrix}$	2

4.2.1.1 *Reward Function*

The chosen reward function is given by the summed instantaneous delay over all road users inside the simulation

$$r_t(s) = -\sum_{i=1}^{N} d_i \tag{4.2}$$

with the instantaneous delay of a single road user given by

$$d_i = 1 - \frac{v_i}{v_{max}}. \tag{4.3}$$

The instantaneous delay or time loss has already been used to control traffic lights [4].

4.2.1.2 *Action definition*

At each step, the agent gets to choose whether to keep the current signal state or to shift to the next position in the cycle. If the agent chooses to change the signal state before the minimal duration of this state is reached, the environment ignores the agent's actions. This action definition has multiple advantages:

- allows enforcing minimum phase durations
- makes it possible to introduce yellow times at the correct position in the cycle

Because the agent receives the time since the last phase change, it can learn, in which states an action results in an actual phase change.

4.2.1.3 *Transition Probabilities*

The transition probabilities between states given actions are given by the simulation of the chosen scenario and will be described separately in the following section.

4.2.2 *The SUMO Scenario*

The microscopic traffic simulation SUMO [12] is used to provide the environment and compute its state transitions in reaction to the actions executed by the agent. Table 4.2 contains the simulation parameters and their values. Next, a more detailed explanation of the scenario's main properties is given.

4.2.2.1 *The Intersection*

The intersection consists of a signalized intersection with two phases. Thus, road users cannot turn left or right. To simplify the scenario, road users enter the intersection only on two approaches. The intersection can be seen in Figure 4.1. The directions from which road users enter the intersection have been marked.

Table 4.2: SUMO scenario parameters.

	value
speed limit	19.4 m/s
vehicle length	5 m
time steps per episode	900
vehicle emission WE	$\frac{1}{30}$
vehicle emission SN	$\frac{1}{10}$

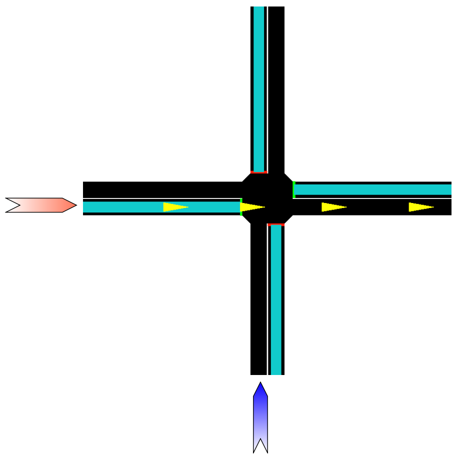


Figure 4.1: Intersection, the active approaches have been marked.

VEHICLE AND ROUTE GENERATION The vehicles are created at two sources at the start of the west to east approach and the beginning of the south to north approach. The vehicle emission of each source is defined by a Bernoulli process that decides at each time step whether a vehicle is emitted or not. There is a fixed list of vehicles and routes generated. For both approaches, a probability for vehicle emission at every given time step is defined. The demand date is created by looping through both approaches at each time step. With each iteration, a vehicle and route is written to the demand file with the defined probability.

It is important to note, that the vehicle and route definitions are generated once and used throughout all episodes.

At this early stage, it is most important to keep as many parameters fixed as possible. The agent is trained over many episodes and time steps, exploration requires to act randomly with a certain probability, which is parametrized by ε and decayed over time. This illustrates why the agent's state is always a function over all previous time steps, while it also determines future actions and their resulting states. Exploration makes it necessary that the agent's actions have to be random over a large number of steps. Thus, a small change of only one parameter can

Algorithm 3 Algorithm to generate vehicles and routes.

```
define 2 routes: SN, WE
for t<episode length do
    for all routes do
        Sample $p \sim U(0, 1)$
        if $p < p_{route}$ then emit vehicle for route at time t
        end if
    end for
end for
```

Table 4.3: Overview of used agent parameters.

Parameter	Value
replay buffer size	50 000
learning rate α	0.001
ε_{max}	0.9
ε_{min}	0.1
ε decay steps	50000
discount factor γ	0.9
activation function	leaky ReLU
target network update frequency in steps	1000
optimizer	Adam
loss function	MSE
number of hidden layers	2
size of hidden layers	16

have unforeseeable effects. The use of seed values for random number generation cannot prevent those effects.

While the decision to use the same route file throughout all episodes was necessary at this early stage, there is no doubt that further experiments with less deterministic demand and a greater bandwidth of traffic pattern have to follow.

4.2.3 *Agent Implementation*

The agent has been implemented in Python and some common supplementary scientific libraries. The ANNs make use of Tensorflow and Keras for code clarity and improved hardware efficiency. Further relevant implementation details are described next.

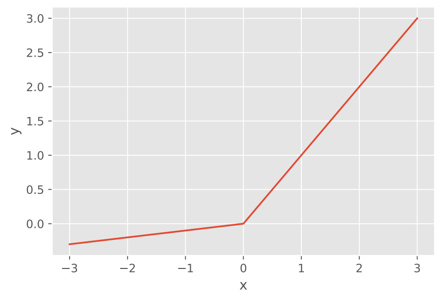


Figure 4.2: Leaky ReLU activation with augmented slope for better clarity.

4.2.3.1 *Optimization*

Over the years several derivatives of the classical SGD algorithm have been published. In supervised settings, the different optimization algorithms are very likely to converge to the same optimum if the learning rate is properly annealed over time step. In RL settings, the choice of the optimization algorithm is much more relevant, because the update path on the error surface influences the agent's behavior and thus also the data for the next updates. The chosen Adaptive Mement Estimation (ADAM) algorithm [11] combines a history of past squared gradients and previous average gradients that decay over time and which function as momentum to prevent getting stuck on local minima. The parameters used are equivalent to those published in the original paper.

4.2.3.2 *Nonlinear Activation Function*

The leaky ReLU [15] used as nonlinearity for the ANN modifies the more common ReLU function by adding a small negative slope, so backpropagated large gradient updates are less likely to cause the dying ReLU problem, which results in permanently non-responsive neurons that do not propagate gradient updates anymore.

$$y(x) = \max(0, 0.01x) \tag{4.4}$$

4.2.3.3 *Replay Memory*

The replay memory's sample capacity is directly related to the amount of decoupling of learning from samples and generating those samples. Higher capacity leads to more stability, but if the memory holds too many samples, an improving policy will not change the distribution of samples the agent learns from.

If the number of samples is too small during the first episodes, the distributions of samples in the memory and those produced by the current policy are too tightly coupled. For that reason, the replay memory is filled with samples that have been generated by a random policy, prior to the actual training.

4.2.3.4 *Exploration*

To explore the agent follows an epsilon-greedy strategy. At each time step the probability of acting greedy and just exploiting the current value function is

$$P_{greedy} = 1 - \varepsilon \,. \tag{4.5}$$

Otherwise, the agent selects a random action with uniform probability from all actions. At each step ε decreases linearly from its initial value to a minimal final value, it can be written as a function of time as

$$\varepsilon(t) = \max(\varepsilon_{max} - a_{dec}t,\ \varepsilon_{min}), \tag{4.6}$$

where a_{decay} is a decay parameter. So if the final value should be reached after a specific number of steps n_{final}, it is possible to calculate it like this:

$$a_{dec} = \frac{\varepsilon_{max} - \varepsilon_{min}}{n_{final}} \tag{4.7}$$

4.3 RESULTS

The following chapter provides visualizations of the most important quantities that have been recorded during the training. It shows important relationships between them and how they provide information about the agent's training process.

4.3.1 *Reward over Time*

The change in collected episode reward over time is the most important measure for the assessment of RL algorithms. Figure 4.3 shows the total reward per episode as defined in section 4.2.1.1 over the course of 1000 episodes.

Besides the decrease of delay over time, it is difficult to estimate the meaning of the absolute values without any context. Therefore, the graph provides additional benchmark results created by

- a completely random policy
- a traffic light actuated by the induction loops shown in 4.4
- a traffic light with fixed timing

Since the same routes file has been used over all episodes, the total reward of all policies with the exception of the random policies is actually constant over all episodes. The results of the random policy have been averaged over all episodes for clarity. Because the random

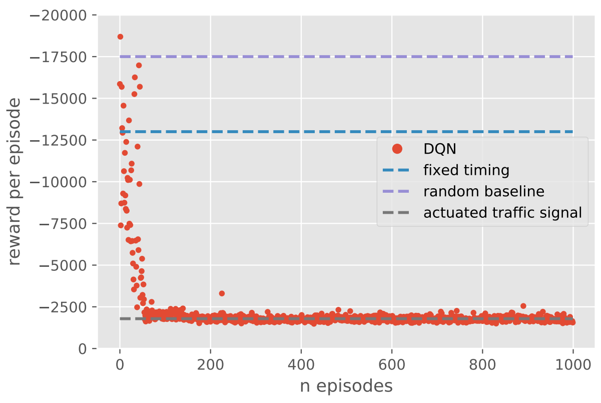


Figure 4.3: Reward over episode number.

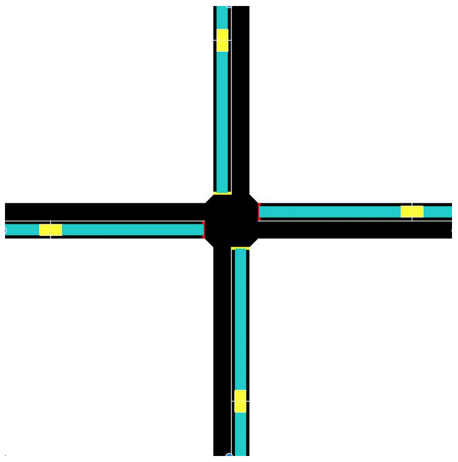


Figure 4.4: Intersection with detectors for actuated traffic light.

policy produced results with very low variance, their average contains almost the same amount of information.

Clearly, the agent's performance over time improves but the experiment's outcomes allow to draw some additional conclusions:

4.3.2 *Exploration Rate*

The episode reward improves right away, while the sampled batches, used to update its value function, almost completely contain the initial samples created by a random policy as explained in section 4.2.3.3. This suggests that improving the policy, at least during the start phase, is possible without any significant amount of exploration. The initial reward equals the reward produced by the random policy. The agent's epsilon-greedy strategy is roughly equivalent to the random policy in the start phase. That the reward the agent converges to is equivalent to the performance of the actuated traffic light suggests, that the scenario might not require to sacrifice immediate reward for possible future rewards. Figure 4.5 shows the decay of the epsilon parameter over time. This is highly relevant to interpret the agent's results. The epsilon parameter is decreased linearly over the number of steps. As the agent improves, the episode length decreases, which results in the nonlinear decrease over the episode number.

Figure 4.5 also shows the exploration parameter over the number of episodes next to the episode reward and reveals that the reward already converges to its final value, while the agent's exploration probability is still comparably high. It is likely that the minimum durations before the environment actually reacts to the agent's decision to change the signal provides one explanation. Here, the environment absorbs too many unrelated, contradictory decisions. Furthermore, controlling a two-phased intersection in combination with the action definition that only allows to switch to the next phase in the cycle or keep the current phase also makes the environment more robust against random actions. Here, the episode reward of the random policy becomes important because it clearly shows that this robustness against a high percentage of random actions implies that a random policy would produce the same episode rewards.

4.3.3 *Learning Loss*

Figure 4.6 shows the total loss over all backup steps to update the ANN. If the agent learned the true value function, the loss would approach zero. But while the episode reward converges, the loss is still far from zero, yet the agent's performance improves. Because the agent samples the training experiences from the memory, there is a temporal decoupling between the current policy and the training. It is also possible that the agent is not able to learn the true value function but still chooses the right actions. This is more likely in environments with smaller numbers of actions to choose from.

It is possible that the true state-action value of both available actions is almost the same for many states. For intersections with more phases, some phase switches would require to cycle to multiple phases of the cycle until the particular phase is reached. Thus, the value function would be less smooth. It is very likely that the high performance in this environment cannot be transferred to more complex environments.

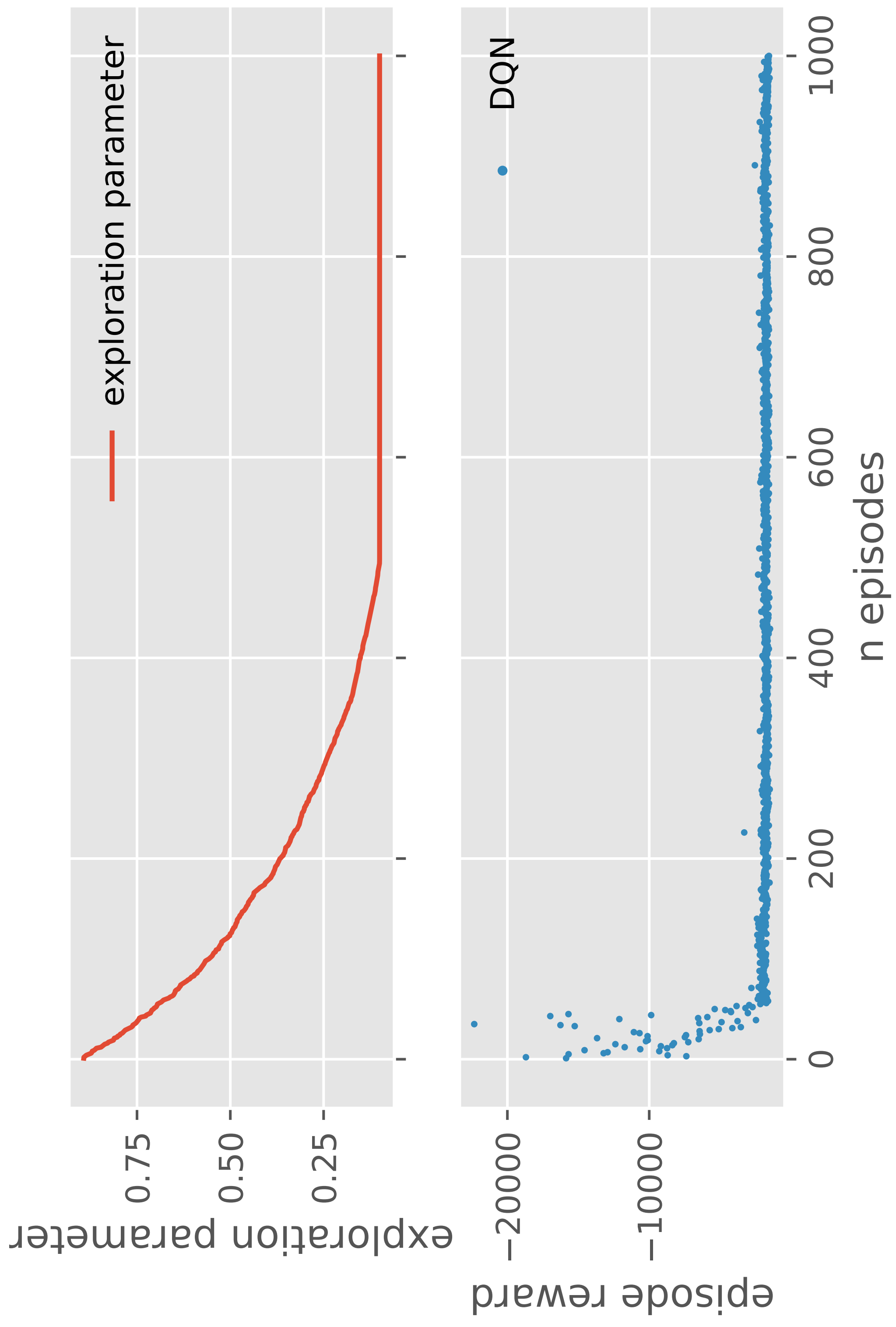


Figure 4.5: Decay of exploration parameter, with episode rewards sharing the same x-axis for context below.

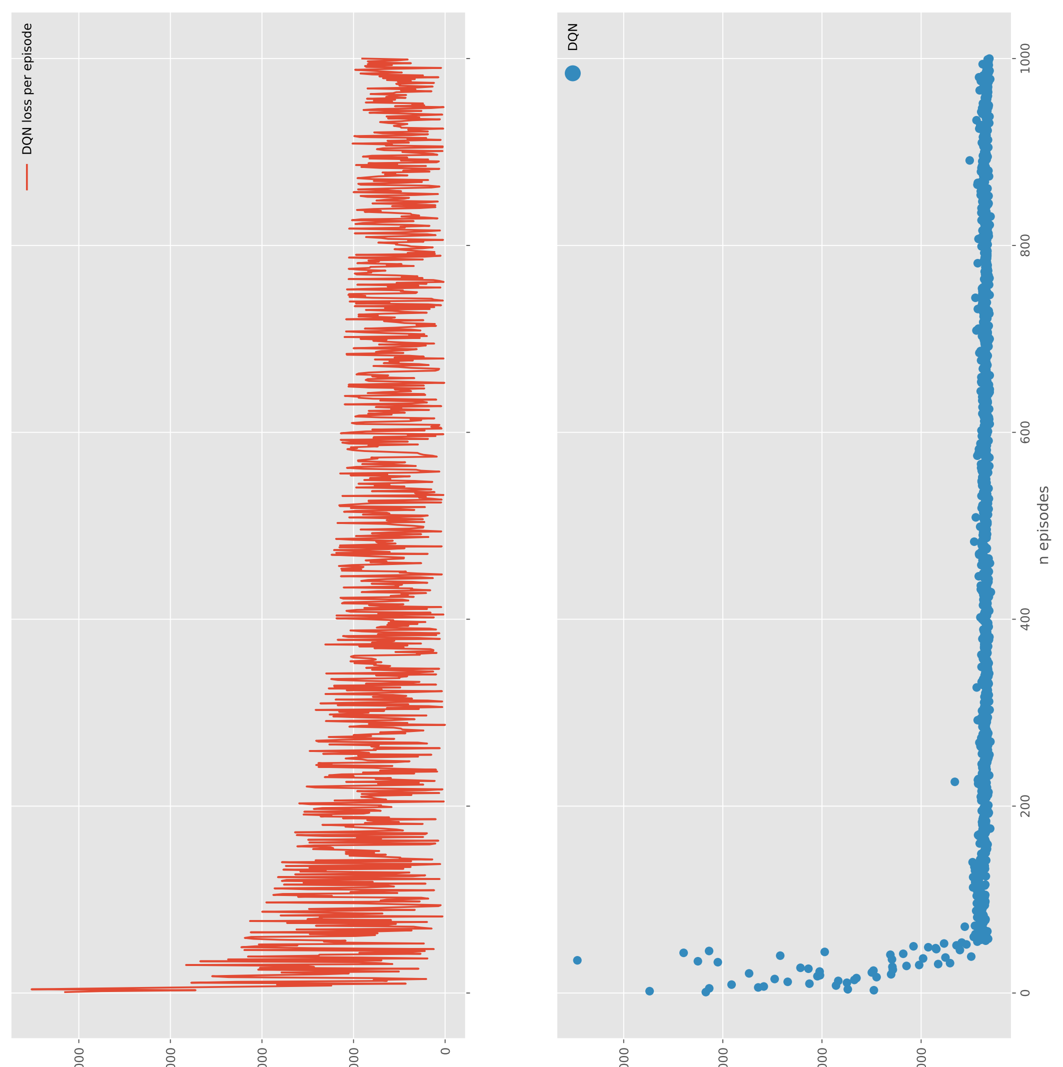


Figure 4.6: Accumulated Learning-Loss per episode, with episode rewards sharing the same x-axis below.

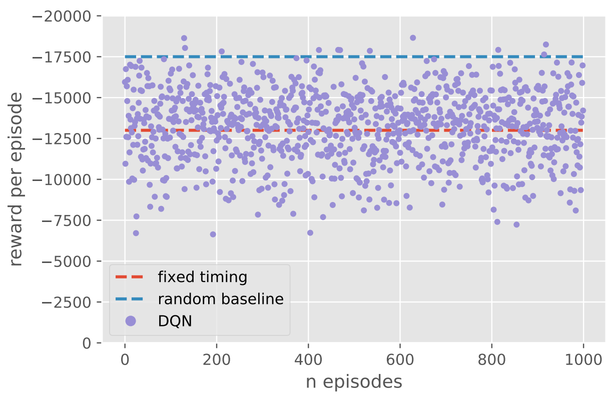


Figure 4.7: Reward over episode number.

4.3.4 *Failed Training Sequence*

Figure 4.7 shows one of many non converging training runs. The particular results in Figure 4.7 were generated with identical training parameters. Only the random number generation has been seeded differently. It should be noted that the same route file has been used and the traffic generation was not affected by the changed random seed. Only the agent's exploration and the weight initialization depended on the seed. This illustrates the fragility of the training process and leaves some questions regarding the practicability of RLTSC.

4.4 DISCUSSION

The last chapter explained the main mechanics, behind the agent's training data. However, to understand the actual meaning of the experiment's data, where it provides answers and how much those answers can be trusted, it is essential to understand the experiment's design and its consequences.

4.4.1 *Design of the Experiment*

The experiment is in many aspects different from a real-world scenario or possible applications. There are two main reasons for simplifications and abstractions.

- clarity
- practicability and implementability

In some cases it has been necessary to sacrifice realism to produce less ambiguous results. In other cases, the size of the experiment concerning the available resources made some concessions necessary. The most important properties and their implications are discussed next.

Constant Routes over all Episodes

The agent's actions depend on past experiences but also determine future experiences, which again effect which actions the agent chooses. Also, a significant amount of randomness is introduced by the agent's epsilon-greedy strategy to ensure enough exploration. This means that even a small change of a single parameter can produce entirely different results. The lack of reproducibility and comparability cannot be improved by seeds for pseudo-random number generation. Because training value-based RL algorithms requires many samples, it is not practicable to address those difficulties by a more significant number of repetitions. For that reason, the route file is generated once and used throughout all episodes. This was necessary to compare and understand different episodes and their results.

Demand Patterns

The traffic has been chosen in such way that the emission probability for each source is constant over all 3600 steps. Furthermore, both approaches have very different traffic volumes. This made it possible to decide if the agent learns and improves compared to a random policy, even at earlier stages with high exploration probabilities although it also makes learning considerably easier.

Action Definition

To reduce the set of actions that are available to the agent, so that the agent is only allowed to cycle through a fixed sequence of phases without changing their order, has a large impact on the action-value function. For more complex intersections with more phases, the individual phases are further apart inside the cycle. Thus, their order is more relevant and learning becomes more difficult because it is necessary to cycle through all phases between the current phase and the target phase. Hence, it is not possible to draw any conclusions for more complex intersections from the current results.

Episodic Task

TSC on an intersection does not consist of any episodes, and ideally, it should not be modeled as a sequence of individual episodes. Episodic tasks have several advantages compared to continuing tasks:

- transparency
  - evaluation episodes with $\varepsilon = 0$ are possible
  - possibility to compare episodes helps to measure learning process
- stability
  - possibility to recover from catastrophic states
  - improved elasticity of environment
- practicability

  - possible to save checkpoints
  - requires less supervision, episodes can be terminated after a certain step count

Unfortunately, the transformation of a continuing task to individual sequential episodes alters the learning process. The backup step for the last state of an episode is reduced to

$$Q(S_t, A_t | t = t_{\text{final}}) \leftarrow R_{t_{\text{final}}} . \tag{4.8}$$

This is problematic because their value function cannot be expressed in terms of future states. Due to of the fundamental mechanics of TD learning, the value of those 'artificial' states flows back to previous states and thus has a global effect on the whole value function.

4.4.2 *Accomplishments*

The implementation's contribution lies on two pillars:

- outcomes in terms of data
- presenting and investigating the methodology

4.4.2.1 *Agent Performance*

The highest priority was to demonstrate the agent's fundamental ability to learn. Therefore, the environment and tasks have been optimized to detect and demonstrate the agent's learning as clearly and unambiguously as possible. This allowed to reduce the number of influencing factors and randomness of the system. Therefore, it became porssible to draw more evident connections between logged data and the agent's learning. The experiment's data shows that the implemented agent was able to control a simplified intersection. Because the traffic definitions have been constant over all episodes, the improving reward per episode points very clearly to the agent's ability to learn and improve its efficiency. Since the experiment has been optimized to demonstrate the agent's fundamental learning ability, its validity for more realistic scenarios is limited.

4.4.2.2 *Methology*

As the implementation aimed to provide clear evidence for the agent's learning ability, it was also possible to demonstrate how to gain insights into the learning process. The implementation gives an overview of important key values and some of their possible interpretations. The demonstrated methods and insights are of universal relevance.

Part III

DELIBERATIONS

5 SPECIFICS OF RLTSC

Rltsc has two origins:

Its methology comes from the field of pure rl. Applications for Rl have been within the realm of the feasible only recently. For this reason the field of rl had only little exposure to possible applications. Combined with the flexibility of rl's abstractions, which can be applied to almost every problem, it can easily happen to miss where rltsc is different from pure rl. The problem domain of rltsc has its own, rich tradition and a comprehensive collection of terminology, methology and theories.

If one approaches rltsc coming from the problem of tsc he might disregard where *training* traffic signal control agents is different from the development and evaluation of other algorithms for those tasks.

This chapter discusses the challenges specific to rltsc and argues that the field has to factor in its specific requirements and challenges in order to move forward.

5.1 RL AND TRAFFIC SIGNAL CONTROL

Even recent publications that implement newer advances of pure rl fail to take into account that rltsc is different from other approaches to atsc [18, 6, 14]. Advanced and flexible self-control algorithms limit the control loop to the current traffic state, introduce heuristics and prevent the absolute failure of the controller [13].

Rl, however, is able to limit the time horizon by the discount factor but has no possibility to limit the effects of catastrophic states or to intervene directly to stabilize itself. This means that even small changes in the system can cascade to completely, sometimes dysfunctional states and policies. There is no direct mechanism to dampen the system or to introduce a policy that stabilizes the system, similar to a stable equilibrium point of mechanical systems.

This has several implications:

- The final performance of a policy becomes almost irrelevant as opposed to the stability under various conditions.
- It becomes very relevant to determine the conditions under which agents fail.
- It is mandatory to implement agents in a way they can interface with different simulations and scenarios.

- It is important to find ways to implement security mechanisms and predict states that lead to permanent dysfunctional policies, network weights etc., therefore research has to abandon its fixation on final rewards.

5.2 RLTSC AND PURE RL

There is not one publication on RLTSC that fails to mention the importance of adaptive TSC in the face of the problems caused by traffic congestions, air pollution and similar problems. Still every recent paper fails to adress the implications that arise from the applied nature of the problem to mention some examples:

- The infinite horizon problem of traffic control is devided into episodes without mentioning potential effects on the outcomes.
- The authors fail to address if agents should be pretrained inside simulations or if they should start learning a policy from scratch on real intersections.
- The authors do not adress if they believe agents can converge to policies that are flexible enough, so they can be frozen if they intent to keep learning over the complete lifetime of the controller.

It is not necessary that RLTSC is able to answer questions like those mentioned above today. But that a field that bases its relevance on the claim of being able to solve actual problems does not even mention the existence of questions beyond final rewards might be evidence that there is still a long road ahead of actual RLTSC.

There are several obstacles that arise from the characteristics of reinforcement learning, some of them will be discussed in the next section.

5.2.0.1 *Difficulties Specific to RL*

The explanation why applying RL is particularly challenging, and why the application to traffic signal control is especially difficult, is necessary to understand why reinforcement learning is fundamentally different from other methods for traffic signal control.

CAUSE AND EFFECT IN RL SYSTEMS Diving deeper into different theories about the complexity of systems is beyond the scope of this thesis, but basic intuition how engineers approach the understanding of systems is sufficient to understand the fundamental difficulties of reinforcement learning.

EXPERIMENTATION IN RL Many common approaches in engineering to deal with matters evolving from complex systems are based on two building blocks:

- isolating subsystems and identifying clear boundaries between those systems
- repetition of experiments to identify the effect of parameters and the relationships between those parameters

The implementation of this thesis has shown that the process of training has several properties that interfere with those building blocks. Some of the main factors and the resulting difficulties are:

- coupling of all variables over time
  - it is hard to isolate and repeat parts of the learning process
  - small changes can have a large impact over time
- large amount of necessary randomness for exploration
  - difficult to identify the effect of specific changes
  - coupling over time means seeding pseudo-random number generators hardly improves repeatability
- large numbers of samples required for training agents
  - very long runtimes of training runs limits the number of repetitions
  - large temporal distances between cause and effect of events
- recent methods like Experience Replay or fixed target networks introduce even higher levels of indirectness
  - further decoupling of the collection of samples and the effects of learning from those samples
  - even larger numbers of samples needed
  - causes even longer runtimes

The field of reinforcement learning compensates for those difficulties by analyzing and developing reinforcement learning methods by selecting and designing training problems that expose several properties:

- episodic
- transparency
- difficulty of training problems can be estimated, parametrized and varied
- small number of parameters for training problems

- available benchmarks of other methods on the same problem

If reinforcement learning has to solve specific problems, it is not possible anymore to freely choose or design training problems to expose specific properties that simplify diagnosing and understanding the training process. It becomes apparent that applied reinforcement learning problems are subject to a trade-off between realism and interpretability.

5.3 SPECIFIC CHALLENGES OF RLTSC

Comparing the task of traffic control to the kind of problems preferred by general reinforcement learning research also explains why traffic control embodies a particularly challenging application for reinforcement learning. The next section outlines the most relevant difficulties that complicate the training process.

5.3.1 *No Obvious Mapping to MDP*

As traffic signal control does not have a single inherent MDP representation, the research uses a wide variety of state representations, action and reward definitions. This accounts for the lack of actual benchmarks and makes it hard to identify mistakes and weaknesses of the design of experiments. Reproducing results or even adapting existing agents to test them on envionments with a different MDP representation is very time-consuming and cannot be done easily.

5.3.2 *Non-Episodic*

Most recent publications split the problem of traffic signal control into episodes. All of them fail to explain the motivation behind this decision and do not discuss the alteration to the actual non-episodic problem of traffic signal control. The implementation in chapter 4 makes it clear, that the training in an non-episodic environment would be extremely difficult. However, this thesis also emphasizes that the split into episodes alters the actual problem and effects its outcomes.

5.3.3 *External Contraints*

Agents do not only need to be efficient, they are also obliged to follow specific traffic roles, for example minimal phase durations. The variety of rules requires the agents to consider many specific cases. This introduces strong coupling between those agents and the problem domain. This complicates the reproducibility and interpretation of results and makes it difficult to reuse or improve agents.

5.3.4 *Coupling of Problem Formulation and Solution*

A common origin of many major difficulties of RLTSC lies in the difficulty to decouple the problem, its MDP formulation and its solution. Therefore, current research struggles to improve the comparability and reproducibility of its results and tends to neglect the specific properties of RL. It often resorts to a black-box approach regarding the training of agents. The coupling between the formulation of problems and their solution constitutes a serious obstacle for the distribution of knowledge among researchers.

6 CONCLUSION

This thesis seeked to strike balance between the mindsets and methods of pure RL and those of RLTSC. The implementation of an RLTSC agent showed that RLTSC agents are capable of learning traffic signal control, even without the use of a CNN for function approximation. The most important findings of this thesis are not the final scores of the implemented agent. RL is still considered a young area of research but compared to RLTSC even pure RL is a rather old field of study. This study shows that some questions that are highly relevant to RLTSC will continue to exist regardless of possible advances in RL. Those problems have to be addressed directly by RLTSC.

Basic reseach is allowed to explore the boundaries of what is possible. But if the authors of RLTSC intend to be mindful about their promise to contribute to the solution of traffic control problems, they must not be overly attached to singular success but have to investigate the limits and flaws of their methods. Section 4.3.4 documents a good example which shows that already minimal perturbations can destabilize the learning process irreversibly. The implementation also revealed that the implemented agent did not depend on the latest computationally expensive technologies like CNNs. Therefore, this thesis argues that RLTSC needs to emancipate from pure RL.

This work wants to point out some of the obstacles that lie ahead of applied RLTSC. While it is almost impossible to predict the future of RLTSC, I believe that all of the following issues are critical. Beginnig to look for answers will only be possible by acknowledging their existence:

COOPERATION RLTSC fails to expand on the ideas of its two origins, yet most RLTSC do not attempt to benefit from the current knowledge of related fields.

REPRODUCIBILITY Compared to fields like physics or medicine RL reseach has the potential to be very accessible, reproducible and transparent. Yet most of the published implementations are tightly coupled to details of the environment. Often authors do not share their implementations at all.

ROBUSTNESS A practical RLTSC solution would have to be robust in different environments, sensor configurations and external requirements. This implies that reproducibility and robustness are central to RLTSC. Despite all that, most RLTSC implementations are extremly brittle and most authors fail to share any code or implementation details at all.

SAFETY Currently, there is little research about safety concerns and the prevention of failure. Publications focus on the creation of impressive scores and neglect to investigate the limits of their solutions.

It is crucial for RLTSC to learn how knowledge and results can be shared and communicated with other reseachers and even related fields. RLTSC needs to adapt the mindset of engineering with regard to the development of modular agents that can be shared, recombined, tested and improved by a large group of researchers. At the same time, it needs as much understanding and insight as possible into the way agents are learning. RLTSC has to emancipate from its origins while it must learn from them at the same time. Some of the major challenges will be impossible to solve by the implementation of the next extension to DQN. That being said, the same problems will not be solvable for RLTSC without a sound understanding of RL.

Part IV

APPENDIX

BIBLIOGRAPHY

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

[2] Yit Kwong Chin, Lai Kuan Lee, Nurmin Bolong, Soo Siang Yang, and Kenneth Tze Kin Teo. "Exploring Q-learning optimization in traffic signal timing plan management." In: *Proceedings - 3rd International Conference on Computational Intelligence, Communication Systems and Networks, CICSyN 2011*. 2011. ISBN: 9780769544823. DOI: 10.1109/CICSyN.2011.64.

[3] George Cybenko. "Approximation by superpositions of a sigmoidal function." In: *Mathematics of control, signals and systems* 2.4 (1989), pp. 303–314.

[4] Jakob Erdmann, Robert Oertel, and Peter Wagner. "VITAL: a simulation-based assessment of new traffic light controls." In: *Intelligent Transportation Systems (ITSC), 2015 IEEE 18th International Conference on*. IEEE. 2015, pp. 25–29.

[5] Brian Everitt. *The Cambridge dictionary of statistics*. Cambridge, UK; New York: Cambridge University Press, 2002. ISBN: 052181099X 9780521810999. URL: http://www.worldcat.org/search?qt=worldcat%7B%5C_%7Dorg%7B%5C_%7Dall%7B%5C&%7Dq=052181099X.

[6] Juntao Gao, Yulong Shen, Jia Liu, Minoru Ito, and Norio Shiratori. "Adaptive Traffic Signal Control: Deep Reinforcement Learning Algorithm with Experience Replay and Target Network." In: *arXiv preprint arXiv:1705.02755* (2017).

[7] Wade Genders and Saiedeh Razavi. "Evaluating reinforcement learning state representations for adaptive traffic signal control." In: *Procedia Computer Science* 130 (2018), pp. 26–33.

[8] Hado V Hasselt. "Double Q-learning." In: *Advances in Neural Information Processing Systems*. 2010, pp. 2613–2621.

[9] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. "Reinforcement learning: A survey." In: *Journal of artificial intelligence research* (1996), pp. 237–285.

[10] S Sathiya Keerthi and B Ravindran. "A tutorial survey of reinforcement learning." In: *Sadhana* 19.6 (1994), pp. 851–889.

[11] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization." In: *arXiv preprint arXiv:1412.6980* (2014).

[12] Daniel Krajzewicz, Jakob Erdmann, Michael Behrisch, and Laura Bieker. "Recent Development and Applications of {SUMO - Simulation of Urban MObility}." In: *International Journal On Advances in Systems and Measurements* 5.3&4 (Dec. 2012), pp. 128–138.

[13] Stefan Lämmer and Dirk Helbing. "Self-control of traffic lights and vehicle flows in urban road networks." In: *Journal of Statistical Mechanics: Theory and Experiment* 2008.04 (2008), P04019.

[14] Xiaoyuan Liang, Xunsheng Du, Guiling Wang, and Zhu Han. "Deep Reinforcement Learning for Traffic Light Control in Vehicular Networks." In: *arXiv preprint arXiv:1803.11115* (2018).

[15] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. "Rectifier nonlinearities improve neural network acoustic models." In: *Proc. icml*. Vol. 30. 1. 2013, p. 3.

[16] Patrick Mannion, Jim Duggan, and Enda Howley. "An experimental review of reinforcement learning algorithms for adaptive traffic signal control." In: *Autonomic Road Transport Support Systems*. Springer, 2016, pp. 47–66.

[17] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. "Human-level control through deep reinforcement learning." In: *Nature* 518.7540 (2015), p. 529.

[18] Elise van der Pol. "Deep reinforcement learning for coordination in traffic light control." PhD thesis. Master's thesis, University of Amsterdam, 2016.

[19] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. Vol. 1. 1. MIT press Cambridge, 1998.

[20] Richard S Sutton and Andrew G Barto. "Reinforcement Learning: An Introduction." 2015. URL: `https://www.dropbox.com/s/b3psxv2r0ccmf80/book2015oct.pdf?dl=0`.

[21] Hado Van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-Learning." In: *AAAI*. Vol. 16. 2016, pp. 2094–2100.

[22] Christopher John Cornish Hellaby Watkins. "Learning from delayed rewards." PhD thesis. King's College, Cambridge, 1989.

[23] Marco Wiering et al. "Multi-agent reinforcement learning for traffic light control." In: *ICML*. 2000, pp. 1151–1158.

[24] Marco Wiering and Martijn Van Otterlo. *Reinforcement learning*. Vol. 12. Springer, 2012.

A IMPLEMENTATION SAMPLES

```python
# Demand generation based on:
# http://sumo.dlr.de/wiki/Tutorials/TraCI4Traffic_Light

Point = namedtuple('Point', ['x', 'y'], verbose=False)
Phases = namedtuple('Phases', ['one', 'two'], verbose=False)


def encode_state(unencoded):
    encoding = [[0, 0, 0, 1], [0, 0, 1, 0], [0, 1, 0, 0], [-1, -1, -1, -1]]
    encoded = encoding[unencoded]
    return encoded


light_state = ([0, 1], [1, 0])


def get_inds(points):
    test = np.asarray(points)
    test2 = np.expand_dims(np.trunc((test[:, 1] - 4.5) / 8), axis=1)
    return np.concatenate((test, test2), axis=1)


def discretize(points):
    grid = np.zeros(12)
    speed_grid = np.zeros(12)
    if len(points) > 0:
        inds = get_inds(points)
        grid[inds[:, 3]] = 1

    return grid


def transform(position):
    x, y = position
    return Point(x - 250, y - 250)


def is_in_bound(vehicle):
    speed, x, y = vehicle
    bound = 96
    return abs(x) < bound and abs(y) < bound
```

```python
def idx_change(action):
    if action <= 1:
        if action == 0:
            return (0, [- 5, 0])
        else:
            return (0, [5, 0])
    else:
        if action == 3:
            return (1, [0, -5])
        else:
            return (1, [0, 5])


class TrafficEnv:
    def __init__(self, traci, gui=True):
        self.traci = traci
        self.netfile = 'data/traffic.net.xml'
        self.routefile = 'data/traffic.rou.xml'
        self.addfile = 'data/traffic.add.xml'
        self.exitloops = ["loop4", "loop5", "loop6", "loop7"]
        self.lanes = ["n_0_0", "s_0_0", "e_0_0", "w_0_0",
                      "0_n_0", "0_s_0", "0_e_0", "0_w_0"]
        self.lane_area_detectors = ["e2det_s_0_0", "e2det_w_0_0"]
        self.loop_variables = [tc.JAM_LENGTH_METERS]
        if gui:
            self.mode = 'sumo-gui'
        else:
            self.mode = 'sumo'

    def start(self):
        args = ["--net-file", self.netfile, "--route-files",
                self.routefile, "--additional-files", self.addfile, "--start"]
        sumoBinary = checkBinary(self.mode)
        sumo_cmd = [sumoBinary] + args
        self.traci.start(sumo_cmd)
        currphase = self.traci.trafficlight.getRedYellowGreenState("0")
        self.light = TrafficLight(currphase)
        jams = self.alternative_obs()
        obs = [jams[:], self.light.state]
        flatobs = [item for items in obs for item in items]

        return flatobs

    def _change_phases(self, action):
        idx, change = idx_change(action)
        phasename = ''
        otheridx = False
        if idx == 0:
            otheridx = 1
        else:
            otheridx = 0

        if 5 <= self.phases[idx] + change[idx] <= 45:
            one = self.phases[0] + change[0]
            two = self.phases[1] + change[1]
            self.phases = Phases(one, two)

        if phasename == 'one':
            return 0
        else:
            return 2

    def step(self, action):
        rewardsum = 0
        done = False
        sig = self.light.act(action)
        self.traci.trafficlight.setRedYellowGreenState("0", sig)
        currphase = self.traci.trafficlight.getRedYellowGreenState("0")
        rewardsum += self.compute_reward()
        self.traci.simulationStep()
        if self.traci.simulation.getMinExpectedNumber() <= 0:
            done = True
        jams = self.alternative_obs()
        obs = [jams[:], self.light.state]
        flatobs = [item for items in obs for item in items]

        return done, flatobs, rewardsum

    def observation(self):
        vehicleIDs = self.traci.lane.getLastStepVehicleIDs("e_0_0")
        vehicle_data = self.get_Values(vehicleIDs)
        positionMatrix = np.zeros((1, 12), dtype=np.int8)
        speedMatrix = np.zeros((1, 12))
        for data in vehicle_data:
            positionMatrix[0, data[3]] = -1
            speedMatrix[0, data[3]] = data[2]
        speedPosition = np.vstack((positionMatrix, speedMatrix))
        reward = self.compute_reward()
        jams = self.jamLengths()
        return speedPosition

    def jamLengths(self):
        jams = [0, 0, 0, 0]
        pos = 0
        for detector in self.lane_area_detectors:
            jams[pos] = self.traci.lanearea.getJamLengthVehicle(detector)
            pos + 1

        return jams

    def compute_reward(self):
        ids = []
        for detector in self.lane_area_detectors:
            ids.append(self.traci.lanearea.getLastStepVehicleIDs(detector))
        flat = [num for elem in ids for num in elem]
        speeds = []
        for vehicle in flat:
            allowed = self.traci.vehicle.getAllowedSpeed(vehicle)
            current = self.traci.vehicle.getSpeed(vehicle)
            speeds.append((allowed - current) / allowed)
        n_vehicles = len(speeds)
        average_delay = 0
        if n_vehicles > 0:
            average_delay = sum(speeds) / n_vehicles
        return -average_delay

    def alternative_obs(self):
        jam = []
        for detector in self.lane_area_detectors:
            jam.append(np.array([self.traci.lanearea.getJamLengthVehicle(
                detector), self.traci.lanearea.getLastStepVehicleNumber(detector)]
        print('jam: ', np.array(jam))
        return np.array(jam).flatten()

    def get_Values(self, ids):
        vehicleData = []
        for ID in ids:
            x, y = self.traci.vehicle.getPosition(ID)
            if abs(x - 250) <= 96.5:
                speed = self.traci.vehicle.getSpeed(ID) / 19.4
                grid_pos = math.trunc((x - 254.5) / 8)
                vehicleData.append((x, y, speed, grid_pos))
        return vehicleData

    def _reward(self):
        res = self.traci.lanearea.getSubscriptionResults()

        reward = 0
        for loop in self.lane_area_detectors:
            for var in self.loop_variables:
                reward -= res[loop][var]**1.2
        return reward

    def get_vehicles_left(self):
        return self.traci.simulation.getMinExpectedNumber()

    def _stop(self):
        self.phases = Phases(10, 10)
        self.traci.close()
        time.sleep(.500)

    def reset(self):
        self._stop()

    def end(self):
        self._stop()
```

Listing 1: Environment class.

```python
# Replay Memory
class ReplayMemory:
    def __init__(self, *, capacity):
        self.samples = deque([], maxlen=capacity)

    def store(self, exp):
        self.samples.append(exp)
        pass

    def get_batch(self, n):
        n_samples = min(n, len(self.samples))
        samples = sample(self.samples, n_samples)
        return samples

    def __len__(self):
        return len(self.samples)
```

Listing 2: Replay Memory.

```
def prep_input(data, n_dimension):
    prep = np.asarray(data)
    transformed = prep.reshape(-1, 6)
    return transformed


def prep_batch(to_prep):
    prep = np.vstack(to_prep)
    return prep


class DoubleNN:
    def __init__(self, n_states=6, n_actions=4, batch_size=64,
                 learning_rate=0.01, activation='tanh'):
        self.model = make_model(n_states, n_actions,
                                learning_rate=learning_rate)
        self.model_t = make_model(
            n_states, n_actions, learning_rate=learning_rate)
        self.batch_size = batch_size
        self.n_states = n_states

    def train(self, X, y):
        X = prep_batch(X)
        y = prep_batch(y)
        loss = self.model.fit(X,
                              y,
                              batch_size=self.batch_size,
                              epochs=10,
                              verbose=0,
                              shuffle=True)

        return loss

    def predict(self, input_state, usetarget=False):
        print(input_state)
        state = prep_input(input_state, self.n_states)
        if usetarget:
            q_vals = self.model_t.predict(state)
        else:
            q_vals = self.model.predict(state)
        return q_vals[0]

    def update_target(self):
        weights = self.model.get_weights()
        self.model_t.set_weights(weights)
        self.save('weights.h5')
        pass

    def best_action(self, state, usetarget=False):
        q_vals = self.predict(state, usetarget)
        best_action = np.argmax(q_vals)
        return best_action

    def save(self, fname):
        self.model.save_weights(fname, overwrite=True)
        pass

    def load(self, fname):
        self.model.load_weights(fname)


def make_model(input_dim, output_dim, learning_rate, activation='tanh', loss='mean
    model = Sequential()
    model.add(Dense(8, input_shape=(6,), activation=activation))
    model.add(Dense(8, activation=activation))
    model.add(Dense(4, activation='linear'))
    opt = SGD(lr=learning_rate)
    model.compile(loss='mean_squared_error', optimizer=opt)
    return model
```

Listing 3: ANN reimplementation in Keras. Functionally equivalent to the used PyTorch version.

```python
class DQNAgent:
    def __init__(self, *, model, memory, n_actions):
        self.memory = memory
        self.n_actions = n_actions
        self.NN = model
        self.episode_count = 0
        self.step_count_total = 1
        self.step_count_episode = 1
        self.epsilon = 0.9
        self.epsilon_min = 0.01
        self.epsilon_max = 0.01
        self.epsilon_decay = 0.0002
        self.target_update = 200
        self.max_steps = 1
        self.n_episodes = 1
        self.epsilon = 1
        self.batch_size = 16
        self.usetarget = False
        self.gamma = 0.9
        self.loss = 0
        self.done = False
        self.reward = 0
        self.reward_episode = 0
        self.learning_switch = False
        self.learning_start = 1

    def act(self, state):
        self.step_count_total += 1
        action = self.choose_action(state)
        return action

    def learn(self, obs):
        if self.step_count_total > self.learning_start:
            self.learning_switch = True
        self.memory.store(obs)
        if self.learning_switch:
            self.backup()
        self.update_epsilon()
        pass

    def backup(self):
        self.replay()
        if self.step_count_total % self.target_update == 0:
            self.NN.update_target()
            self.usetarget = True
        pass

    def replay(self):
        X, y = self._make_batch()
        self.loss = self.NN.train(X, y)
        if np.isnan(self.loss.history['loss']).any():
            print('Warning, loss is {}'.format(self.loss))
        pass

    def choose_action(self, state):
        if np.random.random() <= self.epsilon:
            choice = self.random_choice()
        else:
            choice = self.greedy_choice(state)
        return choice

    def greedy_choice(self, state):
        greedy_choice = self.NN.best_action(state, usetarget=False)
        return greedy_choice

    def random_choice(self):
        random_choice = np.random.randint(0, self.n_actions)
        return random_choice

    def _make_batch(self):
        X = []
        y = []
        batch = self.memory.get_batch(self.batch_size)
        for state, action, newstate, reward, done in batch:
            X.append(state)
            target = self.NN.predict(state, False)
            q_vals_new_t = self.NN.predict(newstate, self.usetarget)
            a_select = self.NN.best_action(newstate, False)
            if done:
                target[action] = reward
            else:
                target[action] = reward + self.gamma * q_vals_new_t[a_select]
            y.append(target)
        return X, y

    def update_epsilon(self):
        self.epsilon -= 0.00002962962
      #  self.epsilon = (self.epsilon_min +
      #                  (self.epsilon_max - self.epsilon_min) * np.exp(-self.epsi
        pass
```

```
class RandomAgent:
    def __init__(self, nActions, memory):
        self.nActions = nActions
        self.memory = memory

    def act(self, obs):
        choosenAction = random.randrange(self.nActions)
        return choosenAction

    def learn(self, obs):
        self.memory.store(obs)

    def __len__(self):
        return len(self.memory)
```

Listing 4: Basic agent class.