Learning Road Traffic Control:

Towards Practical Traffic Control Using Policy Gradients

Diplomarbeit

Silvia Richter

Albert-Ludwigs-Universität Freiburg
Fakultät für Angewandte Wissenschaften
Institut für Informatik

July 2006

Erklärung
(Declaration)

Hiermit versichere ich, dass ich diese Abschlussarbeit selbstständig verfasst habe, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Diese Abschlussarbeit wurde nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt.

(I hereby declare that I wrote this thesis on my own, only making use of the sources mentioned, and that I have indicated all places where parts from other published documents were used in word or in meaning. This thesis has not been used for any other examination so far.)

Canberra, im Juli 2006 (July 2006) Silvia Richter

Abstract

The optimal control of traffic lights in urban road networks is a highly complex problem. Many factors influence the flow of traffic, and hence the performance of a traffic network, of which few can readily be measured. Currently used control systems are often relatively simple and date back several decades, while more sophisticated optimisation methods fail for large networks.

Reinforcement learning algorithms are a means of learning control strategies for complex environments, requiring no pre-specified knowledge about possible solutions. Policy-gradient algorithms are reinforcement learning methods that are particularly useful for learning control strategies (policies) for large and only partially observable environments. They use a parameterised function to represent the policy, and perform gradient ascent on the parameters of this function. Convergence to a (local) optimum is guaranteed, under appropriate conditions.

In this work, we examine how policy-gradient ascent can be used to learn the control of traffic signals, with the goal of optimising the traffic flow in a road network. We show that our methods perform very well, are able to scale up to large networks and can achieve better results than other commonly used approaches such as saturation balancing algorithms.

Acknowledgements

I thank Douglas Aberdeen, my main advisor, for his guidance and help, and Bernhard Nebel for making this thesis possible. Thanks to Olivier Buffet for valuable advice throughout this thesis, and for his patient help during long hours of debugging. Conrad Sanderson, Simon Günter and Malte Helmert have all proof-read parts of this thesis and deserve thanks for their dedication and helpful suggestions. Finally, thanks to all at the Statistical Machine Learning group of NICTA, Canberra, for making me feel at home in the group, and to the New South Wales Roads and Traffic Authority for providing the basis for this thesis through a cooperation project with NICTA.

Contents

Chapter 1

Introduction

Traffic jams have become a familiar sight for many drivers travelling to work in the morning or going for a trip on the weekend. As car ownership rates and traffic volume have steadily increased over the last decades, existing road infrastructure today is often strained nearly to its limits. Continuous expansion of this infrastructure, however, is not possible or even desirable due to spacial, economical and environmental reasons. It is therefore of paramount importance to try to optimise the flow of traffic in a given infrastructure.

However, the inputs to this optimisation problem can be arbitrarily complex: to find an optimal solution, it would be necessary to know the position, speed and route of every vehicle in the system. Furthermore, all reactions of drivers to changing traffic conditions would need to be known and taken into account. In reality, traffic light controllers need to work with far less information about the current traffic situation. In most cities around the world today, the only sensors available are inductive loop detectors, which are embedded in the roads and count the number of cars passing over them.

The difficulty of controlling traffic lights optimally, along with the importance of the problem, have led to a great number of research approaches [19]. This makes it quite surprising that the systems used today often date back some 20 or 30 years. Simple controllers still perform quite well as opposed to more sophisticated methods, as the latter are in many cases not able to deal with more than a handful of intersections at the same time [19].

One possible method for finding control strategies in complicated domains is reinforcement learning [32]. Reinforcement learning has been a popular field in machine learning throughout the last few decades. Its main attraction is that it is a general framework, capable of addressing virtually any “real-world problem”, and particularly requiring little previous knowledge about the solutions to be learned. It can deal with incomplete information and stochastic changes in the environment, which are two complications present in the traffic control problem. Reinforcement learning can furthermore cope with delayed feedback about the control strategy that is being pursued.

In reinforcement learning, the goal for the learner is to learn the best reaction to any situation that it may find itself in at some point in time. This so-called *policy* is acquired through interaction with an environment in a trial and error process. The learner, often called an *agent* in this context, continually chooses actions based on its current perception of the world, and then receives feedback on these actions which modify its concept of good behaviour in the corresponding situations or *states* of the environment. With this technique it is possible to learn in complex and highly stochastic domains without having previous knowledge about possible solutions. *Policy-gradient* (PG) methods are a class of reinforcement learning algorithms that are particularly well suited for use in large real-world problems. They calculate policies by performing gradient-ascent on a parameterised policy function.

In this work, we use PG methods to learn signalling policies for traffic lights. Performing gradient ascent on the policy parameters enables us to efficiently search through the large space of possible policies. Our approach scales up to a large number of signalled intersections by having each intersection controlled by a separate controller. At the same time, the controllers can work together to optimise the overall system through the use of common world information and performance feedback. We show that we are able to automatically find effective policies, learning from simple observations and without the need of an explicit model of the system. Our methods outperform a popular controlling technique used around the world, at least within the limitations of our simulation.

This thesis is structured as follows: In Chapter 2, we define the traffic control problem, introduce some vocabulary used in traffic literature and give an overview of existing techniques for traffic control. In particular, we point out some weaknesses of current control systems. In Chapter 3, we describe the reinforcement learning framework and some classical algorithms. We introduce policy-gradient methods in general and natural policy gradients in particular, and present the two algorithms we use in our experiments.

The core of this work is described in Chapter 4, where we show how we apply policy-gradient methods to traffic control. We discuss the expected advantages of our approach as opposed to existing control techniques, and define optimisation criteria. Then we develop the architecture of the learning system and present a traffic simulation program that we implemented for training the traffic light controllers. Chapter 5 contains a set of experiments we conducted and their results. We describe the baselines we compare against, and develop test scenarios appropriate for demonstrating the particular strengths of PG methods. Subsequently, we present the results of our experiments and analyse the performance of PG compared to the baselines. We also compare our two PG algorithms against each other and draw conclusions about their respective usefulness for traffic control. Chapter 6 summarises the work presented and gives an outlook on possible future work in this area.

Chapter 2

Traffic Control

The flow of traffic in a road network can be influenced by several measures, such as signalling at intersections, messaging to the drivers, collecting tolls, and marking of the roads, with signalling being however the most important control measure [19]. The narrowed-down *traffic control problem* thus consists of finding an optimal signalling schedule for all intersections in a network, typically with the aim of minimising the waiting time of cars at intersections, the total travel time or other performance criteria such as the number of stops, fuel consumption (and hence emissions), etc.

2.0.1 Why Traffic Control is Hard

The traffic control problem is difficult for a variety of reasons, with its sheer size being the prime reason. Many variables influence the performance of a network even when the signalling policy is fixed. Factors that come into play, for example, are irregular and unpredictable incidents like pedestrians, accidents or illegal parking, and the weather. Driver characteristics are a further source of variance, as they influence the choice of routes as well as driving behaviour like speed and distance keeping.

A further aspect making optimal traffic control difficult is that information about the current manifestation of traffic in the network can in most cases only be *partially* attained through (noisy) measurements. In fact, most signal controllers in use today rely solely on the information gained from inductive loops embedded in the roads. Finally, tight real-time constraints exist, e. g., decisions based on current information may need to be made within 2 seconds [19].

2.0.2 Traffic Models

In order to deal with the problem of incomplete information about the current traffic situation, many control techniques in use today build and maintain an internal *model* of the traffic situation, derived from available data such as loop detector counts [9]. The model then serves as the basis for decision making, predicting

the reaction of traffic on a possible signalling strategy, and deriving performance criteria such as queue lengths at intersections.

Models can operate on different levels of abstraction. Some view traffic as made up of single vehicles (the microscopic view), while others describe the flow of traffic in terms of streams (using fluid theory), or calculate characteristics on the network level, such as the traffic intensity in terms of cars per space unit (macroscopic view) [28].

A popular traditional approach is to combine a deterministic component of traffic flow based on fluid theory with a stochastic component based on steady-state queueing theory [28]. Steady-state queueing theory calculates car arrival distributions at an intersection and derives estimations of delays and queue lengths. This is done locally, i. e., for each intersection separately, taking into account only the upstream neighbour intersection. This theory cannot model the coordinating effects of the interaction between various intersections, and it has further limitations when traffic is dense (when the average flow exceeds the average capacity for a certain time interval, and no stochastic equilibrium exists).

A further difficulty for accurate modelling arises from the fact that signalling policies and traffic flow interact. It is not sufficient for models to fix the expected routes of vehicles and optimise the signalling for the resulting traffic pattern – models must also take into account that drivers adapt their behaviour, and possibly their route choices, to changed conditions. This is what is called the traffic assignment problem [11].

Much research has gone into developing ever more sophisticated models (for some recent work see [9, 10]). The problem remains, however, that models are seldom perfect and that the performance of controlling strategies depending on such models may suffer from this inaccuracy.

2.1 Existing Control Methods

In this section we give an introduction to the basic vocabulary used in traffic theory, an overview of existing approaches to traffic control, and some examples of popular controlling systems in use today.

A large part of the following glossary as well as of the descriptions of current systems is based on Markos Papageorgiou's "Review of Road Traffic Control Strategies" [20].

2.1.1 Glossary

A *stream* is a sequence of cars entering an intersection from a certain direction and leaving it in a certain direction. If two streams can cross the intersection without interfering they are called *compatible*, else they are called *antagonistic*. A *phase* or *stage* is a time interval where a certain subset of the lights at an intersection are green, such that a certain set of compatible streams have the right of way during

that time. The term phase is also used to denote the set of streams corresponding to it, while the terms *phase length* or *green-time* are used to denote the duration of the phase. A *signal cycle* traditionally is completed when each phase has been on once, the *cycle time* being the time needed for the completion of one cycle. *Inter-green times* are the few seconds needed in between different phases, where no stream has the right of way, to avoid interference of antagonistic streams.

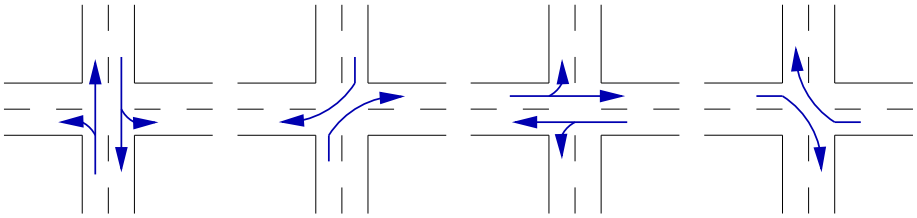
Traditionally, control algorithms aim to optimise traffic flow via the *phase scheme*, the *split*, and the *offset*. The *phase scheme* or *staging* determines which streams will have the right of way together, i. e., it groups the signal lights into subsets that will be green at the same time, and it determines the order in which the corresponding phases will come on. Popular groupings of streams into phases in Australia include the *split approach* and the *diamond overlap*. In the split approach (SA), all streams coming from one direction have the right of way together, while in the diamond overlap (DO) there is one phase where straight and left turns of two opposite directions go together, followed by a phase where the right turns of those two directions go together. Examples for phase schemes made up of two SAs and two DOs respectively are shown in Figure 2.1. Phase schemes combining the two (DOSA) are also used.

A *split* gives a distribution of the cycle time to the individual phases, thus determining the length of each phase. An *offset* refers to the time interval between a reference point in time and the start of a new cycle. Offsets can be introduced to coordinate neighbouring intersections: if, for example, along a street (also called a *link* in network terms) several intersections have the same phase schemes and similar cycle times, then offsetting subsequent intersections by the average travel time between their predecessors and themselves creates a “green wave” for the vehicles travelling along this link, minimising the number of stops and waiting times at intersections.

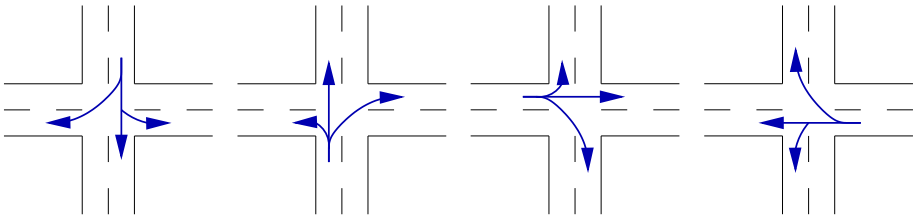
The *demand* denotes the number of cars that want to enter the road network at a given point in time or over a certain period of time. If we assume that the demand is independent of the control strategy used in the network, we assume *fixed demand*. The *saturation* of a stream s during a phase p is the ratio of actual (current) traffic volume on the stream and the maximum possible traffic volume, given the current length of the phase:

$$\mathit{saturation}(s,p) := \frac{\mathit{current\_traffic}(s,p)}{\mathit{max\_traffic}(s,p)} \tag{2.1}$$

Under-saturated traffic conditions are those where the demand for any link in the network does not exceed the capacity of the link, and queues that build up during red phases can be emptied during green phases. *Saturated* and *over-saturated* are both used to denote the opposite of under-saturated. A popular traditional approach to traffic control is based on balancing the saturation of the (streams in) different phases. Algorithms following this approach are the so-called *saturation-balancing* algorithms.



(a) NS straight (b) NS right (c) EW straight (d) EW right



(e) N (f) S (g) E (h) W

Figure 2.1: The double diamond-overlap (DODO) phase scheme (above), and the double split approach (SASA) phase scheme (below). N, S, E, and W denote north, south, east and west respectively.

2.1.2 Categories of Traditional Control Techniques

According to Gartner [11], scientific research on traffic began in the 1930s with the application of probability theory to the description of road traffic. In the 1950s, various approaches were pursued to develop advanced traffic models, including traffic wave theory and queueing theory [11]. The first computer based signalling controller was installed in Toronto, Canada, in 1963 [10].

Existing systems can historically be grouped into three main categories [10, 11]. *Fixed time* control strategies are calculated off-line, based on historical data about traffic flow at particular times of day at given intersections. Because they cannot adapt to the actual traffic situation, they are only appropriate for dealing with under-saturated traffic conditions. TRANSYT and MAXBAND are well-known members of this category [20, 25]. *Adaptive* strategies employ real-time control systems, calculating their policies based on the actual current traffic as determined from sensor readings. SCOOT and SCATS are two prominent members of this group.

More modern methods of the so-called *third generation* differ from traditional approaches in that they are not concerned with separately optimising cycle times, splits and offsets. Given a pre-specified phase scheme and a sophisticated dynamic traffic prediction model, they try to find optimal lengths for all phases, solving a dynamic optimisation problem on a discretized time representation. CRONOS and COP use are two examples that use dynamic programming [19]. The exponential complexity of the problem, however, limits these approaches to the local view of a few intersections at a time, instead of aiming for a global optimum. Heuristic optimisation methods as employed in PRODYN allow scaling up to bigger networks [19].

A fourth approach that has become popular are the so-called *store-and-forward* based techniques [19]. They use simplified traffic models, thus trading accuracy of the models against computation time for policy calculation. They are based on the fact that optimisation with real-valued variables can be solved in polynomial time in contrast to the NP-hardness of problems involving integer valued variables. One simplification of the optimisation problem used by these methods, for example, is to describe the outflow of an intersection with an average value at any time, instead of making the distinction of flow zero during a red phase and a positive flow during green.

2.1.3 TRANSYT

TRANSYT (the Traffic Network Study Tool) is probably the best-known off-line system available [20]. It was developed in the 1960s by Robertson [25] and substantially extended later on. It is a global optimisation method, i. e., it employs a macroscopic model of the traffic network based on historical data and calculates the control policies for all intersections jointly, using a mix of heuristic search algorithms like hill-climbing and genetic algorithms. Given a cycle length, it computes

split and offset, optimising a performance index consisting of waiting times and the number of stops in the whole network. Other performance criteria like fuel consumption can also be optimised. Because of its use of heuristic search techniques it is not guaranteed to find the optimal control policy [20].

2.1.4 SCOOT

SCOOT (the Split Cycle Offset Optimisation Technique) [26] is often referred to as the traffic adaptive version of TRANSYT, and is among the most popular systems deployed today. It was developed by the Transport and Road Research Laboratory (TRRL), UK, in 1973 and is in use in many European cities like London and Madrid, as well as in Bangkok, Beijing, Toronto and other major cities throughout the world. It has since been under continuous development, with Siemens Traffic Control Ltd. and Peek Traffic Ltd. being industrial partners in the project.

SCOOT works with loop detectors located at a certain distance from each intersection (typically some 100–300 meters), from which it calculates the expected car arrival profile. Using this information, it optimises cycle time, split and offset with a hill-climbing algorithm. Like TRANSYT, it aims for minimisation of waiting time and number of stops across the whole network. The cycle time here is identical for groups of intersections in the network, facilitating offset calculation. In addition to this global base schedule, SCOOT makes small incremental optimisation steps for each controller. Every controller can decide to advance or retard a scheduled phase change by up to 4 seconds, if that improves the local performance index. Once every cycle, the controller also checks whether incrementing or decrementing the offset by 4 seconds would be preferable. Similarly, the cycle time of groups of intersections may be adapted by a few seconds up or down.

One disadvantage of having the loop detectors installed far from the intersections is that saturated traffic conditions are registered late, when queued traffic backs up to the detectors. This means that SCOOT cannot employ measures that keep traffic back at the borders of a congested subsystem. Instead, it switches to special measures once it registers the saturated conditions, its primary goal then being not minimisation of travel time but the dispersion of queued up traffic [20].

2.1.5 SCATS

SCATS (the Sydney Coordinated Adaptive Traffic System) [30] is one of the oldest adaptive systems. It was introduced in 1964 by the New South Wales Roads and Traffic Authority (RTA), and runs computer-based since 1972, being employed in most major Australian and New Zealand cities as well as in many cities throughout South East Asia and North America.

SCATS is a decentralised system and does not involve a complicated traffic model. It uses a fixed phase scheme and recalculates the cycle time, split and offset of each intersection once every cycle. It works on two levels: the tactical level, where cycle time and splits are determined for each intersection separately, and the

strategic level, where intersections are coordinated. The system is supported by a set of pre-specified plans, which contain phase schedules for all intersections for certain times of the day, as calculated from historical data. These plans are loaded regularly and are subsequently modified according to current conditions.

At each intersection, a local SCATS controller measures the traffic volume on every link with the help of loop detectors. These are located closely to the stop lines of the intersection. Once every cycle, new target values for the cycle time and split of each intersection are derived from a saturation-balancing algorithm (see the glossary in Section 2.1.1), which calculates the phase lengths in such a way that a certain level of saturation is achieved for the most used stream of any phase. The cycle time is then simply the sum of the phase lengths. SCATS does not employ the new values directly, but adjusts its current plan by a small step towards the new values.

For achieving offsets, the network is divided into sections of up to 10 – 20 intersections which can be coordinated. Again, SCATS relies on pre-specified plans for offset values. If neighbouring intersections have a similar cycle time, they can decide to *marry*, both adopting the longer cycle time and reading the corresponding offset from one of the offset plans.

Systems that perform small incremental changes of their schedules like SCATS are especially suitable for stable and slowly changing traffic demands. However, rapidly changing traffic conditions pose a problem to them, as they are not able to adapt quickly enough. Furthermore, SCATS is purely *reactive*: because the loop detectors are located closely to the intersections, SCATS can only adapt its policy to traffic that has already arrived at the intersection, instead of anticipating changed demand. SCATS does not optimise the global network performance. A further disadvantage is that it needs prior knowledge about expected traffic volumes and involves a substantial amount of hand-tuning for calculation of the base phase and offset plans.

We are particularly interested in SCATS, as our work was inspired by a project launched in cooperation with the New South Wales Roads and Traffic Authority. Our focus, when setting up the learning system, was largely motivated by the desire to compare against SCATS and demonstrate better performance in those areas where SCATS is known to have weaknesses. We therefore evaluated the performance of our controllers under conditions compatible with SCATS, and designed an approximation to the adaptive component of SCATS as a baseline. However, due to the fact that SCATS is a proprietary system, our emulation of SCATS is limited to the publicly known facts about the system, which means that our emulation might differ slightly from the way SCATS actually works. Section 5.1.1 describes our implementation in detail.

2.1.6 Shortcomings of Traditional Systems

Summarising the description of traditional approaches to traffic control, we can identify the following weaknesses:

1. Most existing methods rely on a model of the traffic flow. This makes them prone to suboptimal behaviour if the employed model lacks accuracy or if unforeseen events occur. Moreover, most systems do not have a mechanism for learning from feedback on the quality of their model, which may lead to systematic errors [9].

2. Many of the more sophisticated optimisation methods do not scale up to large networks. Instead, independently optimised subsets of intersections are combined, possibly resulting in sub-optimal overall performance.

3. Over-saturated traffic conditions can rarely be handled adequately [20].

4. Many systems have problems with traffic incidents or rapidly changing demands, as they are slow to react, only allowing incremental changes to their pursued policy.

Chapter 3

Reinforcement Learning

The key idea of reinforcement learning is very simple: Humans or animals often learn from positive or negative feedback for their actions (see Rescorla and Wagner [23] for an introduction and examples in psychological studies). Using the same idea, an artificial agent can be made to learn from so-called *reward signals* about the utility of its actions in a given situation or *state*, thereby weakening or strengthening the probability of the corresponding behaviour in that state in the future.

The property of needing little or no previous knowledge about a problem distinguishes reinforcement learning from other forms of machine learning. In traditional supervised learning, for example, a learning algorithm is given examples for correct answers to questions and the goal is to generalise from the seen examples [29]. In this setting the human supervisor needs to know the correct answer to at least a subset of the questions, in order to construct training examples. In many cases, however, when the problem is complex, we do not have the necessary knowledge to construct explicit training examples. At the same time we do often have some idea about the desirability of a given world state. The advantage of reinforcement learning is that it suffices to be able to judge the utility of a state relatively to others, in order to ultimately learn the right behaviour in any state. Supervision here only consists of giving the learner rewards or punishments upon reaching desirable or undesirable states, respectively. It is then the learner's task to determine which of its actions were responsible for leading to that state, and hence which actions were good or bad in the corresponding previous states.

In more detail, the framework for reinforcement learning is as follows: at every time step, the agent chooses an action applicable in the current state. The environment then reacts to that action by transitioning into a new state (deterministically or stochastically), and the agent receives a scalar reward signal. In the general case, this reward can depend on the action chosen in a given state and the next state reached. Figure 3.1 depicts the interaction of agent and environment graphically.

Most algorithms for reinforcement learning assume the environment to be a Markov decision process (MDP), i. e., a system in which the state transitions only depend on the action chosen by the agent and the current state, not on past events.

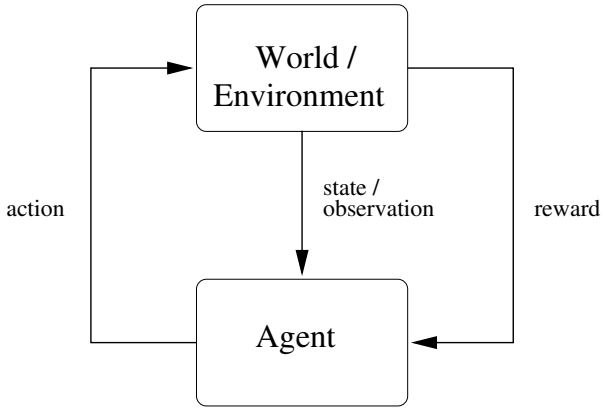


Figure 3.1: Interaction between the learning agent and its environment.

If this is the case, the *Markov property* (after the Russian mathematician Andrei A. Markov) is said to hold, or the environment is *Markovian*. A formal definition of an MDP will be given in the next section. There are numerous variations of MDP-based problems:

- The world model (i. e., state transition probabilities and rewards) may or may not be known to the learner. If the world model is known, learning is not necessarily needed and a solution can be calculated analytically.
- The learner may or may not be able to reliably identify the current state. If this is not the case, we speak of *partial observability* of the environment.
- Environment interaction sequences (so-called *episodes*) can have finite or infinite length. In the first case, the problem often is to find a sequence of actions that lead to a "goal state", while in infinite episodes we want to maximise the accumulated rewards.

Traffic light optimisation is indeed a very difficult task, considering the above points. The learner does not know the exact model of the environment (how will traffic flow react to a certain signalling policy?), we only have partial observability (it is not possible to measure location, speed, and intended route for every vehicle in the system), and we must deal with an infinite horizon, since traffic does not stop.

We will now define Markov decision processes formally and present some popular algorithms for finding policies in them. Following that, Section 3.2 describes some of the general difficulties involved with reinforcement learning, while Section 3.3 extends the MDP formalism to include partial observability. Policy-gradient methods for finding policies in MDPs or POMDPs are presented in Section 3.4. The following section and Section 3.2 are largely based on the textbook by Sutton and Barto [32].

3.1 Markov Decision Processes

A Markov Decision Process (MDP) is defined by a tuple $M = \langle S, A, T, R \rangle$, where

- S is a set of states,
- A is a set of actions,
- $T : S \times A \rightarrow \prod(S)$ is a state transition function mapping a state and an action to a probability distribution over the states, and
- $R : S \times A \rightarrow \mathbb{R}$ is a reward function mapping a state and an action to an immediate scalar reward.

The agent interacts with the system defined by the MDP by selecting, at each time step, an action $a \in A$ in the current state $s \in S$ of the MDP. The system then transitions to a new state $s' \in S$, as sampled from the probabilistic function $T(s, a)$, and the agent receives the reward signal $r \in \mathbb{R}$, where $r = R(s, a)$. While we defined the reward function to be deterministic, all equations and algorithms in this thesis hold for stochastic reward functions as well, if $R(s, a)$ is seen as the *expected* reward of action a in state s.

The action-selection of the agent is defined by its *policy*, which in general is a function π mapping states to a probability distribution over the actions, $\pi : S \rightarrow \prod(A)$. The policy thus specifies, for each state, a probability for the execution of each action. We will need the stochasticity of policies later on, when we introduce partial observability. However, for now we assume *deterministic* policies, hence the probability $\pi(s, a)$ equals one for exactly one action in state s and zero for all other actions.

We also assume finite MDPs, i. e., MDPs with finite state and action spaces. We will use subscripts to denote states, actions and rewards at particular time steps, where r_{t+1} is the reward obtained by the agent after choosing action a_t in state s_t at time step t.

3.1.1 Acting Optimally

The goal of the agent is to find an optimal policy, i. e., a policy that maximises the long-term reward sequence obtained from the environment. When trying to specify this high-level goal in more detail, however, several models of optimality are possible [32].

In a *finite-horizon* scenario, where the length of each interaction with the environment is limited by a fixed number of time steps T, we can simply aim to maximise the expected *sum* of rewards obtained in an interaction episode:

$$E\left(\sum_{t=1}^{T} r_t\right).$$

This is the easiest case, but in many real-world problems we cannot guarantee a finite horizon. Note also that the value of this expectation depends on the starting state.

In infinite-horizon cases, where the interaction with the environment is unbounded in length, the *discounted* model has become popular. Here, a discount factor γ, where $0 \leq \gamma < 1$, is used to discount future rewards:

$$E\left(\sum_{t=1}^{\infty} \gamma^{t-1} r_t\right).$$

By discounting future rewards, we ensure convergence of the infinite sum of rewards. However, this has further implications for the agent: since future rewards are worth less, it is preferable for the agent to obtain a reward immediately than to obtain the same reward a few time steps later. In some applications this behaviour is indeed desirable. For example, the rewards could represent money that loses some of its value over time through inflation. In many cases, the discounting of rewards is simply used as a computational trick to solve an infinite-horizon problem. However, the implications of choosing this model have to be examined carefully. Like the expected sum of rewards, the expected discounted reward also depends on the starting state of the episode.

A third, and perhaps the most natural model, is the *average reward model*,

$$\lim_{T\to\infty} \frac{1}{T} E\left(\sum_{t=1}^{T} r_t\right),$$

where an agent aims to maximise its expected average reward per time step. This value does not depend on the starting state. However, algorithms for finding policies that match this criterion seem more complicated and have not been investigated thoroughly yet [13]. The term *return* is generally used to denote a target function of the reward sequence, like one of the above.

3.1.2 Classical Algorithms for Solving MDPs

In this section, we present some of the most popular algorithms for finding policies in MDPs, as they form the historical and theoretical basis of algorithms for the more general case of partially observable MDPs. We introduce the concepts of value functions, greedy policies, temporal difference, and eligibility traces, which we will build on later.

The definition of a learning environment as an MDP prescribes that it is stationary, i. e., the transition function and reward function do not change in time. When the transition function and the reward function (the so-called *model* of the MDP) are known to the agent, no learning is actually needed. An optimal behaviour in this system can be calculated by solving a set of linear equations. It has been shown that for every finite MDP there exists a stationary deterministic policy that is optimal [27]. All of the algorithms we present are guaranteed to asymptotically find such an optimal policy.

Classes of algorithms

One fundamental issue when talking about MDP algorithms is whether or not they require the agent to know the model of the MDP. Assuming this knowledge, solutions can be calculated analytically, as mentioned above, or by *dynamic programming* (DP) methods. If the model is not known beforehand, a *reinforcement learning* approach is needed that finds out about the environment through interacting with it. In this case, one can either try to learn the model through experience, observing the frequency of the respective state transitions and rewards, and then employ a DP-like method (this is the *model-based* approach). Or, one can try to learn a state-action mapping directly by searching the space of possible policies, bypassing model estimation. Such algorithms are *model-free* methods. We will discuss the relative merits of the latter class in more detail in Section 3.4, where we introduce policy-gradient methods as members of this category.

In the following, we present some classical algorithms for finding policies in MDPs, including both model-based and model-free approaches. We first define the *value* or *utility* of being in a certain state, and similarly the values of actions in a given state. These values correspond to the total return expected from the state or action, respectively. We then present two dynamic programming algorithms, Value Iteration and Policy Iteration, that calculate state values from the environment model and derive policies from them. Policies are derived by selecting, in each state, the action that leads to the state with highest utility. Temporal difference (TD) learning methods are introduced next. They do not assume a model of the environment, but learn the values for states or actions through environment interactions. As all of these methods work by calculating values for states or actions, they are called *value-based* methods.

Value Functions

In the following, we use the discounted reward measure used in most classical algorithms [32], allowing however the discount factor γ to equal 1 if termination of each episode is guaranteed. That way our definitions encompass the sum of rewards measure in those cases where it is well-defined.

We define the value of a state s under policy π, $V^\pi(s)$, as the expected return when starting in state s and following π [32]:

$$V^\pi(s) := E\left(\sum_{t=1}^{\infty} \gamma^{t-1} r_t \middle| s_0 = s\right). \tag{3.1}$$

Similarly, the value of a state-action tuple under π, $Q^\pi(s,a)$ is defined as the expected return when starting in state s, taking action a there and following π from then on:

$$Q^\pi(s,a) := E\left(\sum_{t=1}^{\infty} \gamma^{t-1} r_t \middle| s_0 = s, a_0 = a\right). \tag{3.2}$$

The functions V^π and Q^π are called the state-value function and the action-value function for policy π, respectively.

Let in the following a and a' always be actions from A, and s and s' be states from S. We simplify notation and write $\sum_a$ for $\sum_{a \in A}$, and use the $Pr(\cdot)$ notation to denote a probability, e. g., $Pr(s'|s,a) = T(s,a,s')$ is the probability of the environment transitioning to state s' given the agent chooses action a in state s.

The famous Bellman equations can be directly derived from the definitions in Equations (3.1) and (3.2). They state that the values of neighbouring states are tied together:

$$V^\pi(s) = \sum_a \pi(s,a) \left(R(s,a) + \gamma \sum_{s'} Pr(s'|s,a) V^\pi(s') \right) \tag{3.3}$$

$$Q^\pi(s,a) = R(s,a) + \gamma \sum_{s'} Pr(s'|s,a) V^\pi(s'). \tag{3.4}$$

An optimal policy π^*, i. e., one that maximises the expected return among all policies, needs to take an optimal action at each step,

$$\pi^*(s,a) = 1 \Rightarrow a \in \arg\max_{a'} \left(R(s,a') + \gamma \sum_{s'} Pr(s'|s,a') V^*(s') \right), \tag{3.5}$$

where $V^*(s)$ is the highest value s has under any policy, $V^*(s) := \max_\pi V^\pi(s)$. The reverse is also true: a policy that chooses an optimal action in each state (i. e., a policy that fulfils (3.5) for all s and a), is an optimal policy [29]. Therefore, all optimal policies have exactly V^* as their state-value function, and in an analogous manner they share the optimal action-value function Q^* [32]. For those optimal value functions in particular, the Bellman optimality equations hold:

$$\begin{aligned} V^*(s) &= \max_a \left(R(s,a) + \sum_{s'} Pr(s'|s,a) \gamma V^*(s') \right) \\ &= \max_a \left(R(s,a) + \gamma \sum_{s'} Pr(s'|s,a) V^*(s') \right) \end{aligned} \tag{3.6}$$

$$\begin{aligned} Q^*(s,a) &= R(s,a) + \sum_{s'} Pr(s'|s,a) \gamma V^*(s') \\ &= R(s,a) + \sum_{s'} Pr(s'|s,a) \gamma \max_{a'} Q^*(s',a') \\ &= R(s,a) + \gamma \sum_{s'} Pr(s'|s,a) \max_{a'} Q^*(s',a'). \end{aligned} \tag{3.7}$$

Value Iteration

The Value Iteration algorithm calculates a state-value function V by iteratively applying the Bellman optimality equation (3.6) to initially random state values. It

updates, at each step, every state value estimate according to its old value estimates of the neighbouring states:

$$V_{t+1}(s) := \max_a \left(R(s,a) + \gamma \sum_{s'} Pr(s'|s,a) V_t(s') \right).$$

This algorithm provably converges to the optimal state-value function V^* [29]. After convergence, we can construct an *optimal* policy by choosing actions greedily with respect to the state-value function, i. e., in each state s we choose an action a that is optimal with respect to V^* as given by the right-hand side of the implication in Equation (3.5).

Policy Iteration

The Policy Iteration algorithm iteratively evaluates and improves a policy. It starts with a random policy. In each iteration, it calculates the state-value function V^π of the policy, and then creates a new policy π' which is greedy with respect to V^π. As the actions chosen by π' lead to at least as much return from each state as the actions chosen by π, π' is at least as good as π. Furthermore, when the policy does not change anymore, the state values V^π fulfil the Bellman optimality equation (3.6), hence $V^\pi = V^*$, and we have converged to an optimal policy [32].

The state-value function V^π can be calculated analytically or iteratively, in a similar way as Value Iteration calculates the optimal state values V^*:

$$V^\pi_{t+1}(s) := \sum_a \left(\pi(s,a) R(s,a) + \gamma \sum_{s'} Pr(s'|s,a) V^\pi_t(s') \right). \tag{3.8}$$

This process provably converges to the true state-value function under π, V^π, as defined in (3.1).

Temporal Difference Learning

Dynamic programming methods like Value Iteration and Policy Iteration are guaranteed to asymptotically find the optimal policy, assuming that a complete model of the environment is known. But as the state space grows, calculating the exact value of every state soon becomes very costly. One iteration of the Value Iteration algorithm, for example, has complexity $O(|S|^2|A|)$, and potentially many iterations are necessary.

However, in very large state spaces we may not need to know the exact value of every state, or of the actions in that state, to act well. This is because the policy only depends on the *relative* values of states and actions, which can usually be estimated long before convergence to the true value function. Furthermore, there may be many states that are rarely ever visited during a typical episode, and for which we do not necessarily need very good estimates. An alternative approach therefore estimates the value functions through experience, by taking actions and

observing their outcome. That way computational efforts are concentrated on the states which occur frequently.

Gathering experience through environment interaction is often termed Monte-Carlo exploration. Sutton and Barto [32] reserve the term Monte-Carlo for a class of algorithms that complete finite interaction episodes, and estimate the transition probabilities and rewards by averaging over the values observed during the episodes. We will however use Monte-Carlo exploration for any learning by experience, as described above.

The Monte-Carlo algorithms in the restricted sense of Sutton and Barto have the drawback that they can only be applied to episodic tasks, since they wait until an episode has terminated to update their estimated values. Another class of algorithms that lies between dynamic programming methods and true Monte-Carlo algorithms are the temporal-difference learning (TD) methods. They learn value functions by iteratively executing actions according to their current policy, and updating their estimates of the value functions after every step, depending on the observed rewards.

TD(λ)

The TD(λ) family of algorithms do not actually learn a policy, but are presented to demonstrate the principle of the temporal difference learning approach. They learn the values of states under a given policy, and can thus be used to evaluate policies. The simplest and best-known temporal-difference algorithm, TD(0), iteratively executes an action according to π and updates its value estimates by a small amount depending on the observed reward. Let V_t be the state-value function estimate at time step t. After executing an action in the current state s_t and receiving the reward r_{t+1}, TD(0) updates its value estimate of s_t by moving its value function towards the equilibrium prescribed by the Bellman equation (3.3):

$$V_{t+1}(s_t) := V_t(s_t) + \Delta V_t(s_t),$$

where

$$\Delta V_t(s_t) := \alpha_t[(r_{t+1} + \gamma V_t(s_{t+1})) - V_t(s_t)]$$

is the so-called temporal difference. Thus, TD(0) updates its value estimates based on the value estimates of neighbouring states, like dynamic programming methods do (this is called bootstrapping). However, unlike DP, TD(0) updates its values based on one *observed* sample transition instead of all possible transitions to neighbouring states. Instead of exactly *averaging* over sampled rewards for an entire sample trajectory like Monte-Carlo algorithms, TD(0) performs updates with small, fixed step sizes α_t possibly decreasing over time. It converges to the true state-value function in the mean, or with probability 1 for decreasing step sizes [32].

The general TD(λ) algorithm does not only observe one transition, but updates its value estimates based on *all future rewards* in the (possibly infinite) episode. A

discount factor λ ensures that the infinite sums of rewards exist, and defines by how much the value of a state is updated for a reward received at a certain later point in time. We define the n-step return following time step t, $R_t^{(n)}$, as the discounted reward we expect when observing n sample steps of the environment and using our current value estimates after that:

$$R_t^{(n)} := \sum_{k=0}^{n-1} \gamma^k r_{t+k+1} + \gamma^n V_t(s_{t+n})$$

The temporal difference of TD(λ) is then defined as follows:

$$\Delta V_{t+1}(s_t) := \alpha_t \left((1-\lambda) \left(\sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(n)} \right) - V_t(s_t) \right) \tag{3.9}$$

Thus, TD(λ) uses a weighted average over all n-step returns. The factor $(1-\lambda)$ ensures that the weights sum to 1. Although the definition of the weighted average in the above update formula seems rather complicated, there is an easy implementation for it. An online algorithm cannot complete, at each time step t, the update corresponding to this time step as given in (3.9) (because $R_t^{(n)}$ depends on future time steps). Instead we have to keep updating $V(s_t)$ during all future time steps. This can be implemented with the help of *eligibility traces*. An eligibility trace here is an additional variable for each state, keeping track of how often and how recently a state was visited, in short: how eligible a state is for updates whenever a reward occurs. This reflects how responsible each state is deemed to be for a reward occurring at a later time step. On each step, the eligibility trace for state s is updated as follows:

$$z_{t+1}(s) = \begin{cases} \gamma\lambda z_t(s) & \text{if } s \neq s_{t+1} \\ \gamma\lambda z_t(s) + 1 & \text{if } s = s_{t+1} \end{cases}$$

The update performed on the value estimates at time step t is then

$$\Delta V_{t+1}(s) = \alpha_t [r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)] z_t(s)$$

for all $s \in S$. For a diminishing step size sequence $(\alpha_t)_{t\in\mathbb{N}}$, this implementation is equivalent to (3.9) [6].

Eligibility traces are thus a means of crediting states for rewards obtained later on. Decaying them exponentially can be seen to express the assumption that rewards are exponentially more likely to be connected to states visited recently than to states visited longer ago. They present one way to solve the temporal credit assignment problem (cf. Section 3.2), and our policy-gradient algorithms will be using them in a similar form.

SARSA and Q-Learning

SARSA and Q-Learning are two temporal-difference learning algorithms that learn state-action values. SARSA learns the action-value function Q^π of its current policy, and continually improves its policy with respect to these values, in a similar way to Policy Iteration. The Q-Learning algorithm, on the other hand, learns the *optimal* action-value function Q^* directly. The policy it follows during learning has no influence on the learned value estimates, hence Q-Learning is a so-called *off-policy* method. We first describe SARSA and subsequently Q-Learning, pointing out the differences between the two algorithms.

SARSA updates its value estimates based on the experience of two time steps, by first executing action a_t in state s_t and observing the reward r_{t+1}, and then executing action a_{t+1} in the resulting next state s_{t+1}:

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha_t \left[(r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1})) - Q_t(s_t, a_t) \right].$$

The name SARSA is derived from this update rule, as it involves the quantities $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$. The Q-values SARSA learns depend on the actions prescribed by its policy, thus SARSA learns the action-value function of its current policy. However, it updates its policy at each time step to exploit the learned Q-values, and is guaranteed to asymptotically converge to the optimal value function and an optimal policy. Conditions for convergence are again decreasing step sizes α_t, and that all state-action tuples are visited infinitely often, to learn their correct values. (A corresponding condition is required for the TD(λ) algorithm and Q-Learning.) To ensure that all state-action tuples continue to be visited, SARSA must use a slightly stochastic policy instead of always choosing a greedy action with regard to Q^π. One possibility to do this is to usually choose greedy actions, but with a small, decreasing probabilities choose a non-greedy action (cf. Section 3.2). The traditional SARSA algorithm as defined above performs its value updates based on *one-step* temporal-difference look-aheads, like TD(0). SARSA can be modified to use discounted infinite look-aheads like a TD(λ) algorithm does, by using eligibility traces. This leads to a family of algorithms SARSA(λ).

The Q-Learning algorithm learns the *optimal* action-value function Q^* directly. Its value estimates are updated at each time step according to the following rule:

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha_t \left[\left(r_{t+1} + \gamma \max_a Q_t(s_{t+1}, a) \right) - Q_t(s_t, a_t) \right].$$

Because of the maximisation operator in the above formula, the update of the value estimate for a state-action tuple is based on the reward observed for that tuple and the estimates for the best neighbouring state-action tuple. The policy pursued thus has no influence on the learned values, as long as it guarantees to visit all states infinitely often. Q-Learning and SARSA are true reinforcement learning algorithms, as they learn a policy without assuming knowledge of the world model.

3.2 Challenges in Reinforcement Learning

Exploration versus Exploitation

In most RL settings, from the beginning, the agent learns by interacting with the environment. It continually needs to choose actions depending on its current state and knowledge about the world. (The only exception being if we can learn from observing someone else interacting with the environment.) This naturally leads to a problem known as the *exploration-exploitation trade-off*: should the agent exploit its current knowledge and choose the actions that seem most promising? Or should it explore the world by choosing actions which it has less information about, in order to accumulate information and possibly find better actions in the future?

A popular approach to solve this problem is to use non-deterministic action selection, i. e., with a small probability select an action that is not deemed optimal, but which helps to gain information about the environment. The ϵ*-greedy* action selection is the easiest implementation of such a strategy: with probability $1-\epsilon$ the learner selects the action it currently thinks best, and with probability ϵ it selects a random action. Another possibility is to use a *softmax* selection rule, which weighs the probability of selecting an action by its estimated utility. Assume a preference function $p(s,a)$ that specifies some estimate of the utility of action a in state s, e. g., $p(s,a) := Q(s,a)$ for all s and a. The most common softmax function is to use a Boltzmann, or Gibbs, distribution of that preference function, where τ is a positive "temperature" parameter (possibly decreasing with time):

$$\pi(s,a) := \frac{e^{\frac{p(s,a)}{\tau}}}{\sum_{a'} e^{\frac{p(s,a')}{\tau}}}.$$

The Temporal Credit Assignment Problem

A further difficulty that arises frequently in reinforcement learning problems is the delay of reward. Often, the agent receives a big positive or negative reward only after a long sequence of interactions with the environment, for example at the end of a game. Which of its actions in the sequence should the agent credit for getting that reward? If the agent loses the game, it may very well be that it played flawlessly during most of the time, but made one very bad move which led to losing the game. This is called the *temporal credit assignment problem*. It is especially hard in infinite horizon tasks, as there is no bound on how much delay rewards may have. Discount factors can be used to impose an artificial horizon by assuming that rewards are exponentially more likely to be due to recent actions (see the explanation of eligibility traces in Section 3.1.2).

Large State Spaces and Partial Observability

More complications arise when we try to solve large and complex real-world problems with reinforcement learning. Many real-world problems have a prohibitively

large state space that cannot be enumerated explicitly. Here, the classical versions of our algorithms become intractable and we need to resort to some kind of approximation for some or all of the relevant variables and functions in the problem, e. g., the state space, the value functions, and the policy representation. The approximate value-function approach, where one tries to approximate the state-value or action-value functions and derive a policy from those approximations, has a significant drawback: as opposed to an exact value function, there is no guarantee anymore that the resulting greedy policy will be optimal. Even if the optimal policy is representable in the chosen approximation framework, no guarantees exist for value-based algorithms to converge to this policy [6, 33]. This is because small changes in the value function may cause discontinuous changes in the policy [6]. Alternatively, the *policy* can be approximated, e. g., by a parameterised function of the state descriptions, instead of explicitly storing an action for each state. This is the approach taken by policy-gradient methods (see Section 3.4).

Finally, in real-world problems the agent may not always be able to determine the current state of the environment exactly. That may be because the agent relies on sensor measurements that only reflect some aspects of the current state. In such *partially observable* environments learning is much harder, and the agent usually requires memory about past actions to act well. Partially observable MDPs are introduced formally in the next section, where we also discuss their complexity and approaches to solving them.

3.3 Partially Observable MDPs

Many types of problems can be cast as MDPs. However, MDPs make strong assumptions about how well the learner can perceive its environment – namely that it can identify every state reliably. In many realistic applications this requirement is not given. Noisy sensor readings, or an infinite state space that can only be perceived through a discrete number of different sensor readings, are reasons why an agent may not be able to identify the exact state of the environment. A generalisation of MDPs that is able to express these kinds of problems are the partially observable MDPs, or POMDPs.

A Partially Observable Markov Decision Process (POMDP) is defined by a tuple $M = \langle S, A, \Omega, T, R, O \rangle$, where

- S is a set of states,
- A is a set of actions,
- Ω is a set of observations,
- $T : S \times A \to \prod(S)$ is a state transition function mapping a state and an action to a probability distribution over states,

- $R : S \times A \to \mathbb{R}$ is a reward function mapping state-action tuples to immediate scalar rewards, and
- $O : S \to \prod(\Omega)$ is an observation function mapping states to a probability distribution over the observations.

Again, we will in the following assume finiteness of the state, action and observation spaces. In this framework, an agent does not know the exact state of the environment at every time step. Instead, it senses an observation associated with the state (where an observation is often a vector of various observation features). States that lead to the same observation can now not be distinguished by the learner unless it has additional information, possibly attained through past interactions with the environment.

One possibility, for example, is to maintain a *belief state* representation in the learner, where a belief state is a probability distribution over all possible states, which is updated at every time step according to the agent's model of the world and the new observations encountered. Other approaches are to keep track of the last n interactions with the environment (finite window methods) or to maintain an internal state in the learner that dictates the choice of action, given an observation (finite state controllers).

POMDPs are much harder to solve than MDPs. While algorithms for explicitly represented MDPs run in polynomial time and the policy existence problem, i. e., the question whether there is a policy with expected reward $\geq c$, is P-complete [18], policy existence for POMDPs is undecidable for all popular performance criteria, even if we assume knowledge of the underlying MDP model [16]. When we restrict ourselves to stationary, i. e., memoryless policies, the policy existence problem is in NEXP for the average reward criterion [24].

Using memoryless policies in POMDPs means that we ignore partial observability and treat observations as sufficient indicators of the underlying world state. However, memoryless policies will most often not be optimal in POMDPs. In fact, the best memoryless policy can be arbitrarily worse than the optimal policy of the underlying MDP [31]. Nevertheless it often makes sense, in the face of intractability, to consider only memoryless policies. Many real-world problems may have good memoryless or low-memory policies, where a simple mapping from the sensings of the learner to actions is sufficient for good performance [15]. For the traffic control problem, we restrict ourselves to low-memory policies, which indeed prove to be effective.

If memoryless policies are deemed to be appropriate for a given POMDP, it is possible to employ traditional MDP algorithms like the ones presented in Section 3.1.2. However, SARSA, Q-Learning and many other value-based methods may fail to converge for POMDPs [6]. Eligibility traces can be used to partially overcome these problems [15]. As they allow updating the value of states based on all future rewards, the uncertainty about the (value of) the current state may be resolved gradually, as future rewards are observed. SARSA(λ) has shown to yield good results in a number of standard problems from the POMDP literature [15]. A

second disadvantage of value-based methods is that they usually produce deterministic policies, while the optimal policy for POMDPs may be stochastic [31]. The inability of the learner to distinguish states makes it easy to construct examples where a stochastic policy outperforms the best deterministic memoryless policy.

3.4 Policy Gradient Ascent

An effective way to avoid the disadvantages of value-based methods under partial observability is to search in the space of policies directly. Policy gradient (PG) ascent methods do this by performing gradient ascent on the parameters of a parameterised policy function, adjusting the parameters into the direction of higher rewards. They are guaranteed to converge to a *local* optimum of their optimisation function (e. g., the expected long-term average reward) under appropriate conditions. In combination with Monte-Carlo-like exploration of the environment, they allow efficient handling of large state spaces.

3.4.1 Definitions and Assumptions

From now on, we assume a stochastic policy $\pi : \mathbb{R}^K \times \Omega \times A \to [0, 1]$ parameterised by a vector $\theta \in \mathbb{R}^K$. Let s_t be the state of the environment at time step t. The agent then perceives observation o_t, generated stochastically from the environment according to the observation function O, and chooses action a_t with probability $\pi(\theta, o_t, a_t)$. We will use the currying notation for functions, e. g., $\pi(\theta)$ to denote the function that takes an observation o and an action a and returns $\pi(\theta, o, a)$. Similarly, $O(s)$ denotes a probability distribution over all possible observations in state s, while $O(s, o)$ specifies the probability of seeing observation o in state s.

In the following, let $P(\theta)$ be the *global state transition matrix* of the POMDP given that the agent follows policy $\pi(\theta)$: $P(\theta)$ then is a stochastic $|S| \times |S|$ matrix where an entry $P(\theta)_{s,s'}$ represents the probability of the environment transitioning from state s to state s' in reaction to an action of the agent:

$$\begin{aligned} P(\theta)_{s,s'} &= \sum_{o \in O} \sum_{a \in A} Pr(o|s) Pr(a|\theta, o) Pr(s'|s, a) \\ &= \sum_{o \in O} \sum_{a \in A} O(s, o) \pi(\theta, o, a) T(s, a, s') \end{aligned}$$

We furthermore make the following assumptions:

Assumption 1: Each $\pi = \pi(\theta)$ for $\theta \in \mathbb{R}^K$ has a unique stationary distribution $d^\pi = [d^\pi(1), ...d^\pi(|S|)]$ satisfying the balance equation:

$$d^\pi P(\theta) = d^\pi.$$

If this holds, the Markov process specified by the POMDP and the policy π has at most one recurrence class, and the probability of being in a state s at time t for $t \to \infty$ is independent of the starting state: $d^\pi(s) = \lim_{t \to \infty} Pr(s_t = s|\pi)$.

Assumption 2: The derivatives

$$\frac{\partial \pi(\theta, o, a)}{\partial \theta_k}$$

exist and the ratios

$$\frac{|\frac{\partial \pi(\theta,o,a)}{\partial \theta_k}|}{\pi(\theta, o, a)}$$

are uniformly bounded by a constant $C < \infty$ for all $o \in \Omega, a \in A, \theta \in \mathbb{R}^K$, and $k = 1, ..., K$.

3.4.2 Objective Functions

Ideally, we would like to find policy parameters that maximise the *average* reward per step that we obtain by following the policy infinitely long, $J_a(\theta)$:

$$\begin{aligned} J_a(\theta) &:= \lim_{T\to\infty} \frac{1}{T} E\left(\sum_{t=1}^{T} r_t\right) \\ &= \lim_{T\to\infty} \frac{1}{T}\left(\sum_{t=1}^{T}\sum_{s} Pr(s_t = s|\pi) \sum_{a} \pi(\theta, s, a) R(s, a)\right) \\ &= \sum_{s} d^{\pi}(s) \sum_{a} \pi(\theta, s, a) R(s, a). \end{aligned} \tag{3.10}$$

However, most policy-gradient algorithms resort to some approximation of the gradient by introducing a discount factor γ in a similar way DP algorithms do (see Section 3.1.2), thus optimising the *discounted* return, which depends on the starting state s_0:

$$\begin{aligned} J_\gamma(\theta, s_0) &:= E\left(\sum_{t=1}^{\infty} \gamma^{t-1} r_t | s_0\right) \\ &= \sum_{t=0}^{\infty} \gamma^t \sum_{s} Pr(s_t = s|s_0) \sum_{a} \pi(\theta, s, a) R(s, a). \end{aligned} \tag{3.11}$$

The *expected* discounted return assumes that the probability of starting in a certain state is given by the stationary state distribution d^π, and is defined as

$$J_\gamma(\theta) := \sum_{s} d^{\pi}(s) J_\gamma(\theta, s).$$

Optimising the expected discounted return $J_\gamma(\theta)$ is equivalent to optimising the average reward $J_a(\theta)$, since $(1 - \gamma)J_\gamma(\theta) = J_a(\theta)$ holds [31]. Also, as γ approaches 1, *for every state* s the discounted return $J_\gamma(\theta, s)$ normalised by $(1 - \gamma)$ approaches the average reward: $\lim_{\gamma\to 1}(1 - \gamma)J_\gamma(\theta, s) = J_a(\theta)$ [6]. If we aim

to maximise this normalised discounted return from a given starting point s_0, we have the following objective function:

$$\begin{aligned} J'_\gamma(\theta, s_0) &:= (1-\gamma) J_\gamma(\theta, s_0) \\ &= (1-\gamma) \sum_{t=0}^{\infty} \gamma^t \sum_s Pr(s_t = s|s_0) \sum_a \pi(\theta, s, a) R(s, a) \\ &= \sum_s d^\pi_\gamma(s) \sum_a \pi(\theta, s, a) R(s, a), \end{aligned} \tag{3.12}$$

where $d^\pi_\gamma(s)$ is the *discounted* distribution of states encountered when starting in s_0 (for better readability, the dependence on s_0 is here implicit):

$$d^\pi_\gamma(s) := (1-\gamma) \sum_{t=0}^{\infty} \gamma^t Pr(s_t = s|s_0, \pi). \tag{3.13}$$

We will in the following generally use $J(\theta)$ to denote an objective function, and specify its definition when specific algorithms are described. Note that $J(\theta)$ does not depend on the state, as opposed to the value functions V and Q used by value-based methods.

3.4.3 A "Vanilla" Policy Gradient Method

Traditional policy gradient methods (also termed "vanilla" PG methods) aim to compute the gradient of the average or discounted reward $J(\theta)$ in the space of the parameters θ and update the policy parameters at each step with the following update rule:

$$\theta_{t+1} := \theta_t + \alpha \nabla J(\theta_t)$$

where

$$\nabla J(\theta) := \left[\frac{\partial J(\theta)}{\partial \theta_1}, ..., \frac{\partial J(\theta)}{\partial \theta_K} \right].$$

The problem is to estimate the gradient $\nabla J(\theta)$. As one example for how this can be done, we present the OLPOMDP algorithm by Baxter and Bartlett [5]. Baxter and Bartlett assume a reward function that only depends on the states. This formulation of MDPs is equivalently powerful to ours, but the state spaces of the two usually differ. However, our application indeed fulfils the requirement that rewards only depend on states, and we will thus assume action-independent rewards throughout this section.

Let $r = [R(1), ..., R(|S|)]$ be the vector specifying the reward for each state. Under Assumption 1, the expected long-term average reward is independent of the starting state and is equivalent to the product of the rewards and the stationary probability of being in each state under policy $\pi(\theta)$:

$$J(\theta) = \sum_{s \in S} d^{\pi}(s) R(s) = {d^{\pi}}^T r.$$

If the model of the underlying MDP were known, $\nabla J(\theta)$ could be calculated exactly with the following formula (for a derivation, see [5]):

$$\nabla J(\theta) = d^{\pi} \nabla P(\theta) [I - P(\theta) + \mathbb{1} d^{\pi}]^{-1} r,$$

where I is the identity matrix and $\mathbb{1}$ is a vector of ones. Baxter and Bartlett show that the term $[I - P(\theta) + \mathbb{1} d^{\pi}]^{-1}$ in the above formula exists and is equal to $\sum_{t=0}^{\infty} (P(\theta)^t - \mathbb{1} d^{\pi})$. Since the entries in each row of $P(\theta)$ sum to 1, we have $\nabla (P(\theta)\mathbb{1}) d^{\pi} = \nabla \mathbb{1} d^{\pi} = 0$, which, after some rewriting (see [1]), leads to

$$\nabla J(\theta) = d^{\pi} \nabla P(\theta) \left[\sum_{t=0}^{\infty} P(\theta)^t \right] r. \tag{3.14}$$

Since we do not know the model (or in cases where we want to avoid the heavy matrix inversion calculations involved with large state spaces), we can do a Monte-Carlo exploration of the environment and iteratively calculate an approximation of the gradient. Baxter and Bartlett [5] suggest to compute a *biased* estimate of the long term average reward gradient, Δ_T, over T time steps. In the limit, this estimate is guaranteed to converge to an *approximation* $\nabla_{\beta} J(\theta)$ of the true gradient:

$$\lim_{T \to \infty} \Delta_T = \nabla_{\beta} J(\theta),$$

where the approximation $\nabla_{\beta} J(\theta)$ satisfies:

$$\lim_{\beta \to 1} \nabla_{\beta} J(\theta) = \nabla J(\theta).$$

The bias factor β cannot be set arbitrarily close to 1 because it influences the *variance* of the estimate, which is proportional to $\frac{1}{(1-\beta)^2}$. The increasing variance, as β approaches 1, results from the increasing difficulty of the temporal credit assignment problem, as the artificial horizon increases (see Section 3.2). The choice of β is therefore a trade-off between the bias and the variance of the estimate. Specifically, the approximation $\nabla_{\beta} J(\theta)$ is given by

$$\nabla_{\beta} J(\theta) = d^{\pi} \nabla P(\theta) \left[\sum_{t=0}^{\infty} \beta^t P(\theta)^t \right] r. \tag{3.15}$$

Again, we refer to Baxter and Bartlett [5] for the proofs of the derivations and convergence statements.

Baxter and Bartlett show that the following equality holds:

$$\nabla_{\beta} J(\theta) = \lim_{T \to \infty} \frac{1}{T} \sum_{t=0}^{T-1} \frac{\nabla \pi(\theta, o_t, a_t)}{\pi(\theta, o_t, a_t)} \sum_{r=t+1}^{T} \beta^{\tau - t - 1} R(s_\tau), \tag{3.16}$$

where o_t and a_t are the observation perceived and the action taken at time step t, respectively, such that the truncation of (3.16) after T time steps,

$$\Delta_T := \frac{1}{T} \sum_{t=0}^{T-1} \frac{\nabla \pi(\theta, o_t, a_t)}{\pi(\theta, o_t, a_t)} \sum_{r=t+1}^{T} \beta^{\tau-t-1} R(s_\tau) \tag{3.17}$$

is the desired estimate converging to $\nabla_\beta J(\theta)$ in the limit for $T \to \infty$. The term $\frac{\nabla \pi(\theta, o_t, a_t)}{\pi(\theta, o_t, a_t)}$ in the above equation is what has been called the *likelihood ratio* in earlier work on the computation of gradients of loss functions. For shortness of notation, we will make use of the fact that

$$\frac{\nabla \pi(\theta, o_t, a_t)}{\pi(\theta, o_t, a_t)} = \nabla \log \pi(\theta, o_t, a_t),$$

where $\log$ will denote the natural logarithm throughout this thesis, and call the term $\nabla \log \pi(\theta, o_t, a_t)$ the "log gradient" of π. One way to compute (3.17) efficiently is to use an eligibility trace z_t (cf. Section 3.1.2) to store the discounted sum of the past "log gradient" terms:

$$z_{t+1} = \beta z_t + \nabla \log \pi(\theta, o_t, a_t).$$

Now $g_t := R(s_{t+1}) z_{t+1}$ is the immediate gradient estimate of $J(\theta)$ at time step t, and averaging over T time steps gives Equation (3.17).

Instead of collecting gradients over a certain time horizon T and averaging them in the end, the online algorithm OLPOMDP [5] uses the gradient $g_t = R(s_{t+1}) z_{t+1}$ at each time step to update its parameters. It thereby trades accuracy of the parameter updates against (often) faster convergence, due to the more frequent updates [7]. Its pseudo code is given in Algorithm 1.

Algorithm 1 OLPOMDP

$t = 1,\ z_1 = [0],\ \theta_0 = [0]$
$\epsilon =$ step size, $\beta =$ bias factor
while not converged **do**
    observe o_t (generated according to $O(s_t)$)
    generate action a_t according to $\pi(\theta_t, o_t)$
    receive $R(s_{t+1})$ (with next state s_{t+1} generated according to $P(\theta_t)_{s_t, s_{t+1}}$)
    set $z_{t+1} = \beta z_t + \nabla \log \pi(\theta_t, o_t, a_t)$
    set $\theta_{t+1} = \theta_t + \epsilon R(s_{t+1}) z_{t+1}$
    set $t = t + 1$
return θ_t

Like all direct policy search algorithms, policy gradient methods have the advantage that they are more suitable for dealing with large state spaces than value-based methods. The disadvantages of "vanilla" PG methods are their sometimes slow convergence and the high variance of the gradient estimates. The discount or bias factor and the step-size parameter are critical values, which are usually determined empirically.

3.4.4 Natural Policy Gradient Methods

As Amari [2] points out, the policy gradient in parameter space as described in the previous section may not be the most desirable policy update direction. Although "vanilla" gradient methods are guaranteed to converge to a local optimum of their performance function, convergence is often very slow, and estimation procedures suffer from high variance of the gradient estimates. We would also prefer an update procedure that is not dependent on the representation of the policy – i. e., if a policy can be represented by two different parameterisations, then we would like the updates made to the policies to be the same in both cases, only depending on the policies rather than the parameters.

This criterion is fulfilled by the *natural gradient* [2]. In a Riemannian space of parameters θ, the natural gradient of a function $J(\theta)$ is defined as

$$\widetilde{\nabla} J(\theta) := G^{-1} \nabla J(\theta),$$

where the matrix G is the metric of the Riemannian space [2]. Amari [2] proved that on the parameter space of a set of probability distributions (as defined, for example, by parameterised policies), the *Fisher information matrix* $F(\theta)$ is such a metric, hence $F(\theta) = G$. Given a family of probability distributions $p(x, \theta)$ parameterised by θ, it is defined component-wise as follows:

$$F(\theta)_{i,j} := E\left[\frac{\partial \log p(\theta, x)}{\partial \theta_i} \frac{\partial \log p(\theta, x)}{\partial \theta_j}\right].$$

For reinforcement learning in MDPs (where a policy is again conditioned on states rather than observations), we can use the Fisher information matrix corresponding to the *probabilities of sample trajectories*, which result from following the policy π, resulting in [22]

$$F(\theta)_{i,j} := E\left[\frac{\partial \log \pi(\theta, s, a)}{\partial \theta_i} \frac{\partial \log \pi(\theta, s, a)}{\partial \theta_j}\right].$$

The expectation here is over the states and actions occurring in an infinite sample path.

The natural gradient in policy space can thus be calculated using the Fisher information matrix as $\widetilde{\nabla} J(\theta) = F(\theta)^{-1} \nabla J(\theta)$. The Fisher information metric represents a distance over the space of policies π where two given policies always have the same distance, regardless of their representation. On the other hand, policies that only differ slightly with regard to their parameters can have a large distance, if the small change in parameter space corresponds to a large change with regard to the action probabilities. Gradient ascent using the natural gradient of policies thus has the advantage that it allows to quickly step over large plateaus in parameter space, where "vanilla" policy-gradient algorithms have difficulties. Like "vanilla" gradient ascent, natural gradient ascent is guaranteed to converge to a local optimum, as the angle between the natural gradient and the ordinary gradient is never larger than ninety degrees [22].

Kakade [14] provides experimental results, showing that in many cases the natural gradient as calculated from the Fisher information matrix is indeed superior to the "vanilla" gradient in terms of faster convergence and less frequent plateaus. His algorithm learns F online, i. e., by updating, at each time step,

$$f_{t+1} = f_t + \nabla \log \pi(\theta, s, a)(\nabla \log \pi(\theta, s, a))^T,$$

so that after T time steps $\frac{f}{T}$ is an unbiased estimate of F.

3.4.5 Policy Gradient With Approximate Value Functions

In Section 3.4.3 we gave an example of how a model-free estimate of the policy gradient can be computed by interacting with the environment. While this is an easy and straightforward approach, it completely ignores the relations of states, i. e., it does not make use of the fact that neighbouring states have similar values, as expressed in the Bellman equation. This contributes to the variance of such gradient estimates.

To remedy this problem, it is possible to calculate the policy gradient $\nabla J(\theta)$ using state-action values, which are in turn approximated from Monte-Carlo rollouts. Sutton et al. [33] proved the decisive property that *approximated* state-action values are sufficient to guarantee convergence to the true gradient under certain conditions. Using a slight adaptation of the *policy gradient theorem* of Sutton et al., the policy gradient in any MDP can be calculated as follows [22]:

$$\nabla J(\theta) = \sum_s d^\pi_\gamma(s) \sum_a \nabla \pi(\theta, s, a)(Q^\pi(s, a) - b(s)),$$

where for the normalised discounted reward model, $J(\theta)$ and d^π_γ are defined as in Equation (3.12) and Equation (3.13), Q^π is the usual discounted action-value function as in (3.2), and $b(s)$ is an arbitrary function of s. For the average reward model the same result holds with $d^\pi_\gamma := d^\pi$ (the stationary distribution), and Q^π also defined differently. The exact definition of Sutton et al. for Q^π in this case is somewhat unusual, but does not matter as Q^π will be replaced by an approximation.

To see that $b(s)$ can indeed be an arbitrary function of s, note that for any state s, $\sum_a \pi(\theta, s, a) = 1$, and hence $\sum_a \nabla \pi(\theta, s, a) = 0$. However, $b(s)$ can be useful in reducing the variance of the gradient estimate, if the Q^π-values are estimated from experience. It is then called a baseline [12]. For example, a baseline of rewards can be calculated by averaging all rewards obtained in an interaction with the environment (here, b does not depend on s). By subtracting the reward baseline from the expected return as in Equation (3.4.5), or directly from the obtained reward at each step, we get a more accurate estimate of the *relative* value of a state-action tuple.

The term $Q^\pi(s, a)$ (or $Q^\pi(s, a) - b(s)$, since $b(s)$ is arbitrary), in the above equation can be replaced by an approximation $f^\pi_w(s, a)$ satisfying

$$f^\pi_w(s, a) = (\nabla \log \pi(\theta, s, a))^T w \tag{3.18}$$

where w is a parameter vector, without affecting the unbiasedness of the resulting gradient estimate [33]. If Equation (3.18) holds, the approximation function is said to be *compatible* with the policy parameterisation. The resulting estimate is thus

$$\nabla J(\theta) \approx \sum_s d_\gamma^\pi(s) \sum_a \nabla\pi(\theta, s, a)(\nabla \log \pi(\theta, s, a))^T w. \tag{3.19}$$

As Peters et al. [22] point out, the fact that $\sum_a \nabla\pi(\theta, s, a) = 0$ means that $f_w^\pi(s, a)$ has zero mean with regard to the action distribution:

$$\begin{aligned}\sum_a \pi(\theta, s, a) f_w^\pi(s, a) &= \sum_a \pi(\theta, s, a)(\nabla \log \pi(\theta, s, a))^T w \\ &= \sum_a \pi(\theta, s, a) \frac{\nabla\pi(\theta, s, a)^T}{\pi(\theta, s, a)} w = 0.\end{aligned}$$

This implies that $f_w^\pi(s, a)$ really approximates an *advantage function* A^π giving the *relative* value of each action in a state, $A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$. It is thus not possible to learn $f_w^\pi(s, a)$ in a TD-like bootstrapping manner from the values of neighbouring states, as the values of states are explicitly subtracted out [22]. Sutton et al. suggested to learn $f_w^\pi(s, a)$ from roll-outs by taking the immediate reward at each time step r_t as an unbiased estimate of $Q(s, a)$, and performing least-squares minimisation between $f_w^\pi(s, a)$ and the estimated Q-values [33]. However, a general problem is that $f_w^\pi(s, a)$ is constrained to be linear in w, which means that it may not be possible to approximate the true advantage function $A^\pi(s, a)$ very accurately.

3.4.6 Natural Actor Critic

RL algorithms that maintain both value functions and a separate representation of the policy (instead of simply using a greedy policy with respect to the value function) are known as *actor-critic* algorithms [32]. The representation of the policy is the actor, responsible for choosing actions at each step, while the value function plays the role of the critic, evaluating the performance of the actor.

We now take a closer look at actor-critic frameworks that use a compatible function approximator $f_w^\pi(s, a)$ to the state-value advantage function $A^\pi(a, s)$ as outlined in the previous section (i. e., $f_w^\pi(s, a) = (\nabla \log \pi(\theta, s, a))^T w$ holds). Such frameworks can be very elegantly combined with the natural gradient described in Section 3.4.4. Peters et al. [22] show that if the simple gradient $\nabla J(\theta)$ is estimated according to Equation (3.19), then $\nabla J(\theta) \approx F(\theta)w$, where $F(\theta)$ is the policy Fisher matrix from Section 3.4.4:

$$\begin{aligned}\nabla J(\theta) &\approx \sum_s d_\gamma^\pi(s) \sum_a \nabla\pi(\theta, s, a)(\nabla \log \pi(\theta, s, a))^T w \\ &= \sum_s d_\gamma^\pi(s) \sum_a \pi(\theta, s, a) \nabla \log \pi(\theta, s, a)(\nabla \log \pi(\theta, s, a))^T w \\ &= E\left[\nabla \log \pi(\theta, s, a)(\nabla \log \pi(\theta, s, a))^T\right] w = F(\theta)w.\end{aligned}$$

Hence, when we calculate the natural policy gradient $\widetilde{\nabla} J(\theta)$, the Fisher information matrix and its inverse cancel each other out, and the result is exactly w:

$$\begin{aligned}\widetilde{\nabla} J(\theta) &= F(\theta)^{-1} \nabla J(\theta) \\ &\approx F(\theta)^{-1} F(\theta) w \\ &= w.\end{aligned}$$

This means that we can avoid estimation of $F(\theta)$ completely. As computing a good estimate of $F(\theta)$ normally requires far more data (i. e., environment interactions) than is needed for a good estimate of w [22], we get a gradient estimate at much lower cost.

Peters et al. [22] recently developed an algorithm called *Natural Actor Critic*, or NAC for short, which exploits this fact. It is based on the LSTD(λ) algorithm (Least Squares Temporal Difference) [8], which approximates state values by minimising the squared error on data gathered from Monte-Carlo roll-outs.

In the following, we give some intuition about how the NAC algorithm works. We begin with the Bellman equation (3.4) for fixed parameters θ where the value of action a in state s is $Q(s, a)$. This can also be written as the value $V(s)$ plus the advantage of action a in state s, $A(s, a)$:

$$Q(s, a) = V(s) + A(s, a) = R(s, a) + \gamma \sum_{s'} \Pr(s'|s, a) V(s'). \tag{3.20}$$

We approximate the value function V linearly using a base function ϕ and a parameter vector v: $V(s) \approx \phi(s)^T v$. The advantage function A is approximated by our compatible approximator, $A(s, a) \approx f_w^\pi(s, a) = (\nabla_\theta \log \pi(\theta, s, a))^T w$. Replacing V and A in the above equation with these approximations, we get

$$\phi(s)^T v + (\nabla \log \pi(\theta, s, a))^T w \approx R(s, a) + \gamma \sum_{s'} \Pr(s'|s, a) \phi(s')^T v.$$

Our goal is to solve for the natural policy gradient w. However, both the parameter vectors w and v are calculated by the algorithm. We reformulate the above equation as a temporal-difference estimate of $Q(s_t, a_t)$, noting in particular that the expected immediate reward from state s, $R(s, a)$, is replaced by the observed next reward r_{t+1}, and the summation expressing the expected (discounted) value of the next state is replaced by an approximation $\gamma \phi(s_{t+1})^T v_t$ of the discounted value of the observed next state. This approximation introduces an additional zero-mean error term σ:

$$\phi(s_t)^T v_t + (\nabla \log \pi(\theta_t, s_t, a_t))^T w_t \approx r_{t+1} + \gamma \phi(s_{t+1})^T v_t + \sigma(s_t, a_t, s_{t+1}).$$

Rewriting as a linear system yields

$$\begin{aligned}(\phi(s_t) - \gamma \phi(s_{t+1}))^T v_t + (\nabla \log \pi(\theta_t, s_t, a_t))^T w_t - \sigma(s_t, a_t, s_{t+1}) &\approx r_{t+1} \\ [(\nabla \log \pi(\theta_t, s_t, a_t)^T, (\phi(s_t) - \gamma \phi(s_{t+1}))^T][w_t^T, v_t^T]^T - \sigma(s_t, a_t, s_{t+1}) &\approx r_{t+1}\end{aligned}$$

Finally, we pre-multiply both sides by an eligibility trace z_{t+1} (cf. Section 3.1.2), leading to

$$z_{t+1}[(\nabla \log \pi(\theta_t, s_t, a_t)^T, (\phi(s_t) - \gamma\phi(s_{t+1}))^T][w_t^T, v_t^T]^T - z_{t+1}\sigma(s_t, a_t, s_{t+1}) \approx z_{t+1}r_{t+1}.$$

The NAC algorithm approximates w by aggregating both sides of the above equation over many time steps H,

$$D_H := \sum_{t=0}^{H-1} z_{t+1}[(\nabla \log \pi(\theta_t, s_t, a_t)^T, (\phi(s_t) - \gamma\phi(s_{t+1}))^T] \tag{3.21}$$

$$c_H := \sum_{t=0}^{H-1} z_{t+1}r_{t+1}, \tag{3.22}$$

and solving the equation $D_H[w^T, v^T]^T = c_H$ for $[w^T, v^T]^T$. Note that the noise term $z_{t+1}\sigma(s_t, a_t, s_{t+1})$ does not appear in D_H. This is because σ is zero-mean and has sufficiently small variance to be averaged out [8].

If we only calculated the state values v in the above manner (and aimed for the undiscounted average reward by setting $\gamma := 1$), we would have Boyan's LSTD(λ) algorithm [8]. LSTD(λ) has been shown to converge with probability 1 under mild assumptions [17]. The conditions are that each state has a stationary probability greater than 0, i. e., is visited infinitely often, and that the matrix containing in each row i the basis function of state i has full column rank. NAC is actually only concerned with finding accurate values for w, but adapting LSTD(λ) in this manner means that NAC inherits the convergence guarantees, while v can be seen to constrain the possible solutions for w. Pseudo code for NAC is given in Algorithm 2.

The NAC algorithm thus has several properties which make it a promising candidate for solving hard RL problems. It is the first algorithm to use the natural gradient in an actor-critic framework, trying to get "the best of both worlds": the natural gradient for faster convergence as opposed to the "vanilla" gradient, and a value function that incorporates the Bellman equation into the policy. This is why NAC is our algorithm of choice for the traffic control problem.

3.4.7 Online Natural Actor Critic

We adapt NAC in two ways to tailor it to our problem. First of all, we need to deal with partial observability, as we can only obtain crude information about world states through the loop detectors. We solve this by assuming that our observations are a *deterministic* function of the state (i. e., assuming noiseless sensors), and by using the observations as state features, i. e., as the basis functions $\phi(s)$ in the NAC algorithm. Hence we assume that our observations can be used to approximate the value of a state, in other words, that in order to act well our observations are sufficiently accurate indicators of the underlying state.

Algorithm 2 Natural Actor-Critic with LSTD-Q(λ)

$t = 0,\ D_0 = [[0]],\ \theta_0 = [0],\ z_0 = [0]$
$\epsilon =$ step size, $\gamma =$ Critic discount, $\lambda =$Actor discount
observe o_0 (generated according to $O(s_0)$)
while not converged **do**
  while w not converged **do**
    generate action a_t according to $\pi(\theta_t, s_t)$
    $z_{t+1} = \lambda z_t + [\nabla \log \pi(\theta_t, s_t, a_t)^T, \psi_t^T]^T$
    $D_{t+1} = D_t + z_{t+1}[\nabla \log \pi(\theta_t, s_t, a_t)^T, \phi_t - \gamma\phi_{t+1}]$
    receive r_{t+1} (generated according to $R(s_t, a_t)$)
    $c_{t+1} = c_t + z_{t+1} r_{t+1}$
    $[w_{t+1}^T, v_{t+1}^T]^T = D_{t+1}^{-1} c_{t+1}$
    $\theta_{t+1} = \theta_t$
    $t = t + 1$
  $\theta_t = \theta_{t-1} + \epsilon w_t$
return θ_t

Secondly, we turn NAC into an online algorithm, updating the parameters at each time step instead of waiting until the gradient estimate has converged. This is motivated by the following observation: when doing *stochastic* gradient ascent in large state spaces, i. e., when gradient estimates are only based on a small subset of the possible examples, they are typically noisy. The computational effort of calculating a very accurate estimate with regard to such a noisy example subset may then not pay off, and computationally cheap online algorithms can often be shown to converge faster, because the policy can improve at every step [7]. Furthermore, to get the most out of a more accurate gradient estimate, a line search or quasi-newton method should be used, which estimates the optimal step size into the direction of the gradient estimate. In our application a line search is not feasible (at least not in a real-world deployment), since during a line search performance of the system can become very poor, e. g. causing traffic jams. Quasi-newton methods, on the other hand, need exact gradients to perform well.

The original formulation of NAC requires an inversion of the matrix A for every parameter update, which naively incurs a computational cost of $O(n^3)$, where $n = |\theta| + |\phi|$. For our online algorithm, which performs parameter updates at every time step, it is therefore crucial to reduce this cost. We use the Sherman-Morrison update of a matrix inverse (as suggested by Nedić and Bertsekas for LSTD(λ) [17]):

$$(D + zy^T)^{-1} = D^{-1} - \frac{D^{-1} z y^T D^{-1}}{1 + y^T D^{-1} z}.$$

In other words, we always work in the inverse space. The update now is $O(n^2)$, which is still expensive compared to "vanilla" PG approaches. However, as we will show in Chapter 5, NAC can make up for expensive computations by requiring orders of magnitude fewer steps to converge to a good policy. Pseudo code for our

online version of NAC is given in Algorithm 3.

Algorithm 3 Online Natural Actor-Critic

1: $t = 0,\ D_0^{-1} = I,\ \theta_0 = [0],\ z_0 = [0]$
2: $\epsilon =$ step size, $\gamma =$ Critic discount, $\lambda =$Actor discount
3: observe o_0 (generated according to $O(s_0)$)
4: **while** not converged **do**
5: generate action a_t according to $\pi(\theta_t, o_t)$
6: $z_{t+1} = \lambda z_t + [\nabla \log \pi(\theta_t, o_t, a_t)^T, o_t^T]^T$
7: receive r_{t+1} (generated according to $R(s_t, a_t)$)
8: observe o_{t+1} (generated according to $O(s_{t+1})$ with next state s_{t+1} generated according to $P(\theta_t)_{s_t,s_{t+1}}$))
9: $y = [\nabla \log \pi(\theta_t, o_t, a_t)^T, o_t^T]^T - \gamma[0^T, o_{t+1}^T]^T$
10: $\alpha_t = 1 - \frac{1}{t}$
11: $u = (1 - \alpha_t) D_{t-1}^{-1} z_t$
12: $q^T = y_t^T D_{t-1}^{-1}$
13: $D_{t+1}^{-1} = \frac{1}{\alpha_t} D_t^{-1} - \frac{uq^T}{1+q^T z_t}$
14: $g = r_t z_{t+1}$
15: $[w_{t+1}^T, v_{t+1}^T]^T = D_{t+1}^{-1} g$
16: $\theta_{t+1} = \theta_t + \epsilon w_t$
17: $t = t + 1$
18: **return** θ_t

We retain the aggregation of D_H from Equation (3.21), using a rolling average implemented by the α weighting (line 10). This means that D_t, the value of D_H at time step t, is an average of the Fisher matrices for many parameter values. Actually, we expected a discounted average to work better, since it should yield a value for D_t that better represents θ_t (placing more emphasis on the Fisher matrix estimations for recent values of θ). However, this performed poorly, perhaps because decaying α mitigates ill-conditioning in the Fisher matrix as parameter values grow [14]. We only use instantaneous gradient estimates $g = R(s_{t+1})z_{t+1}$ instead of the accumulated gradients c_H from Equation (3.22), to avoid multiple parameter updates based on the same rewards.

The OLPOMDP algorithm of Baxter and Bartlett [5] that we introduced in Section 3.4.3, produces per step gradient estimates from the discounted sum of the past likelihood ratio terms $\nabla \log \pi(\theta_t, o_t, a_t)$ for all t, multiplied by the instant reward r_{t+1}. This is exactly w_{t+1} if we set $\beta := \lambda$ and $D_t := I,$ for all t. Other PG approaches [14, 33] are also specialisations of NAC [22]. As the simplest and fastest infinite-horizon algorithm we use OLPOMDP for comparisons.

3.4.8 Factored Learning With Policy-Gradients

The policy-gradient algorithms we described, online NAC and OLPOMDP, both learn a stochastic memoryless policy $\pi(\theta)$ mapping observations o to actions a.

In order to learn traffic control strategies, we need in fact a policy that defines the behaviour of all intersection controllers jointly. Hence an action a is a vector $a = [a_1, \ldots, a_N]$, specifying actions for all controllers in the network, where N is the number of controllers. Similarly, let $o = [o_1, \ldots, o_N]$ be a concatenation of observation vectors and $\theta = [\theta_1, \ldots, \theta_N]$ be a concatenation of the parameter vectors for all intersection controllers. We define that the action a_i for controller i only depends on o_i and θ_i:

$$\begin{aligned}\pi(\theta, o, a) &= Pr(a_1, a_2, \ldots, a_N | \theta_1, \ldots, \theta_N, o_1, \ldots, o_N) \\ &= \prod_i Pr(a_i | \theta_i, o_i).\end{aligned}$$

Then the "log gradient" term used in the updates of our algorithms is

$$\begin{aligned}\nabla \log \pi(\theta, o, a) &= \nabla \log \prod_i Pr(a_i | \theta_i, o_i) \\ &= \frac{\nabla \prod_i Pr(a_i | \theta_i, o_i)}{\prod_i Pr(a_i | \theta_i, o_i)}.\end{aligned}$$

Let $\nabla_{\theta_j}(\theta, o, a)$ be the "log gradient" of π with respect to θ_j, $\nabla \log \pi(\theta, o, a) := [\nabla_{\theta_1}(\theta, o, a), \ldots, \nabla_{\theta_N}(\theta, o, a)]$. Then

$$\begin{aligned}\nabla_{\theta_j}(\theta, o, a) &= \frac{\nabla_{\theta_j} \prod_i Pr(a_i | \theta_i, o_i)}{\prod_i Pr(a_i | \theta_i, o_i)} \\ &= \frac{\nabla_{\theta_j} Pr(a_j | \theta_j, o_j) \prod_{i \neq j} Pr(a_i | \theta_i, o_i)}{\prod_i Pr(a_i | \theta_i, o_i)} \\ &= \frac{\nabla_{\theta_j} Pr(a_j | \theta_j, o_j)}{Pr(a_j | \theta_j, o_j)} \\ &= \nabla_{\theta_j} \log Pr(a_j | \theta_j, o_j).\end{aligned}$$

Hence, the global "log gradient" can be split into "log gradients" for each controller, using only the parameters, observations and actions of that controller. Similarly splitting the other variables used in NAC and OLPOMDP lets us learn the control policy of each controller *separately*, i. e., with an independent instance of NAC or OLPOMDP. This tremendously reduces the complexity of learning: in the case of NAC, for example, the complexity is reduced to O(Nn^2) instead of O(N^2n^2), where $n = \frac{1}{N}|\theta| + |o|$ is the size of the parameters and observations for one controller.

This approach is only correct as long as all controllers receive the same, global reward at each time step. However, learning can be sped up significantly by using *local* rewards, which reward each controller separately for its respective policy [3]. Although this usually implies losing the guarantee of finding a local optimum (across the entire system), local rewards are often preferable in large systems, at least if maximising the local rewards separately can be shown to lead to a good

global performance of the system. As Bagnell and Ng show [3], using local rewards becomes necessary as the number of agents grows: they demonstrate that with global rewards, the number of training iterations needed to achieve near-optimal results for MDPs is linear in the number of agents, while with local rewards a logarithmic number of iterations is sufficient.

3.5 Reinforcement Learning for Traffic Control

To our knowledge, policy-gradient algorithms have so far not been applied to learning control policies for traffic lights. Previous work suggesting the use of reinforcement learning includes that by Wiering [35], who uses Q-learning (see Section 3.1.2), and extensions of this approach by Bakker et al. [4]. In the approaches of Wiering and Bakker, like in our approach, each intersection has a controller acting as a separate agent. The actions of each agent are to switch certain traffic lights green or red during the next time step. Wiering and Bakker assume fixed routes for the vehicles in their system, i. e. drivers do not adapt to traffic conditions. The state representation they use is then *car-based*: every car has a separate state space encoding its current location and final destination. Every car also learns a value function that relates the actions of intersection controllers (whether to turn the light this car waits for green or red in the next time step) to the expected remaining travel time for that vehicle. If the intersection controller decides to turn the light red, the expected travel time of a vehicle waiting for this light will be higher than if the intersection controller turns the light green. The intersections controllers then choose their actions to maximise the summed value functions of all cars currently waiting at that intersection. Hence they learn to minimise the average travel time of the waiting vehicles.

Wiering compares his approach against a number of baseline strategies, two of which achieve good performance. The first is a throughput based strategy, which sets the lights at any time step in such a way that a maximum number of cars can pass through the intersection. The second strategy always gives the right of way to the longest queue at the intersection. In under-saturated conditions, the performance of the RL approach is similar to that of the two baselines. As the traffic volume grows, the estimated travel times of cars increasingly differ, depending on congested roads at later stages of their trip. The RL approach profits from learning these estimates and beats the baselines notably under conditions of heavy traffic. Bakker enhances the method of Wiering by adding communication between intersections, and improves the performance of the RL method further.

However, Wiering and Bakker train their algorithms on a very simple traffic simulator, and furthermore derive state information from the simulator that would not be accessible in the real-world. As their approach is model-based, they assume knowledge about all the cars waiting at an intersection and about the destinations of those cars. In a real-world deployment, estimation of these values would require the use of a highly sophisticated traffic prediction model. As mentioned before, this

may incur loss of performance through imperfection of the model. Furthermore, Wiering and Bakker ignore partial observability by assuming, for example, that a few values like the current location and destination of a car are sufficient indicators of that car's remaining travel time. If this assumption is not fulfilled in a real traffic system, a value-based method like Q-learning might do arbitrarily badly (see Section 3.3).

Another RL approach to traffic light control is pursued by Thorpe and Anderson [34], who employ the SARSA algorithm (see Section 3.1.2). The controllers learn to minimise the maximum travel time of any vehicle in the system. One controller is trained on a single intersection, and the state of that controller is replicated to all intersections of a network after training is completed. This means that every controller optimises its intersection only locally, and furthermore implies the assumption that every intersection controller should pursue the same policy. The method of Thorpe and Anderson beats the performance of a fixed duration strategy, where all phases are assigned equal, fixed length. A second baseline they compare against is a strategy of "greatest volume", where the most-used stream always gets the right of way. This is a simple version of a saturation-balancing algorithm (see Section 2.1.1). Assuming knowledge about the exact locations of all cars in the network, the RL method beat the greatest volume strategy. With a simpler traffic representation assuming only the number of cars on any road to be known, RL did not perform as good as the greatest volume strategy.

Chapter 4

Policy-Gradient for Traffic Control

In Chapter 2, we introduced the traffic control problem, presented some current control systems and discussed their limitations. Subsequently, we introduced policy gradient methods as a class of reinforcement learning techniques, and stated their general properties (see Section 3.4). In this chapter, we do the connecting step and show why policy-gradient methods are appropriate for learning traffic control strategies, and how learning can be achieved.

4.1 Expected Strengths of Policy Gradient Methods

Policy-gradient (PG) techniques are able do deal with large state spaces, stochasticity and partial observability. Apart from these general properties, there are a number of possible further advantages that PG methods can demonstrate specifically in the traffic control area, and which give our approach a strong position against other current controllers (see Section 2.1.6):

- **PG does not need a traffic model**, hence it is not dependent on the accuracy and reliability of complicated traffic predictions. PG does not need an explicit model because the parameters of the control policies implicitly model only the information relevant to acting well, rather than modelling details that have little impact on the policy.

- **PG can quickly react to changing traffic conditions** by recognising patterns. Its policy can be as rich as pre-specified plans, but is learned automatically.

- **PG can scale up to large networks** by learning each intersection's control policy separately, while making cooperation between neighbouring intersections possible through common observations.

- **PG can learn to optimise the overall network**, as e. g., the use of a common, global reward achieves automatic cooperation of all controllers.

In order to demonstrate that policy-gradient methods indeed exhibit all of these features, we conducted a set of experiments requiring the above-mentioned capabilities. The following section describes the traffic simulation system we used for training the controllers and the architecture of our learning system.

4.2 A Simple Traffic Simulator

In order to learn traffic control via reinforcement learning, we need an environment in the form of a simulated traffic system. As described in Chapter 2, finding accurate models for traffic flow has been the subject of extensive research for more than half a century. Various (mostly commercial) programs exist that simulate traffic based on such advanced models and that can be used to test the effectiveness of traffic control techniques. The input to these programs typically consists of a road network, the control policy for the signalled intersections at each time step, and a specification of the traffic volume and direction (e. g., origin and destination nodes for each car, or turn ratios at intersections). From the simulated traffic flow, performance criteria such as queue lengths at intersections or average travel times can be calculated, which can be used to evaluate the quality of a controlling policy.

First trials with such systems unfortunately were not successful. The simulation software *Artemis* by Peter Hidas, for example, is designed to be operated via a graphical user interface, and provides little support for automation. At our request, the author extended the program to allow running it from the command line. However, after investing several weeks of time, the software was deemed too slow and unstable for our purposes. As the process of tuning parameters by Monte-Carlo experimentation demanded that the simulation process be as fast as possible, speed was a major requirement for our simulator. *Paramics*, a popular test bed used by the Road Travel Authority, Sydney, was also ruled out for this very reason. It was only accessible to us through a client-server connection at the time, which would have slowed down experimentation beyond tolerable limits. However, after establishing appropriate algorithms and control models with this work, the next step will be to re-investigate these simulation systems.

The advantage of a policy gradient method (or any direct policy search method) is that it learns a policy without constructing an explicit internal model of the environment. Hence we do not need a very realistic traffic simulation to demonstrate the general capability of our technique. As long as the simulator displays the essential concepts of a traffic system, we can be optimistic that good results for this system mean that we would also achieve positive results in a more realistic setting. (This is essentially an assumption made by all traffic control systems to varying degrees, since no model or simulator can approach the complexity of a real road network.) We therefore decided to implement our own traffic simulation system, aiming at fast simulation speed rather than at an accurate model of traffic flow. Of

course one could argue that learning in a simple system is easier than in a realistic traffic system. The main difference is that a more realistic traffic model would include many more variables influencing the behaviour of the individual vehicles. To a PG learning agent these variables would not be represented explicitly, hence the system would seem noisier, complicating learning. However, since conditions in a traffic system change relatively smoothly with the flow of traffic, we assume for now that our PG controller would be able to cope with the additional noise, and restrict our experiments to the mentioned simple simulator.

We represent the central components of a traffic system: roads with limited capacities, intersections with controllers reacting to current conditions observed through inductive loop detectors, and vehicles travelling from specified origins to specified destinations on the map. We take particular care to create a system that is appropriate for being controlled by a SCATS controller, since it is our aim to be comparable to SCATS: the loop detectors are located at the stop lines of intersections, and the signal lights are controlled by SCATS-compatible controller commands.

To arrive at a basic representation of a traffic system, we made, amongst others, the following simplifying assumptions:

- All vehicles move at uniform speed.
- Road lengths are multiples of a space unit, where one unit is exactly the distance a car travels within one time step.
- We do not care about the relative positions of cars within one road segment, or about interactions between cars. Hence the number of cars in a road segment has no influence on the travel speed of the cars.

Apart from these, there are many more simplifications in our system, as described in the following.

In our simulator, roads are placed in a grid, and signalled intersections may be defined at grid nodes. Sparse grids are possible, i. e., not every grid line or node needs to be occupied by a road or intersection, respectively. This allows quite general maps to be specified, but limits roads to be horizontal or vertical, and a maximum of four roads can be connected by an intersection. The run time of our algorithms does not depend on the size of the map, but of the number of controlled intersections in it. At every intersection, there are two separate queues for the cars from each incoming direction: one queue for cars aiming to turn right, and one for cars aiming to go straight or turn left (stemming from the fact that we are modelling Australian traffic, which is left-hand sided). In the following we will use the term *turn* for any of the possible movements at an intersection, i. e., for turning left, right, or going straight.

Cars always enter the system at intersection nodes rather than at arbitrary locations along the roads. Such new cars exit their source intersection independently of the current phase. Every vehicle has a fixed destination, which is also an intersection. In order to get there, drivers choose a shortest path and also try to minimise

waiting times at intersections. If at an intersection two different turns would lead to equally short routes, drivers will prefer to use one turn over the other if it is green by the time they arrive, and if the queue for that turn is not too long (no more than half of the maximum possible length as given by the road capacity). If none of the turns is preferable, drivers decide randomly for one of them. This *greediness* of drivers, when choosing turns, is one step towards modelling the adaption of drivers to control policies (see the description of the traffic assignment problem in Section 2.0.2). To our knowledge, other simulators usually fail to model such effects. Considering driver adaption is however very important when we want to show the effect that different control policies may have, as we will do in one of our experiments.

Each time step in our simulator corresponds to roughly 5 – 10 seconds in the real world. Using a more finely grained time discretisation would have led to greater accuracy of the model, but also to slower simulation and longer learning This is because more time units would lie between an action and rewards corresponding to it, increasing the temporal credit assignment problem (see Section 3.2). Hence, choosing the length of the time unit was a trade-off between speed and accuracy of the simulation.

At each such time step – or, more precisely, at the beginning of each time unit – the intersection controllers make the decision about which signal phase to turn on during the next time step. They can choose between 4 different phases: north-south-straight, north-south-right, east-west-straight, and east-west-right. The phases for going straight always include the left turn as well (cf. the explanation of the diamond overlap phase scheme in Section 2.1.1). Controllers can stay in their current phase or switch to any other phase – we do not restrict the order of phases. To ensure a reasonable policy, however, the simulator enforces the constraint that within one cycle each phase has to come on at least once. The cycle length here is an arbitrary fixed number of time steps, which we usually chose to be 16. Also, we set an upper limit on the number of time steps a phase may be on during a cycle without interruption (usually 13). If the action a controller decides to take, i. e., the phase it wants to switch on, violates these restrictions, the simulator ignores the controller and switches on a phase that satisfies the constraint. Imagine for example that phase p has not been on during a cycle, and in the last step of the cycle the controller wants to turn on phase q. Then the simulator turns on phase p instead. If the simulator has the choice between various phases that have not been on during the last two or three steps of the cycle, it chooses a phase according to a fixed order of phases.

After the signal lights have been set, the locations of the cars are updated. The order in which this happens is fixed, depending on the time of creation of the cars, thus it does not matter where the cars currently are in the network. All cars travelling along roads move forward one space unit into the next road segment. When arriving at an intersection, a car passes through immediately if the light corresponding to its desired turn is green, and if there is no queue for that turn. Otherwise, it lines up in the queue. To account for the fact that queued cars require time for

acceleration, we limit the number of cars that can pass through an intersection in one time step, if there is a queue. In the first time step of a phase this limit is 2, while in subsequent time steps it is 5. This has the effect of additionally modelling inter-green times, i.e., we account for the time lost by switching phases. A car queued up in front of an intersection can pass through, during a time step, if the throughput limit has not been reached and if the queue is not *blocked*: as a rough model of limited road capacity, we only allow cars to enter the next road segment if the number of cars in that segment does not exceed a certain road capacity parameter (20 in our experiments). This allows us to represent saturated traffic conditions. Cars that cannot move forward due to a blocked segment stay at their current positions. A queue in front of an intersection gets blocked as soon as any car in it cannot pass through due to a blocked road. If a new car is scheduled to enter the network, but the corresponding road where it would be entered is blocked, that car does not enter the system but is ignored.

From the point of complexity, our simulation model is comparable to that used in other studies that applied reinforcement learning to traffic [4, 35]. Implementing our own system, however, allowed us to build in the simulation of the SCATS-like loop detectors and to implement the greediness of drivers mentioned above, as well as producing a very fast simulator. It is capable of simulating 3,800 time steps per second with 100 intersections and an average of 930 cars, on a 3.2 GHz AMD Athlon 64 processor.

4.2.1 The Graphical User Interface

We also implemented a Graphical User Interface (GUI) for the simulator. It allows examination of the network topology, traffic volume and policies pursued by the controllers graphically, and thus supported the development of the system substantially. Figure 4.1 shows a snapshot of our simulator as displayed by the GUI. As mentioned before, each intersection has eight queue lines with loop detectors, where the latter are displayed as thin blue rectangles. The current settings of the lights are displayed by the green and red rectangles in front of each queue, corresponding to green and red lights, respectively.

The four small lines in the middle of intersections reflect the policy pursued by the intersection controller over the last few cycles, showing how much time (relatively) the controller has given to each of the four phases. The longer a line, the more time the controller has devoted to the corresponding phase in the past. The snapshot displayed in Figure 4.1 is taken at the beginning of learning. The controllers on the horizontal road have just begun to learn giving most time to the east-west-straight phase (the third phase), while the controllers on the vertical road are learning to devote most time to the north-south-straight phase (the first phase). The controller of the central intersection has experienced most traffic on the vertical road so far, which is why it has devoted more time to the third phase than to the first phase.

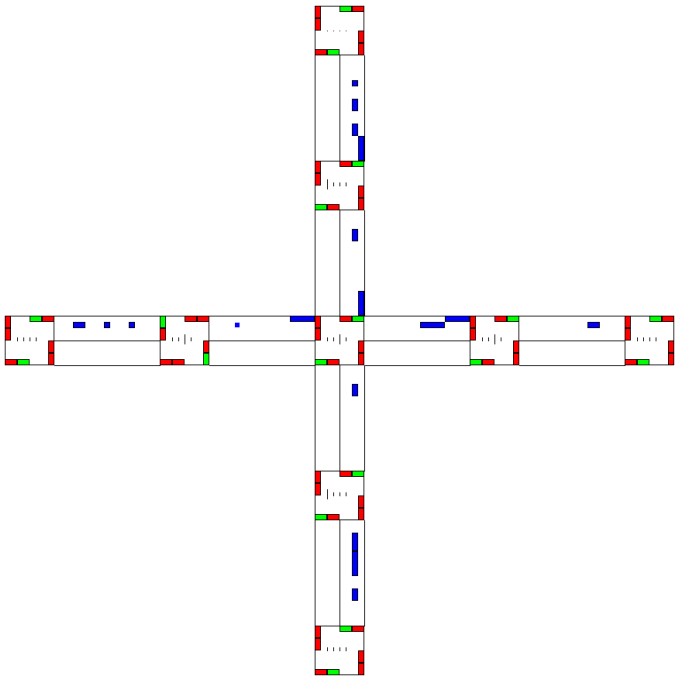


Figure 4.1: A snapshot of our simulator as displayed by the GUI. Note that traffic is left-hand sided.

4.3 Architecture of the Learning System

In our setup, each intersection is controlled by a separate controller using one of the policy-gradient algorithms introduced in Section 3.4, NAC or OLPOMDP, to learn a policy. Cooperation between the controllers is achieved through common observations and, in some scenarios, common rewards. Each controller represents its policy through a *linear function approximator*, which takes as input the observations corresponding to a state, and has four outputs corresponding to the four possible phases. The value it assigns to each output corresponds to the likelihood with which it wants to turn that phase on during the next time step. We turn these likelihoods into well-formed probability distributions by using the softmax function described in Section 3.2 on the network's outputs to decide the next phase. The policy parameters θ form the weights of the linear approximator. At each time step, they are updated by back-propagation into the direction of the reward gradient, as calculated by the PG algorithm. We have also experimented with non-linear approximators, i. e., with multi-layer neural networks, but we could not observe a benefit from that.

In detail, the controller of intersection i maps the observations $o_{t,i}$ at time step t to a probability distribution $\pi_{t,i}$ over P possible phases as follows (in our case, $P = 4$). Let $x_{t,i}$ be the output vector of the linear approximator, where the pth

entry is denoted as $x_{t,i}[p]$, and let θ_i be the $P \times |o_{t,i}|$ matrix of parameters for intersection i. Let u_p be the unit vector with a 1 in row p, and $a_{t,i}$ the action for controller i resulting from the calculation. Then

$$x_{t,i} = \theta_i o_{t,i}\,, \qquad \Pr(a_{t,i} = p|\theta_{t,i}, o_{t,i}) =: \pi_{t,i}(\theta_{t,i}, o_{t,i}, p) = \frac{e^{x_{t,i}[p]}}{\sum_{p'=1}^{P} e^{x_{t,i}[p']}},$$
$$\pi_{t,i}(\theta_{t,i}, o_{t,i}) := [\pi_{t,i}(\theta_{t,i}, o_{t,i}, 1) \ldots \pi_{t,i}(\theta_{t,i}, o_{t,i}, P)].$$

The "log gradients" $\nabla_{\theta_{t,i}} \log \pi_{t,i}(\theta_{t,i}, o_{t,i}, p)$, needed by our algorithms for updating the parameters of each controller (see Algorithm 1 in Section 3.4.3, and Algorithm 3 in Section 3.4.7), are then as follows:

$$\nabla_{\theta_{t,i}} \log \pi_{t,i}(\theta_{t,i}, o_{t,i}, p) = (u_p - \pi_{t,i}(\theta_{t,i}, o_{t,i}))o_{t,i}^T.$$

The intersection controllers interact with the simulator as follows: at the beginning of a time step, all controllers receive observations corresponding to the current state of the traffic simulator (these are different for each controller). Subsequently, the controllers calculate the probabilities for the different signal phases, according to their policy parameters and the softmax function, as described above. The next phase for each controller is decided by taking a random decision according to the calculated phase probabilities. Over time, the policies generally become more deterministic (although this will not happen if a stochastic policy is really optimal). As mentioned in the last section, the simulator may however turn on a different phase, if the phase calculated by the controller violates the restrictions we placed on policies. This over-ruling of policies is simply part of the environment and does not violate the assumptions needed for PG methods to be valid.

Given all actions of the controllers, the simulator simulates one time step by setting the signal lights accordingly and moving the cars, as well as entering new cars and taking out cars that have arrived at their destination. Then the controllers receive the rewards corresponding to the new state of the simulator, and update their policies according to NAC or OLPOMDP. The decision of the controllers at each time step is based on a subset of the following possible observation features:

- **Cycle duration**: specifies how many time steps the current cycle has lasted already. This is important to support time based decisions like offsets. This information, as well as all other features, are presented in a binary format to facilitate learning of the function approximators. It consists of 16 bits, where the nth bit is on exactly in the nth step of the cycle.

- **Current phase**: specifies what phase we were in during the last time step. It consists of 4 bits, where each bit corresponds to a phase.

- **Current phase duration**: indicates how many time steps the current phase has *continously* been on up to now. It is made up of 5 bits, indicating that we have spent more or equal to 1, 2, 4, 8 or 13 continuous time steps in the current phase (hence several bits may be on).

- **Phase durations**: specifies for every phase how many time steps it has been on *in total* during the current cycle. It consists of 5 bits per phase, in the same format as the current phase duration.

- **Detector active**: indicates, for every one of the eight loop detectors at the intersection, whether it is active at the moment (i. e., whether there is at least one car waiting). This information is made up of 8 bits, one for each detector, where the bit is one if the detector is active.

- **Detector history**: specifies for every detector how busy traffic has been during the current cycle, i. e., how saturated the corresponding stream of traffic has been. It consists of 3 bits per detector, indicating a saturation level of more than 0, more than half capacity, or capacity.

- **Neighbour information**: gives a comparison of the detector counts of neighbouring intersections, indicating where traffic is expected from. In our phase scheme, traffic coming in from the north and south always has the right of way together, and so has traffic from the east and west. Hence we specify whether the *sum* of expected traffic from the north and south is greater than the sum of expected traffic from the east and west. For all neighbouring intersections, the detector counts of those lanes are summed that could potentially send cars towards this intersection. The two numbers that result from summing all traffic from the north and south, and summing all traffic from the east and west, are compared for several time steps in the past. By looking back in time as many time steps as correspond to the road length, a controller gets the information about incoming cars just before they arrive. In the experiments, the controllers always got information about a window from 3 to 5 time steps in the past, covering all different road lengths used. This information is then encoded in 2 bits for each of the past time steps, where the first bit is on if more traffic is expected to come towards this intersection from the east/west, and the second bit is on if more traffic is expected from the north/south. By looking back into the past as many time units as the longest road has space units, the controllers can learn to know exactly at what time step traffic will be arriving at their intersection.

4.4 Real-World Deployment of our System

One of the goals of this work was to develop a method that could theoretically be employed in the real world. Hence we only used information for the observation features (see last section) that would be available in the real world, such as detector counts and information about the past actions of the controllers. As will be explained in the following section, the additional use of (simple) traffic models could however improve performance, as that allows for more specific *rewards* which support learning.

Note that, if our methods were to be employed in practice, actions have to be taken which avoid using a very bad policy in the beginning of learning (causing traffic jams on the streets). One way to solve this could be to simulate the road network and expected traffic beforehand, and pre-train our controllers on that simulation. Our controllers would continue learning in the subsequent deployment, in order to make up for the imperfection of the simulation and to adapt to changing traffic demands. Alternatively, our algorithms could learn an initial policy by observing SCATS before they start executing their policies, e. g., by back-propagation.

4.5 Performance Criteria

One of the most important choices when setting up a learning system is which performance measure to use, i. e., how to reward the learners. As described in the previous chapter, frequently used optimisation criteria for traffic systems include the average or total travel time, waiting times, number of vehicle stops and queue lengths.

The *travel time* criterion is especially appealing since it is a very general measure, encompassing both waiting times and stops. There are, however, two problems with using the (negated) average travel time as reward for our learning system. First of all, calculating this value requires knowledge that we would not have in the real world. Our simulator is able to deliver this value, but in a real-world deployment it would only be accessible through (imperfect) traffic modelling. Even if we did decide to employ a traffic model in practice for estimation of this value, the travel time criterion still poses a problem in our test scenarios due to their small size. The problem is that travel times can only be calculated for vehicles once they have arrived at their destination. We would, however, like to feed reliable rewards to our learners as soon as possible after relevant actions, to ease the temporal credit assignment problem (see Section 3.2) – preferably at every time step. In our smaller test scenarios we often deal with only a few cars in the system at any point in time, which means that there are time steps where no car arrives. If we fed a reward of zero to the learner at those time steps, that would be the best negative travel time possible, thus we would falsely reward our learner for the fact that no car arrived. Furthermore, even in those time steps where cars do arrive, the average travel time will only be calculated from a small number of vehicles, leading to high variance in the rewards.

In order to solve the averaging problem in our small test scenarios, we decided to use the *number of cars* in the system at any time step as a criterion. This is equivalent to the total travel time criterion if we assume that the number of cars entering the system is independent from our control policy. It can be seen as follows (using the notation of Papageorgiou [20]).

Let $k = 0, 1, ...$ be a discrete time index, $N(k)$ the number of cars in the system during time interval k, $d(k)$ the demand (i. e., the number of cars entering the

system) and $e(k)$ the exits, i. e., the number of cars leaving the system or arriving at their destination, during time step k. We count cars towards $N(k)$ if they are produced before or in time step k and leave the system after time step k, i. e., cars that arrive at their destination in time step k are not counted for that time step anymore. We aim to minimise the total travel time T_K, i. e., the total number of time steps vehicles spend in the system, which is equivalent to minimising the average travel time if measured over an infinite horizon. Over a limited time horizon of length K, the total travel time is simply the sum of cars at each time step:

$$T_K = \sum_{k=0}^{K} N(k) \tag{4.1}$$

Hence, using the number of cars as the immediate reward at each time step has exactly the effect we want, namely to reward the learner for minimisation of the total travel time.

The problem remains, however, that in practice we would not know the exact number of cars in the system at any point in time. In the real world, we would need to maintain at least a simple model of the traffic flow in our system, and estimate the number of cars from the total detector counts in the system.

We decided therefore to use a second, *local* performance criterion in some of our experiments. In this setting, each intersection controller is optimised independently, using rewards calculated from the local detector counts. In addition to the advantage of using a completely model-free performance criterion, this factorised learning leads to a tremendous speed-up in terms of computation time per iteration (see Section 3.4.8). If our system were to be employed in practice, local rewards would also avoid the need of communicating rewards to the controllers from a central server. Apart from saving communication overhead, this decentralisation leads to a more robust system. However, with the local rewards we cannot guarantee global optimisation of the system anymore (see Section 3.4.8). An obvious way of trying to amend this problem would be to combine local and global rewards. However, we have not pursued this approach in our experiments.

The local reward we use is the *throughput* of an intersection, i. e., the number of cars passing through it during one time step. This criterion can be related to the travel time by making several strong assumptions. These do not hold in reality, but demonstrate that the use of this criterion should generally improve the average travel time. The travel time of a car is comprised of two components: the time it spends travelling along roads (in the following called the road travel time) and the time it spends waiting at intersections. If we assume *fixed routes* for the vehicles (i. e., we assume that the routes drivers choose to get to their destinations are independent of our control policy), and if we assume *constant road travel times*, i. e., we assume that the time it takes to travel along the links in the system is independent of our control policy, then the total road travel time over all cars is constant. Thus minimising the total waiting time at intersections is equivalent to minimising the total travel time.

Looking at each intersection separately, the total *local* waiting time over all cars during one time step is exactly the number of cars waiting at that intersection, hence this measure equals the summed *queue lengths* at an intersection. Local minimisation of the queue lengths at intersections would be an appealing optimisation criterion, since it is still closely related to the travel time, but allows for the speed-up in learning that local rewards offer. However, knowing this value in practice would still imply the usage of traffic prediction models, as otherwise we would not know how many cars are queued up at the intersection.

To arrive at our completely model-free criterion, we therefore make a further simplifying assumption: we assume that the arrival process of cars at an intersection is independent of its control policy. In practice, the control policies of neighbouring intersections will usually influence each other, such that the assumption does not hold. Given the assumption, however, locally maximising the throughput of an intersection at any time step is exactly the same as locally minimising queue lengths (in our framework). Here, a special property of our algorithms comes into play: the discount factor used when attributing rewards to former actions. It is not sufficient to maximise the throughput *on average* in order to optimise travel time (under the mentioned assumptions) – it is necessary to achieve throughput *as early as possible*. Only if cars pass through an intersection as early as possible will they arrive as early as possible, optimising their travel time.

This requirement is fulfilled by our learning algorithms. Because they discount rewards, it is more desirable for the intersection controller to get, say, three cars through in time step k, than to get two cars through in time step k and one car in time step $k + 1$. That way we usually achieve the desired effect of passing cars through as early as possible. Depending on the discount factor, however, a policy that leads to a great reward at a later time step but sacrifices a small reward in an earlier time step may be overall preferable to the learner.

Hence, our local rewards are an approximation to the travel time criterion. But even though we cannot guarantee global optimisation of the travel time when using the throughput rewards, they have led to very good results in our experiments.

Note, however, that our above arguments only hold true if the time window we optimise the throughput over (cf. Equation (4.1)) starts with an empty system. To see this, consider a pathological example. Imagine a system consisting of only one intersection and imagine that at every time step one car arrives, from alternating directions. In the optimal case, the controller switches to the corresponding phase just for the time step when a car arrives, thus no car ever needs to stop. If the controller is not synchronised with the arriving cars, but one step behind, for example, it can obtain the optimal throughput of one car per time step – over a certain time window – even though cars have to wait. Hence, in this case maximising the throughput over a time window does not lead to minimisation of the total travel time during that time window, independently of the discount factor. Assuming optimal behaviour of the controller, this situation can only occur if the time window starts when one car is already waiting at the intersection. Otherwise the controller would have had to pass that car through as soon as it arrived. If the policy of

the controller is not optimal, such situations may also occur during the optimisation period, leading to ongoing suboptimal behaviour of the controller. However, SCATS does not consider policies at such a fine grain level and would not do better in similar scenarios.

Both our global and our local reward measures do not guarantee *fair* treatment of the vehicles, i. e., they do not balance travel times, or waiting times respectively. As an example, imagine a system consisting of one intersection with a maximum possible throughput of m cars per time step. Imagine that in time step zero, m cars arrive from the north and m cars arrive from the west. In all subsequent time steps, further m cars arrive from west. Whether the controller lets the m cars from the north through early, or whether they wait for a long time while cars from the west pass through–the number of cars in the system will be the same, which means that none of the policies is preferable from the point of view of the intersection controller. Although this is sensible if we only care about minimising the total travel time, in the real world this effect would not be desirable. Criteria like the mean squared travel time could be used to tackle the problem. In our toy examples, however, travel time criteria do not work well for the reasons explained above, which is why we made the conscious decision not to care about unbalancedness of waiting times in our experiments. The policy restriction of visiting all phases once per cycle ensures that no car waits forever.

Chapter 5

Experiments

Our goals in this work are threefold. Firstly, we want to demonstrate the general applicability of policy-gradient (PG) methods to traffic control, and demonstrate that we can achieve good performance. Secondly, we want to target known weaknesses of the system that motivated this research, the Sydney Coordinated Adaptive Traffic System (SCATS), as described in Section 2.1.5, and show that we can outperform a SCATS-like algorithm (within the limitations of our simulator). Thirdly, we aim to compare the performance of the recently proposed Natural Actor Critic algorithm NAC (see Section 3.4.6) to the standard or "vanilla" policy-gradient algorithm OLPOMDP (see Section 3.4.3). We also compare the two PG algorithms to the three baselines described in the following section.

5.1 Baselines

In all experiments, we quote the performance of the initial policy of our algorithms, i. e., the policy our algorithms start out with before learning begins. This policy results from initialising the weights of the linear function approximator that represents the policy to zero. Since both PG algorithms use a neural net of the same structure, the initial policy is the same for both algorithms. Because of the softmax function (see Section 3.2) all phases are thus assigned equal probabilities, which means that the initial policy corresponds to a uniform random policy (with one exception – in one scenario we had to bias the initial policies). We call this baseline RANDOM. A second simple baseline is the performance of a *uniform policy*, i. e., a policy that assigns equal lengths to all phases, where the phases follow a fixed order. We hand-tune this policy to use the best possible phase length (given restricted cycle length) for all of our experiments. This baseline is referred to as UNIFORM.

Our main goal is to compare the performance of our learning approach against a SCATS-like approach. To this end, we implemented a baseline emulating the adaptive part of SCATS. It is called SAT and is described in the following section.

Algorithm 4 SAT

```
1: for cycle = 1 to ∞ do
2:   for p = 1 to PHASES do
3:     throughput = throughput_last_cycle( max_stream( p))
4:     target_max_throughput = throughput / TARGET_SATURATION
5:     target_length[p] = length_for_throughput( target_max_throughput)
6:     target_length[p] = max(1, target_length[p])
7:     target_length[p] = min(target_length[p], MAX_PHASE_LENGTH)
8:   for p = 1 to PHASES do
9:     if length[p] < target_length[p] then
10:      length[p]++
11:    else if length[p] > target_length[p] then
12:      length[p]- -
13:  cycle_length = Σ_p(length[p])
14:  index = 0
15:  while cycle_length >MAX_CYCLE_LENGTH do
16:    if length[index] > 1 then
17:      length[index] - -
18:      cycle_length - -
19:    index = (index + 1) mod PHASES
20:  run_cycle( length)
```

5.1.1 SAT: A Simple Saturation-Balancing Technique

SAT is inspired by the SCATS system described in Section 2.1.5 in that it tries to achieve an equal saturation of the traffic flow on all phases. Like SCATS, it is an adaptive method, adjusting its policy at each decision point in small discrete steps, depending on the current traffic. It also uses the same kind of traffic information from detector loops located closely to the stop-lines of intersections as SCATS. It is, however, much simpler than SCATS as it is purely automatic–it does not include any hand-tuned plans for optimal behaviour given certain traffic patterns, or pre-specified offset plans for coordinating intersections. Our goal was to implement SAT as closely as possible to the adaptive part of SCATS, given limited access to information about the proprietary system (see Section 2.1.5).

SAT has a fixed phase scheme, namely the double diamond overlap DODO (see Section 2.1.1), where the order of the four phases is as follows: north-south-straight, north-south-right, east-west-straight, and east-west-right. Once every cycle, it calculates new targets for the lengths of all phases and adjusts its policy into that direction. It aims for a saturation flow of 90%. Given a maximum cycle length, it starts by allocating time units to each phase in such a way that the most used stream in the phase is as close as possible to 90% saturation. If the resulting plan exceeds the maximum cycle length, time units are subtracted iteratively from all phases until the maximum length requirement is met. This then forms the new

goal plan, and SAT modifies its current plan by adjusting each phase by one time unit towards the goal plan. Each phase must however be allocated at least one time unit.

Pseudo code for SAT is given in Algorithm 4. In line 3 of the algorithm, the function "max_stream" returns the most used stream of a phase, and the function "throughput_last_cycle" then returns all detector counts of that stream during the last cycle. In line 4, a target value is derived for the maximum possible throughput of a phase that would lead to the 90% saturation goal (see the definition of saturation, Equation (2.1) in Section 2.1.1). In line 5, the function "length_for_throughput" calculates the minimum phase length that allows for the target maximum throughput. In line 20, the function "run_cycle" uses the calculated new policy for one cycle, before returning for a new re-calculation of the policy.

In spite of its simplicity, SAT is a surprisingly effective method, in most cases beating the performance of the hand-tuned uniform controller by far, and even beating non-uniform static policies that were hand-tuned (*static* policies are those that do not change over time.) This is mostly due to the fact that SAT is able to (slowly) adjust its policy. For example, by oscillating its policy continually, it is able to give, for example, an *average* length of 2.5 time units to a certain phase, while a fixed policy can only be chosen to either give 2 or 3 time units to that phase. As such, SAT has more flexibility in finding good phase length values even for steady traffic demands.

5.2 Test Scenarios

For our experiments, we develop a test bed of specific traffic scenarios appropriate for demonstrating the advantages of our approach. Our first four test scenarios are designed in such a way that we expect PG learning to outperform the SAT controller. The weaknesses of SCATS and SAT that we target (cf. Section 2.1.5 and Section 5.1.1) are

1. SAT's inability to adjust to rapidly changing demand ("Fluctuation Scenario"),
2. the reactiveness of SAT ("Sudden Influx Scenario"),
3. the fact that SAT cannot calculate the values for offsets between intersections automatically ("Offset Scenario"), and
4. the fact that SAT only optimises locally instead of globally ("Adaptive Driver Scenario").

In three of these four scenarios (2,3, and 4) we show that we can learn from specific observation features by restricting ourselves to a particular subset of the observation features.

The fifth scenario ("Large Scale Optimisation") is a large scale experiment aiming to show the general applicability of PG methods to traffic control. In this scenario we had no particular prior reason to expect PG to outperform SAT. The five scenarios are described in detail in the following sections.

Global rewards are only used if the scenario demands it (in two cases), with local rewards being used in the remaining cases. In all scenarios, we choose the car production rates (i. e., the number of cars entering the system) to be small enough such that SAT can cope with the traffic, i. e., roads never fill up to the point where entering cars must be blocked (cf. Section 4.2). A rather short road length (between 2 and 5) is used in all scenarios. In the global reward case, this facilitates learning for our algorithms as rewards follow the respective actions sooner (due to shorter travel times). The short road lengths and the restrictions to few observation features both speed up learning of our algorithms, which was helpful given the limited time scale of our experiments. However, the same results could probably be obtained with longer roads and full sets of observations, by using a smaller step size parameter and giving the algorithms proportionally more time to learn.

5.2.1 Fluctuating Scenario

The Fluctuating scenario focuses on an intersection of two roads, where horizontal traffic flows from west to east, and vertical traffic flows from north to south. A snapshot of our graphical user interface depicting the network is given by Figure 5.1. The traffic volume entering the system on the horizontal and vertical traffic axes is proportional to a sine and cosine function of the time, respectively. Thus the demand at the centre intersection also oscillates with time. This scenario models a rapidly changing demand that has a pattern. It is realistic because traffic control can lead to "bunching" of traffic: upstream intersections release periodic bursts of traffic, which then disperse as they travel along the road. Pedestrians at a zebra crossing also have the effect of compacting a stream of traffic into waves.

SCATS adapts too slowly to changing demand to deal well with such situations. Since it is purely reactive and has no means of recognising traffic patterns, it can only begin to adapt its policy once it notices changed demand. This means, for example, that when demand on the horizontal axis grows, SAT lags behind, allocating too little time to the horizontal phase, thus building up queues at the central intersection. On the other hand, during times of decreasing demand it allocates more time than necessary to the horizontal phase, unnecessarily creating queues on the vertical axis.

In detail, the traffic volume in the scenario is created as follows. Let $c_n(t)$ be the number of cars entering the system from the north at time step t, and $c_w(t)$ the number of cars entering from the west. Then $c_n(t)$ and $c_w(t)$ are calculated as follows:

$$c_n(t) := \lfloor(\sin(f(t)) + 1)/2 \cdot base\_num\rfloor$$
$$c_w(t) := \lfloor(\cos(f(t)) + 1)/2 \cdot base\_num\rfloor,$$

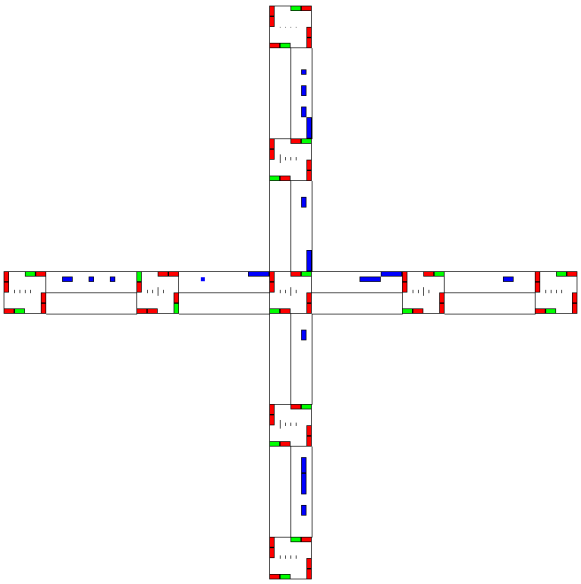


Figure 5.1: The network used in the Fluctuating scenario and the Sudden Influx scenario. Note: as discussed in Section 4.2, we model left-hand sided traffic.

where $f(t)$ is a function of the current time step and $base\_num$ is the average number of cars to be produced. Adding 1 to the sine and cosine functions achieves that the result is always positive, varying between 0 and 2, and hence dividing by 2 and multiplying by $base\_num$ leads to the production of a total of $base\_num$ cars per time step on average in the scenario. In the experiment reported in the next chapter, we set $base\_num := 3$ and $f(t) := \pi/10 \cdot t$, so that after 20 time steps we have completed one period.

Each road consists of 4 pieces linked by intersections, so that there are 5 intersections on each road: 3 central ones and 2 end intersections. As mentioned in Section 4.2, the end intersections only serve the purpose of providing origin and destination nodes for the cars. The control policy at these intersections does not influence the network, so that we do not need to train controllers for them. The road length is 3 units, leading to an optimal travel time of 12 time units for every vehicle. We use all observation features and local rewards in this scenario.

5.2.2 Sudden Influx

The Sudden Influx scenario shows how intersection controllers can learn to "cooperate" by using common observations. In this scenario we make use of the *neighbours* feature described in Section 4.3, meaning that a controller may use the detector counts registered at its neighbour intersections to anticipate traffic coming towards it. The road network is the same as in the Fluctuation scenario (see Figure 5.1), i. e., a crossing of two roads with three controllable intersections and two end

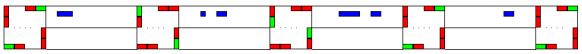


Figure 5.2: The Offset scenario.

intersections on each road, and a road length of 3.

The only regular traffic in this scenario is a steady stream of cars (1 per time step) travelling horizontally from east to west. Hence, usually it is a good policy for the controller of the centre intersection to give maximum time to the east-west-straight phase. Every now and then (with probability of 0.02 per time step), however, we enter a group of 15 cars from the north, with destination south. When this happens, the controller of the centre intersections should change its behaviour temporarily and give more time to the north-south-straight phase, otherwise those cars will wait for a long time before they can pass through. Optimally, the controller learns to anticipate the sudden strong demand for the north-south-straight phase (using the information from its neighbours), and changes its policy in time *before* the cars arrive. We use only the neighbours feature and local rewards for this scenario.

5.2.3 Offset

Many drivers have been frustrated by driving along a main street, to be constantly interrupted by red lights. In this scenario the goal is to learn an offset between neighbouring intersections, a feature that needs to be *hand-tuned* in most other controllers. While SCATS can automatically decide whether to use offsets or not, candidate roads for offsets and the corresponding offset values must be pre-specified (see 2.1.5).

The scenario consists of one road with three consecutive controlled intersections, where we neglect any traffic flowing in from side roads. The network is depicted in Figure 5.2. At every fourth time step, a car enters the system from the west, its destination being the eastern end point. Optimally, the controllers thus learn to be in the east-west-straight phase exactly at those time steps when a car arrives at their intersection. However, due to the constraint that every phase has to come on once during a cycle (where the cycle length is 8 time steps in this scenario), the controllers cannot always stay in the east-west-straight phase. They must thus learn to visit the other phases at those time steps where no traffic is expected, which corresponds to traffic being held up at an upstream intersection. The road length is 2 in this scenario, resulting in an optimal travel time of 8. We restrict the observations to the *cycle duration*, meaning that our controllers learn from time information only, and use global rewards to facilitate cooperation of the controllers. The fact that cars are always entered at the same steps of a cycle makes it possible to learn from the cycle duration feature only. Otherwise the neighbours feature would be needed to tell controllers when to expect traffic.

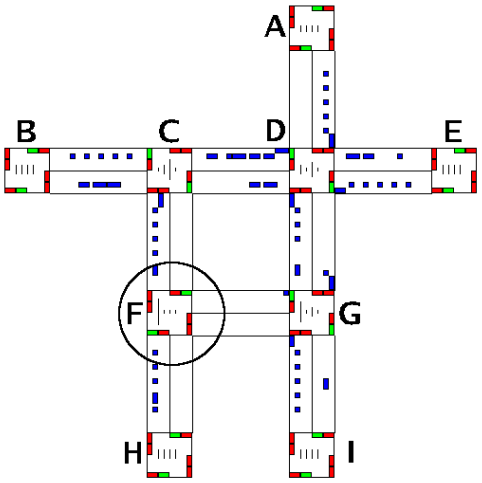


Figure 5.3: The Adaptive Driver scenario.

5.2.4 Adaptive Driver

The Adaptive Driver scenario is an example of a network where optimising all controllers independently does not lead to optimal global performance of the network. The network is depicted in Figure 5.3. Like in the Fluctuation scenario and the Sudden Influx scenario, we have north-south and east-west streams that interact only at a central intersection, Intersection D. Both streams have the same high volume, modelling two main roads, so that the controller of the central intersection should devote approximately equal time to both corresponding phases (north-south-straight and east-west-straight). An additional stream of cars is generated in the south-west corner, at Intersection H, and travels diagonally to the east, to Intersection E. For a vehicle of that stream, two equally short routes are available by going straight, or turning east at Intersection F. However, cars that turn east to join the northbound traffic flow at Intersection G must then turn east again at the central Intersection D, forcing the controller of Intersection D to devote time to a third north-south-right phase, and forcing the main volume of traffic to pause. (This is because in our model of left-hand sided traffic the right turn is an extra phase, see Section 2.1.1). For the overall network it is actually preferable if the controller of the critical Intersection F routes more cars north than east, as those cars can join the main eastbound traffic flow at Intersection C.

This scenario relies on a driver model that prefers, among routes of equal distance, the route with shorter waiting time at intersections. If the controller of the critical Intersection F gives equal time to both the north-south-straight phase and the east-west-right phase, then the numbers of vehicles travelling north and east will be approximately equal (due to random route decisions). A locally optimising system like SAT, which starts out with equal phase lengths, will thus usually con-

tinue to follow a policy that gives equal lengths to both phases. On the other hand, by giving more time to the first of these phases than to the second, the intersection controller can influence the route decisions of the drivers. As the drivers will choose the turn which incurs less waiting time, the controller can actively route cars north.

The road length is again 3, so that 12 would be the optimal travel time. One car per time step is produced at each of the end intersections A, B, E, H and I. Furthermore, a second car per time step is produced with probability 0.15 at each end intersection. This is the highest car production rate that SAT can handle well. Since this scenario is explicitly designed to require optimisation of the global network, we use global rewards. Our observations for this scenario consist only of the *phase durations*, informing the controller how much time it has spent so far in each phase during the current cycle. That way our controllers are forced to learn a good *average* policy which is, for example, not influenced by specific detector counts.

5.2.5 Large Scale Optimisation (10 x 10)

This scenario aims to show the general applicability of PG methods in a relatively large-scale setting. In a 10×10 intersection network, perhaps modelling a city centre, each node potentially produces two cars at each time step according to fixed small probabilities between 0 and 0.25. The production probabilities and the two destinations for the cars from each source are chosen randomly at the beginning and then stay fixed during the run of the simulation. This way we create a system with semi-random traffic patterns: stochastic route choices for the vehicles and the stochasticity in the car production create variance in the system. On the other hand, since the sources and destinations for all cars are chosen when the scenario is constructed, the system has some regularity which the controller can learn to optimise. Compared to the size of the intersection subgroups that SCATS' base plans optimise (10 – 20 intersections, see Section 2.1.5), 100 intersections is a remarkable number. The road length is 3, and we use local rewards and all observations.

5.3 Setup of the Experiments

On all five test scenarios, we run NAC, OLPOMDP and the three baselines and report the best result achieved by each algorithm within restricted run time. The parameters for the two PG algorithms are hand-tuned for best performance. As explained in Section 4.5, our goal is to minimise the total travel time of vehicles in the system, even if our algorithms optimise this quantity indirectly. Performance is hence measured in terms of travel time.

In a second set of experiments, we compare the convergence properties of the two PG algorithms by showing the average performance over several runs in two of the scenarios. In this case, our parameters are tuned for maximum speed of convergence rather than best performance. Thirdly, we evaluate the usefulness of

our observation features by training with several different subsets of the full set of features, comparing the resulting performances.

5.3.1 Particular Design Decisions

The tuning of the parameters is a critical task. If the step sizes, for example, are small, learning takes a very long time. On the other hand, if the step sizes are too big, the PG algorithms may quickly get stuck in a suboptimal solution. Because of the difficulty of determining an efficient rate of decrease for the step sizes, we followed common practice and used constant step sizes in our experiments. However, setting this parameter is still a trade-off between rapid convergence and quality of the achieved results. For comparing the best performances of NAC and OLPOMDP we therefore use conservative step sizes, while we experiment with bigger step sizes to compare the convergence properties of the two algorithms.

Following standard RL practice [12], a reward baseline is used for both PG algorithms (see Section 3.4.5), i. e., instead of feeding the learners the actual rewards obtained at every step, they receive the difference between the reward obtained and the estimated average reward. This is known as an additive control variate method and has the effect of reducing the variance in gradient estimates. Intuitively, a positive difference between a reward and the baseline is "good" and a negative difference is "bad" [12]. This baseline is reset every 1000 – 100,000 iterations, depending on the overall run time of the algorithms for a given scenario. Especially for NAC this baseline improves the results considerably.

We noticed that sometimes the PG algorithms produce policies that block cars. As described in Section 4.2, we assume a limited road capacity, so that cars that would enter the system on a certain road are not permitted to enter if that road is already full. Controllers may thus block cars by giving, for example, minimum possible green time to a certain phase. This leads to queues building up at the intersection, until the incoming road is full. Because blocked cars never enter the system, they never count towards the travel time. Our rewards are closely linked to the travel time and are affected similarly. Cars that do not enter the system result in a lower number of total cars, our global reward. Blocking cars can also lead to an increase in local rewards, because a controller may be able to maximise the throughput of an intersection by always preferring a phase with heavy traffic over a phase with little traffic.

One way to avoid this problem in the case of global rewards would be to not let the simulator ignore cars that cannot enter the system, but instead accumulate them in a queue and enter them as soon as possible at some later point in time (while counting them towards the number of cars in the system). However, this significantly impedes learning. In the beginning, while our algorithms follow a near-random policy, long queues of waiting cars build up in such a system, and it may take a long time for the system to reach a state of "normal" traffic volume, hindering the ability of the algorithms to learn a correct policy. An alternative solution (which also works for local rewards) is to immediately punish the algorithms for

Scenario	Random	Uniform	SAT	NAC	OLPOMDP
Fluctuating	250.0	102.0	21.5	14.3	13.4
Sudden Influx	197.0	35.0	18.4	13.4	13.5
Offset	17.9	15.0	12.0	8.0	8.0
Adaptive Driver	251.0	74.2	17.2	15.8	16.0
Large Scale	60.5	54.7	35.1	29.8	27.9

Table 5.1: Travel times for PG and the baseline algorithms.

each blocked car with a high negative reward. We chose this approach, using a punishment of -100 for *every* controller whenever a car was blocked. This is because it is usually not possible to identify a single intersection that causes a blocked road – it is rather the whole network that does not work well. This approach led to the desired effect of mostly producing policies that do not block cars. With unfortunate parameter settings (e. g., a big step size), the problem may still occur. However, for the results reported in this work we took care to use only parameter settings that did not produce such unwanted policies.

Finally, in the Large Scale scenario our algorithms had difficulties to learn because of a "fill-up" effect of the system. Even if blocked cars do not accumulate outside the system, a bad initial policy leads to over-saturated traffic conditions, which in a large system may persist for a long time. We therefore *biased* the policy in this scenario by adding a constant (of 4) to each output of the function approximator that represents the policy. Hence, instead of using a completely random policy in the beginning, we bias the policy towards uniformity. The function approximator can still learn to completely overrule this bias by increasing its parameters accordingly, but profits from the bias in the beginning of learning. This helped to overcome the learning problems in the Large Scale scenario and would be a sensible trick in a real world problem.

5.4 Results and Analysis

Our results quote the average travel time (TT) of vehicles in the system. Table 5.1 contains the results for the three baselines and the best results for NAC and OLPOMDP that could be obtained within restricted run time. The table quotes single runs with tuned parameters. The parameters used are given in Table 5.3. In Appendix A, we also show the graphs of the PG runs for all quoted results.

The results in Table 5.1 show that NAC and OLPOMDP both improve upon the uniform controller and SAT in all scenarios. The two PG algorithms mostly achieve similar travel times. NAC sometimes shows slightly worse performance, which may however be due to the time restriction (see the detailed discussion in the following section). Table 5.2 contains the computation time and the number of iterations needed by both algorithms to arrive at the travel time quoted in Table

	NAC			OLPOMDP		
Scenario	TT	n	secs	TT	n	secs
Fluctuating	14.3	$4.5 \cdot 10^6$	860,549	13.4	$1.1 \cdot 10^9$	491,298
Sudden Influx	13.4	$4.4 \cdot 10^6$	25,454	13.5	$9.7 \cdot 10^8$	35,572
Offset	8.0	$2.1 \cdot 10^6$	1,973	8.0	$6.3 \cdot 10^8$	8,546
Adaptive Driver	15.8	$9.3 \cdot 10^7$	867,267	16.0	$2.2 \cdot 10^9$	807,496
Large Scale	29.8	$2.9 \cdot 10^5$	1,077,151	27.9	$3.0 \cdot 10^8$	1,029,428

Table 5.2: Run times and iterations for all scenarios for the PG algorithms. Optimisation was performed for n iterations of the algorithm. 'Secs' is wall-clock time. Cut-off time was, in order, 887,150, 45,264, 15,967, 888,388, 1,077,151 seconds for the 5 scenarios. Experiments were run on a 3.2 GHz AMD Athlon 64 processor.

5.1, and also specifies our cut-off limit, i. e., the maximum allowed run time, for each scenario. SAT always found its policies very fast, i. e. within a few thousand iterations, which is why we did not include run times for SAT in the table. As can be observed, NAC can require more computation time as OLPOMDP, but achieves its results in up to 3 orders of magnitude fewer learning steps. In a real-world deployment, both NAC and OLPOMDP would have no difficulty of keeping up with real-time. This is because each iteration (and hence decision on the next phase) corresponds to a few seconds of real-time. Furthermore, in a real deployment each intersection policy would be calculated in parallel instead of sequentually. In our simulation NAC only required fractions of a second (about 0.03 seconds) for each iteration per controller. The results for NAC are thus very appealing, because it needs fewer learning steps and hence fewer interactions with the environment for good performance, which in a real system would mean faster adaption to shifting traffic patterns.

The parameters in Table 5.3 are very similar across all scenarios. In accordance with theory, lowering the discount factor β for OLPOMDP achieved faster learning, but worse end results as the responsibility for rewards is not propagated as far back to earlier states than with a high discount factor 3.4.3. For NAC, λ and γ both discount the eligibility trace, which explains why high values for both parameters lead to good results. As the critic discount γ also indicates how related neighbouring states are, lowering this value could sometimes lead to more stable convergence. This was the case in scenarios where observations could change drastically from one time step to the next. Overall, we found that NAC's critic discount did not have a very big influence, such that NAC is not much harder to tune than OLPOMDP, even though it requires one parameter more.

5.4.1 Results Per Scenario

Fluctuating. As can be seen in Table 5.1, NAC and OLPOMDP both achieve good travel times of 14.3 and 13.4 respectively, which is close to the theoretically

	NAC			OLPOMDP	
Scenario	ϵ	λ	γ	ϵ	β
Fluctuating	10^{-5}	0.9	0.95	10^{-3}	0.9
Sudden Influx	10^{-4}	0.9	0.95	10^{-4}	0.9
Offset	$5 \cdot 10^{-5}$	0.98	0.9	$5 \cdot 10^{-6}$	0.98
Adaptive Driver	10^{-7}	0.98	0.95	10^{-6}	0.98
Large Scale	10^{-4}	0.9	0.95	10^{-5}	0.9

Table 5.3: Optimisation parameters for all scenarios for the PG algorithms.

optimal time of 12. These performances are far better than those of the uniform controller and SAT.

The result of NAC, however, is notably worse than that of OLPOMDP. This may be due to the fact that NAC would have needed more run time, while we terminated the runs after ten days. As can be observed in Figure 5.4, NAC was still improving notably at that point and had not reached a "quasi steady-state" (i. e., a level of performance from where improvement is substantially slower than before). However, the learning rate had slowed down, so that it was hard to predict just how long we would have needed to continue the runs in order to achieve the same result for NAC as for OLPOMDP. Given restricted time we thus opted to terminate the experiments, but point the reader to the graph in Figure 5.4 for a visual estimate of NAC's learning rate. The effect that NAC does not manage to reach the same performance as OLPOMDP within the limited time of our experiment is also present in the Large Scale scenario. It can be attributed to the fact that we are using all observation features in these two scenarios, while using considerably less features in the other scenarios. One iteration of OLPOMDP is linear in the number of observation features, but NAC needs quadratic time (see Sections 3.4.3 and 3.4.7). The rather long computation time was also the reason for using local rewards in this scenario and most others (see Section 4.5).

Sudden Influx. Both NAC and OLPOMDP learned a good policy in terms of travel time using local rewards (see Table 5.1). When examining the learned policies, we noted in particular that for both algorithms the centre intersection controller had learned to switch to the north-south-straight phase just in time to let the group of cars from the north pass without waiting. This is something that SAT and SCATS cannot do.

Offset. Both PG algorithms learned an optimal policy in this scenario using global rewards. SAT performed badly because it has no means of implementing an offset. We discovered, however, that learning an optimal policy is difficult. For a road length of 3, for example, we failed to do so (given limited time). We also had to lower the maximum cycle length from its usual value of 16 down to 8, and to set

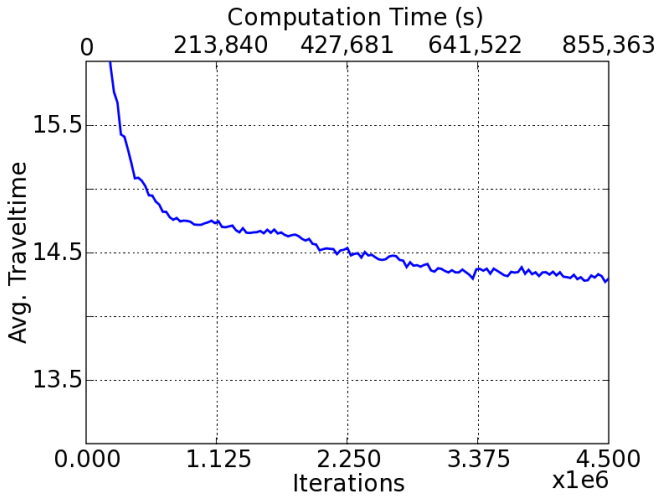


Figure 5.4: Quoted run for NAC in the Fluctuating scenario.

the maximum phase length to 5 accordingly. Otherwise our controllers always preferred to stay in the east-west-straight phase for the maximum time possible, which is not an optimal policy. Our local reward measure also did not lead to the optimal policy in this case. What makes this scenario difficult is that intersection $n+1$ can only begin to learn its part of a network-wide optimal policy when intersection n has already converged to an approximately correct policy.

Adaptive Driver. Although the average travel time of the PG algorithms was only slightly better than that of SAT, their policies were radically different. SAT routed cars equally north and east at the critical intersection F (see 5.3). The PG algorithms, on the other hand, routed most cars north. In this scenario a slightly larger volume of vehicles made SAT cause permanent traffic jams, while the PG algorithms still found the correct policy. To verify our claim that routing cars north is advantageous in this scenario, we hand-coded a corresponding optimal static policy. Although a static policy has the restriction that it cannot spend a fractional number of steps (on average) in any phase (see Section 5.1.1), this policy beat SAT slightly, achieving a travel time of 17.1. The PG algorithms, in turn, beat this hand-coded policy, achieving what we believe is near-optimal performance in this scenario.

Large Scale Optimisation. OLPOMDP gave an average travel time improvement of 20% over SAT even though this scenario was not tailored for our controller. NAC also beat SAT notably, although the results quoted for it suffer again from the fact that we cut NAC off after 12 days, when it was still improving notably.

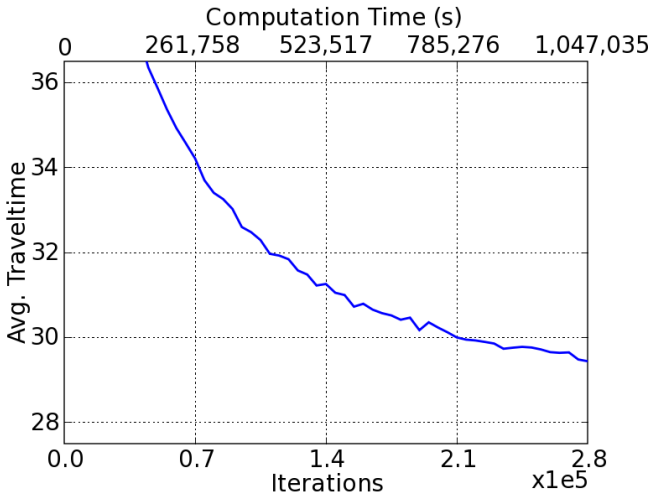


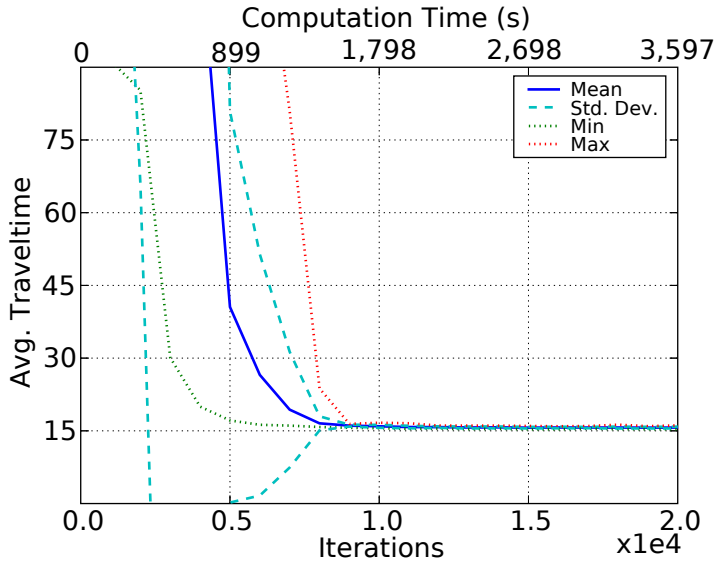
Figure 5.5: Quoted run for NAC in the Large Scale scenario.

5.4.2 Convergence Rates

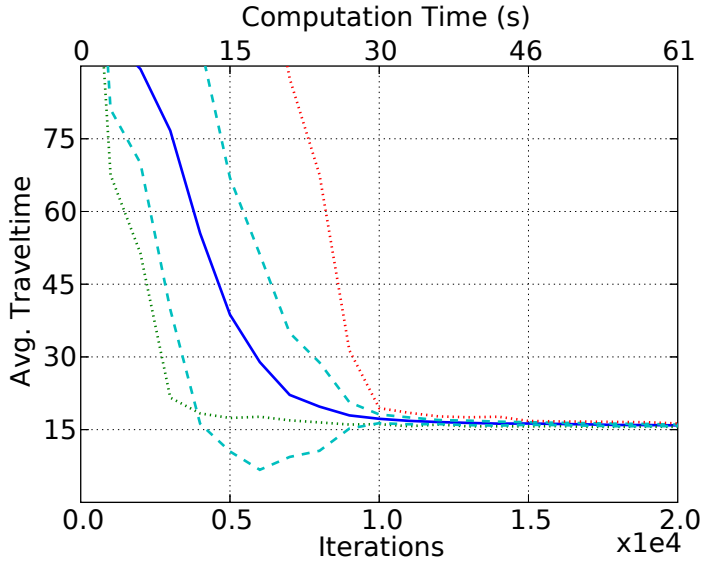
To check the reliability of convergence and compare the properties of the two algorithms, Figures 5.6 and 5.7 display the results of 30 runs for both algorithms in two of our scenarios that demonstrates extremes of behaviour for NAC and OLPOMDP. Here, we tuned the parameters for maximally fast convergence. As can be observed, both algorithms achieve approximately the same policy quality, but their relative convergence behaviour differs substantially in the two scenarios. In the Fluctuating scenario, both algorithms reach a "quasi steady-state" at around 7000 – 8000 iterations. The OLPOMDP algorithm converges slightly faster in terms of iterations, and within a fraction of the computation time needed by the NAC algorithm. In the Offset scenario, on the other hand, NAC is far superior to OLPOMDP both with regard to iterations and computation time. In the three remaining scenarios NAC always required orders of magnitude fewer iterations, but sometimes more computation time than OLPOMDP (see Table 5.2).

5.4.3 Assessment of Observation Features

To examine the relative usefulness of the observation features, we analysed the respective performances resulting from using various subsets of the features. We ran tests with OLPOMDP on the Fluctuation scenario (which usually uses all features), removing in each run of the algorithm one of the observation features. Removing a single feature always resulted in a slight degradation of performance as opposed to the full set, regardless of which feature was removed. Successively removing features caused an initially smooth degradation of the policy performance, until the performance suddenly deteriorated with the removal of the last two or three

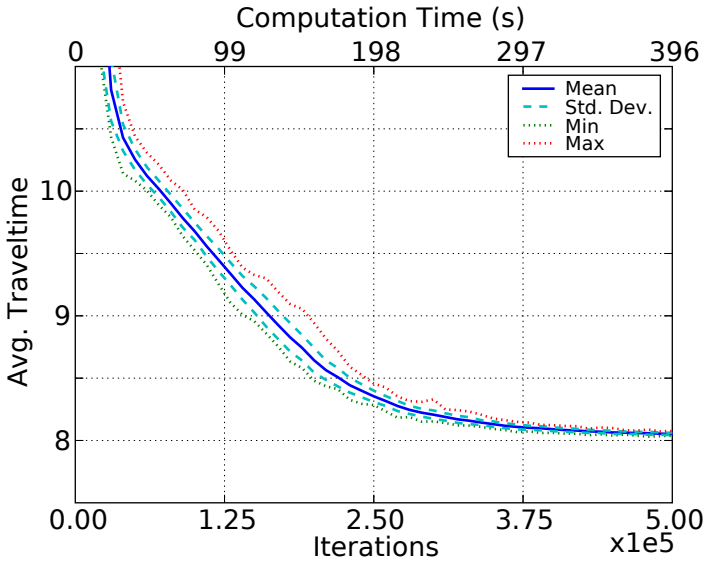


(a) Fluctuating NAC

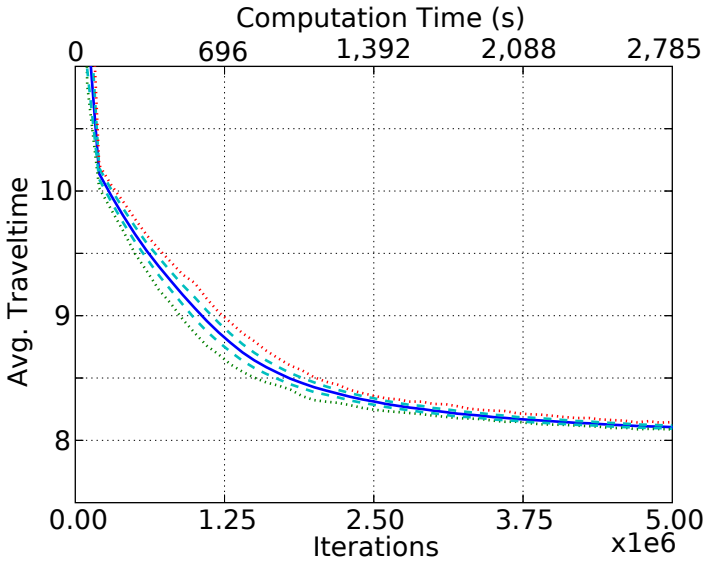


(b) Fluctuating OLPOMDP

Figure 5.6: Convergence properties of NAC (top) compared to OLPOMDP (bottom) over 30 runs in the Fluctuation scenario. Step size ϵ is 5×10^{-4} for both algorithms, the discount factors β and λ is 0.98, and NAC's critic discount γ is 0.9.



(a) Offset NAC



(b) Offset OLPOMDP

Figure 5.7: Convergence properties of NAC (top) compared to OLPOMDP (bottom) over 30 runs in the Offset scenario. Step size ϵ is 1×10^{-4} for both algorithms, the discount factors β and λ are 0.98, and NAC's critic discount γ is 0.9.

Successively Removed Features	Travel Time
none (using all features)	13.5
neighbours	13.8
cycle duration	14.2
detector active	14.3
current phase	16.8
current phase duration	17.0
phase durations	67.4
detector history (using no features)	70.4

Table 5.4: Performance degradation of the OLPOMDP algorithm when removing observation features in the Fluctuation scenario.

features. Again, we noted that the *order* in which observations are removed did not matter much. Hence we can establish the fact that indeed all of our observation features can be useful for learning. No feature can be singled out to be particularly helpful, but all of them contribute to learning a good policy in this scenario. As an example, we show in Table 5.4 one particular order of feature removal and the resulting travel time values.

Chapter 6

Conclusion and Outlook

In this thesis, we have used reinforcement learning techniques to learn the control of traffic lights in a simulated traffic system. Our experiments show that policy-gradient algorithms can learn to control large networks, while achieving a very good network-wide performance. In particular, we have demonstrated that some of the problems that saturation-balancing algorithms suffer from can be avoided with our approach. As a saturation-balancing algorithm is the heart of SCATS, one of the world's most widely used traffic control systems, we have shown that our methods can potentially improve traffic control in a real-world deployment.

We have examined the performance of the recently proposed Natural Actor Critic (NAC) algorithm, and concluded that this algorithm can be especially useful for traffic control. It reduces the number of learning iterations greatly as opposed to a "classical" policy-gradient method, while retaining all the benefits of policy-gradient methods. However, as NAC needs far more computation time per learning iteration, a classical policy-gradient algorithm might be more suitable in environments that can be simulated quickly and where the number of learning iterations is of secondary importance.

A practical next step will be to test the performance of our controllers in a more realistic environment, using sophisticated commercial traffic simulations like Paramics (see Section 4.2). This system is also used by the New South Wales Roads and Traffic Authority for testing SCATS. Implementing the control protocol used by SCATS, we hope to be able to directly compare the performance of our controllers against SCATS in Paramics and other environments. Learning off-line from real data could be a further step towards testing the appropriateness of our approach for real-world deployment. An assessment of the robustness of our controllers in unexpected conditions is important to answer the question whether reinforcement learning controllers are reliable enough to be deployed in a real world system.

In future research, we would also like to further explore the topic of learning policies in a distributed fashion. Ideally, we want to optimise the global performance of a traffic network. However, because in large systems a global perfor-

mance criterion incurs very long learning times for the controllers, we have also used local criteria in this work, optimising each controller independently. This can lead to good global results if the local criteria are appropriate, as we have shown to be the case for our criteria. However, there are no theoretical guarantees for the global quality of policies resulting from such local optimisation. It has been shown that *propagation* of local rewards across a network can lead to global optimisation, if the global performance criterion is the sum of the local criteria [21]. Such a propagation of local rewards thus leads to exactly the same result as using a global reward: it guarantees global quality of the solution, but it also results in the long learning time needed for a global criterion. By propagating local rewards only *partially* through the network we might be able to find an intermediate solution between local and global rewards, a trade-off between quality of the solution and learning time. It would be interesting to determine whether *discounted* propagation of local rewards, for example, could lead to a significant improvement of performance, and whether theoretical bounds can be established for the quality of the resulting solution.

Finally, comparing our policy-gradient techniques against other reinforcement learning approaches for traffic control (see Section 3.5) would be interesting from an algorithmic point of view, and could give further answers about the specific advantages and disadvantages of policy-gradient methods in complex systems.

Appendix A

Detailed Results

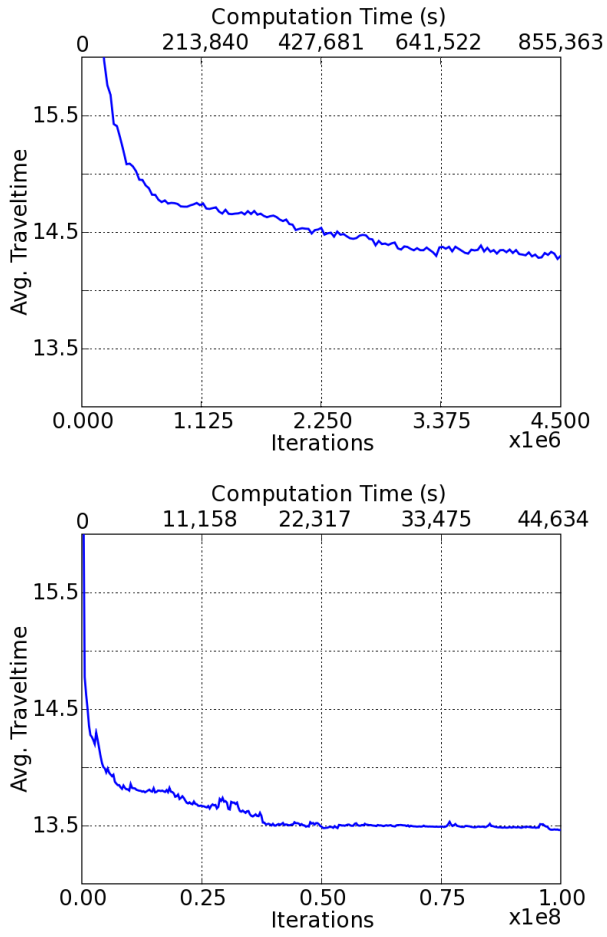


Figure A.1: Fluctuating scenario. Quoted run for NAC (top) and OLPOMDP (bottom).

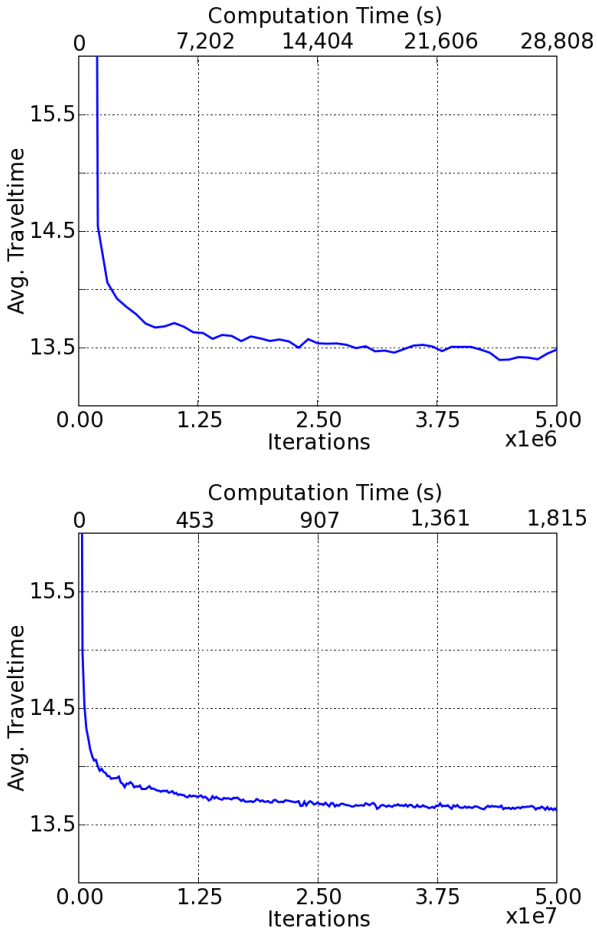


Figure A.2: Sudden Influx scenario. Quoted run for NAC (top) and OLPOMDP (bottom). Note the different scale.

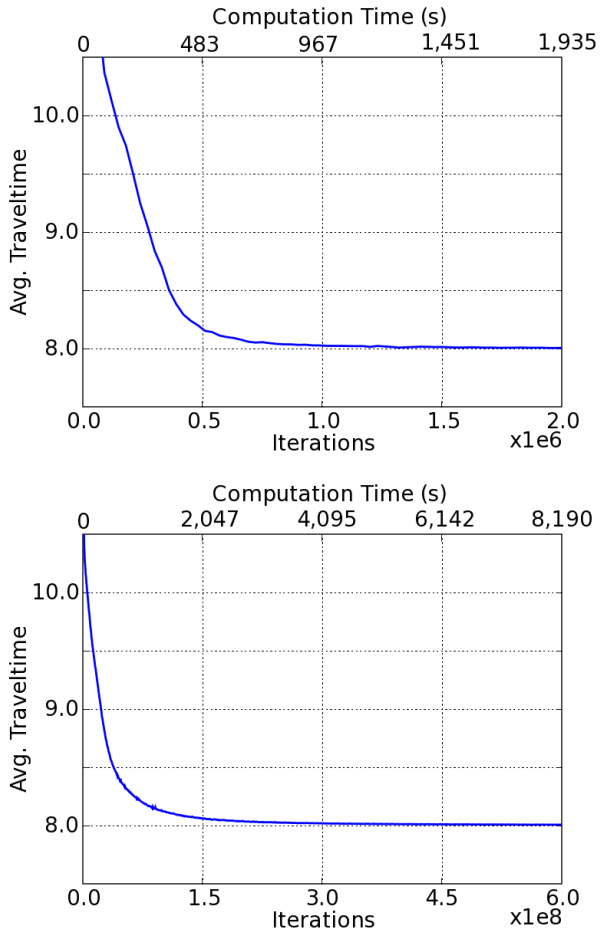


Figure A.3: Offset scenario. Quoted run for NAC (top) and OLPOMDP (bottom).

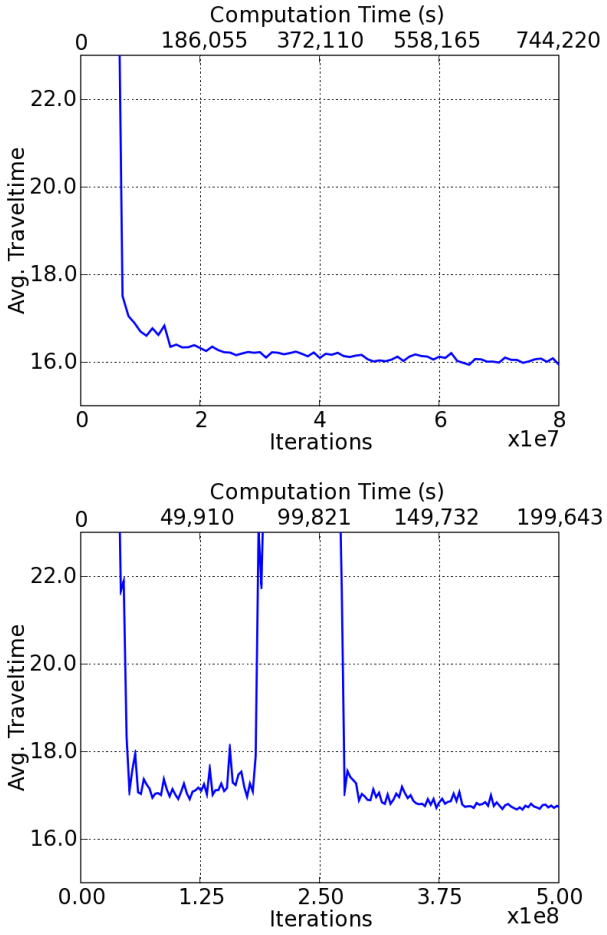


Figure A.4: Adaptive Driver scenario. Quoted run for NAC (top) and OLPOMDP (bottom).

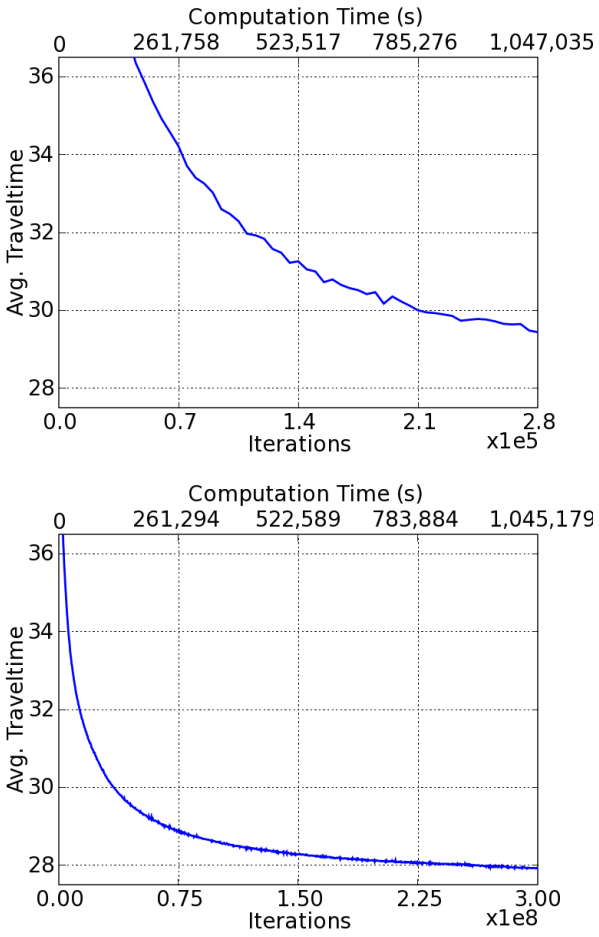


Figure A.5: Large Scale scenario. Quoted run for NAC (top) and OLPOMDP (bottom).

Bibliography

[1] Douglas Aberdeen. *Policy-Gradient Algorithms for Partially Observable Markov Decision Processes*. PhD thesis, Australian National University, Canberra, Australia, March 2003.

[2] Shun-ichi Amari. Natural gradient works efficiently in learning. *Neural Computation*, 10(2):251–276, 1998.

[3] J. Andrew Bagnell and Andrew Y. Ng. On local rewards and scaling distributed reinforcement learning. In *Advances in Neural Information Processing Systems, Proceedings of the 19th Neural Information Processing Systems Conference (NIPS'2005)*, pages 91–98, 2006.

[4] Bram Bakker, Merlijn Steingröver, Roelant Schouten, Emil Nijhuis, and Leon Kester. Cooperative multi-agent reinforcement learning of traffic lights. In *Proceedings of the Workshop on Cooperative Multi-Agent Learning, European Conference on Machine Learning (ECML'05)*, pages 24–36, Porto, Portugal, 2005.

[5] Jonathan Baxter and Peter L. Bartlett. Infinite-horizon policy-gradient estimation. *Journal of Artificial Intelligence Research*, 15:319–350, 2001.

[6] Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-dynamic Programming*. Athena Scientific, Belmont, MA, 1996.

[7] Léon Bottou and Yann Le Cun. Large scale online learning. In *Advances in Neural Information Processing Systems, Proceedings of the 17th Neural Information Processing Systems Conference (NIPS'2003)*, pages 217–224, 2004.

[8] Justin A. Boyan. Least-squares temporal difference learning. In *Proceedings of the 16th International Conference on Machine Learning (ICML'99)*, pages 49–56, Bled, Slovenia, 1999.

[9] Bernhard Friedrich, Irina Matschke, Essam Almasri, and Jürgen Mück. Data fusion techniques for adaptive traffic signal control. In *Proceedings of the 10th IFAC Symposium on Control in Transportation Systems*, Tokyo, Japan, 2003.

[10] Bernhard Friedrich and Mohamed Shahin. Adaptive traffic control in metropolitan areas. In *Proceedings of the 4th International Conference on the Role of Engineering Towards a Better Environment (RETBE'02)*, Alexandria, Egypt, 2002.

[11] Nathan H. Gartner, Carroll J. Messer, and Ajay K. Rathi, editors. *Traffic Flow Theory: A State of the Art Report – Revised Monograph on Traffic Flow Theory*. U.S. Department of Transportation, Transportation Research Board,Washington, D.C., June 1992.

[12] Evan Greensmith, Peter L. Bartlett, and Jonathan Baxter. Variance reduction techniques for gradient estimates in reinforcement learning. *Journal of Machine Learning Research*, 5:1471–1530, 2004.

[13] Leslie Pack Kaelbling, Michael L. Littman, and Andrew P. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[14] Sham Kakade. A natural policy gradient. In *Advances in Neural Information Processing Systems, Proceedings of the 15th Neural Information Processing Systems Conference (NIPS'2001)*, volume 2, pages 1531–1538, 2002.

[15] John Loch and Satinder Singh. Using eligibility traces to find the best memoryless policy in partially observable Markov decision processes. In *Proc. 15th International Conference on Machine Learning (ICML'98)*, pages 323–331. Morgan Kaufmann, San Francisco, CA, 1998.

[16] Omid Madani, Steve Hanks, and Anne Condon. On the undecidability of probabilistic planning and related stochastic optimization problems. *Artificial Intelligence*, 147:5–34, 2003.

[17] Angelia Nedić and Dimitri P. Bertsekas. Least-squares policy evaluation algorithms with linear function approximation. *Discrete Event Dynamic Systems: Theory and Applications*, 13(1–2):79–110, 2003.

[18] Christos H. Papadimitriou and John N. Tsitsiklis. The complexity of Markov decision processes. *Mathematics of Operations Research*, 12(3):441–450, 1987.

[19] Markos Papageorgiou. Traffic control. In R. W. Hall, editor, *Handbook of Transportation Science*, pages 233–267. Kluwer Academic Publishers, Boston, 1999.

[20] Markos Papageorgiou, Christina Diakaki, Vaya Dinopoulou, Apostolos Kotsialos, and Yibing Wang. Review of road traffic control strategies. *Proceedings of the IEEE*, 91(12):2043–2067, 2003.

[21] Mark Paskin and Carlos Guestrin. A robust architecture for distributed inference in sensor networks. Technical Report IRB-TR-03-039, Intel Research Berkeley, 2003.

[22] Jan Peters, Sethu Vijayakumar, and Stefan Schaal. Natural actor-critic. In *Proceedings of the 16th European Conference on Machine Learning (ECML'05)*, pages 280–291, Porto, Portugal, 2005.

[23] R. A. Rescorla and A. R. Wagner. A theory of pavlovian conditioning: variations in the effectiveness of reinforcement and nonreinforcement. In A. H. Black and W. F. Prokazy, editors, *Classical Conditioning II*, pages 64–99. Appleton Century Croft, New York, NY, 1972.

[24] Jussi Rintanen. Complexity of probabilistic planning under average rewards. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 503–508, Seattle, Washington, 2001.

[25] Dennis I. Robertson. TRANSYT method for area traffic control. *Traffic Engineering & Control*, 10:276–281, 1969.

[26] Dennis I. Robertson and R. David Bretherton. Optimizing networks of traffic signals in real time — the SCOOT method. *IEEE Transactions on Vehicular Technology*, 40(1):11–15, February 1991.

[27] Sheldon M. Ross. *Introduction to Stochastic Dynamic Programming*. Academic Press, New York, 1983.

[28] Nagui Rouphail, Andrzej Tarko, and Jing Li. Traffic Flow at Signalized Intersections. In Nathan H. Gartner, Carroll J. Messer, and Ajay K. Rathi, editors, *Traffic Flow Theory: A State of the Art Report – Revised Monograph on Traffic Flow Theory*, chapter 9. U.S. Department of Transportation, Transportation Research Board, Washington, D.C., June 1992.

[29] Stuart Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, New Jersey, 1995.

[30] A. G. Sims and K. W. Dobinson. The Sydney coordinated adaptive traffic (SCAT) system – philosophy and benefits. *IEEE Transactions on Vehicular Technology*, 29(2):130–137, May 1980.

[31] Satinder P. Singh, Tommi Jaakkola, and Michael I. Jordan. Learning without state-estimation in partially observable Markovian decision processes. In *Proceedings of the 11th International Conference on Machine Learning (ICML'94)*, pages 284–292, 1994.

[32] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[33] Richard S. Sutton, David McAllester, Satinder P. Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems, Proceedings of the 13th Neural Information Processing Systems Conference (NIPS'1999)*, pages 1057–1063, 2000.

[34] Thomas L. Thorpe and Charles W. Anderson. Traffic light control using SARSA with three state representations. Technical report, IBM Corporation, 1996.

[35] Marco Wiering, Jilles Vreeken, Jelle van Veenen, and Arne Koopman. Simulation and optimization of traffic in a city. In *IEEE Intelligent Vehicles Symposium (IV'04)*, pages 453–458, Parma, Italy, 2004.