Improving Description-based Q-learning

UNIVERSITÄT DES SAARLANDES

Computer Science Department
Saarland University
Germany

A thesis presented for the degree of M. Sc. Computer Science

Author: Lukas Maria Schaller

Supervisor: M. Sc. Timo P. Gros,
Ph.D. student in the Modeling and Simulation Group

Examiner: Univ.-Prof. Dr. Verena Wolf,
Modeling and Simulation Group

Second Examiner: Univ.-Prof. Dr. Jörg Hoffmann,
Foundations of Artificial Intelligence (FAI) Group

Date of submission: January 23, Saarbrücken, Germany

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine andere als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in der Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with my passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 23.01.2023 [illegible]

(Datum, Date) (Unterschrift/Signature)

Abstract

Deep reinforcement learning (DRL) has made promising advances in recent years solving complex sequential decision-making problems, e.g. by beating the grandmasters in the game of Go, predicting protein folding or speeding up matrix multiplication. However, applying DRL to new problems successfully requires careful tuning and adaptation to find a suitable algorithm. This was demonstrated by the creators of MoGym, a tool which allows training DRL agents on any problem formulated as Markov decision process in the JANI specification language. They applied Deep Q-learning (DQN) on a set of problems of the Quantitative Verification Benchmark Set (QVBS) to test their tool and noticed that the DRL algorithm could not successfully learn on some instances. This thesis aims to find the main challenges of such instances loaded with MoGym, provide the community with DRL algorithms and techniques to address these issues and especially aims to contribute to the community by improving on existing approaches.

It is shown that the DQN agent was likely unable to learn due to an exploding action-space (as a result of loading a problem using MoGym) and the lack on sophisticated exploration during training. Description-based Q-learning is proposed as a possible solution to replace DQN, because this variant can mitigate the effect of the action-space dimension by learning state-values instead of action-values. To solve the exploration problem, the thesis proposes a new version of the Go-Explore framework called Stochastic Go-Explore. The new variant drops some limiting assumptions on the training environment, providing the DRL community with a powerful exploration technique that is applicable to a larger variety of training environments than its predecessor. The experiments show that this new technique is the only one of all tested exploration techniques which enables the agents to learn on all tested problems. In addition, DBQL is extended to use n-step learning, which aims to improve sample efficiency by proposing three new n-step DQBL agents. Experiment findings establish their feasibility and provide some evidence to show that their use can lead to sample efficiency gain. However, the experiment results further suggest that this improvement might lead to higher training times. A discussion on when to use n-step learning when training time is a limiting factor is included. All findings are based on experiments made on a subset of QVBS problems and are evaluated using deep statistical model checking (DSMC) and training returns. It is shown that there exists a discrepancy in the measurements of both approaches, hinting that DSMC can be a valuable tool for the DRL community when evaluating experiments.

Acknowledgments

First I want to thank Timo Gros, who has introduced me to the topic of reinforcement learning, for supervising my thesis. Furthermore, I would like to thank Prof. Dr. Verena Wolf and Prof. Dr. Jörg Hoffmann for reviewing my thesis.

I also thank my father for supporting me in my studies, financially as well as mentally. His productivity and resilience has greatly inspired my work ethic. Further, I would like to thank my sister for her commitment to extensively proofread my thesis despite the language and subject difficulties. Finally, I thank my friends, be it the ones I made at the university or older friends, for supporting me and making my time at the university significantly more fun.

Contents

1 Introduction 1
- 1.1 Research Background . . . 1
- 1.2 Research Problems and Questions . . . 2
- 1.3 Limitations and Thesis Structure . . . 4

2 Theoretical Background 5
- 2.1 Reinforcement Learning . . . 5
- 2.2 Deep Q-learning . . . 7
- 2.3 Expected vs Sample Updates . . . 9
- 2.4 Description-based Q-learning . . . 10
- 2.5 N-step Learning . . . 12
- 2.6 Action-space Explosion . . . 13
- 2.7 Deep Statistical Model Checking . . . 15

3 Literature Review 16
- 3.1 N-step Learning . . . 16
- 3.2 Exploration . . . 17
  - 3.2.1 Count-based Exploration . . . 17
  - 3.2.2 Intrinsic Motivation . . . 18
  - 3.2.3 Random Networks . . . 19
  - 3.2.4 Planning and Exploration . . . 19

4 N-step Description-based Q-learning 21
- 4.1 Sample Description-based Q-learning . . . 21
- 4.2 Recursive Description-based Q-learning . . . 23
- 4.3 Matrix Description-based Q-learning . . . 25

5 Go-Explore 29
- 5.1 Original Go-Explore . . . 29
  - 5.1.1 Exploration Phase . . . 29
  - 5.1.2 Robustification Phase . . . 30
- 5.2 Stochastic Go-Explore . . . 30
  - 5.2.1 Stochasticity During Training . . . 31

5.3 Implementation Details . . . 32
5.3.1 Exploration Archive . . . 32
5.3.2 Heuristic . . . 32
5.3.3 Exploration Technique . . . 33
5.3.4 Pseudocode . . . 34

6 Methodology **35**
6.1 General Methods and Limitations . . . 35
6.1.1 Scope of Experiments . . . 35
6.1.2 DRL Limitations . . . 36
6.1.3 Hyperparameter Tuning . . . 38
6.2 First Research Question . . . 39
6.3 Second Research Question . . . 39
6.4 Summary . . . 43

7 Results and Evaluation **44**
7.1 Action-space Explosion and Exploration . . . 44
7.1.1 DSMC Evaluations . . . 45
7.1.2 Training Returns . . . 47
7.1.3 Conclusion . . . 50
7.2 N-step Experiments . . . 51
7.2.1 Sample Efficiency . . . 51
7.2.2 Time Efficiency . . . 53
7.2.3 Conclusion . . . 56
7.3 Timer Data Analysis . . . 57
7.3.1 Conclusion . . . 62

8 Conclusion and Future Work **63**

Appendix **69**
9.1 Hyperparameters . . . 69
9.2 Discussion: Stochastic Go-Explore vs Existing Approaches . . . 71
9.3 DSMC Exploration Plots using GRP . . . 75
9.4 DSMC N-step Plots using GRP . . . 78
9.5 Timeout Data vs Non-Timeout Data . . . 80

Chapter 1 Introduction

Deep learning (DL), especially deep reinforcement learning (DRL), has become a very promising field to solve sequential decision-making problems in recent years. Starting with Deep Q-learning (DQN) [1], reinforcement learning (RL) in combination with deep neural networks, known as deep reinforcement learning, has shown to be capable of finding good policies in very complex tasks, e.g. in playing games like Atari games [1, 2], Chess or Go [3, 4]. Apart from games, DRL achieved further advances in the field of robotics [5] or in scientific problem settings, e.g. by solving the protein folding problem [6] or speeding up matrix multiplication [7]. However, careful engineering and tuning of DRL algorithms is needed in order to either achieve the desired results or to solve such problems at all. MoGym [8] is a framework which enables training any (D)RL algorithm on problems described in the JANI format [9], e.g. from the *Quantitative Verification Benchmark Set* (QVBS) [10, 11]. Although well established algorithms like DQN can be successfully applied to complex problems, like playing Atari games, the authors of MoGym show that applying DQN on some QVBS benchmarks is unsuccessful [8]. This research aims to identify and address the challenges of such unsolvable benchmark instances. In addition, it aims to provide DRL algorithms which make use of the unique properties of MoGym and recent advancements in DRL literature to solve most QVBS benchmarks. This chapter provides a high level background of reinforcement learning, planning as well as quantitative verification and briefly states how these research fields are connected to this thesis. Then an outline of the research problem, the research aim, objectives and questions, the significance of this work as well as an outline of the thesis is given.

1.1 Research Background

In our world, we are often faced with sequential decision-making problems. When playing chess for example, we have to decide which move to make to win the game, given the current state of the chessboard. Depending on the problem's complexity, finding a good strategy to solve such decision problems can be very hard for us humans and it often takes time and practice to achieve good performance. Thus, sequential decision-making is a key challenge addressed by the artificial intelligence (AI) research community [12], which develops formal methods to solve such problems. In recent years, computer programs were developed that can reach superhuman level of gameplay in e.g. Chess and the game of Go, consistently beating the human grandmasters of the respective games [3]. Other areas include training AI programs to play computer games, most prominently Atari games, where similar techniques achieve great success [4]. More recently, the insights gained by solving games is applied in a more scientific context e.g. by solving the problem of protein folding [6] or speeding up matrix multiplication in software to make better use of the available hardware [7].

Due to starting from different assumptions, two AI communities have emerged which address sequential decision-making problems: the AI planning community and the reinforcement learning (RL) community [12, 13]. The most successful AI advances of the recent years, like the Alpha Zero AI beating the grandmasters of Go [3], are a product of combining planning and learning techniques [13]. Planning is a huge research area with many methods [14, 15]. This thesis focuses on state-space planning which includes methods to "[...] search through the state space for an optimal policy or an optimal path to a goal" [16]. The focus is on non-deterministic problems where the outcome of an action is paired with a probability. Such probabilistic problems can be modeled as Markov decision processes (MDPs), which are also the base for RL algorithms and thus enables the mixing of both techniques [16].

To test the performance of state-of-the-art planning algorithms, benchmarks have been developed which provide a fixed set of well-defined planning tasks. MoGym [8] is a framework which provides a Python interface resembling the OpenAI interface, the most common interface to train (D)RL algorithms. This allows applying reinforcement learning on planning benchmarks without rewriting each benchmark to be usable for (D)RL training, thus bringing both communities closer together. However, the planning benchmarks must be available in the JANI format [9] before they can be used with MoGym. The QVBS [10] is a benchmark set originally designed for quantitative verification but it includes many famous planning benchmarks in this format, e.g. the *PRISM* [17] benchmark set.

Quantitative model checking is concerned with the analysis of quantitative models on specified properties. A quantitative model checker, such as PRISM [17], can analyze MDPs which allows model checking of planning tasks. Statistical model checking is a form of quantitative model checking and can answer questions such as "What is the probability of reaching a goal state from a start state of the MDP?" [18]. This is interesting, as it allows reasoning about the underlying MDP on which an (D)RL agent trains on, especially since the MDP is readily available in the QVBS and not implicitly hidden in an environment written in Python or any other language. In addition, MoGym provides an interface for deep statistical model checking (DSMC), which allows checking "reach-avoid" properties of DRL agents and deep neural networks [19]. Thus, DSMC is used to evaluate the experiments of this thesis.

1.2 Research Problems and Questions

While the authors of MoGym could show that DQN [1], one of the most prominent RL algorithms, can be successfully trained on some instances of the QVBS, there are instances where the DQN agent is not able to learn at all [8]. The paper does not explore why this is the case, however it states that these agents where able to see the goal during training [8]. This is problematic because DQN can be successfully applied to complex problems such as playing Atari games [1]. Consequently, this could indicate that either planning tasks are especially hard to solve using DQN and other standard DRL algorithms or that loading the planning tasks using MoGym makes the problems harder than the original problems. Furthermore, it might also be the case that the DQN agent is not properly tuned to solve the benchmarks or that any possible combination of the problems is true. This thesis aims to find the reasons why some benchmarks are not learnable using DQN, by discovering challenges to overcome using MoGym on those benchmarks. Moreover, the thesis provides the reader with DRL agents and solutions to address the challenges by improving on existing approaches. We think that this is an important contribution to bridge the gap between planning and (D)RL further, demonstrating that combining ideas from both communities is worthwhile.

In particular, we think that the key challenges for DRL agents trained on planning benchmarks using MoGym are (1) the action space explosion, which results from converting a MDP to a RL environment, and (2) the lack of sophisticated exploration techniques. Compared to the original planning problems, the exploding action space for the DRL agents makes the tasks much more difficult to learn. It is well known that DQN struggles with large action-spaces [20]. The reasons for the action-space explosion are covered in Section 2.6. For this thesis Description-based Q-learning (DBQL) [21] is used as a DQN agent variant which is not affected by the action-space explosion problem. To test if the aforementioned challenges are the two main limitations for DRL on hard, unsolvable tasks using the DQN of MoGym, this thesis covers the following research questions:

Research Question 1.1 Is the action-space explosion the main factor for the unsolvability of some planning tasks?

Research Question 1.2 Is exploration a key factor when learning fails, even if the agent accounts for the action-space explosion?

This thesis also aims to provide the reader with DRL agents as a solid basis to learn other, here unaddressed, planning benchmarks using MoGym. For this purpose, the thesis introduces an adapted version of an exploration framework called Go-Explore [22] which drops constraints of the environment used and makes the approach applicable to a broader set of training environments. Thus, we hope to provide the reader with tools to apply this exploration technique to similar problems compared to the ones presented in this thesis. Another aim of the thesis is to improve on existing approaches to these problems. In particular, we try to improve on DBQL by adding n-step learning. Since this is the first attempt to n-step DBQL, three new n-step DBQL algorithms are proposed, one of which is based on existing DQN adaptions. N-step learning can lead to faster learning if tuned correctly [2, 16], and several n-step DQN implementations have shown that this can also be achieved with DRL [2, 23–26]. N-step learning is interesting in the scope of DBQL because this algorithm can make use of its knowledge of the environment's transition function while most DRL algorithms do not have this advantage. Typically, the underlying MDP is treated as a black box by the (D)RL agents and is not available during training. With MoGym, since it loads a description of an MDP to create an environment, the MDP is explicitly known. Thus, MoGym allows a (D)RL agent to see the outgoing transitions of a state. The n-step learning algorithms proposed abuse this feature to improve their learning capabilities. A disadvantage of this approach is that the proposed agents are reliant on environments which can provide the underlying transition function. The thesis answers the following research questions:

Research Question 2.1 Does n-step learning increase sample efficiency?

Research Question 2.2 Does n-step learning lead to faster learning with regard to training time?

By proposing three new n-step DBQL agents the effect of planning ahead using real-world data in form of the MDP's transition function can be tested. While model-based reinforcement learning has similar approaches already by trying to learn the underlying MDP, the access to the actual transition function is such a unique property that these algorithms are a unique mixing between planning and reinforcement learning which is not broadly studied. Due to this, we think it is worthwhile to test this approach by testing the possible advantage of sample efficiency increase while also looking at the potential disadvantage of high complexity and training time.

1.3 Limitations and Thesis Structure

Since the research questions are very broad, in the scope of this thesis, certain limitations are necessary. For the exploration experiments, we limit experiments to three hard benchmarks from the QVBS and use them to generalize our findings. Lots of exploration techniques have been developed in recent years, and it is impossible to test all of them. The thesis limits itself to a few techniques that we think cover the main ideas in the field. For the n-step experiments, we limit the number of n-steps trained to two simple benchmarks to save time. Furthermore, since DRL algorithms are very sensitive to the hyperparameters used, it is not possible to find the best hyperparameters for each agent on each benchmark. However, hyperparameter tuning is used to at least choose some good configurations. The results of the experiments are also dependent on randomness. To account for this, we set three different seeds for all random generators used to make the results reproducible.

The thesis is structured as follows: First the theoretical background is covered which includes a brief introduction to reinforcement learning. This is followed by an introduction of Deep Q-learning (DQN) because DBQL, on which we improve, can be seen as an extension of DQN. Afterwards a distinction of two update types, which can be used by DBQL, is outlined before introducing DBQL and the concept of n-step learning. The chapter ends with an outline of causes of the action-space explosion problem when porting QVBS instances to an environment using MoGym and a brief introduction of deep statistical model checking (DSMC), which is used to evaluate the experiments. After the theory is covered, the thesis is positioned in related work in the literature review chapter. In this chapter related work with regard to n-step learning is introduced and the all necessary exploration techniques used for the set of second research questions are outlined. The fourth chapter introduces three DBQL agents using n-step learning for the first set of research questions while the fifth chapter describes the exploration technique Go-Explore and introduces and contrasts the new Stochastic Go-Explore variant developed for this thesis. The sixth chapter describes in detail which experiments are made, why these experiment designs were chosen and how their findings can be used to answer the research questions. Chapter seven discusses the experiment results. Finally, the last chapter draws a final conclusion and provides an outlook on future work.

Chapter 2
Theoretical Background

2.1 Reinforcement Learning

In reinforcement learning a decision-making entity, typically called agent, interacts with an environment, which represents everything outside of the agent. The agent continuously observes the current state of the environment, decides which action to take and then observes a numerical reward based on the action and the next state of the environment. This interaction is done in discrete time steps [16]. The reinforcement learning process is summarized in Figure 2.1, which is an illustration from Barto and Sutton [16].

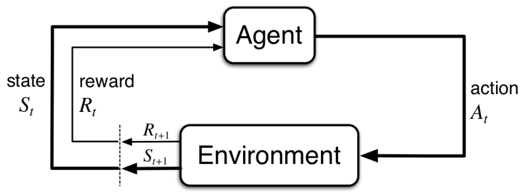


Figure 2.1: Diagram depicting the interaction between a RL agent and the environment as shown in the book of Barto and Sutton [16].

The interaction process of an agent with an environment is formally modeled as a finite Markov decision process (MDP).

Definition 1. *Let $D(S)$ denote the set of probability distributions over a non-empty set S. A Markov decision process (MDP) is a tuple $M = \langle S, A, R, T, \mu \rangle$ consisting of a finite set of states S, a fine set of actions A, a partial transition probability function $T : S \times A \rightharpoonup D(S)$, a reward function $R : S \times A \times S \to \mathbb{R}$ and an initial distribution $\mu \in D(S)$ [21, 27].*

In some environments, an agent cannot perform every action in every state. Actions that can be taken in a state are called *applicable*.

Definition 2. *An action $a \in A$ is applicable in a state $s \in S$ if the transition function $T(s, a)$ is defined. $Act(s) \subseteq A$ denotes the set of applicable actions in a state s [21].*

To act in an environment, RL agents have to decide which action to take given an observation of the environment. A policy resolves this decision-making problem by stating the probability of the agent taking an action in a state. We say that an agent follows a policy.

Definition 3. *A policy $\pi : S \times A \to [0, 1]$ is a probability distribution over all actions $a \in A$ given a state $s \in S$, denoted $\pi(a|s)$ [16].*

This notation already hints that a policy is only dependent on the current state and not on any previously seen states. The agent's goal is to learn a policy which maximizes the overall expected (discounted) return. This policy is called the *optimal policy*, denoted π_*. The reward function must thus be chosen to encourage the agent to learn the desired behavior.

The discounted return is the accumulated, discounted reward an agent receives when interacting with an environment, starting in the current time step t until the end of an episode T. The discount factor $\gamma \in [0, 1]$ ensures that the discounted return has a finite value given that $\gamma < 1$ and that the reward function is bounded [16].

Definition 4. *Let G_t denote the discounted return an agent receives starting in time step t. Let $\gamma \in [0, 1]$ be the discount factor and R_t be the reward received at time step t. Then the discounted return is defined such that $G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ [16]. Note that T denotes the total number of time steps needed to end an episode. In some cases the agent is limited to a maximum of allowed time steps during training while in other cases an episode is infinite such that $T = \infty$.*

RL agents typically use value functions which estimate how good it is for an agent either to be in a state or to take an action in a state with regard to maximizing the expected return [16]. There are two kinds of value functions, the *state-value function* and the *action-value function*. The state-value function $v_\pi(s)$ represents the expected return if the agent follows a policy π when starting in a state $s \in S$ at time t.

Definition 5. *Let $s \in S$ be a state and $t \in \mathbb{N}$ be the current time step of the current episode. Then the state-value function $v : S \to \mathbb{R}$ is defined such that $v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$ [16].*

The action-value function $q_\pi(s, a)$ describes the expected return when taking an action in a state under the assumption that the agent follows policy π afterwards.

Definition 6. *Let $s \in S$ be a state, $a \in A(s)$ be an applicable action of s and $t \in \mathbb{N}$ be the current time step of the current episode. The action-value function $Q : S \times A \to \mathbb{R}$ is defined such that $q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$ [16].*

Value functions satisfy a recursive relationship, a fact which is used to obtain good estimates of value functions using dynamic programming. This recursive relationship is often expressed using the Bellman equations [16]. The idea is to split the return G_t into the rewards R_t received for a transition from the current state and the return G_{t+1} of the successor state of the transition. This is a direct result from Definition 4.

Definition 7. *Let π be a policy such that $\pi(a|s)$ is the probability of choosing action a in state s. Further, let $T(s, a)(s')$ be the probability of transitioning to state s' when starting in state s and taking an action a and r be the reward associated with this transition in the underlying MDP. Then*

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\ &= \sum_{a \in Act(s)} \pi(a|s) \sum_{s' \in S} T(s, a)(s')[r + \gamma v_\pi(s')] \end{aligned}$$

recursively defines the state-value function for policy π. The equation is called the Bellman equation for v_π [16].

Definition 8. *Similar to Definition 7,*

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \\ &= \sum_{s' \in S} T(s, a)(s')[r + \gamma\, \pi(a'|s') \sum_{a' \in Act(s')} q_\pi(s', a')] \end{aligned}$$

recursively defines the action-value function for a policy π. This equation is called the Bellman equation for q_π [16].

Notice that these equations describe the respective value function when following any policy π. Since the agent tries to find the optimal policy, the associated value function is especially important as finding the optimal policy can be solved by finding the optimal value function. The Bellman equations for the optimal policy are called Bellman optimality equations.

Definition 9. *The Bellman optimality equation for the state-value function*

$$\begin{aligned} v_*(s) &= \max_a \mathbb{E}_{\pi_*}[G_t|S_t = s, A_t = a] \\ &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1})|S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} T(s,a)(s')[r + \gamma v_*(s')] \end{aligned}$$

can be derived using the definition for the state-value function (see Definition 5) where the discounted return is defined recursively using the optimal state-value function [16].

Definition 10. *Similar to Definition 9,*

$$\begin{aligned} q_*(s,a) &= \mathbb{E}[R_{t+1} + \gamma \max_{a' \in Act(s')} q_*(S_{t+1}, a')|S_t = s, A_t = a] \\ &= \sum_{s' \in S} T(s,a)(s')[r + \gamma \max_{a' \in Act(s')} q_*(s', a')] \end{aligned}$$

defines the Bellman optimality equation for q_* [16].

The main difference in these equations is that the policy probability terms $\pi(s, a)$ are replaced by a maximization, as the optimal policy is the one that maximizes the expected return.

2.2 Deep Q-learning

One of the most well-known (D)RL algorithms is the DQN algorithm, which is based on the tabular Q-learning algorithm. To understand the differences of Description-based Q-learning and DQN, the mathematical intuition of the latter is briefly outlined.

Recall that the goal of an agent is to find the optimal policy π_* which maximizes the expected return. The action-value function for the optimal policy, called the *optimal action-value function*, is denoted q_*. Following Definition 6, the equation

$$q_*(s,a) = \mathbb{E}_{\pi_*}[G_t|S_t = s, A_t = a]$$

expresses the optimal action-value function mathematically.

If the agent knows $q_*(s, a)$ for all state-action pairs, following the optimal policy is easy, as it suffices to take the action with highest action-value in every state. This ensures that the agent always chooses actions with the highest expected return.

If the agent can learn q_* it thus has learned π_*. This is the main idea of Q-learning, which starts with a random action-value function q and continuously improves its estimates of the action-values to eventually match q_*. The Bellman optimality equation for action-values q_* (Definition 10) is used to improve the value estimates.

As the agent interacts with the world, it gains information about the current state $s \in S$, the state of the next time step $s' \in S$ after taking an action $a \in Act(s)$ and the reward $r \in R$.

This n-tuple will be referred to as *trajectory* (s, a, r, s') from now on. With each trajectory, a temporal difference (TD) target

$$y = r + \gamma \max_{a' \in Act(s')} q_\pi(s', a')$$

can be constructed which combines an actual reward r and an estimation of the future action-value. By including the actual reward, the target is grounded in reality and thus the target is used to update our current estimate $q_\pi(s, a)$.

In Deep Q-learning, the action-value function is represented as neural network (NN) with parameters θ. The notation $Q_\theta(s, a)$ indicates that this is an action-value estimated by the NN. Updating the current q-value estimate towards the target is the same as optimizing the Mean Squared Error (MSE) between the two [1]. This is done by minimizing the loss function

$$L(\theta_i) = \mathbb{E}_{\theta_i}[((r + \gamma \max_{a'} Q_\theta(s', a')) - Q_\theta(s, a))^2]$$

using stochastic gradient descent.

This loss function is updated in every iteration i where θ_i denotes the weights of the NN in the i-th iteration. Note that the loss function is an expectation. Looking at the Bellman equation of Definition 10, it would be possible to directly compute this expectation if the agent had knowledge of the transition function T and the reward function R. Such updates are called *expected updates* [16]. DQN does not assume knowledge of the underlying MDP, hence it is a model-free RL algorithm which treats the environment as a black box. Updating based on a sample successor state instead of using all outgoing transitions is considered a *sample update* [16] (see Section 2.3 for a more detailed discussion). To optimize this expectation, the agent has to ensure that it sees and updates all state-action pairs infinitely often [16]. In other words, the agent has to keep exploring instead of just following the best policy it has currently found.

Estimating the action-value function Q (or value functions in general) with non-linear approximators like deep neural networks can make learning unstable. In some cases, the estimated value function even diverges [1]. The DQN paper by Mnih et al. [1] states that this is due to the agent learning on correlated observations and the fact that even small changes of the value function can significantly change the policy. By changing the policy, the agent obtains different observations and thus the data distribution has also changed significantly. One common measure to mitigate these problems is the use of experience replay, where the agent stores each experienced trajectory (the current observation, the action taken, the reward observed and the next observation seen) into a replay buffer. The neural network is trained by sampling randomly for the buffer. If the buffer is large, this reduces correlation between the sampled trajectories. Another problem is that the computed target values and the corresponding value estimate, which is to be updated, are also correlated. TD target values bootstrap using q-value estimates of possible future observations. Updating q-value estimates using a deep neural network can also change the estimates for similar observations, thus the target values do not remain fixed which can cause divergence. Mnih et al. use a different network to estimate q-values for the target computation called a target network [1]. This network copies the parameters of the learned neural network every few steps and keeps the parameters fixed in-between. Instead of copying the parameters every few steps, it is also possible to update the parameters of the target network in every step slightly in the direction of the parameters of the learned value network [20]. Let $\tau \in (0, 1)$ be the update factor for the target network, θ be the parameters of the learned network and θ' be the parameters of the target network. In every step the formula $\theta' = \tau\theta + (1 - \tau)\theta'$ is used to update the target network [20].

2.3 Expected vs Sample Updates

RL algorithms, which estimate a value function, need to update their estimates based on experiences gathered by interacting with an environment to improve their current policy and eventually learn an optimal policy. This thesis is focusing on Temporal Difference (TD) Learning, where estimates are used in combination with gathered experience (rewards) to compute a new estimate of the value function, called a TD target. The target is calculated by computing the Bellman optimality equation. Recall for example the Bellman optimality equation for q_* from Definition 10

$$\begin{aligned} q_*(s,a) &= \mathbb{E}_{\pi_*}[G_t | S_t = s, A_t = a] \\ &= \sum_{s' \in S} T(s,a)(s')[r + \gamma \max_{a' \in Act(s')} q_*(s',a')] \end{aligned}$$

which recursively defines the expectation over all returns when following policy π_*. Instead of unrolling this equation completely, TD learning makes use of bootstrapping. The idea is that the value function $q_*(s',a')$ is substituted with the agents current estimate of the value at some point of the computation. In the simplest case, the equation is not unrolled and $q_*(s',a')$ is estimated immediately. Note that bootstrapping has the benefit of computing targets without playing out full episodes, but it also introduces estimation error. The TD target does not replace the current estimate. Instead, the current estimate is corrected in the direction of the target. If a neural network is used as estimator, this is often done by minimizing the MSE between target and current estimate.

Multiple ways of computing TD targets exist which are based on different assumptions. Assume an agent wants to update its state-value estimate of state s. One update idea is to directly apply the Bellman equation of the state-value function (Definition 9). To compute the expectation, the agent must require knowledge of the transition function and the reward function. (Technically, we can only compute the expectation if we know v_*. In the following we say that we compute the expectation to say that we calculate the sum over all transitions using estimates for v_*). Updates, which directly compute the expectation over all possible next states are called *expected updates* [16]. In RL, the environment is often assumed to be a black box and thus the underlying MDP is assumed to be unknown. In this case, *sample updates* [16] can be used.

Sample updates consider only a single successor state per state. Instead of summing over all possible next states, the agent takes one action per state and uses the seen rewards along the way to compute an update [16]. In other words, sample updates use a single transition path instead of all outgoing transition paths for the target computation. Due to not knowing the environments transition function, sampling a single trajectory by interacting with the environment is often the only possibility. If all trajectories are sampled infinitely often, the agent can still compute a good estimate of the expectation and thus of the optimal value. Figure 2.2 illustrates the difference of both updates. Sample updates are also used by the DQN algorithm which uses only the seen successor state and the seen reward after taking an action to update a state. While expected updates suffer from estimation error due to bootstrapping, sample updates in addition also suffer from a sampling error [16].

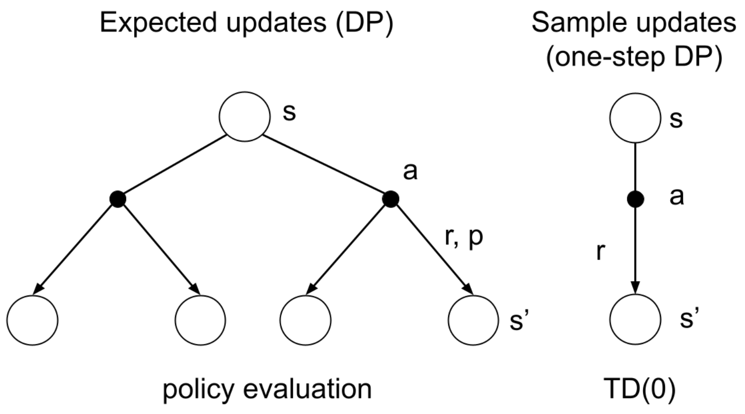


Figure 2.2: Difference between expected update and sample update to compute a target for $v_\pi(s)$ [16].

2.4 Description-based Q-learning

Description-based Q-learning (DBQL) as introduced by Gros et al. [21], learns the state-value function $V(s)$ instead of the action-value function $Q(s, a)$. This makes the agent's learning less dependent on the size of the action space, as fewer values need to be estimated. The state-value is then used to compute the q-values for all applicable actions in the state. From these the policy is reconstructed again by always choosing the action with the highest q-value (see Section 2.2). This requires knowledge of the transition function T, an unusual case in a RL setting but not in the planning domain. One common approach between planning and RL is dynamic programming (DP), where state-value estimates are updated with expected updates by constantly iterating over the complete state-space. This is a core difference to RL agents, which interact with the environment to generate trajectories, which are then used to update the value function estimates. *Real-Time Dynamic Programming* (RTDP) is an algorithm which performs expected updates like DP but uses trajectories sampled by interacting with the environment [16, 28]. This makes RTDP an RL algorithm which makes use of the know transition relation, in contrast to e.g. DQN. The Description-based Q-learning paper [21] introduces a novel version of RTDP, which can use non-linear function approximators, like neural networks, for the value function. This is achieved by using the same ideas which have already enabled Q-learning to be a deep learning algorithm.

The optimal action-value function

$$q_*(s, a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1})|S_t = s, A_t = a]$$

can be formally expressed with the optimal state-value function [16]. The optimal state-value function

$$v_*(s) = \mathbb{E}_{\pi_*}[G_t|S_t = s]$$

is defined analog to the optimal action-value function and follows directly from the value-function definition [16]. Just like DQN, this algorithm also bootstraps using TD learning. Recall that DQN uses sample updates to compute a target since this does not require knowledge of the transition function or a resettable environment on which simulations can be run to get this knowledge. The same TD method can be used to update the state-value function. In its simplest form,

$$y_\theta(s) = r + \gamma V_{\theta'}(s') - V_\theta(s)$$

defines the TD target. The adapted loss function

$$L(\theta_i) = \mathbb{E}_{\theta_i}[((r + \gamma V_\theta(s')) - V_\theta(s))^2],$$

which is used to optimize in every iteration, remains the MSE of the target $y_\theta(s)$ and current estimate of the value function $v_\theta(s)$.

If the agent has access to the true transitions in the MDP of the environment, an expected update is possible. Recall from Section 2.3 that an expected update computes the expectation of the used Bellman equation instead of using a single successor state as a sample of the expectation. A TD-target based on this equation will be referred to as expected target. If this target is used to compute the MSE-Loss, the agent optimizes for the *transition-based loss function* [21]

$$L(\theta_i) = \mathbb{E}_{\theta_i}[((\max_{a' \in Act(s')} \sum_{s' \in S, r} T(s,a)(s')(r + \gamma V_\theta(s'))) - V_\theta(s))^2]$$

in each iteration. The full Description-based Q-learning algorithm shown in Algorithm 1 highlights the differences between both approaches in different colors. If DBQL uses sample updates to compute the TD-target, the algorithm will be denoted as DBQL$_S$ to distinguish both ideas. While DBQL$_S$ stores trajectories in the replay buffer which the agent has actually seen, the agent using expected updates needs to store all possible transitions from the states it visits (line 14). Apart from that, the main difference between DBQL and DBQL$_S$ remains the computation of the TD target. The code specific to DBQL$_S$ is highlighted in blue and is replaced with the code highlighted in orange for the full DBQL version with expected updates.

Algorithm 1 Description-based Q-learning

```
 1: Initialize replay buffer $D$
 2: Initialize value network $V_\theta$ and target network $V_{\theta'}$ where $\theta$ and $\theta'$ are the network parameters
 3:
 4: **for** episodes i=0 to M-1 **do**
 5:     Reset environment to start state $s$
 6:     **for** step t=0 to T-1 **do**
 7:         **with** probability $\epsilon$:
 8:             Select random action $a_t \in A(s_t)$
 9:         **else with** probability $1 - \epsilon$:
10:             Compute $Q_\theta(s', a)$ using Equation 2.4
11:             Select $a_t = \operatorname*{argmax}_{a \in A(s_t)} Q_\theta(s, a)$
12:         **end with**
13:         Execute $a_t$, observe $r_{t+1}$ and $s_{t+1}$
14:         Store $\begin{cases} (s_t, a_t, r_{t+1}, s_{t+1}) \\ (s_t, a_t, T(s_t)) \end{cases}$ in replay buffer $D$
15:         **every** $C$ steps **do**
16:             Sample a minibatch of samples $(s_j, a_j, r_{j+1}, s_{j+1})$ from $D$
17:             Set target $y_i = \begin{cases} r_{j+1}, & s_{j+1} \text{terminal state} \\ r_{j+1} + \gamma \cdot V_{\theta'}(s_{j+1}) - V_\theta(s_j), & \text{otherwise} \end{cases}$
18:             Set target $y_i = \begin{cases} r_{j+1}, & s_{j+1} \text{terminal state} \\ \max_{a'} \sum_{s',r} T(s,a)(s')[r + \gamma V_{\theta'}(s')], & \text{otherwise} \end{cases}$
19:             Optimize using gradient descent on loss $(y_j - V_\theta(s_j))^2$
20:             Update target network weights $\theta' = (1 - \tau) \cdot \theta + \tau \cdot \theta'$
21:         **end every**
```

2.5 N-step Learning

N-step learning modifies an existing algorithm based on value-function estimation by changing the update function to use multiple environment rewards before bootstrapping of value function estimates [16]. Due to the recursive definition of the value function in the form of the Bellman equations, the latter can be unrolled multiple times. If the update considers only the immediate successors of a state, thus looking only one step into the future to learn from rewards before bootstrapping, this is considered a 1-step TD learning update, sometimes also abbreviated as TD(0) [16]. Updates do not need to rely on bootstrapping. An agent can play out or simulate a whole episode and use the real return of the finished episode for the update. Methods that do this are called *Monte Carlo* (MC) methods. MC methods need to play out full episodes, which is more time-consuming than looking only one time step into the future. It also introduces higher variance, as there can be a lot of episodes which start in a state and the episodes can have very different returns. N-step learning is a compromise between 1-step TD methods and MC methods. The idea of n-step learning is to look only n time steps into the future and gather multiple rewards for the update computation before relying on a value estimate.

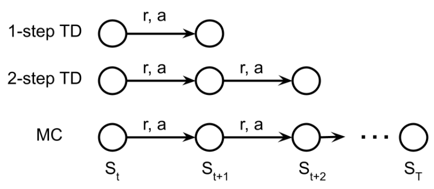


Figure 2.3: Trajectories used for sample updates. For n-step TD updates, the trajectories end after n steps, even if the state is not terminal. Monte Carlo updates use full trajectories until an episode ends.

Recall from Section 2.3 that TD updates compute a TD target estimate using the Bellman equation of the associated value function. In 1-step TD learning, the update bootstraps by substituting the value function for the successor states after one transition with estimates. In 2-step learning, the Bellman equation is unrolled one time, such that estimates are used after two rewards are collected on each outgoing path from the current state. A n-step return unrolls the Equation $n-1$ times, such that bootstrapping occurs after collecting n future rewards. This is illustrated in Figure 2.3 for sample updates. In case of Q-learning, the Bellman optimality equation

$$\begin{aligned} q_*(s,a) &= \mathbb{E}_{\pi_*}[G_t|S_t = s, A_t = a] \\ &= \sum_{s' \in S} T(s,a)(s')[r + \gamma \max_{a' \in Act(s')} q_*(s',a')] \end{aligned}$$

is utilized. If sample updates are used, the agent samples the return G_t from the expectation and uses it as the TD target. In 1-step TD learning, G_t is approximated with the formula

$$G_t^1 = R_{t+1} + \gamma \max_{a \in Act(S_{t+1})} q_*(S_{t+1}, a),$$

where every appearing q-value is substituted using estimates.

The n-step TD target approximates G_t, which is based on n returns, before bootstrapping. The n-step sample return for Q-learning and Description-based Q-learning is the following:

Definition 11. *[16] Let R_t denote the reward received by the agent at timestep t and let $\gamma \in [0,1]$ be the discount factor. The n-step sample return for Q-learning is defined such that*

$$G_t^n = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n \max_{a \in Act(S_{t+n})} q_*(S_{t+n}, a).$$

Definition 12. *[16] Let R_t denote the reward received by the agent at timestep t and let $\gamma \in [0,1]$ be the discount factor. The n-step sample return for Description-based Q-learning is defined such that*

$$G_t^n = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n v_*(S_{t+n}).$$

Expected updates do not sample returns from the expectation over the returns when following a policy. Instead, the target directly computes the expectation. The update principle remains the same: unrolling the Bellman equation multiple times before bootstrapping. Hence, the n-step target is the associated Bellman equation unrolled n-1 times. For a 2-step update in Transition Learning, the target computation

$$\begin{aligned} v_*(s) &= \max_{a \in Act(s)} \sum_{s',r} T(s,a)(s')[r + \gamma v_*(s')] \\ &= \max_{a \in Act(s)} \sum_{s',r} T(s,a)(s')[r + \gamma \max_{a' \in Act(s')} \sum_{s'',r} T(s',a')(s'')[r + \gamma v_*(s'')]] \end{aligned}$$

approximates the Bellman optimality equation for v_* (Definition 9) unrolled once. It cannot equal the equation as the true optimal value function is unknown.

N-step learning for sample updates requires the agent to keep track of n states and the transition rewards during the interaction with the environment. With expected updates, the agent needs to keep track of the whole tree of all possible successor states and rewards for n-steps in the future before being able to perform an update. With each additional step, a n-step expected update is getting exponentially more expensive. The parameter n has to be chosen such that the additional computational overhead is not having a negative effect on the learning time of an agent. Compromise between full expected n-step updates and n-step sample updates, like e.g. proposed by the n-step tree backup algorithm [16], can ease this problem.

To give a complete picture, it is worth noting that this thesis focuses only on n-step learning which looks into the future to perform an update. Based on the same idea of n-step learning, it is also possible to look at the states visited in the past during interaction with the environment and update them based on the rewards seen. This can be done with eligibility traces [16], a concept that is not introduced here because this *backward view* works well with sample updates only. For expected updates which use an expectation over the full state distribution of the successor states, looking only at a single traces of states visited in the past by taking single actions is not useful.

2.6 Action-space Explosion

This section outlines the reasons why the action space might explode when porting a JANI MDP from e.g. the Quantitative Verification Benchmark Set (QVBS) to an RL environment using MoGym. The problem is best explained with an example. Assume a (D)RL agent controls a robot walking in a simple gridworld, where each cell corresponds to a state of the environment and the underlying MDP respectively [16]. In each cell, the agent can move **north, south,**

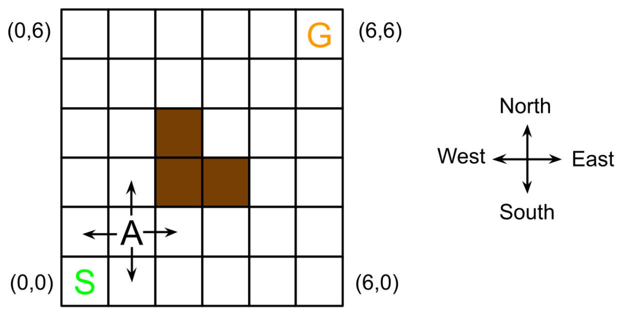


Figure 2.4: A simple gridworld where the agent starts in the cell with a green S and has to reach the goal cell marked with an orange G. There are some walls on the map, marked in brown, to make the problem more interesting.

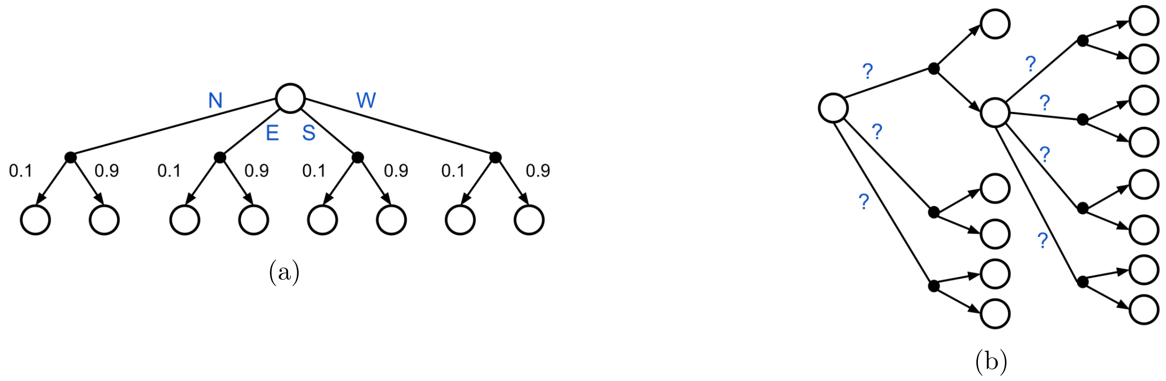


Figure 2.5: (a) MDP schematic of the gridworld described in Section 2.6. In each state four actions (north, east, south, west) can be chosen. (b) MDP without action labels where it is not possible to match transitions to the general actions of the problem.

east and **west** to enter an adjacent cell. This makes a total of four general actions. The agent's goal is to reach a specified cell, the goal cell. To add stochastic behavior, there is a small chance that an action can fail, e.g. that the agent decides to go north but stays in the cell with 10% probability. See Figure 2.4 for a visualization of an example gridworld environment.

A schematic of the MDP describing the gridworld environment is depicted in Figure 2.5a, which shows one state of the MDP and its outgoing transitions to the immediate successor states. There are only four actions in each state. We know that there are only four actions as described in the previous paragraph, so each transition can be labeled with the corresponding action. Now suppose the MDP does not provide the action labels for each transition. In this case, there is no way to know whether a transition of state $s_1 \in S$ corresponds to taking the same action of a transition of another state $s_2 \in S$. It is also possible, that the agent cannot perform the same set of actions in each state. Suppose the agent is at the left bottom corner of the gridworld, which is surrounded with walls. Then the agent cannot take action *west* and *south*. But potentially on this state, it can also take a unique action *mark* to leave a sign in the current cell. The resulting MDP without labels (see Figure 2.5b) has lost all of this information described because the transitions cannot be mapped back to the general, applicable actions.

The JANI format specifies that information about the action of an edge in the MDP is optional. It is also optional to provide all general actions available in the whole MDP [29]. If these optional cases are missing, the action space explodes to the number of transitions in the MDP as each edge is considered a unique action. In the gridworld example, given that the gridworld has

$6 \cdot 6 = 36$ cells and four general actions, the MoGym environment interprets this as $36 \cdot 4 = 144$ unique actions if the necessary action information in the MDP is not provided.

2.7 Deep Statistical Model Checking

Throughout this thesis deep statistical model checking (DSMC) [19] is used to evaluate learned policies of DRL agents. Statistical model checking (SMC) is a version of probabilistic model checking (PMC), both of which can determine the probability with which an MDP satisfies a property considering all possible action policies [19, 30]. While PMC uses exact methods to infer the probabilities, SMC is based on Monte Carlo simulation and hypothesis testing and provides an estimation of the probability together with some statement of the error, e.g. by a predefined error-bound or by providing a confidence interval [18, 30]. Statistical model checking has the advantage over PMC that its memory consumption is not dependent on the state-space of the MDP. However, the amount of samples needed for the simulation to fulfill some error bound might explode for some rare events of a property [18, 19].

Deep statistical model checking, in contrast to statistical model checking in general, calculates probabilities for properties in a MDP given a single policy instead of reasoning over all policies. The policy used is neural network which resolves the non-determinism of the MDP by choosing an action in a state, thus inducing a Markov chain on which stochastic model checking is performed [19]. In the scope of deep reinforcement learning, the agent uses its learned policy network for decision-making. DSMC allows answering questions like “What is the probability of reaching a goal state?”. Such probabilities can be used to compare policies with each other. Note that DSMC (and PMC in general) is also capable of estimating expectations instead of probabilities [19], such that it also enables answering questions like “What is the expected return when following this policy?”.

Chapter 3
Literature Review

3.1 N-step Learning

Reinforcement learning agents need to be trained on lots of data before their policy converges to an optimal policy. Faster learning can be achieved if agents can build more accurate value function estimates to update their policy [24]. This is why n-step returns are very popular, especially in tabular settings. In DRL, where computing value estimates through neural networks is computationally expensive compared to table lookups and an experience replay buffer is often needed for off-policy learning, n-step methods can be less efficient if not designed carefully [25]. If training time is not crucial, n-step returns still play an important role in DRL. A very prominent example is AlphaZero [3], which uses Monte Carlo tree search (MCTS) as an approximation technique for n-step returns to achieve remarkable results in e.g. the game of Go.

On environments where simpler algorithms like DQN are able to successfully learn a good policy in much less time than e.g. AlphaZero, it is crucial that the disadvantage of longer computation times for n-step value estimates does not outweigh the advantage of faster convergence. Thus a careful implementation of n-step methods, e.g. for DQN, and the choice of the parameter n is important.

Replay-based methods are problematic for n-step return computation since sampling random mini-batches does not allow re-using computations from previous steps. Daley and Amato [25] propose an adapted replay memory which stores the n-step return, in this case the λ-return, together with the corresponding transitions. The return is computed recursively while the agent interacts with the environment. They also propose dropping the need for a target network by partitioning their replay buffer into blocks which can be efficiently updated using the non-target value network. A similar idea is used in RainbowDQN [2], a DQN variant which also uses an adapted replay buffer for n-step learning. Their replay buffer has a build in cache of size n, which is used to automatically compute the n-step return using recursion. The return, together with the state, the action taken and the final state after n steps is then stored in the buffer once the return is computed internally. This moves the task of keeping track of the last n rewards and the computation of the n-step reward from the agent to the buffer. In addition, the RainbowDQN agent also keeps a 1-step replay buffer from which it samples. These samples together with the corresponding samples in the n-step buffer are used for learning. This is done to reduce variance as there is no off-policy correction mechanism. The RainbowDQN buffer will be tested with the sample-update based Description-based Q-learning algorithm. Both of the given solutions require an online algorithm based on sample updates like DQN.

Since MoGym provides the agent information of the complete transition function, computing n-step returns with expected updates is the main focus of n-step learning in this thesis. Because knowledge of the full transition function is not common in (D)RL, to the best of our knowledge, there is not much related work which tries to make computation of such expected n-step returns efficient with replay buffers in DRL literature. In the tabular case however, multi-step updates for Real-Time Dynamic Programming (RTDP) has been investigated e.g. by Efroni et al. [31]. Due to this, the two approaches proposed for Description-based Q-learning using n-step expected updates are not based on any prior work.

3.2 Exploration

RL methods, which rely on value estimates, need to ensure sufficient exploration. It is important to see all states or state-action pairs to have accurate value estimates which can then be used by the agent to act greedily and thus follow an optimal policy. Deciding when to act according to the currently best known policy and deciding when to explore states and actions, which are believed to be worse, is called the *exploration-exploitation dilemma* [16]. A simple way to solve this problem for agents estimating any kind of value function is acting ϵ-greedy [16]: with a small probability $\epsilon \in (0, 1)$ the agent chooses a random action in a state and with probability $1 - \epsilon$ the agent greedily chooses the action with the highest action-value. A variant of this simple approach was also used in the MoGym paper [8], where the agent is incentivized to explore a lot at the start of the training and gradually rely more and more on the policy it learns as the training progresses. To do this, the epsilon parameter starts of with high and is decayed in every step until it reaches a lower threshold. Thus, this is referred to as ϵ-greedy decay.

Providing an intelligent exploration mechanism is especially important in environments where rewards are sparse. RL agents perform poorly when a large sequence of actions has to be taken before a non-zero reward is provided [22]. Simple heuristics like ϵ-greedy are inadequate in sparse reward settings [32], especially when the action space is large [20]. In this thesis, for the second research question, more sophisticated exploration techniques are tested on the hard action-space problems of the QVBS benchmark. Additionally, a new approach is introduced and tested based on ideas of recent, successful exploration techniques. The aim is to enable learning on benchmarks which were impossible to learn in MoGym because the agent hypothetically did not see goal states often enough to learn a good policy.

3.2.1 Count-based Exploration

One intuitive idea to ensure that every state is explored is to keep track how often a state, or state-action pair, was visited already. This information is used to decide on an action either directly or by altering the reward in favor of less visited states. This method is often referred to as *Count-based Exploration*. In tabular RL, where a value function is represented by storing the value for each state or state-action pair in a table-like data structure, this approach is easy to implement. However, explicitly storing value functions makes tabular RL infeasible to use on problems with large state spaces as such RL agents lack the ability to generalize learned values to similar, unseen states [1, 16]. In DRL, where value functions are represented by deep, non-linear function approximators like neural networks, generalizing to non-tabular visit counts is not straight forward. Furthermore, the effectiveness of count-based methods in high-dimensional state spaces is questionable since most states are typically not visited more than once [33, 34].

Nevertheless, several methods have emerged for DRL which are based on the idea of count-based exploration and achieve good results in sparse reward problems like Montezuma Revenge, a popular problem from the Atari games benchmark which is often used to compare exploration techniques [33]. Bellemare et al. introduced *pseudo-counts*, which are generalized visit counts derived using a density model over the state space [34] and thus circumvent the need to store visit counts for each state. While their implementation with DQN, using a density model called CTS [35], has shown very promising results in Montezuma's Revenge, this method requires choosing an appropriate density model for each environment. Furthermore, implementing a density model is a challenging task. Other exploration techniques have shown similar or even better results, generalize easier across different problem domains and are easier to implement. Thus, pseudo-counts will not be used in this thesis.

Another count-based approach introduced by Tang et al. [33] has shown to be successful in continuous action spaces and is based on hashing. The idea is to keep track of visit counts on state-hashes. Through hashing, the large state space is partitioned into a feasible small space which enables the usage of tabular visit counts again. Similar to pseudo-counts this technique is difficult to generalize as a suitable hash-function has to be chosen for each environment.

3.2.2 Intrinsic Motivation

Another popular method to tackle exploration in sparse reward settings is through intrinsic motivation. The agent tries to maximize rewards to complete a task, which is defined through the specific reward structure of the environment. If the environment rewards are sparse the agent is exploring aimlessly for long sequences of actions until it receives useful feedback. The idea with intrinsic motivation is that an agent should not only optimize for the environment task, but it should also have its own goals which it can additionally pursue to explore the environment. This is done by letting an agent calculate an additional *intrinsic reward* based on its own goals, which it also seeks to maximize. In theory, the agent then optimizes the sum of both intrinsic and environment reward: $r = r_{env} + r_i$. Achiam and Sastry [32] describe three of the most used intrinsic motivation goals:

- *Empowerment*: An agent enjoys the level of control it has about its future.
- *Surprise*: An agent is excited to see outcomes that run contrary to its understanding of the world.
- *Novelty*: An agent is excited to see new states.

Note that novelty and surprise are very similar intrinsic motivations. For this thesis, curiosity-driven exploration as introduced by Pathak et al. [27] will be used and implemented as a reference implementation using intrinsic motivation. The idea is similar to the surprise motivation described by Achiam and Sastry. Two neural networks are used to compute the curiosity reward. The first NN predicts the next state given the current state and the chosen action by the agent while the second NN predicts the action taken by the agent given the state prior to the action and the state the agent has seen after taking the action. By combining the prediction error of both NNs, the agent receives a bigger positive reward the higher the error of the predictions is. Both NNs are updated constantly during training which ensures that predictions for states which have often been explored get better over time. This method to compute the intrinsic reward is abstracted into a structure called *Intrinsic Curiosity Module* (ICM).

Similar to count-based methods, surprise-based intrinsic reward, e.g. using ICM, enables the agent to have a measure quantifying which states have not been explored as often. This is similar to states with a low visit count but with the advantage that storing separate data for each state is not required. Furthermore, the use of NNs enables generalization to similar states. An ICM module is independent of the underlying environment which makes it a good alternative solution to count-based methods. However, a benchmark study from Bellemare et al. [36] has shown that *Random Network Distillation* (RND) [37], another intrinsic reward technique based on novelty, performs even better on the notorious Montezuma's Revenge environment. The idea is that the intrinsic reward is the loss between a randomly initialized target network $t : S \to \mathbb{R}^k$ and a predictor network $p : S \to \mathbb{R}^k$. The predictor network is trained on the expected *mean squared error* (MSE) between the output of the target network and the predictor network itself. Thus, it learns the function t, getting more accurate for states which are visited often by the agent, thereby decreasing the corresponding loss and hence giving an agent less incentive to visit such states again. The authors claim that, by learning a deterministic function t instead of predicting observations, RND is more robust to the noisy TV problem [37]. In stochastic settings, predicting observations and using the error of such predictions as indication for novelty can be troublesome. An example is an agent standing in front of a TV showing random pixels. Predicting the next observation cannot be learned due to the randomness of the environment, hence the agent is stuck watching TV as its curiosity never reduces.

3.2.3 Random Networks

By adding an intrinsic reward to the environment reward, which the agent seeks to maximize, the processes of exploration and finding a policy for the reward structure of the environment are not decoupled anymore. Without careful consideration, intrinsic rewards can thus alter the optimal policy for the combined reward such that it is not optimal for the pure environment reward function [22, 38]. Additionally, using neural networks for prediction tasks to compute intrinsic motivation requires the agent to see lots of samples. *NoisyNet* [38] is a technique which tries to avoid these pitfalls. By adding noise to the parameters of a neural network representing a value function, an agent can use its value function directly to act in the environment. This is different to e.g. ϵ-greedy strategies, where the agent would not act greedily with respect to its value function with probability ϵ.

A linear layer of a neural network has the form $y = wx + b$, where y is the output vector, w is the weight vector of the layer, b is a bias vector and x is the input to the layer. With NoisyNet, the network has at least one noisy layer where $w = \mu^w + \sigma^w \odot \epsilon^w$ and $b = \mu^b + \sigma^b \odot \epsilon^b$ [38]. Both, weight vector and bias vector are compromised of learnable parameter vectors μ and σ, while ϵ is a noise vector. Note that $\odot$ represents element-wise multiplication. By minimizing the loss of the networks estimated value function, the noise parameters of the noise layers are also trained. This is a key distinction to methods adding noise to the value estimations after a forward pass through a network. The noise vector ϵ can be sampled e.g. from a (factorized) Gaussian distribution and is held constant in between learning.

NoisyNet can be implemented independent of the environment and algorithm used and has only little computational overhead. It has shown promising results on the Atari benchmark [38] and was implemented in RainbowDQN [2], a DQN agent using multiple techniques to improve upon the base DQN algorithm as it was first introduced [1]. For these reasons, NoisyNet seems a good non-intrinsic reward technique which can be tested on hard exploration instances of the QVBS loaded with MoGym.

3.2.4 Planning and Exploration

While intrinsic motivation utilizes the fact that RL agents seek to maximize their expected reward, adding an intrinsic reward is an indirect exploration method. A more direct, planning based exploration approach, called Go-Explore, has shown a huge improvement in hard exploration environments like Montezuma's Revenge [22]. The idea behind this method is to remember interesting states such that an agent returns to these states before further exploration begins. The authors state that intrinsic motivation methods suffer from two problems they call *detachment* and *derailment* [22]. Go-Explore is specifically designed to avoid these problems.

Detachment states that an agent might explore interesting states until the intrinsic reward is smaller than for other states. Since the agent acts greedily, maximizing the combined intrinsic and extrinsic expected reward, it might never return to these states again, hence being detached from high intrinsic reward frontiers.

If an agent discovers an interesting state, it is desirable to return to this state again. This is often done by acting out the same (or slightly updated) policy which has lead to this state. In most exploration methods, the policy is mixed with some stochasticity, e.g. by adding random noise to the network (see NoisyNet) or by sometimes acting randomly instead of greedily (ϵ-greedy). The added stochasticity, and the fact that the environment itself might have stochastic behavior, often prevents an agent to return to promising states. This problem is called *derailment*.

As a high level description of Go-Explore, an agent stores the paths to newly discovered, interesting states in a buffer. Then the agent samples from said buffer and follows the same actions as stored in the sampled path to return to the promising state. Once the state is reached, the agent can start exploring using any exploration technique. Note that the algorithm also uses a form of imitation learning to make the algorithm more robust in stochastic settings. For a more detailed description of the specific implementation, the interested reader is referred to the original paper by Ecoffet, Huizinga et. al. [22], as only the intuition is relevant for this thesis. In their paper, the authors suggest that this specific version of the algorithm needs an environment which can disable stochasticity during the return phase or allows resetting to a chosen state [22]. Alternatively, a deterministic simulation environment can be used. The authors left whether this idea works in purely stochastic training environments and a corresponding adaption of Go-Explore for future work [22]. The thesis tries to fill this research gap by introducing a new stochastic version of Go-Explore which we call *Stochastic Go-Explore*. It is based on the same basic principle of letting an agent return to interesting states first before explicit exploration starts, but it skips the imitation learning phase of the original algorithm.

Other algorithms using the Go-Explore principles exist which allow training a policy while training in stochastic environments. The authors of Go-Explore published *policy-based Go-Explore* in later revisions of their original paper [39] and another team of researchers developed a similar algorithm called *Diverse Trajectory-conditioned Self-Imitation Learning* (DTSIL) [40]. Both approaches train goal-conditioned policies, include a supervised learning loss (SIL) and use a concept called soft-trajectories. Since Stochastic Go-Explore does not rely on any of these concepts, we think that our approach is an interesting alternative and a valuable contribution to the DRL community. A distinction of the three algorithms and a further discussion on the topic can be found in the appendix (see Section 9.2).

Chapter 4
N-step Description-based Q-learning

Both the sample update and the expected update version of DBQL as introduced in Section 2.4 can be extended to use n-step updates. In the scope of this thesis, three new agents are proposed for this purpose. One new agent extends DBQL_S and is based on an implementation of a n-step DQN agent. For DBQL with expected updates, two agents are introduced: DBQL^n_{DP} which uses dynamic programming and DBQL^n_M which uses matrix operations to compute the n-step updates.

4.1 Sample Description-based Q-learning

The n-step DBQL_S, denoted DBQL^n_S, uses a special buffer which allows for an efficient update computation. This type of n-step buffer was introduced in RainbowDQN [2, 41] and is referred to as NBuffer from now on. It computes the discounted n-step return $G_{t:t+n}$ which is used to compute the sample n-step update target. Recall from Section 2.5 that the n-step sample target for DBQL equals the return approximation

$$G^n_t = R_{t+1} + \gamma^2 R_{t+2} + \ldots + \gamma^{n-1} R_{t+n} + \gamma^n v_*(S_{t+n}).$$

G^n_t uses n future rewards until $v_*(S_{t+n})$ is used to approximate the return from timestep $t+n$ onward. Since the agent does not know the true optimal state-value function v_*, the agent uses its current best estimate. The subscript θ indicates that v_* is approximated, where θ are the current weights of the approximator. Note that the accumulated reward prior to the value function estimate does not change during training, while the agent wants to use the most accurate estimate of the value function. The accumulated n-step reward is defined as [2]

$$R^n_t = \sum_{k=0}^{n-1} \gamma^k R_{t+k+1}$$

such that the n-step agent minimizes the loss

$$L(\theta) = [R^n_t + \gamma^n \, V_{\theta'}(S_{t+n}) - V_\theta(S_t))]^2$$

using gradient descent with the help of a target network to estimate $V(S_{t+n})$.

Since the agent samples a single trajectory of length n to compute this return, the agent could sample by looking into the future and compute R^n_t directly. Instead of sampling from the future, the agent can withhold the computation of the n-step reward for n steps and use the experienced trajectory, gained by acting out its current policy, as a sample. This has the advantage that the agent does not need to know the underlying MDP to plan ahead. In RL, the agent often treats the environment as a black box which makes sampling from the future impossible. While MoGym, the environment used for this thesis, enables the agent to use both kinds of sample strategies, gathering experiences by interacting with the environment is much faster than building a n-step lookahead tree to sample from.

The NBuffer computes the n-step reward using an internal cache of size n, which stores the last n trajectories $(s_t, a_t, r_{t+1}, s_{t+1}, ..., s_{t+n})$ of the current episode. In each step, the agent stores the current trajectory (s, a, r, s') in the NBuffer, which will insert it into the cache using the first-in-first-out principle. If the cache is full, the buffer then computes the associated n-step reward R_t^n and stores the trajectory $(s_t, a_t, R_t^n, s_{t+n})$ in the buffer itself. The agent can then sample trajectories from the NBuffer and use them to calculate the target value which equals G_t^n.

The pseudocode for DBQL$_S^n$ is depicted in Algorithm 2 which highlights the new changes compared to 1-step DBQL$_S$ in orange. Note that the NBuffer of the DBQL$_S^n$ agent does not replace the replay buffer of the 1-step DBQL$_S$ agent. Both buffers are used in conjunction to reduce variance of the n-step loss [41]. The agent still computes the 1-step TD target with the help of the replay memory, which equals G_t^1, just like in the 1-step DBQL$_S$ version. By the definition of the discounted return (Definition 4), the return calculated using G_t^1 and G_t^n, the 1-step and n-step targets, are the same if the optimal state-value function v_* is used. Thus, the agent can minimize the sum of both n-step and 1-step loss as a total loss

$$L(\theta) = ([R_t + \gamma\ V_{\theta'}(S_{t+1}) - V_\theta(S_t)] + [R_t^n + \gamma^n\ V_{\theta'}(S_{t+n}) - V_\theta(S_t)])^2$$

as it should converge to zero if the optimal value function is found.

Algorithm 2 N-step sample DBQL (DBQL$_S$, DBQL)

```
1:  Initialize state-value network and target network $V_\theta$ and $V_{\theta'}$
2:  Initialize NBuffer $N$ and replay buffer $D$
3:
4:  for episodes i=0 to M-1 do
5:      Reset environment to start state $s$
6:      for step t=0 to T-1 do
7:          with probability $\epsilon$:
8:              Select random action $a_t \in A(s_t)$
9:          else with probability $1 - \epsilon$:
10:             compute $Q_\theta(s', a)$ using equation 2.4
11:             select $a_t = argmax_{a \in A(s_t)} Q_\theta(s, a)$
12:         end with
13:         Execute $a_t$, observe $r_{t+1}$ and $s_{t+1}$
14:         Store $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay buffer $D$
15:         Store $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay buffer $N$
16:         every $C$ steps
17:             Sample minibatch $(s_i, a_i, r_{i+1}, s_{i+1})$ of D
18:             Set target $y_i = \begin{cases} r_{i+1}, & s_{i+1}\text{terminal state} \\ r_{i+1} + \gamma \cdot V_{\theta'}(s_{i+1}) - V_\theta(s_i), & \text{otherwise} \end{cases}$
19:             Sample minibatch $(s_i, a_i, R_i^n, s_{i+n})$ of N
20:             Set target $y_i^n = \begin{cases} R_i^n, & s_{i+1}\text{terminal state} \\ R_i^n + \gamma^n \cdot V_{\theta'}(s_{i+n}) - V_\theta(s_i), & \text{otherwise} \end{cases}$
21:             Optimize using gradient descent on loss $(y_i - V_\theta(s))^2 + (y_i^n - V_\theta(s)))^2$
22:             Update target network weights $\theta' = (1 - \tau) \cdot \theta + \tau \cdot \theta'$
23:         end every
```

4.2 Recursive Description-based Q-learning

The Description-based Q-learning agent uses expected updates to compute a target state-value. In the 1-step case, the target is computed as

$$y_i = \max_a \sum_{s',r} T(s,a)(s')[r + \gamma V_{\theta'}(s')]$$

which approximates

$$v_*(s) = \max_a \sum_{s',r} T(s,a)(s')[r + \gamma v_*(s')]$$

by bootstrapping without unrolling the Bellman optimality equation. To use n-step updates, the equation has to be unrolled $n-1$ times. The resulting equation considers the tree of all outgoing paths of length n starting in the state s, for which the value function is estimated. Every state which can be reached in at most n steps is a node of this tree. Since the Bellman optimality equation is recursively defined, the final value can be obtained using dynamic programming. At each leaf node state the value of the function is estimated using the agent's value function estimator. In deep reinforcement learning, the estimator is typically a deep neural network. The estimate is then propagated to the parent node, which solves for Equation 9 and propagates the result back to its parent. This is done until the value of the current state is computed. Figure 4.1 visualizes this tree computation for a 2-step expected update.

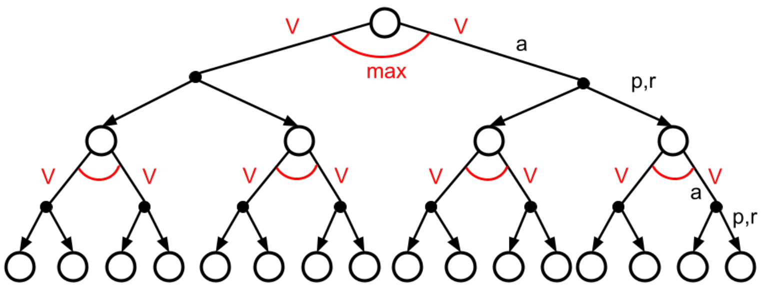


Figure 4.1: Illustration of a 2-step expected target computation using the Bellman optimality equation for $v_*(s)$ (see Definition 9). In this example, the agent can take two actions in each state. For each leaf-node state the state-value is estimated. For each subtree of the first level state nodes, the Bellman optimality equation is applied. The resulting values are used when applying the Equation again for the root node state. For each increasing step size, the computation tree increases by one level. The state-values are backed-up to the next higher level where the highest value is chosen (here marked in red) as state-value of the subtree. The process continues until the value for the root is found.

If the agent has knowledge of the transition probability function $T(s,a)$ and the reward function $R(s,a,s')$ of the environment, it can calculate the expected n-step update using dynamic programming as shown in Algorithm 3. Since MoGym provides us with the necessary information, it allows implementing a n-step DBQL agent using this method, which will be referred to as DBQL^n_{DP}. The pseudocode for this agent is depicted in Algorithm 4, where the agent specific code is highlighted in blue. Since the agent has access to the transition and reward function of the environment at any time, the only information needed to compute the expected n-step return using the dynamic programming procedure (Algorithm 3) is the state for which the value function should be updated. Thus, in line 15, only the visited states are stored in the replay buffer. Line 18 indicates that Algorithm 3 is used to compute the target using the sampled states from the buffer. Note that this requires to build the corresponding search tree and invoke the DP method for every sampled state before the batch update to the neural network can be made.

Algorithm 3 Recursive procedure to compute n-step expected updates

```
1: procedure DP($s \in S$, $n \in \mathbb{N}$)
2:     if $n = 1$ then
3:         return $\max_{a \in Act(s)} \sum_{s' \in S} T(s,a)(s')[R(s,a,s') \cdot V_\theta(s')]$
4:     else
5:         return $\max_{a \in Act(s)} \sum_{s' \in S} T(s,a)(s')[R(s,a,s') \cdot DP(s', n-1)]$
```

4.3 Matrix Description-based Q-learning

Since building a search tree and applying DP to calculate an n-step expected update is expensive, an agent is proposed which uses matrix arithmetic to compute updates for a batch of states more efficiently. Unrolling the Bellman optimality equation for v_* using DP has two disadvantages: (1) it requires the agent to build the same search tree for a state every time it is sampled and (2) it requires as many forward passes through a neural network as the search tree has leaf nodes. It is noted that both problems can be eased by using caches. However, caches do not eliminate the problem entirely and introduce memory overhead. TL_M^n is an agent which tries to solve the two issues mentioned by (1) saving the search tree for a state in the buffer and (2) using a single forward pass through the NN to compute all value estimates of the leaf nodes at once.

Algorithm 3 computes the update target by unrolling the Bellman optimality equation using recursion on the computation tree of depth n. The unrolled equation consists of multiple maximization steps. This can be seen in Figure 4.1 and in the equation

$$v_*(s) = \max_{a \in Act(s)} \sum_{s' \in S} T(s,a)(s')[R(s,a,s') + \gamma[\max_{a' \in Act(s')} \sum_{s'' \in S} T(s',a')(s'')[R(s',a',s'') + \gamma v_*(s'')]]],$$

which is the 2-step unrolled Bellman equation serving as an example in the following paragraphs. Instead of taking the maximum at each level in the computation tree, it is also possible to move all maximization steps to the beginning of the formula. Applying this reformulation to the running example results in

$$v_*(s) = \max_{a \in Act(s), a' \in Act(s')} \sum_{s' \in S} T(s,a)(s')[R(s,a,s') + \gamma[\sum_{s'' \in S} T(s',a')(s'')[R(s',a',s'') + \gamma v_*(s'')]]].$$

The main problem with this equations are the summations as they must be computed in order, from the innermost summation outwards. Thus, when using this formula in a non-recursive algorithm, one has to keep track which results must be summed together. Consequently, this results in tedious and possibly complex computational overhead. As a result, the matrix agent uses a trick which is also used in the sample update DBQL agent DBQL$_S^n$. Recall from Section 2.3 that sample updates do not compute the expectation by summing over all outgoing transitions. Instead, a single transition from the expectation is sampled to compute the update target. This is only possible if all transitions are encountered infinitely often during training to ensure a good approximation of the expectation. The n-step DBQL agent of Section 4.1 relies on exploration techniques to meet this assumption. The matrix DBQL algorithm uses n-step sample updates like DBQL$_S^n$, but it samples every possible outgoing path of transitions of length n instead of a single transition path. Once the sample n-step update target is computed for every outgoing n-path, the highest target value is chosen as the state-value estimate for the state. This novel update target does not equal the update target represented by the Bellman optimality equation and a discussion if the resulting policy converges to the optimal policy is left for future work. While the vague theoretical foundation of the proposed update is a disadvantage, it has the advantage of being easier to compute and potentially reducing training time while still converging faster than 1-step DBQL methods to good policies. The quality of the found policies is subject of the experiments of this thesis. The following serves as an intuitive justification for this novel n-step update.

Recall that the optimal state-value can be formulated as an expectation over returns when following a policy (Definition 9). The return G_t can be substituted by the n-step return G_t^n of Definition 12, such that

$$
\begin{aligned}
v_*(s) &= \max_a \mathbb{E}_{\pi_*}[G_t | S_t = s, A_t = a] \\
&= \max_{A_t,\ldots,A_{t+n}} \mathbb{E}_{\pi_*}[G_t^n | S_t = s, A_t = a] \\
&= \max_{A_t,\ldots,A_{t+n}} \mathbb{E}_{\pi_*}[R_{t+1} + \gamma^2 R_{t+2} + \ldots + \gamma^n v_*(S_{t+n}) | S_t = s, A_t = a]
\end{aligned}
$$

expresses the optimal state-value using the n-step return [31].

By computing the n-step sample return G_t^n for every possible outgoing trajectory of state s, the agent obtains m n-step returns. In the following, the subset $i \in \{0, \ldots, m-1\}$ indicates the i-th n-step return calculated for this update. Then let

$$G_{t,i}^n = R_{t,i}^n + v_*(S_{t+n,i})$$

be the n-step return and let $R_{t,i}^n$ be the n-step reward as defined in Section 4.1. Let $\boldsymbol{R}$ denote a $m \times 1$ matrix containing the n-step rewards and $\boldsymbol{S}$ denote the $m \times 1$ matrix of all successor states $S_{t+n,i}$. The associated n-step returns can be computed in one addition as a $m \times 1$ matrix $\boldsymbol{G}$ using equation

$$
\begin{aligned}
&\boldsymbol{G} = \boldsymbol{R} + V_{\theta'}(\boldsymbol{S}) \\
\Leftrightarrow &\begin{bmatrix} G_{t,0}^n \\ G_{t,1}^n \\ \vdots \\ G_{t,m-1}^n \end{bmatrix} = \begin{bmatrix} R_{t,0}^n \\ R_{t,1}^n \\ \vdots \\ R_{t,m-1}^n \end{bmatrix} + V_{\theta'}(\begin{bmatrix} S_{t+n,0} \\ S_{t+n,1} \\ \vdots \\ S_{t+n,m-1} \end{bmatrix}).
\end{aligned}
$$

Note that $V_{\theta'}$ represents the target neural network which approximates v_* and returns a $m \times 1$ matrix of estimations for $v_*(S_{t+n,i})$ when given a batch of m states. With $\boldsymbol{G}$ available, the target value $v_*(S_t)$ is the highest return for any action sequence $A_t, \ldots, A_{t+n}$ over a set of states $S_t, \ldots, S_{t+n}$. Taking the maximum return instead of the maximum over expectations of returns is used to approximate the state-value efficiently. The resulting target update is $v_*(S_t) \approx \max(\boldsymbol{G})$. Calculating the n-step update by using this equation compared to the exact dynamic programming approach of TL_{DP}^n has the disadvantage that maximizing over returns instead of expected returns leads to overestimation. Assume for example a small MDP where one transition after taking an action a is very unlikely, e.g. the transition has a probability of 0.01 but a very high reward of e.g. 100, while the only other transition has probability $1 - 0.01 = 0.99$ with reward 1. For simplicity, in the example there is only one other action a' from the start state with two transitions. Both transitions have probability 0.5 and a reward of 10. Assume further that the MDP only has terminal states after one transition. The MDP is depicted in Figure 4.2. The expected return for action a is $0.01 \cdot 100 + 0.99 \cdot 1 = 1.99$ and the expected return of action a' is 10. Now the update target using the expected update is $v_*(S) = max(1.99, 10) = 10$. The matrix agent computes the returns $\boldsymbol{G} = [100, 1, 10, 10]^T$ and thus the update target $v_*(S) = \max(\boldsymbol{G}) = 100$.

To reduce the overestimation, the known transition function is used by weighting the rewards with the cumulative probability of the associated transition when taking the sequence of actions $A_t, \ldots, A_{t+n}$ starting in state S_t (for which the update target is calculated). The n-step unrolled Bellman optimality equation can be formulated as

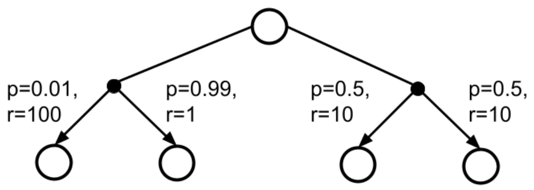


Figure 4.2: Example MDP where a matrix agent overestimates the value of a state compared to a dynamic programming agent.

$$v_*(s) = \max_{A_t,\dots,A_{t+n}} [[\sum_{k=t}^{t+n} \gamma^{k-t} \prod_{j=t}^{k} (T(S_j, A_j)(S_{j+1})) R(S_k, A_k, S_{k+1})] + \gamma^n \prod_{k=t}^{t+n} (T(S_k, A_k)(S_{k+1})) v_*(S_{t+n})].$$

For reference, the 2-step example formula can be retrieved by setting n to two. Ignoring the summation symbol as we sample a transition from the equation, the formula is weighting the rewards and is the basis of the matrix n-step backup. Let $\hat{R}_t^n$ be the weighted n-step reward such that

$$\hat{R}_t^n = \sum_{k=t}^{t+n} \gamma^{k-t} \prod_{j=t}^{k} (T(S_j, A_j)(S_{j+1})) R(S_k, A_k, S_{k+1}).$$

Let $\hat{G_{t,i}^n}$ be the weighted return, which uses the weighted rewards and also weights the state value estimate by its transition probability such that formula

$$\hat{G}_{t,i}^n = \hat{R}_{t,i}^n + \gamma^n \prod_{k=t}^{t+n} T(S_{t+n-1}, A_{t+n-1})(S_{t+n}) v_*(S_{t+n,i})$$

defines the weighted return. Furthermore, let $\boldsymbol{G}$ and $\boldsymbol{R}$ be the weighted matrices of the n-step return and reward. To compute the discounted return $\hat{G_{t,i}^n}$, the state-value is also weighted with its transition probability. Let $C_{t,i}^n$ be the probability used to estimate the final state-value, such that

$$C_{t,i}^n = \gamma^n \prod_{k=t}^{t+n} T(S_{t+n-1}, A_{t+n-1}).$$

Let $\boldsymbol{C}$ be the $m \times 1$ matrix storing these state-value weights for all outgoing transition paths of length n from S_t, then $\boldsymbol{G}$ can be computed with the equation

$$\boldsymbol{G} = \boldsymbol{R} + \boldsymbol{C} \odot V_{\theta'}(\boldsymbol{S})$$

$$\Leftrightarrow \begin{bmatrix} G_{t,0}^n \\ G_{t,1}^n \\ \vdots \\ G_{t,m-1}^n \end{bmatrix} = \begin{bmatrix} R_{t,0}^n \\ R_{t,1}^n \\ \vdots \\ R_{t,m-1}^n \end{bmatrix} + \begin{bmatrix} C_{t,0}^n \\ C_{t,1}^n \\ \vdots \\ C_{t,m-1}^n \end{bmatrix} \odot V_{\theta'}(\begin{bmatrix} S_{t+n,0} \\ S_{t+n,1} \\ \vdots \\ S_{t+n,m-1} \end{bmatrix}),$$

where $\odot$ is the row-wise matrix multiplication. The remaining target update is the maximum weighted return, which can be expressed as $v_*(s) = \max(\boldsymbol{G})$. Algorithm 4 depicts the pseudocode for the n-step matrix algorithm, where the algorithm specific parts are colored in orange.

Algorithm 4 N-step expected DBQL (DBQL^n_S, DBQL^n)

```
 1: Initialize replay buffer $D$
 2: Initialize value network $V_\theta$ and target network $V_{\theta'}$ where $\theta$ and $\theta'$ are the weights
 3:
 4: for episodes i=0 to M-1 do
 5:     Reset environment to start state $s$
 6:     for step t=0 to T-1 do
 7:         with probability $\epsilon$:
 8:             Select random action $a_t \in A(s_t)$
 9:         else with probability $1 - \epsilon$:
10:             Compute $Q_\theta(s', a)$ using equation 2.4
11:             Select $a_t = argmax_{a \in A(s_t)} Q_\theta(s, a)$
12:         end with
13:         Execute $a_t$, observe $r_{t+1}$ and $s_{t+1}$
14:         $\boldsymbol{R_t}, \boldsymbol{C_t}, \boldsymbol{S_t} \leftarrow$ update_matricies($s_t$)
15:         Store $\begin{cases} (s_t) \\ (s_t, \boldsymbol{R_t}, \boldsymbol{C_t}, \boldsymbol{S_t}) \end{cases}$ in replay buffer $D$
16:         every $C$ steps do
17:             Sample minibatch of samples $\begin{cases} s_j \\ (s_j, \boldsymbol{R_j}, \boldsymbol{C_j}, \boldsymbol{S_j}) \end{cases}$ from replay buffer $D$
18:             Set target $y_j = \begin{cases} DP(s_j) \\ \max(R_j + C_j \odot V_{\theta'}(S_j)) \end{cases}$
19:             Optimize using gradient descent on loss $(y_i - V_\theta(s))^2$
20:             Update target network weights $\theta' = (1 - \tau) \cdot \theta + \tau \cdot \theta'$
21:         end every
```

Chapter 5
Go-Explore

In this section, the stochastic version of Go-Explore is introduced. First, the original Go-Explore framework is outlined as it was introduced in the initial paper of Go-Explore [22]. Afterwards the main changes in the new stochastic version are explained and finally implementation details for the Stochastic Go-Explore DBQL agent, which is the agent used for this thesis, are specified. Note that the explanation in this chapter does not include policy-based Go-Explore which was introduced in subsequent revisions of the original paper [39], because Stochastic Go-Explore was developed on the base of the original Go-Explore framework. While there are existing approaches which address the same issue as Stochastic Go-Explore, they differ in the concepts used. A discussion can be found in the appendix (see Section 9.2).

5.1 Original Go-Explore

Go-Explore is an exploration framework which can be split into two distinct phases during training: (1) the *exploration phase* and (2) the *robustification* phase. The main exploration ideas of Go-Explore are addressed in the first phase whereas the second phase is used to train policies by applying self-imitation learning (SIL) on the best trajectories found in phase one to make them more robust in stochastic environments. Figure 5.1 extracted from the paper introducing Go-Explore [22] illustrates an overview of the framework. Note that no policy is trained in the exploration phase [39] of the original algorithm.

5.1.1 Exploration Phase

The exploration phase consists of four high level steps which are continuously repeated. First, at the start of an episode, the agent samples a state from an archive. The archive is a buffer which stores interesting or promising states together with instructions on how to reach them. A heuristic is used to assign each state a value indicating how interested the agent is to return to this state. This value is also stored in the archive and is used to ensure that states with a higher value have a higher probability to be sampled.

In the second step, the agent tries to go to the sampled state by using the additional information stored in the archive. The information could be a concrete sequence of actions to perform to reach the state. By applying the stored actions, the agent is only guaranteed to reach the sampled state again if no form of stochasticity is present during this phase. This includes any form of random exploration mechanisms and the stochasticity of an environment. Due to this, the original Go-Explore framework assumes that either a deterministic version of the problem environment exists during training, e.g. a simulation environment, or that the environment allows resetting to any specified state [22]. Note that this assumption is dropped in later versions, e.g. for policy-based Go-Explore [39].

Once the agent has reached the state of interest, the third step of the phase begins. For the following actions, the agent uses an arbitrary exploration technique to finish the episode and to keep exploring the state space. Any newly encountered states are evaluated with the heuristic and are added to the archive. The archive also updates states where the newly generated

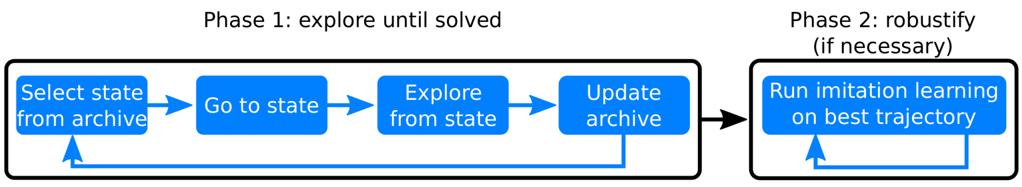

Phase 1: explore until solved
Select state from archive
Go to state
Explore from state
Update archive
Phase 2: robustify (if necessary)
Run imitation learning on best trajectory

Figure 5.1: Original Figure from the Go-Explore paper [22] presenting a high-level overview of the algorithm.

trajectory is "better" than the one stored. What constitutes a better trajectory is a design choice. The authors propose that possible criteria could be e.g. the trajectory length or the cumulative reward [22]. Updating the buffer represents the fourth and final step. The exploration step and the archive update step are intertwined and cannot be clearly separated from each other. The exploration phase of Go-Explore is specifically designed to mitigate the problems of derailment and detachment.

Derailment is a situation where the agent has discovered a promising state from which it wants to explore further but is unable to return to it [22]. The authors of Go-Explore claim that RL agents typically try to return to such states by re-enacting the policy that led to the interesting state but with some sort of exploration perturbations, e.g. through acting ϵ-greedy or also following intrinsic motivation [22].

Detachment describes a problem occurring with agents using intrinsic motivation (IM) as an exploration strategy. IM agents which discover multiple high intrinsic reward areas may sequentially explore each area for a while. During the exploration the agent continuous to lower the intrinsic reward (IR) of each area until another area is more interesting. Detachment occurs if an agent is not able to rediscover the first high IR areas encountered even though exploring them further would still yield high rewards [22].

5.1.2 Robustification Phase

In the first phase the environment is explored either on a completely deterministic version of the environment or the stochasticity was at least deactivated in step two when the agent returned to a promising state sampled from the buffer. Note that policies are only trained in the robustification phase. Since the agent explored a simplified environment with relaxed assumptions, the trajectories found might not perform as well in the stochastic environment. To obtain a more robust policy, the authors of Go-Explore [22] suggest using a form of imitation learning after the exploration phase. Given a few of the best trajectories found in phase one, the agent is trained to mimic or even improve on them while training in the stochastic environment. The imitation learning algorithm used in the paper is the Backward Algorithm from Salimans and Chen [42], but can be substituted by any other algorithm. The paper remarks that the robustification phase may not be needed if a policy obtained in the first phase is already robust to stochasticity. This is an important observation used in the stochastic version introduced in Section 5.2.

5.2 Stochastic Go-Explore

The Stochastic Go-Explore agent implemented for this thesis drops some concepts proposed in the original paper as well as the assumption that a deterministic or resettable environment is needed during the training of phase one. One big concept which is not used in the implemented Stochastic Go-Explore agent is the robustification phase. The original paper states that this phase might not be needed if the trained policy in the first phase is already robust with regard to stochasticity of the environment [22]. Since in this proposed stochastic version of Go-Explore the agent is directly training on a stochastic environment rather than training on a deterministic environment, this is the case. Note that applying the robustification phase might still improve the found policy but any discussions or experiments regarding this are beyond the scope of this thesis.

Since Go-Explore was designed to solve hard, sparse-reward Atari benchmarks, where the observations of the environment include images, the authors suggest to use cell representations [22] to lower the high state-space dimension. The idea is to group similar states together while keeping “meaningfully different” states separate [22]. Cell representations change for each environment, which results in the need to tune this concept for each environment before training. In very high state-space dimensions using images as observations, the training speed-up justifies this pre-training overhead. In the context of training QVBS benchmarks using MoGym, which does not use images as observations, the benefit of having a more generally applicable agent when dropping the concept of cell representations outweighs potential training speed-ups. It is also not trivial to design good cell representations by using only the information extracted from the MDP loaded through MoGym without any further knowledge of the original problem description. Thus, the implemented Stochastic Go-Explore agent trains on the original state space.

5.2.1 Stochasticity During Training

Apart from dropping some minor concepts and the robustification phase, the main abstract steps of phase one do not change much when adapting Go-Explore to exclusively use a stochastic environment. The agent still samples a state from the archive at the beginning of an episode in the first step. As a second step, the agent then aims to go the sampled state by following the stored information in the archive. The agent has to account for the fact that it might not be able to imitate the steps from the archive due to environment noise, which is a main difference to the original framework. If the agent could not completely imitate the stored trajectory, it has to start step three (the exploration step) earlier. Updating the archive remains the same as in the previous fourth step.

One of the main motivations for developing the Go-Explore framework was to mitigate the problem of derailment and detachment. Recall that derailment describes the inability of the agent to return to a promising state. By using a stochastic environment in step two, derailment is still possible, which was the reason why the authors of Go-Explore advocated to use deterministic environment or an environment which can reset to the desired state directly [22]. A discussion about the feasibility of using a stochastic environment during step two was left unanswered by the authors in the original paper [22], but addressed in subsequent publications with the introduction of policy-based Go-Explore [39]. The experiments of this thesis also indicate that the idea of Go-Explore remains a powerful approach, even in a fully stochastic setting.

5.3 Implementation Details

Go-Explore is a framework instead of a precise algorithm. This is why implementations of agents using the Go-Explore framework can differ in various aspects. There are key design choices which have to be made. They include:

- The data structure used for the exploration archive and the way it is updated.
- The method used to return to a sampled state. Popular choices are e.g. a stored sequence of actions, which can be imitated, or storing a copy of the policy network.
- The heuristic used to determine interesting and/or promising states.
- The imitation learning method used in the robustification phase.
- The exploration technique used to discover new states.

5.3.1 Exploration Archive

The exploration archive is a buffer which stores information about how to get back to a state. In this case it stores a full trajectory and an interesting state together with some priority value indicating how interested the agent should be to return to the stored state. The data structure representing this archive must provide functionality to sample a state and the corresponding trajectory based on a probability distribution over the stored priorities. Sampling a single trajectory is done roughly every episode, which requires sampling to be efficient. The archive must also be able to update the priority for entries efficiently as this is also done frequently during training. The same requirements are needed when using prioritized experience replay (PER), a technique first introduced by Schaul et al. [43]. In their paper a *sum-tree* data structure is used which enables efficient sampling and updating. The same data structure is used for the implementation of the Stochastic Go-Explore agent for this thesis.

The choice of storing trajectories in the archive instead of a sequence of actions or a policy network is a direct consequence of using a stochastic environment for training. As outlined in Section 5.2.1, an agent can only try to return to a sampled state. Due to the noisy environment, taking an action does not always result in the agent visiting the same successor states. The agent needs to be able to realize that it cannot reach the stored state by imitating the same actions as before. If the agent is only provided with a sequence of actions to follow, this realization is only possible after all actions are performed by checking if the agent has reached the sampled state or any other state. Furthermore, if the agent has gone off-path, the stored actions might not be applicable in the new states encountered. An agent which has access to the full trajectory can check at each time step if the current state matches with the corresponding state stored in the trajectory. In case of a mismatch, the agent can thus start with random exploration (step three) earlier. A similar argument can be made against storing a policy network. One disadvantage of storing trajectories is that they can vary in length. The archive must be a data structure which supports this. This is the case with the sum-tree used for this implementation.

5.3.2 Heuristic

As a heuristic to determine which states are interesting and should be revisited either the Intrinsic Curiosity Module (ICM) [27] or Random Network Distillation (RND) [37] is used. The advantage of using ICM is that it is very easy to implement and curiosity is a general heuristic which is suitable for most environments. The same applies for RND, thus the experiments include a Stochastic Go-Explore agent for each of the two. One big disadvantage of both heuristics is that they require a forward pass through a neural network each time a state is evaluated. The computational overhead of heuristic evaluation must be carefully considered as multiple states are evaluated in every episode during the third and fourth step.

The choice of a suitable heuristic function is not trivial. It is not obvious if curiosity by itself is a good choice for Stochastic Go-Explore. Once a goal state was found in a sparse-reward stochastic environment the agent should be interested to see the goal state again and again, such that it can update its policy accordingly. Since neural networks require multiple backward propagation runs until a good approximation is learned, it is desirable to sample goal reaching trajectories multiple times, even when their curiosity value is already low. This is especially the case if the trajectories that reach a goal are long because imitating them completely in stochastic environments might be unlikely. Incorporating the cumulative reward into the heuristic value could help to increase the priority of such trajectories permanently and provide a threshold such that the priority for high rewarding trajectories is never reduced to zero. During step three and four, the agent constantly adds new trajectories to the archive. The priorities of old entries encountered during an episode are updated.

5.3.3 Exploration Technique

The distinct idea of Go-Explore is that an agent should not always explore blindly, but should first return to promising states before exploring. However, it does not specify how it should explore from there. The implemented Stochastic Go-Explore DBQL agent uses an ϵ-greedy policy for exploration. Note that any other exploration technique is also feasible but intrinsic motivation methods are arguably a bad choice because Go-Explore tries to mitigate the problem of detachment. By letting the agent optimize for a combined intrinsic and extrinsic reward, the main principles on which Go-Explore is build on is necessarily compromised. A suitable alternative to intrinsic motivation methods and the rather simple ϵ-greedy strategy would be e.g. NoisyNet. Using ϵ-greedy for exploration is preferred for this thesis to demonstrate that the concept of Stochastic Go-Explore is powerful enough to enable learning on difficult instances of the QVBS without adding another, more powerful exploration technique.

5.3.4 Pseudocode

Algorithm 5 shows the pseudocode for Stochastic Go-Explore. Due to the size of algorithm, we limit the pseudocode to the DBQL variant using ICM as heuristic. All lines which remain from the original DBQL algorithm are shown in gray to highlight the new parts of Go-Explore.

Algorithm 5 Stochastic Go-Explore with DBQL using ICM as heuristic

```
 1: Initialize replay buffer $B$ and exploration buffer $E$
 2: Initialize ICM module
 3: Initialize value network $V_\theta$ and target network $V_{\theta'}$ where $\theta$ and $\theta'$ are the weights
 4:
 5: for episodes i=0 to M-1 do
 6:     $s \leftarrow$ Reset environment to start state
 7:     Set explore to true with probability (1-$\epsilon$) or if number of samples in $E$ less than $P$
 8:     if not explore then
 9:         trajectory $\leftarrow$ Priority-sample trajectory $[(s_0^i, a_0^i, s_1^i), ..., (s_{n-1}^i, a_{n-1}^i, s_n^i)]$ from $E$
10:     steps_done $\leftarrow$ Initialize list to keep track of trajectories done in current episode
11:     for step t=0 to T-1 do
12:         if trajectory was imitated successfully $(t > n)$ then
13:             Set agent to explore
14:         if explore then
15:             $a_t \leftarrow$ Choose action using simple exploration technique, e.g. $\epsilon$-greedy
16:         else
17:             $a_t \leftarrow$ Choose action $a_t^i$ from sampled trajectory
18:         Execute action $a_t$ and observe $r_{t+1}, s_{t+1}$
19:         Store $(s_t, a_t, T(s_t))$ in replay buffer $D$
20:         Store $(s_t, a_t, s_{t+1})$ in steps_done
21:         if explore then
22:             $p \leftarrow$ Compute intrinsic motivation for $s_t, a_t, s_{t+1}$, using ICM
23:             Store steps_done in $E$ with priority $p$
24:         else if $s_t = s_t^i$ and end of sampled trajectory was reached $(t = n)$ then
25:             $p \leftarrow$ Compute intrinsic motivation for $s_t, a_t, s_{t+1}$, e.g. using ICM
26:             Set priority of trajectory in $E$ to $p$
27:         every $C$ steps do
28:             Sample a minibatch of samples $(s_j, a_j, r_{j+1}, s_{j+1})$ from $D$
29:             Set target $y_j = \begin{cases} r_{j+1}, & s_{j+1}\text{terminal state} \\ r_{j+1} + \gamma \cdot V_{\theta'}(s_{j+1}) - V_\theta(s_j), & \text{otherwise} \end{cases}$
30:             Optimize using gradient descent on loss $(y_j - V_\theta(s_j))^2$
31:             Update target network weights: $\theta' = (1-\tau) \cdot \theta + \tau \cdot \theta'$
32:             Compute inverse loss $L_i$ and forward loss $L_f$ of ICM
33:             Optimize ICM module using gradient descent on $\beta\ L_f + (1-\beta)\ L_i$
34:         end every
35:     Update buffer (Optionally introduce a small penalty each time a trajectory is executed)
```

Chapter 6 Methodology

This chapter provides an overview of all experiments made to answer the research questions and provides reasoning why the specific experiment designs are chosen. Recall from Section 1.2 that this thesis aims to find the reasons why some QVBS benchmarks are not learnable using DQN, thereby discovering challenges to overcome using MoGym on those benchmarks. In addition, the thesis aims to provide the reader with DRL agents and solutions to address the challenges by improving on existing approaches. In particular, recall that the hypothesis of this thesis is that the main two challenges of hard benchmarks are (1) the action space explosion, which results from loading a MDP and converting it to an environment, and (2) that sophisticated exploration is required.

To test this, the thesis proposes to use DBQL, a DQN variant which mitigates the action-space explosion problem, in combination with several exploration techniques like Random-Network Distillation, ICM or NoisyNet and the newly introduced Stochastic Go-Explore. The thesis aims to answer two research questions (RQs) related to this hypothesis, namely RQ 1.1 and RQ 1.2 defined in Section 1.2. Another objective of the thesis is to improve on DBQL by extending it to use n-step learning, which is why three new algorithms are introduced in Chapter 4. These are used to answer research questions RQ 2.1 and RQ 2.2, which ask whether n-step learning leads to faster learning with regard to sample efficiency and training time.

In this chapter, first general methods used for all experiments regardless of the specific research questions are outlined. This is done by first discussing the scope of the experiments and then the limitations of deep reinforcement learning. After that the experiments and methodology for the first set of research questions (RQ 1.1 and 1.2) and for the second set of research questions (RQ 2.1 and 2.2) are discussed in separate sections. Finally, a short summary of all the experiments is given.

6.1 General Methods and Limitations

The time constraints of the thesis have a direct impact on the possible depth of the experiments to answer the research questions. If not planned carefully, trying to prove or disprove the claims of the research questions can become very time-consuming. Thus, the scope of the experiments has to be limited carefully and limitations in regard to typical challenges of deep reinforcement learning have to be addressed first.

6.1.1 Scope of Experiments

The QVBS has over 25 benchmark instances which can be used to answer the research questions [10, 11]. This is infeasible given the time constraints and hardware access for the thesis. Consequently, experiments for each set of research questions are made on a small subset of benchmarks and the findings are then used to generalize and answer the research questions. The quality of the generalization depends on the choice of benchmarks used. To test the hypothesis stating that the action-space explosion as well as exploration are the main challenges of unlearnable instances (RQ 1.1 and 1.2), we decided on three benchmarks: *Elevators, Firewire* and *Blocksworld*. The first two were benchmarks where the DQN agent of the MoGym paper failed to learn. Thus, they provide a good baseline to check if the discussed approaches enable learning on hard instances. The search space of Elevators consists of 909 states [11] and 41 actions while Firewire has 290017 states [11] and 24 actions. While Firewire has a huge state-space, both environments have a rather low action-space. Thus, to test the impact of action-space explosion we decided to include Blocksworld as more extreme example which has 1125 [11] states and 260 actions. To test the introduced n-step agents of DBQL, we decided to use two environments called *Cdrive* and *Racetrack*. These environments are chosen because the DQN agent of the MoGym paper was able to learn on them [8], which indicates that the DBQL agents can learn on them as well with the same simple exploration technique called ϵ-greedy. Note that only Cdrive is a benchmark from the QVBS which is unproblematic as research questions RQ 2.1 and RQ 2.2 are only concerned with the general properties of the n-step agents. No other environment is considered as we expect the n-step agents to have a much higher training time with increasing n-steps compared to e.g. the exploration agents.

Environments For completion, the following is a short outline of the environments. For more information on the QVBS benchmarks the interested reader can visit the official QVBS website [11]. In Blocksworld an agent tries to rearrange blocks on a table from a starting configuration to a goal configuration. An agent can only move a block which has no block placed on top of it, and it can only place it on other blocks with no block on top of it. In the version used for the experiments five blocks are present in the environment (blocksworld.5.jani). In Elevators, a number of elevators must carry coins to predefined levels. The version used in the experiments has three levels and three coins (elevators.a-3-3.jani). Stochasticity is introduced as an elevator can fail and fall down a level. In Firewire, more specifically *firewire_dl*, the Tree Identify Protocol for the IEEE 1394 High Performance serial bus is modeled, which transports media data across a network. The environment allows to set two properties before loading. For the experiments we set delay to 3 and deadline to 800 because this induces a high state-space and was one of the suggested configurations on the QVBS website [11]. In Racetrack the agent controls a car which is driving on a racetrack. The racetrack is a gridworld, where the agent starts in starting cells and has to reach goal cells. In each cell, nine actions can be performed. The agent can accelerate in each direction (including diagonals) or decide not to accelerate. The current velocity is changed based on the direction of the acceleration. The problem is interesting as the agent can reach velocities where it cannot come to a standstill after taking a single action. Furthermore, on the noisy variant of Racetrack, there is a small probability that the action fails and the agent cannot change its current velocity. For the experiments, the map *barto-small* is used to create the environment. In Cdrive the agent drives a car around a city with 6 locations (cdrive.6.jani). Roads with traffic lights connect the locations. The goal of the agent is to reach a specified location without an accident [11]. Note that the authors of MoGym have used a simplified variant of Cdrive with only two locations [8]. This version is not used here as it is expected that the 1-step agents can already learn this environment with very few episodes of training [8] such that recognizing learning improvements becomes difficult.

6.1.2 DRL Limitations

Hyperparameters Drawing general conclusions on the performance of a DRL agent is complicated because DRL agents are very sensitive to the hyperparameters used for training and for each environment the hyperparameter configuration to achieve the best performance can vary [44, 45]. This means that ideally, one wants to find the best hyperparameters for each environment tested for each agent before comparing the performance of different agents. To evaluate if a certain configuration of hyperparameters is better than another, the agent needs to train on an environment as there is no way to infer this otherwise. In addition, some parameters itself might vary on a continuous scale and are dependent on other parameters. This makes searching the best hyperparameters infeasible in most cases. This problem is well known in the DRL community and the deep learning community in general [45, 46]. We address this problem by applying hyperparameter tuning to find, among a predefined set of values for each parameter, a configuration that works reasonably well and assume that the performance difference to better configurations is not significant for the results. Further information on this can be found in Section 6.1.3.

Randomness Another problem is randomness during training as the environment itself is noisy and additionally an agent can use randomness, e.g. for exploration, during training. Due to this, for each experiment the random number generators used are set to a specified seed. This makes the experiments replicable and allows interested readers to verify the results. Since an agent might perform better on one seed than another, each experiment is done over multiple seeds. With each additional seed the number of experiments grows exponentially as this adds an experiment for each agent on each environment (and each n-step). Consequently, the experiments are limited to three random seeds, although more seeds are desirable.

Reward Structure The reward structure is an important aspect of the experiments because the DRL agents' goal is to maximize the expected return. Since agents are trained on multiple environments for each research question, we fix the reward structure used for training. This has two advantages: (1) comparing the results of experiments is easier as they do not need to be normalized and (2) the reward structure is not optimized for specific environments which reduces design bias. However, it allows only for a simple reward structure that can be applied to all problems. This neglects the fact that results could improve on some environments with different rewards. Nevertheless, the ability to compare the results more easily is more important for this thesis than finding the best policies on each environment. For all experiments, the agents receive a positive reward of 100 for reaching a goal state and a negative reward of -20 for reaching a bad state. For any intermediate action which does not result in ending the episode either by winning or loosing, the agents receive a reward of 0. This specific reward structure was chosen because it has worked well for training various agents in projects outside this thesis. It would also be reasonable to set the positive reward to 1 and the negative reward to -1, which is not done here due to personal preference.

Neural Network Since this thesis uses deep reinforcement learning, the specifc neural network used to represent the learned policy influences the learning capability and training performance of the agents. Analogue to the reward structure, we decided to use the same basic neural network structure for each agent and experiment. The network is a multilayer perceptron (MLP) consisting of two hidden layers with 64 nodes each. The size of the input layer varies depending on the state representation of the used environment. Since DQN estimates action-values, the output layer equals the action dimension of each environment, e.g. for Racetrack the output layer consists of nine nodes. DBQL estimates state-values such that the output layer always consists of one node. Every layer except the final output layer passes their results through a ReLu (Rectified Linear Unit) activation function. The network is learned through backpropagation using Adam [47] as optimization algorithm.

Performance Evaluation When training (D)RL agents, it is difficult to evaluate how well the trained agent can actually perform the task it is supposed to learn. Recall that the main goal of (D)RL agents is to find policies that maximize the expected return received when interacting with the environment. To measure if an agent is learning, the returns received during training are tracked. If the average returns received increase the agent is learning something. However, the reward structure can be difficult to define in a way that an agent learns the actual, abstract problem it is supposed to solve. Sparse-reward functions like the one used for this thesis tend to lead the agent to learn the desired problem. However, they can be problematic as exploration is a problem if the agent gets no useful feedback (zero reward) for a long period of time [48, 49]. With that in mind, we can say that an agent is learning to solve a problem by looking at the training rewards. Deep statistical model checking (DSMC) can help to assess the (training) performance of an agent independent of the training rewards. To do this, policies are saved during training. DSMC is a rather new technique used in the scope of DRL. Furthermore, the discrepancy between correctly maximizing a reward structure and solving the actual abstract problem makes comparing training returns and DSMC results both challenging and interesting. Consequently, the training returns are valued higher in the evaluation as the DSMC results.

Still, using the extracted training returns and DSMC evaluations directly to evaluate the training performance of an agent is not very useful. This is due to the high variance of the measurements. (D)RL agents want to learn policies maximizing the expected return received. Even for good policies, there might be episodes with a very low return and in the next episode the agent receives a high return due to e.g. exploration and environment noise. To account for this variance and the initial goal to maximize expected return, usually one averages the returns using a simple moving average (SMA) first to evaluate the training process. A SMA calculates the current average by summing up the previous k returns (or all data points if less then k are available) and dividing the result by the amount of data points used for the computation. For this thesis the parameter $k = 100$ is used. Similarly, the DSMC evaluations for the extracted policies can also have a high variance. However, due to sparser data availability since far fewer policies than training returns are extracted and evaluated, applying SMA does not seem appropriate. Thus, we decided to use regression to smooth the data and evaluate the training process of agents using DSMC. To do this, *locally estimated scatter plot smoothing* (LOESS) is used which is a local regression algorithm [50].

6.1.3 Hyperparameter Tuning

Deep reinforcement learning is sensitive to the hyperparameters of the algorithms [44, 45]. Since training a DRL agent takes a significant amount of time, tuning many hyperparameters remains a difficult task in reinforcement learning. Consequently, in the scope of this thesis, not all hyperparameters are tuned. Since DQN and DBQL are very similar, the common hyperparameters are tuned by using DQN on Racetrack as DQN needs less time to train the same amount of episodes as DBQL. No hyperparameters are tuned by using n-step agents. Tuning is done by sampling 20 configurations of predefined parameter values and evaluating the policies using DSMC. Only the best policy for each configuration is used for comparison. After that a stochastic DBQL Go-Explore agent is tested with the chosen parameters of the DQN tuning experiments and is trained on Elevators to tune the size of the exploration buffer[1]. Similarly, a DQN agent using ICM and RND for curiosity driven exploration is trained on Racetrack

[1]Go-Explore is not based on DQN for tuning in contrast to the other agents because implementing Go-Explore is difficult and error-prone. Due to this, it was decided to use the final DBQL Go-Explore agent to tune the archive size. Go-Explore as implemented works only if the environment has a single initial state. Thus, tuning is not done on Racetrack as the used Racetrack version has issues when specifying a specific start state.

to tune the exploration specific parameters. This is also done by sampling 20 configurations from predefined parameter ranges. Each agent trained for tuning trains for a total of $100,000$ steps on an environment and on a single seed for the random number generators. The final hyperparameters can be found in Section 9.1 in the appendix.

6.2 First Research Question

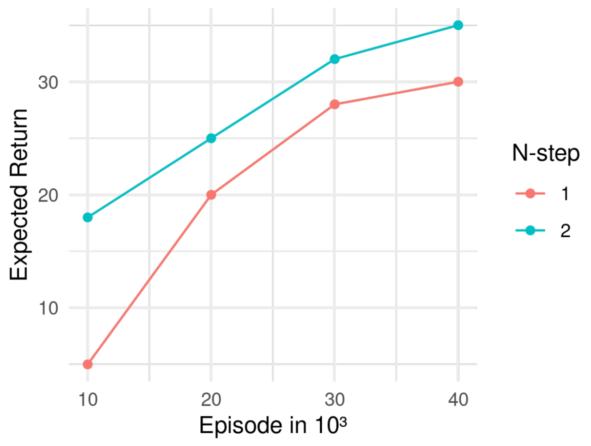
Recall that research question RQ 1.1 asks whether the action-space explosion is the main factor for the unsolvabililty of some benchmarks of the QVBS. Similarly, RQ 1.2 asks whether exploration is a key factor if learning fails even after accounting for the action-space explosion. To test both questions, DBQL agents with different exploration techniques are trained on Elevators, Firewire and Blocksworld. The exploration techniques are ϵ-greedy, RND, ICM, NoisyNet and Stochastic Go-Explore in two variants: one with ICM and with RND as prioritization heuristic. The ϵ-greedy variant is important to verify RQ 1.1 because the DQN agent of MoGym is also using this simple technique for exploration. By using it in conjunction with DBQL which tries to mitigate the influence of the action-space size, it allows drawing conclusions for the direct influence of the action-space explosion without exploration. As a baseline, DQN using ϵ-greedy is trained on each environment as well. For each experiment the agents train for $100,000$ episodes, extracting policies every $1,000$ episodes for DSMC evaluation in addition to all training scores received.

If an agent is receiving a constant average training score above zero from some time onward, we say that the agent was able to learn on an environment. In particular, to answer the research questions it is not important that agents can learn an optimal policy, although comparison between the quality of the agents might be interesting. As a secondary measure, the extracted policies are evaluated using DSMC for two criteria: (1) the expected return and (2) the goal-reaching probability (GRP). The expected return is closely related to the training scores and can give direct insight into the quality of a policy in regard to the agent's objective to maximize the expected return. The GRP gives insight into the quality of the policy in regard to the underlying abstract problem that should be solved by stating how often an agent reaches a goal-state when starting in any starting state of the environment. Since DSMC provides only estimates we specify that the estimation of returns can at most be wrong by 1 and the GRP estimate by 0.01 with a confidence of 95%. Similar to the training score, an agent is able to learn if it learns policies with an expected return and a GRP higher than zero from some time point onward.

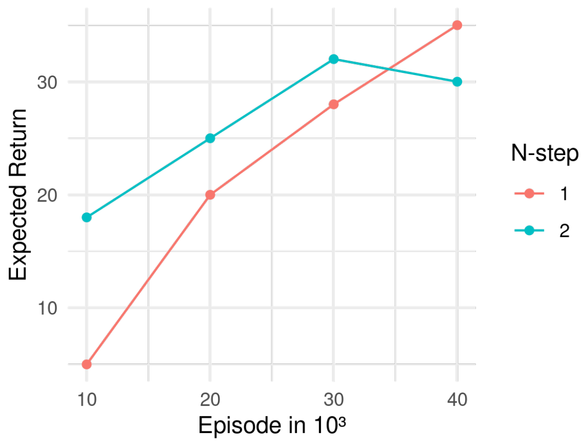
6.3 Second Research Question

Research questions RQ 2.1 and RQ 2.2 ask whether n-step learning leads to faster learning. To compare the learning of two agents, the quality of a learned policy is put in relation to the time needed to learn a policy. We say that an agent learns faster, or that learning improves, if it is able to obtain a better policy in the same amount of time as another agent. There are two timescales we use for this comparison: (1) the episodes needed during training to obtain a policy and (2) the execution-time needed for the training to finish. This is also reflected in both research questions because RQ 2.1 asks if sample efficiency increases with higher n-step, which consequently uses episodes as timescale, while RQ 2.2 asks if agents can learn policies faster concerning training time. The quality of a policy can also be classified using two different measures: (1) The average expected return and (2) the goal-reaching probability. In both cases, higher values are better. Due to this, whenever we say learning improves it must be denoted which timescale and performance measure is used.

To answer RQ 2.1, first a definition of sample efficiency is needed. If an agent learns better policies than another agent given a fixed amount of episodes to train on, we say that it is more sample efficient because it can use the provided data in a more efficient way for learning good policies. In the ideal case, every agent considered finds an optimal policy. Thus, the average returns received during training should converge eventually. This should also be noticeable using DSMC as evaluation method, where the expected return or expected GRP of the extracted policies converge. Thus, we can also say that sample efficiency increases if the policy of an agent converges faster than the policy of another agent. This definition can only be applied if both policies converge to the same limit. Research question RQ 2.2 is reasoning about a similar concept to sample efficiency, where we want to check whether an agent learns better policies than another agent given a fixed amount of time. We call this concept time efficiency in this thesis and note that this is not a common term in DRL literature to our knowledge. To be able to reason about learning improvements more precisely, we define that learning has improved *strictly* if it has improved for each extracted policy found until convergence. If learning only improves on a training section, e.g. the first $30,000$ episodes, we say that training improves *partially*. An example is given in Figure 6.1. Note that learning can improve partially while it may not improve in total. This allows a more fine-grained analysis of the results. The same definition is applied when using training returns as evaluation measurement instead of DSMC evaluations. We say learning improves strictly if the average return is constantly higher during training until convergence compared to the training of another agent. If the average return is only higher during some time periods, we say that learning improves partially. Note that due to the high variance of the received training return we use a sliding mean for this comparison and similarly, the regression line for comparing DSMC results, again due to the high variance. The choice of the mean window and regression function can thus influence the results, but due to the high variance of the measurements, the definitions provided would make no sense otherwise. We fix the window size of the sliding mean as well as the regression function used before evaluating any experiment (see Section 6.1.2) to make sure that both are not chosen in favor of desired results.



(a) Learning strictly improved with regard to expected return and episodes trained.



(b) Learning improved partially for the first 30,000 episodes but not in total.

Figure 6.1: Example plots to distinguish whether we say that learning has improved in total, strictly or partially. In this case improvement is analyzed with regard to the average expected return and the episodes trained for the same agent on the same environment using different n-step parameters.

Recall that this thesis introduces three new DBQL n-step agents. The first agent, denoted as DBQL^n_S, uses sample updates and a n-step replay buffer as proposed for RainbowDQN [2] (see Section 4.1). The second agent is DBQL$^n_{DP}$ (Section 4.2) which computes expected n-step updates using recursion. It stores only visited states in its replay buffer and computes the entire n-step update during learning. The third agent is DBQL^n_M (Section 4.3) which also uses n-step expected updates. It stores all n-length transitions for each state observed in its buffer while acting with the environment. This is done to save time when computing the expected n-step return during learning. This agent only approximates the Bellman target update and suffers from overestimation bias (see Section 4.3 for more details). Due to this, it is unclear if this agent can learn at all.

To investigate the effect of n-step learning on sample efficiency (RQ 2.1), the n-step agents are trained on Cdrive and Racetrack for a fixed amount of episodes. Since the DQN agent of the MoGym paper [8] is capable of learning an optimal policy within $10,000$ episodes on these environments using an ϵ-greedy exploration strategy and DBQL is an improvement of DQN, the training episodes are fixed to $10,000$ and the n-step agents also use ϵ-greedy for exploration. To test whether time efficiency improves (RQ 2.2) the agents are trained on the same environments but for a fixed amount of time instead of a fixed amount of episodes. Deciding on a timeout for the experiments is not trivial because the n-step agents are expected to differ significantly in the average execution time needed to train on a single episode. For example, the matrix agent DBQL^n_M is designed to have a faster target computation than DBQL$^n_{DP}$ and the sample agent DBQL^n_S is not using the known transition function to plan ahead which should boost its training speed for a single episode. However, since n-step DQBL agents should improve on their plain (1-step) DBQL counterpart (either with sample or expected updates), we decided to train the base DBQL algorithms on both environments and use the time needed for these agents until their policies converge. The aim of the n-step agents is to converge faster than their non n-step base algorithm.

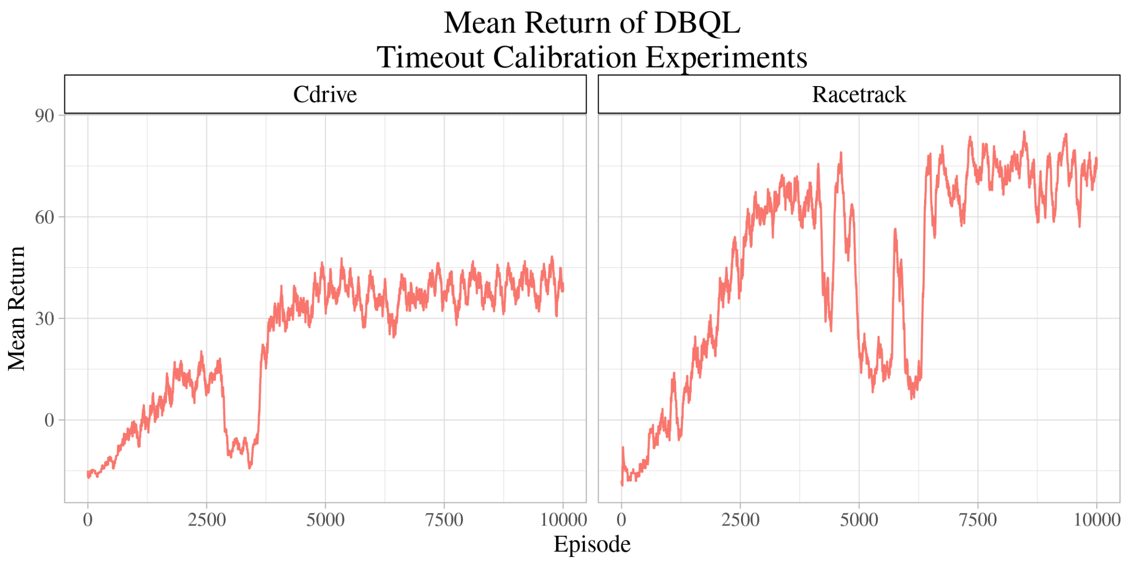


Figure 6.2: The training returns (using sliding moving average) of DBQL using expected updates on both Cdrive and robustification. These plots are used to set a timeout for DBQL$^n_{DP}$ and DBQL^n_M.

Both DBQL$^n_{DP}$ and DBQL^n_M are based on DBQL using expected updates. To figure out the timeout on each environment for both agents the base DBQL agent is trained on Cdrive and robustification for $10,000$ episodes on a single seed, extracting the execution time every 500 episodes. Figure 6.2 plots the mean training returns received by the base agent on Cdrive and Racetrack. For Cdrive, the returns have converged after approximately 5000 episodes which equals an execution time of about 283 seconds. On Racetrack the returns converge after approximately 7500 episodes which equals an execution time of 1404 seconds. For both the matrix (DBQL^n_M) and the DP agent (DBQL$^n_{DP}$) the timeout experiments use these convergence times as threshold on Cdrive and Racetrack respectively.

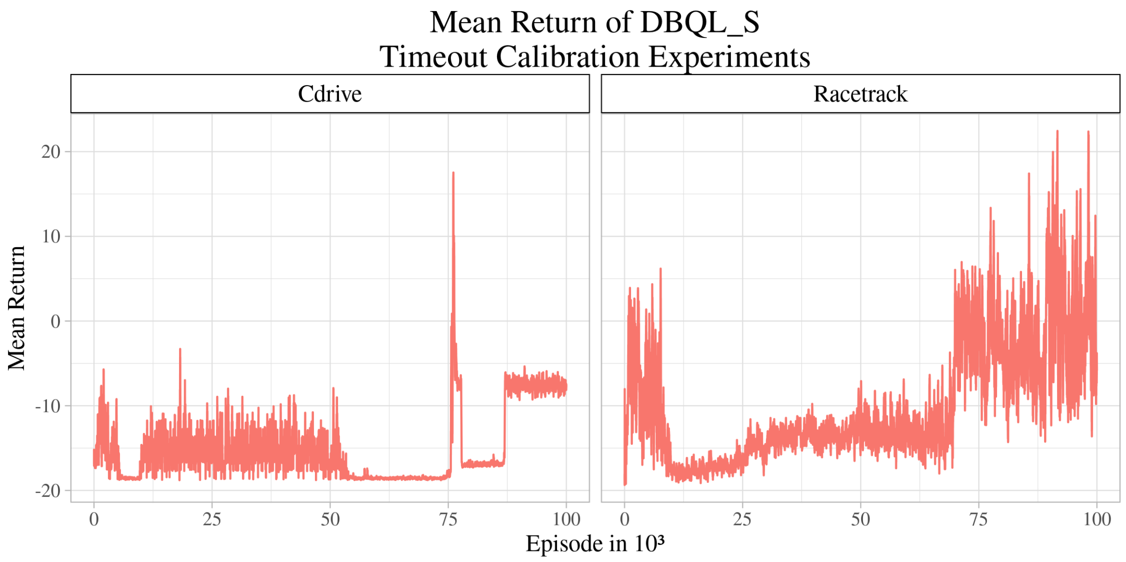


Figure 6.3: The training returns (using sliding moving average) of DBQL$_S$ using sample updates on both Cdrive and Racetrack. The plots are used to set a timeout, but show that the agents were only able to learn a suboptimal policy.

The remaining n-step agent DBQL^n_S is based on DBQL using sample updates and thus this agent is trained on both environments to infer which timeout to use. The agent is not able to learn on both environments even after 100, 000 episodes. Figure 6.3 shows the average training returns of the sample agent on both environments. While it shows that returns are increasing, the agent has not learned a good policy and therefore needs even more episodes to train on. We decided to drop the tests for the sample agent due to two reasons: (1) the time needed for the base algorithm to learn and (2) the novelty and usefulness of this approach. As DBQL^n_S is the only n-step agent proposed, which is based on an already existing n-step DQN counterpart (RainbowDQN [2]), and it does not make use of the known transition function which makes n-step learning interesting in this context, we don't think the possible insights of the experiments justifies the amount of time needed to execute them. Consequently, all experiments for DBQL^n_S are dropped. The other two agents are trained on each environment using three n-steps ($n \in 1, 2, 3$) and for each n-step on three random seeds. We decided to limit the n-steps to three as the DBQL$^n_{DP}$ agent is expected to become very slow with higher n-steps. Additionally, the n-steps used are enough to reason about the effect of higher n-steps to sample and time efficiency.

The returns received during training are stored for every experiment. In case of the sample efficiency experiments, policies are extracted every 500 episodes and for the timeout experiments the policy extraction time is chosen such that 25 policies are gathered each experiment. Additionally, timers keep track of the average execution times needed for various parts of the algorithms. Both training returns and extracted policies are used to infer if the novel agents are able to learn and to test if sample or time efficiency improves depending on the experiment set used. The timers can help reasoning about time efficiency differences by outlining the time needed for specific parts of the agents' algorithm depending on the n-step parameter.

6.4 Summary

To summarize the experiments, for each research question some DRL limitations are addressed. This includes using hyperparameter tuning, using the same neural network for every agent and using the same reward structure for each experiment for comparability. To make the experiments replicable each agent is trained on each environment with multiple fixed seeds for the random generators. In addition, since the time and the resources for this thesis are limited, experiments are run only on a subset of environments and the results are generalized. On a high level, the experiments are split into two sections: the exploration experiments used to answer RQ 1.1 and RQ 1.2 and the n-step experiments used to answer RQ 2.1 and RQ 2.2 where the latter experiments are again split into experiments for each research questions by either using a timeout or by training for a fixed amount of episodes. Figure 6.4 provides an overview of all experiments made for the evaluation chapter.

Exploration Experiments Summary

Environment:	Cdrive, Racetrack
Agents :	DBQL + {ICM, RND, ϵ-greedy, NoisyNet, Stochastic Go-Explore + {ICM/RND}}, DQN + ϵ-greedy
N-step:	1
Seeds:	1,2,3
Data:	Training returns, policies
Time-scale:	Episodes
Number of Episodes:	100,000
PET:	1,000
Timeout:	None
Research question:	RQ 1.1 and RQ 1.2

N-step Experiments Summary

Environment:	Cdrive, Racetrack
Agents :	DBQL^n_{DP}, DBQL^n_M
N-step:	1
Seeds:	1,2,3
Data:	Training returns, policies, timers

Timeout vs No-Timeout experiments

Research question:	RQ 2.1
Timescale:	Episodes
Episodes:	10,000
Policy extraction:	every 500 episodes
Timeout:	None

Research question:	RQ 2.2
Timescale:	Time
Episodes:	-
PET:	**Cdrive:** 11s **Racetrack:** 56s
Timeout:	**Cdrive:** 283s **Racetrack:** 1404s

Figure 6.4: Summary of all experiments made for this thesis.

Chapter 7
Results and Evaluation

This chapter provides an overview of all findings of the experiments described in the methodology chapter. The findings are used to answer the research questions while the next chapter discusses the relevance of the findings and draws final conclusions. The chapter is structured to answer the research questions in order, similar to the methodology chapter. Consequently, first the results of the exploration experiments are presented and discussed answering RQ 1.1 and RQ 1.2. Then the findings of the n-step experiments are outlined, starting with the experiments designed to test the effect of varying the n-step parameters on the sample efficiency (RQ 2.1) and followed by the n-step timeout experiments which enable reasoning about time efficiency of the n-step approaches (RQ 2.2). Finally, the timer data is used to gain further insight in the reasons of the respective time efficiency of the algorithms.

7.1 Action-space Explosion and Exploration

Recall that the thesis aims to identify the challenges of hard QVBS benchmarks which made learning for DQN impossible in the MoGym paper [8]. More precisely, the thesis hypothesis is that the two main problems are the action space explosion and the lack of sophisticated exploration techniques, which results in the first two research questions:

Research Question 1.1 Is the action-space explosion the main factor for the unsolvability of some planning tasks?

Research Question 1.2 Is exploration a key factor when learning fails, even if the agent accounts for the action-space explosion?

The objective of the thesis, after establishing the main problems of hard instances, is to provide the reader with tools to overcome these obstacles by suggesting and improving on existing approaches. Due to this, DBQL is used as a DQN extension addressing the action-space explosion and is mixed with several exploration techniques, one of which is a newly adapted version of Go-Explore. With this new Stochastic Go-Explore the DRL community is provided a powerful and less confined version of the Go-Explore framework for which we hope to verify its feasibility. To test the hypothesis and Stochastic Go-Explore, we trained agents on three environments for $100,000$ episodes. The results for each environment are analyzed separately, first using DSMC and then using training returns.

7.1.1 DSMC Evaluations

Figure 7.1, 7.2 and 7.3 plot the results for each agent on each environment for each seed. DSMC is used to evaluate the mean return achieved by the extracted policies every 10,000 episodes. Similarly, Figure 9.2, 9.3 and 9.4 plot the same data but using GRP as DSMC evaluation criteria and can be found in the appendix. Both criteria show a high correlation. Due to this, in the following, whenever the score plots are referenced, the same effect can also be observed in the plot of the GRP counterpart. The analysis confines itself mostly to the plots using the return criteria such that the GRP plots can be put in the appendix to prevent them from cluttering this chapter due to their size.

Elevators Figure 7.1 shows that on Elevators the DQN agent using ϵ-greedy as well as the ICM, RND and NoisyNet DBQL agents are not able to learn. In contrast, both Stochastic Go-Explore variants can learn on this environment. DBQL using the simplest exploration technique, ϵ-greedy, is also successful while the plain DQN agent is not. This is a first indicator that the action-space explosion is a major challenge for DQN, because DBQL uses the same exploration as DQN but mitigates the effect of large action spaces. Since all other agents are based on DBQL and add a more sophisticated exploration technique than ϵ-greedy, it is rather surprising that only the new Stochastic Go-Explore agents can learn on Elevators. Moreover, plain DBQL has on average better policies, meaning they achieve higher average returns (and GRP), than the Go-Explore variants.

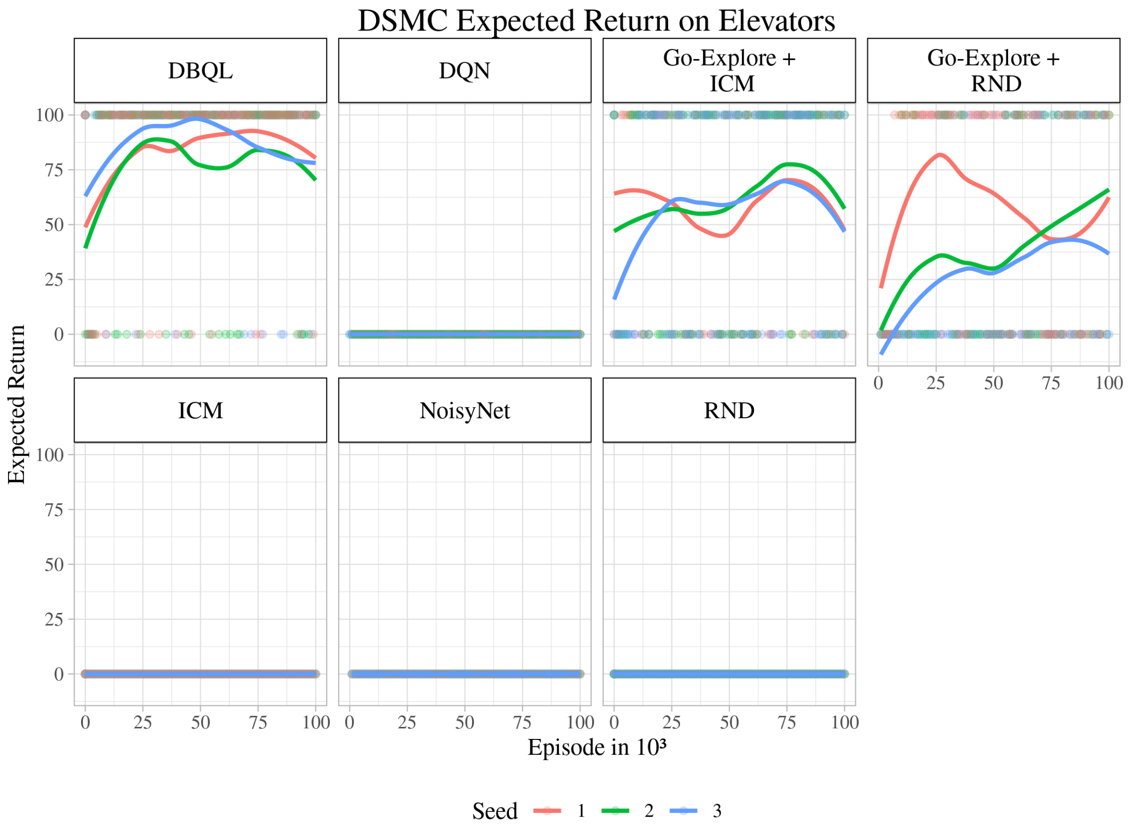


Figure 7.1: DSMC evaluation of experiments on Elevators using expected return as criterion.

Firewire The plots for Firewire presented in Figure 7.2 (and 9.3) show that all agents except the DQN agent can learn successfully on this environment. Compared to Elevators, Firewire has a much larger state-space with 290017 states compared to the 909 states of Elevators and a lower action-space of 24 to 41 actions which makes the findings of Elevators seem surprising at first thought. Intuitively, Firewire is a much harder exploration problem due to the larger state-space and thus if ICM, NoisyNet and RND can learn successful here they should be able to learn on Elevators as well. Note that this interpretation is error-prone. Firstly, a larger state-space does not equal a harder exploration problem if goal states are plentiful and easy to reach and secondly, it is not trivial to understand how intrinsic motivation agents like RND and ICM behave on different environments. We conclude from the results that Firewire is an easier exploration problem than Elevators but the improvements of DBQL over DQN are the significant factor that enable learning on both environments. In other words, the invariance of DBQL to the action-space explosion seems to be a crucial factor to be able to learn on some instances of the QVBS which affirms research question RQ 1.1.

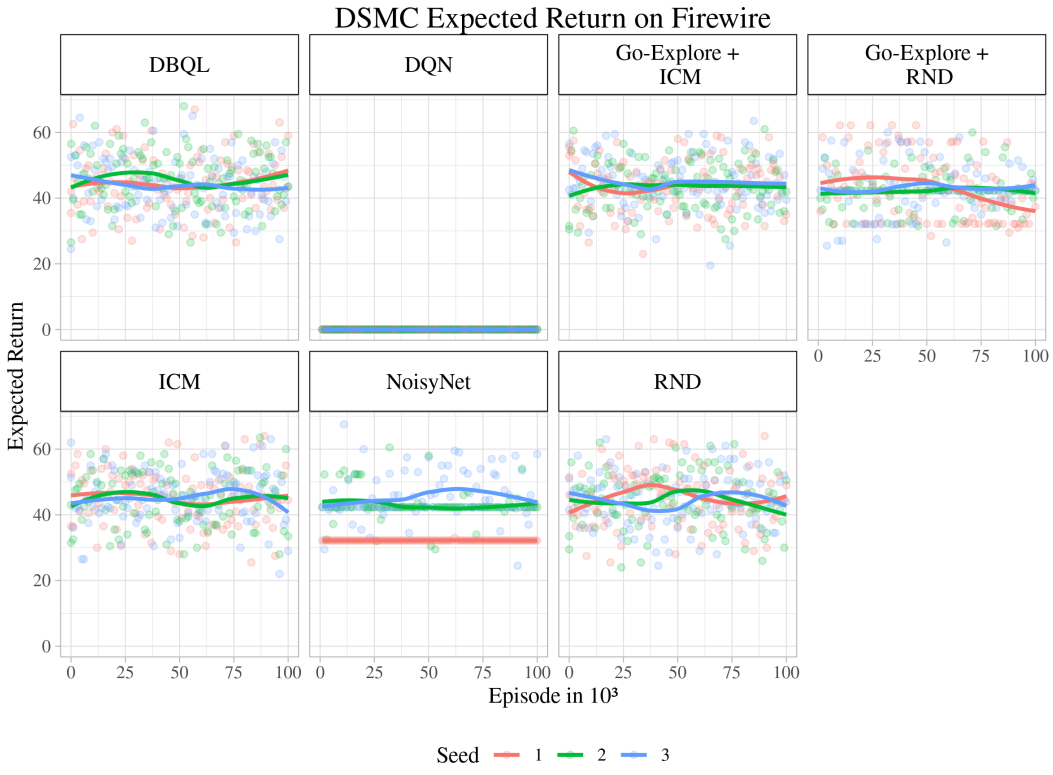


Figure 7.2: DSMC evaluation of experiments on Firewire using expected return as criterion.

Blocksworld Figures 7.3 and 9.4 show that only Stochastic Go-Explore using ICM as prioritizing heuristic is able to learn on Blocksworld. This indicates that exploration is a key factor when learning fails, even if the agent accounts for the action-space explosion, and thus confirms research question RQ 1.2 because in this case both DBQL and DQN using the same exploration technique were not able to learn. In addition, since Stochastic Go-Explore using ICM is able to learn successfully on all three environments we can confirm that this newly proposed variant of Go-Explore remains a powerful and feasible exploration technique, even though it drops a core principle of the original framework. However, by observing that for one random seed Stochastic Go-Explore cannot learn, it seems that the success of the agent is still affected by the randomness of the environment and the exploration method used.

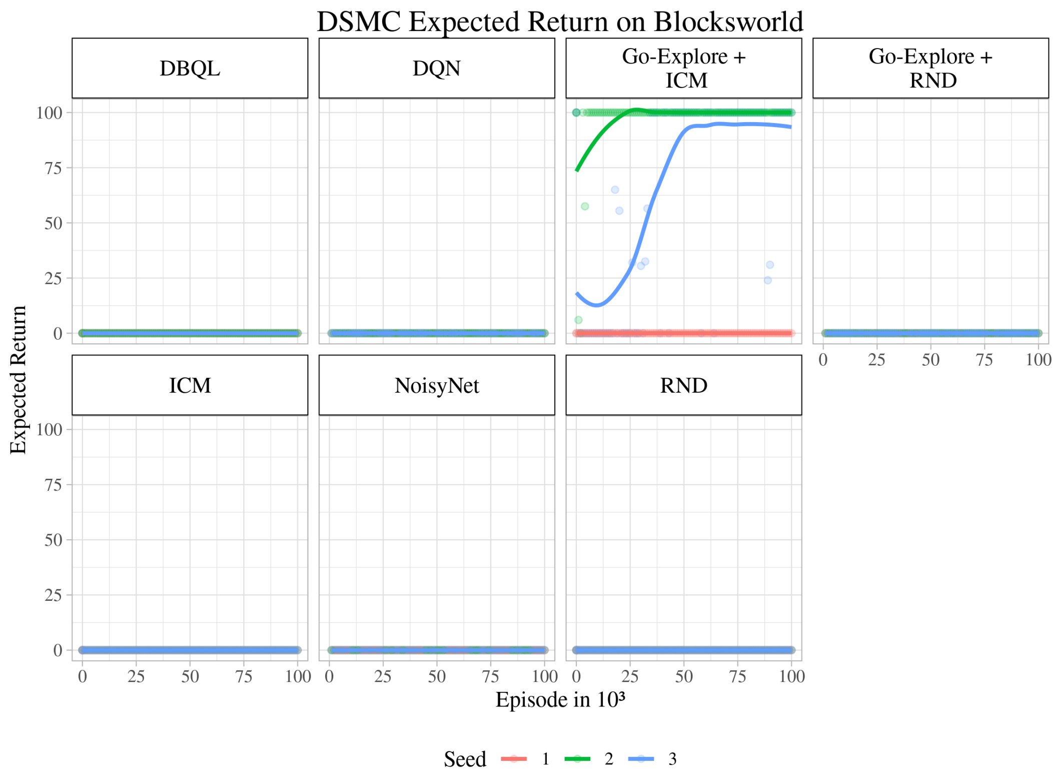


Figure 7.3: DSMC evaluation of experiments on Blocksworld using expected return as criterion.

7.1.2 Training Returns

As mentioned in the methodology chapter in Section 6.1.2, DSMC is a new technique for DRL analysis and there can be a discrepancy between the DRL agents goal to find policies that maximize the expected return and the goal to learn the actual abstract problem. This discrepancy occurs if the used reward structures are poorly designed. Thus, in this thesis we value the training returns higher than the DSMC evaluations if they show different results. Recall that a sliding mean (SMA) is applied to the training returns to analyze the training performance. Figures 7.4, 7.5 and 7.6 show the mean returns received during training on an environment for each agent and seed. This is similar to the DSMC plots prior.

Elevators By comparing the training return plots for Elevators of Figure 7.4 to the DSMC plots of Figure 7.1 using the score criteria, the results are mostly the same. Still, only the DBQL agent using ϵ-greedy as well as both Stochastic Go-Explore agents can learn on the environment. While the DSMC evaluations suggest that the DBQL agent has learned the policies with the highest expected return, the training returns show that the RND variant of Stochastic Go-Explore converges to the same, if not slightly better, average training return. It is not trivial to infer the meaning of this finding for both analysis methods. One possible explanation is that the DSMC regression line is fitted to a sparse data vector with only 100 DSMC evaluations per experiment which might be insufficient to show the training performance accurately. Another theory is that policies can change quickly and that the neural network is very unstable during training, such that DSMC evaluation results suffer from a high variance. Extracting and evaluating more policies would help to plot potentially smoother training curves.

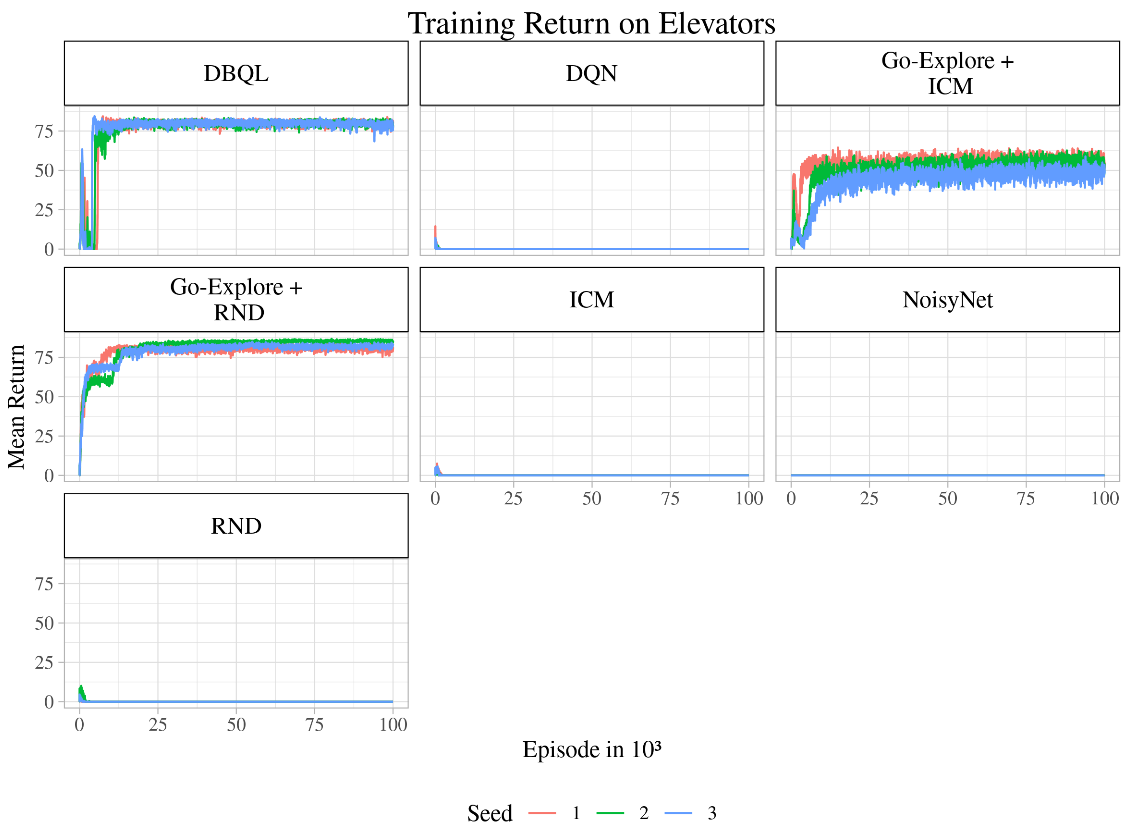


Figure 7.4: Training returns of experiments on Elevators.

Firewire For Firewire the training return plots (Figure 7.5) and the DSMC plots (Figure 7.2) show very similar results. However, while the extracted policies of DQN are estimated to have a mean return of zero using DSMC, the agent constantly receives a training score above zero. This could be a result of exploration in the beginning leading an agent to receive positive returns without having learned anything. However, as the exploration factor epsilon decays until the agent exploits its learned policy with probability 0.95 in each step, the constant positive returns at the end of training suggest that exploration is likely not the main cause of this discrepancy (especially if the environment is hard to explore). Furthermore, the DSMC estimations could be too inaccurate. Another observation is that the estimated expected returns for the extracted policies tend to be higher than the received rewards during training. For both Stochastic Go-Explore variants, the training returns converge to be between about 15 and 25 while the DSMC evaluations indicate that the policies learned to achieve an estimated average return between 35 and 60, or on average slightly above 40. This further indicates that the training score can be an inaccurate measure for the quality of the learned policies and learning in general, as mentioned in Section 6.1.2. The likely cause for a lower mean training score is exploration, as agents are forced to explore during training while for DSMC, only the learned policy is exploited.

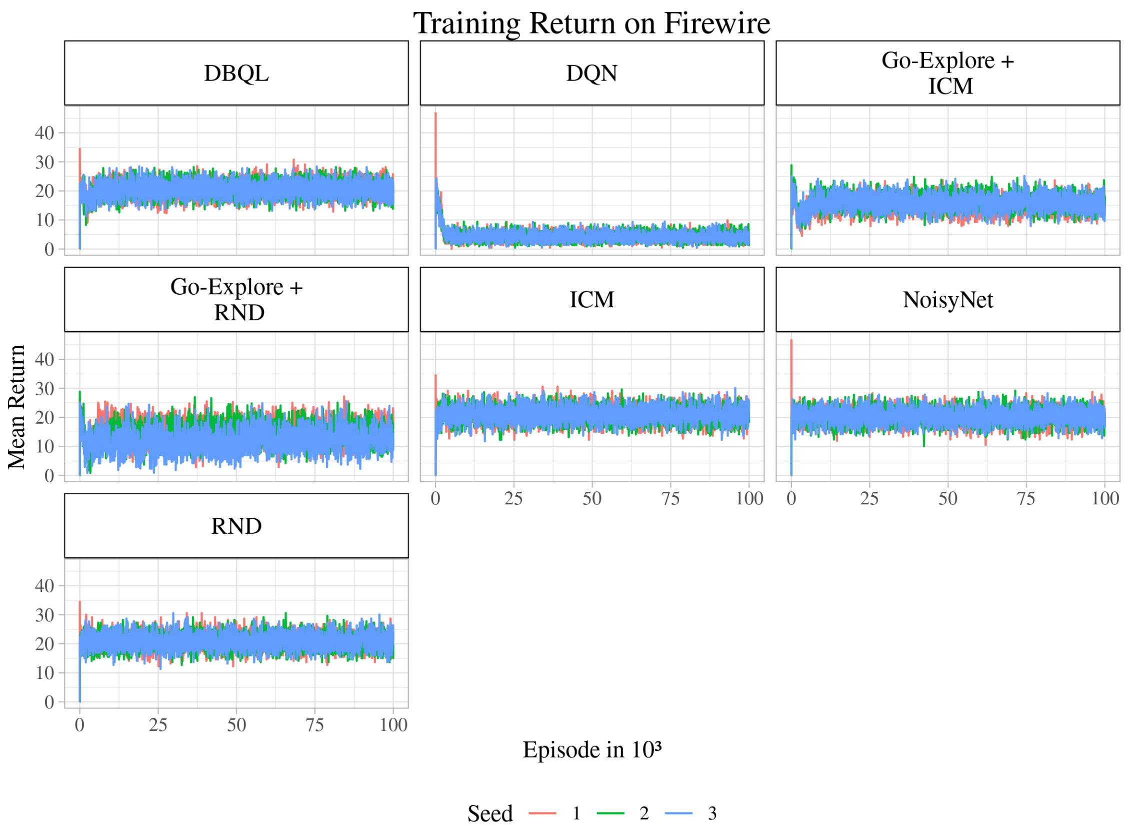


Figure 7.5: Training returns of experiments on Firewire.

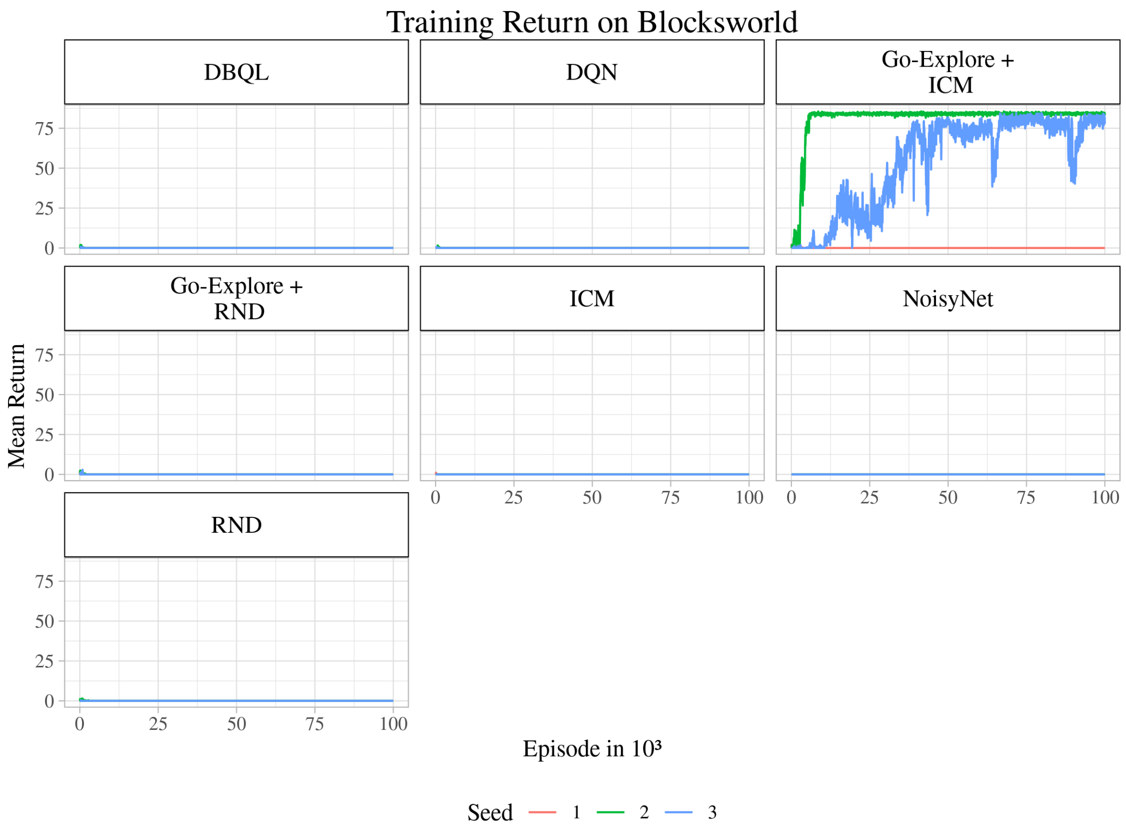


Figure 7.6: Training returns of experiments on Blocksworld.

Blocksworld Comparing the training return plots of Figure 7.6 with the DSMC plots of Figure 7.3, no differences in the results can be identified. Both DSMC regression lines and the mean training return lines match well except that, as already mentioned for Firewire, DSMC estimations converge to a higher expected return than the mean training returns.

Negative Rewards The reward function used for training specifies that an agent receives a positive rewards of 100 if a goal state is reached and -20 if a bad state or a dead end is reached. For any intermediate step the reward is zero. For all exploration experiments, observe for all DSMC plots that no policy is estimated to have a negative expected return nor do any training return plots show returns below zero, although rewards were set correctly for the experiments. Consequently, the only explanation is that the agents did not reach dead or bad states during training, they simply could not reach a goal state in the worst case. This makes the problems even more difficult because useful feedback by rewards being different from zero is even more sparse compared to environments were agents can get negative feedback often, e.g. on Cdrive and Racetrack for the n-step experiments.

7.1.3 Conclusion

Independent of the performance measure used, be it DSMC using expected returns, GRP or the extracted training returns, the same conclusion can be made. First, research question RQ 1.1, asking whether the action-space explosion is a major factor which leads to DQN not being able to learn on some QVBS benchmarks, is confirmed. The experiments show that for Firewire and Elevators the DQN agent is not able to learn in contrast to the DBQL agent using the same exploration technique as DQN. Since DBQL is a DQN extension designed to mitigate the performance dependency on the size of the action-space, we conclude that this must be the main reason for its success in contrast to plain DQN. Research question RQ 1.2, asking whether exploration is a key factor when learning fails, even if the agent accounts for the action-space explosion, is also confirmed. This can be observed with the experiments on Blocksworld, were only Stochastic Go-Explore using ICM as prioritizing heuristic is able to learn. Since for all environments at least one DBQL agent is able to learn while DQN cannot learn on any environment, we accept our hypothesis that the two main challenges for hard QVBS instances are the action-space explosion and exploration. Furthermore, we demonstrated that Stochastic Go-Explore remains a feasible and powerful exploration framework. Apart from the research questions, the experiments also hint that tracking training returns and evaluating policies using DSMC show similar results, but the mean training score tends to be lower than the estimated returns using DSMC. This can be explained by the agents constant need for exploration during training and which results in suboptimal behavior. In contrast, DSMC measures the expected return by exploiting the learned policy completely without any added exploration applied. Note that a proof for this hypothesis remains for future work.

7.2 N-step Experiments

One objective of this thesis is not only to apply existing approaches to the problem of hard instances of the QVBS, but also to improve on them. Although Description-based Q-learning itself is a new algorithm, the thesis extends it to use n-step learning by providing three new algorithms. Apart from testing their feasibility, the potential advantage in sample efficiency is tested while their usefulness in regard to training time is examined by answering the following two research questions:

Research Question 2.1 Does n-step learning increase sample efficiency?

Research Question 2.2 Does n-step learning lead to faster learning with regard to training time?

In this section, the n-step experiments with a fixed number of episodes for training are evaluated to test whether sample efficiency increases with higher n-steps (RQ 2.1). Next, the experiments with a timeout are evaluated to check if n-step agents can learn faster than their 1-step counterpart or if the 1-step base agents are more suitable for training in a time-constraint setting. For both experiment sets, first DSMC evaluation is used for the analysis and then the extracted training returns are used. Finally, additional insights into the time efficiency and affects of the n-step parameter are presented using the timer data gathered during training of all n-step agents.

7.2.1 Sample Efficiency

Recall that DBQL^n_{DP} and DBQL^n_M are trained on Cdrive and Racetrack for $10,000$ episodes with n-steps $1, 2$ and 3 to test the sample efficiency gain of the algorithms related to the n-step parameter. Further recall that sample efficiency increases if an agent learns better policies than another agent given a fixed amount of episodes to train. Additionally, we say that sample efficiency increases if the policy of an agent converges faster than the policy of another agent. Similar to the exploration experiments, the DSMC plots using expected return as measurement and using GRP correlate well together, which is why the GRP plots can be found in the appendix and are only referenced here for the interested reader. Every insight gained using the DSMC plots with the expected return criterion can also be gained by analyzing the GRP plots. While the interested reader can verify this, this section is much more readable if the GRP plots are not shown here.

Figure 7.7 plots the DSMC evaluation using the expected return criterion for the fixed episode experiments on both Racetrack and Cdrive for each n-step and agent. Since for each agent and n-step three experiments were made (one for each random seed), each data point shown in the plot is the mean over the DSMC return evaluated for the corresponding policy of the three experiments. Consequently, the plots show a mean performance of an agent for the specified n-step on each environment. The corresponding GRP plots are shown in Figure 9.5 in the appendix.

Looking at DBQL^n_{DP} on Racetrack, the policies for all n-steps converge and the policies with a higher n-step converge faster. Thus for the DP agent on Racetrack, sample efficiency increases. For the matrix agent on Racetrack, the policies do not converge within the trained $10,000$ episodes. However, with increasing n-step learning improves for the first 3000 episodes which indicates that sample efficiency also increases for the matrix agent. On Cdrive, learning

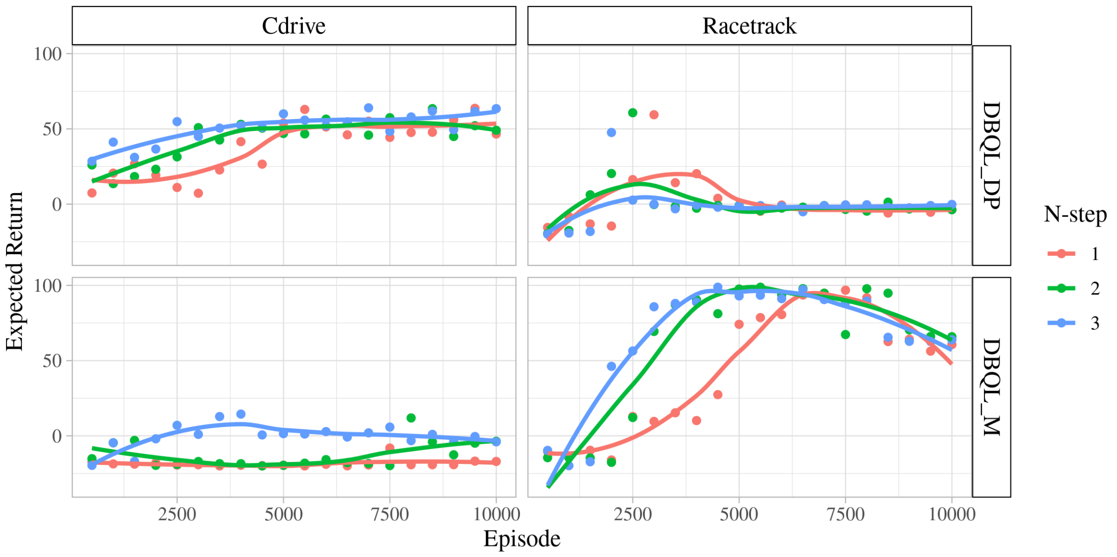


Figure 7.7: Mean expected return of both n-step agents plotted by n-step and environment, estimated using DSMC on the extracted policies.

improves for each agent with increasing n-step. While learning improves nearly strictly for the DP agent except at the very end of training between n-step one and two, training improves only partially for the matrix agent because for n-step two the agent is not performing better than the 1-step matrix agent between episodes 3000 and 6000. That said, the experiments confirm that sample efficiency does increase with higher n-steps for both agents, thereby confirming research question RQ 2.1.

An interesting observation is that DBQL^n_{DP} learns better policies than DBQL^n_M on Cdrive, while the opposite is true for Racetrack. It seems rather surprising that the DP agent's policies converge to zero because the agent has found policies with a higher expected return at the start of training. In addition, the DQN agent of MoGym seems to be able to learn better policies and DBQL is thought to be an extension of DQN. This could indicate that DBQL does not always outperform DQN. A precise answer to this is difficult because the experiments of this thesis do not use the same random seeds and the exact same ϵ-greedy parameters as in MoGym. Further investigations are needed to support this claim, which is outside the scope of this thesis and its research aims. Apart from answering research question RQ 2.1, the experiments show that both proposed n-step agents are able to learn, which is especially interesting for the matrix agent because the target update does not follow the Bellman equations and a proof that the agent can converge to optimal policies is not given. However, at least on Racetrack the agent can find seemingly optimal policies, e.g. after about 5000 episodes of training with n-step three.

Figure 7.8 plots the returns received by the agents during training for each agent on each environment for every n-step. For each experiment a sliding mean is applied to the training returns first to account for high variance. The plots show the average return over all three seeds for each episode. This is done by taking the mean return for each corresponding return of every seed by episode. Looking at the results on the plots for Racetrack first, the mean training return converges faster the higher the n-step for both agents. This confirms the results of the DSMC evaluation and thus we say that on Racetrack sample efficiency increases, noting that the effect is more noticeable for the DP agent. On Cdrive, tendency of increased sample efficiency with higher n-steps can also be observed. However, for the DP agent learning improves only partially. For the Matrix agent, the policies seem to converge to different levels but it is

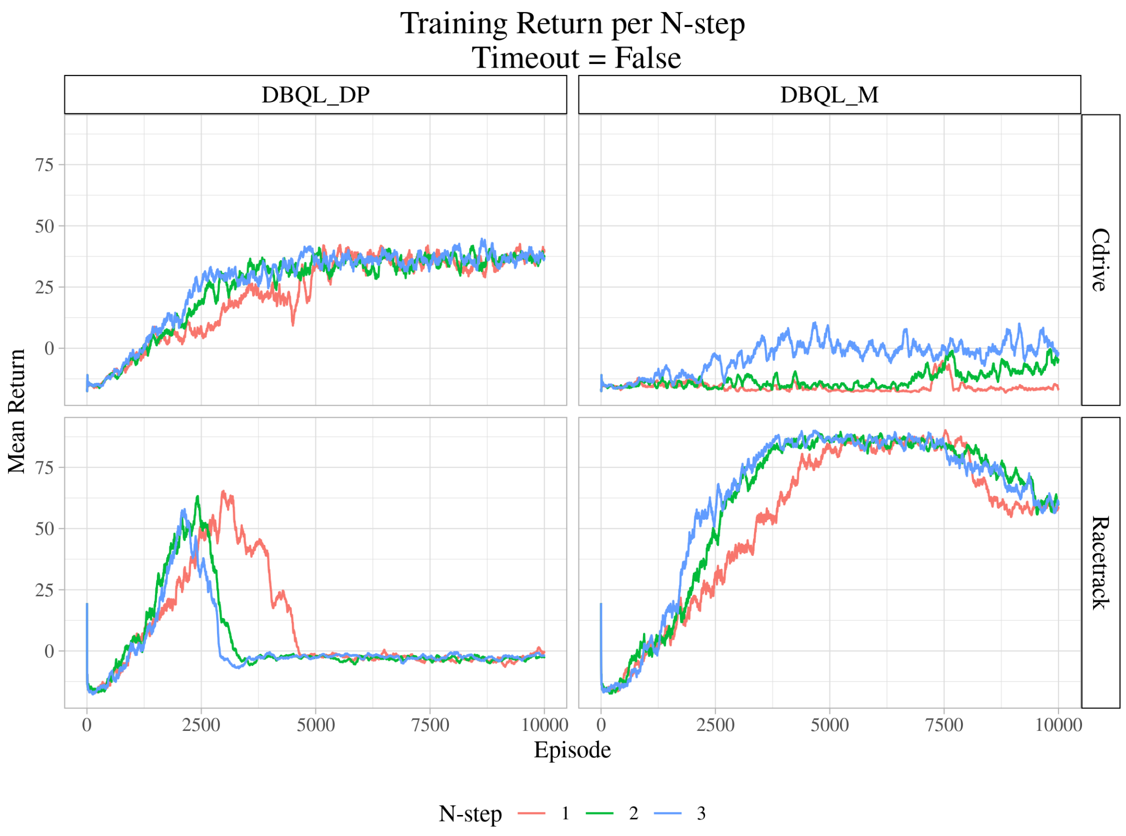


Figure 7.8: Average training return of both n-step agents on both Cdrive and Racetrack plotted for every n-step for fixed episode experiments. The returns are averaged over the random seeds.

uncertain if the policies converge to the same level when trained for more episodes. All in all the training return plots seem to correlate well with the DSMC plots but the same issue as noticed for the exploration experiments can be seen: the training returns converge to lower limits in the training plots compared to the DSMC plots. Again, the difference is likely due to the agent being forced to keep exploring during training while for DSMC the learned policies are exploited fully. To summarize, the training returns support the findings using DSMC that sample efficiency increases with higher n-steps in general.

7.2.2 Time Efficiency

To test whether timeout efficiency increases or decreases for the proposed n-step agents, experiments on Cdrive and Racetrack were made where the agents could train for a fixed amount of time instead of a fixed amount of episodes. Recall that we say that time efficiency increases if an agent learns better policies than another agent given a fixed amount of time. Research question RQ 2.2 is confirmed if time efficiency increases. If the aim is to learn good policies as fast as possible, increasing sample efficiency is only beneficial if the benefit of needing fewer episodes to train on to achieve policies of the same quality is not mitigated by the increase in training time needed to train a single episode. By answering RQ 2.2, we aim to reason about this trade-off and provide advice when to use the newly introduced n-step agents. In an optimal case, higher n-steps should not only induce a higher sample efficiency but also a higher time efficiency.

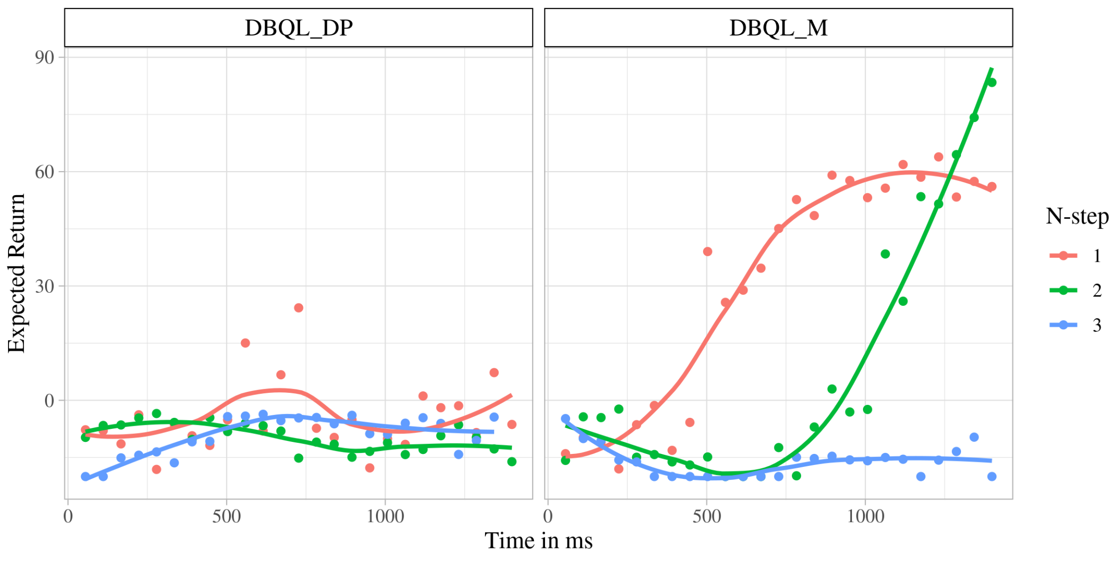


Figure 7.9: Mean expected return of both n-step agents plotted by n-step on Racetrack, estimated using DSMC on the extracted policies. The agents trained for a fixed amount of time.

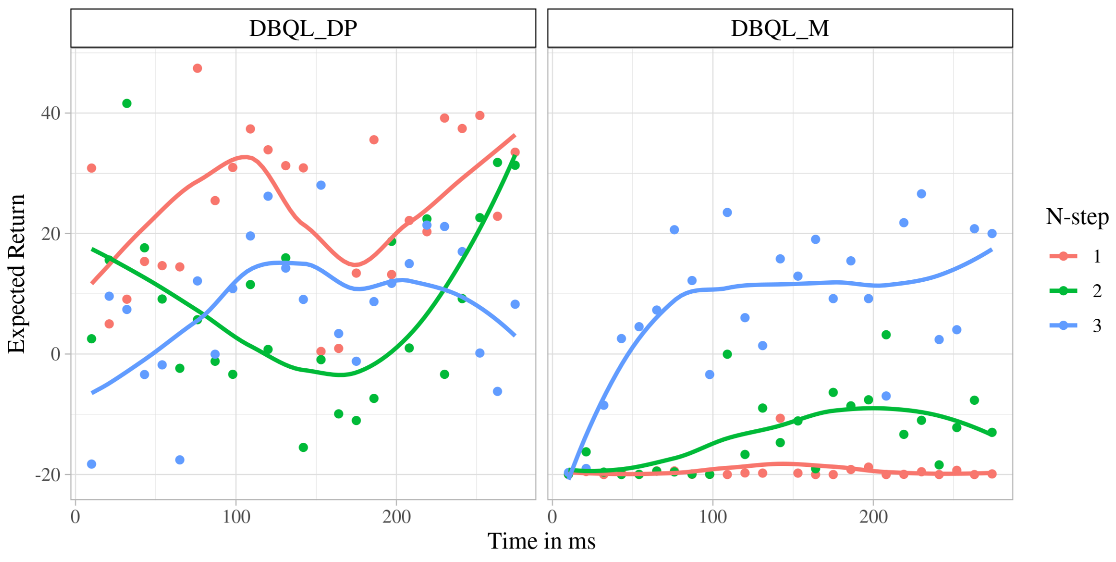


Figure 7.10: Mean expected return of both n-step agents plotted by n-step on Cdrive, estimated using DSMC on the extracted policies. The agents trained for a fixed amount of time.

Figure 7.9 depicts the expected return, estimated using DSMC, for the extracted policies for each n-step, grouped by agent and on Racetrack. The corresponding plots for Cdrive are shown in Figure 7.10. The plots are split into two figures because a different timeout was used for the environments. For Racetrack, learning does not generally improve for both agents. For the DP agent, it seems that the 1-step variant has learned the best policies by the end of training, then the 3-step agent follows and the worst policies where found by the 2-step agents. This pattern is not the same for the matrix agents. Although the 1-step variant has learned good policies fast, the 2-step agent has learned better policies at the end of training while the three-step agent performs the worst by a large margin. A similar pattern applies to the DP agent on Cdrive, where the 1-step agent is also learning the best policies fast but the 2-step agent can catch up at the end of training. Again the 3-step agent performs worst. In contrast to this and in contrast to the matrix agents trained on Racetrack, the matrix agent on Cdrive time efficiency increases clearly with each n-step. Consequently, there is no general answer to research question RQ 2.2 as time efficiency might increase with each n-step on some benchmarks for the matrix agent while on others it does not. The experiments suggest that for the DP agent, choosing the 1-step agent seems to be the most time efficient, while for the matrix agent on Racetrack, the 1-step agent is also a good choice until a certain training time threshold. It appears that the specific timeout set also influences how to answer of RQ 2.2, again making a general answer difficult.

As mentioned above the DSMC plots using GRP as criteria correlate to the plots using the expected return criteria and can be found in the appendix (Figure 9.6 and 9.7). Figure 7.11 and 7.12 plot the mean training returns for the agents on Racetrack and Cdrive, again averaging over the random seeds for each n-step. The main observation is that the higher the n-step for each agent the fewer episodes could be trained until the timeout was reached. This effect is stronger for the DP agent as for the Matrix agent but it holds on both environments and for both agents. On the same environment, the matrix agent can train for much more episodes than the DP agent for the corresponding n-step. This is not surprising because the matrix agent is designed to have faster target computations than the DP agent. Moreover, on Racetrack the difference in episodes trained between each n-step seems to be bigger than for Cdrive for the Matrix agent. The higher action-space dimension of Racetrack could be the cause of this and will be investigated further in the next section where the available timer data is analyzed.

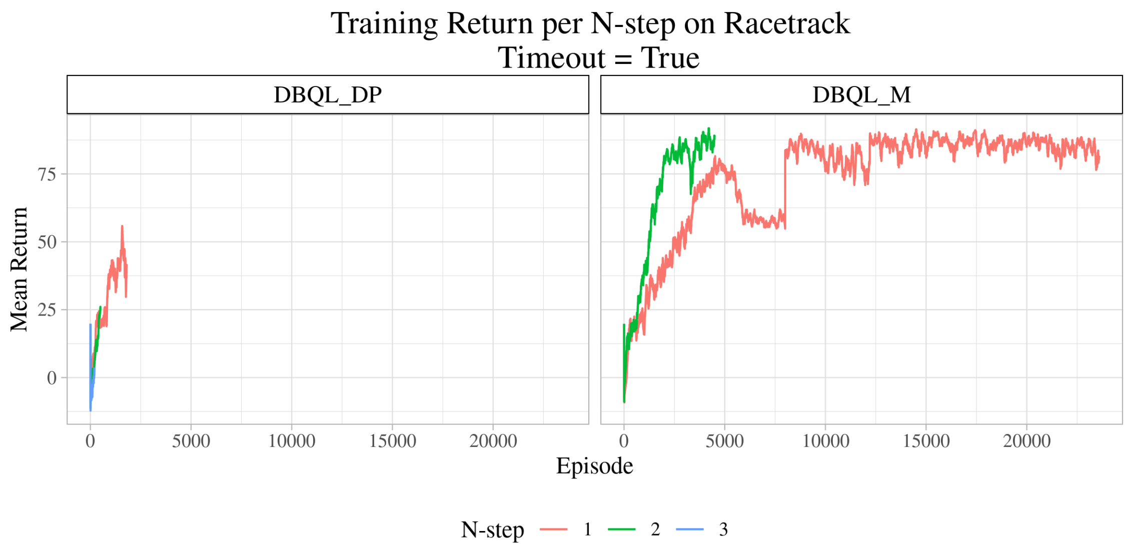


Figure 7.11: Average training return of both n-step agents on Racetrack plotted for every n-step for timeout experiments. The returns are averaged over the random seeds.

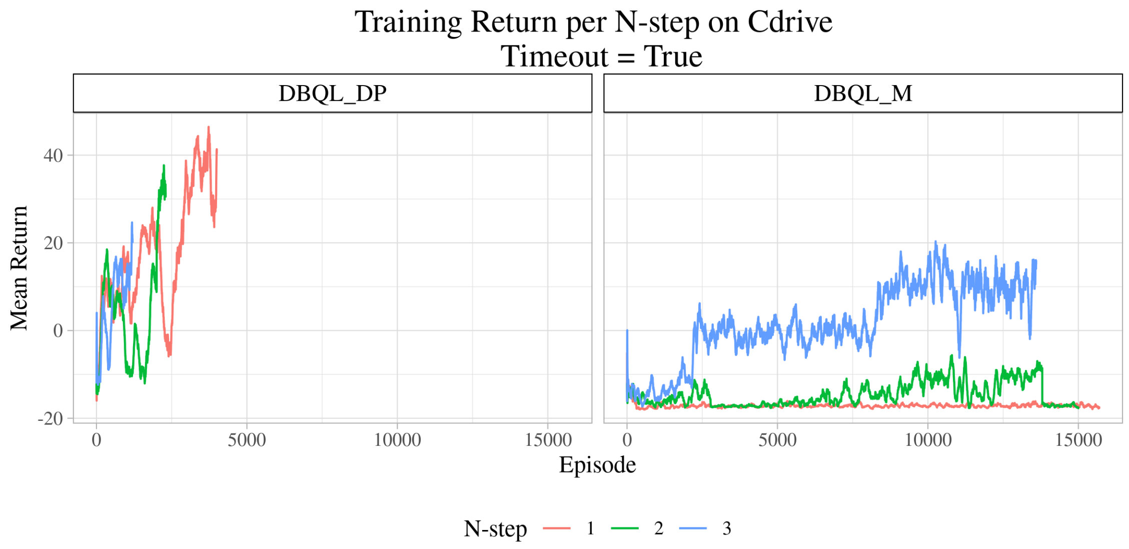


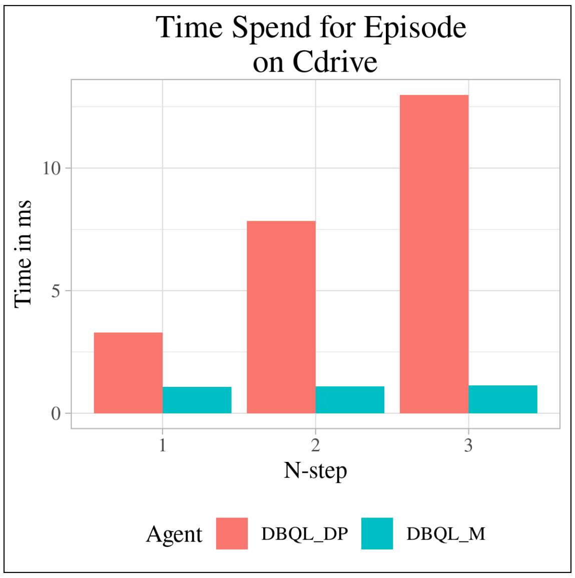
Figure 7.12: Average training return of both n-step agents on Cdrive plotted for every n-step for timeout experiments. The returns are averaged over the random seeds.

7.2.3 Conclusion

The experiments confirmed that sample efficiency does increase with higher n-steps. Nevertheless, answering research question RQ 2.2 remains difficult and no general answer can be given as learning does not generally improve with each higher n-step given a time-constraint. However, the experiments suggest preferring the 1-step DP agent over its higher n-step counterparts if sample efficiency is not as important as learning good policies as fast as possible. For the matrix agent, the 1-step agent is either a good or the worst option and the same applies for the 3-step matrix agent. If the matrix agent cannot be tested for each n-step on an environment, which is advised due to the inconclusive results, we advise using the 2-step variant.

7.3 Timer Data Analysis

In this section, the data gathered by multiple timers of the n-step agents is used to investigate the impact of the added complexity of increasing n-steps to the learning process in regard to training time. The aim of this is to get a better understanding of the differences between the DP and the matrix agent and to infer when to use which n-step if training time is a limiting factor. Note that the research questions are answered in the previous sections and that this section just provides additional insights into the newly introduced n-step agents. If not stated otherwise, all plots of this section are created only with the n-step experiments with a fixed amount of training episodes. The reason for this is the usage of buffers in both agents which affect the training time. In the timeout experiments buffers are not used to their full potential which would affect the results of this section. The interested reader is referred to the appendix, where this issue is outlined in more detail in Section 9.5.



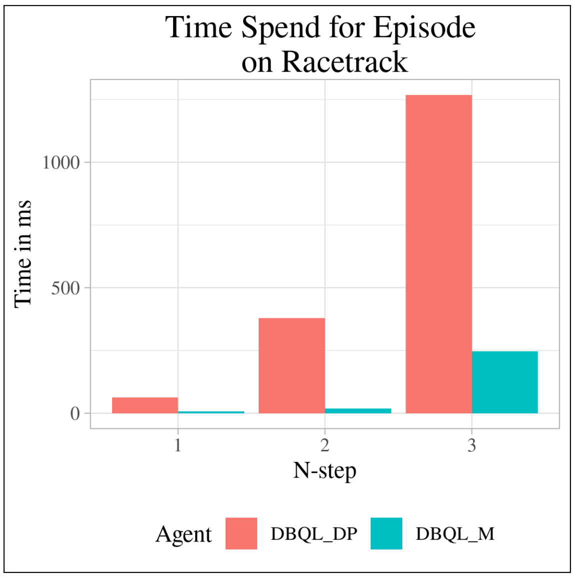


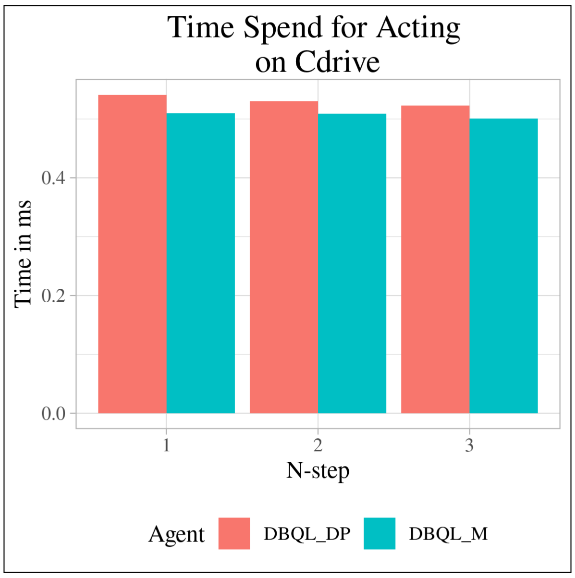
Figure 7.13: Bar charts plotting the average execution time of both n-step agents for training on a single episode, dependent on the n-step parameter and environment used.

On a high level, the difference of training time between both agents can be seen when looking at the average time needed for each agent and n-step to train on a single episode. The barcharts of Figure 7.13 show that the average time per episode is dependent on the environment. For Cdrive, the DP agent using n-step three has the longest average execution per episode with about 13 ms while the time for the same agent on Racetrack increases roughly by factor 100. Apart from that, the added complexity by increasing the n-step parameter can clearly be seen by the increase in execution time needed to train a single episode for the DP agent. On Cdrive, the effect seems to be linear while on Racetrack the execution time follows a seemingly exponential increase per n-step. In contrast, the execution time per episode for the matrix agent on Cdrive seems unaffected by the n-step parameter. On Racetrack, a sudden increase is noticeable from n-step two to three. The differences in the amount of episodes trained between the matrix agent and the DP agent on each environment given the same timeout (which can be seen in Figure 7.11 and 7.12) can be broken down to the execution time differences needed for each agent to train a single episode. The less time is needed to train a single episode, the more episodes can be trained until the timeout is reached. The matrix agent is faster for each n-step in each environment and additionally the effect of higher n-steps is not as big as for the

DP agent. Thus, it is able to train on more data than the DP agent. Note that training on more episodes does not equal learning better policies. On Cdrive, while the 1-step DP agent only trained on roughly a third of the episodes of all matrix agents, the DSMC evaluation (see Figure 7.10) indicates that the better sample efficiency of DP still leads to the agent learning better policies than the matrix agents. That said, the plots raise two questions which need further investigations:

1. What is the main cause for the increasing execution time per n-step?
2. What is the cause for the sudden increase of matrix agent from step two to three?

Each agent constantly performs two high level steps while interacting with the environment for each episode during training. First, the agent decides on an action to take based on its current policy and environment observations, executes that action and observes the environment state and reward. We refer to this as the acting parts of the algorithm. After that, the agent uses its action decision and the gathered reward to update its policy. This part is very algorithm specific and is referred to as the learning step in the following paragraphs and plots. Since acting is the same for both the DP and the Matrix agent and independent of the choice of the n-step parameter, the difference in execution time has to be caused in the learning step part of the algorithms. For this analysis it suffices to provide evidence of the claim that the time for acting is independent of the agent and n-step used. This can be seen in Figure 7.14. Note that on Racetrack, while there are differences for the matrix agent and n-steps, the differences can be neglected as the times vary only at most 2 ms which does not affect the overall execution time. By looking at Figure 7.13, the episode time for the matrix agent on Racetrack increases slightly from n-step one to two, while the average acting time decreases. Thus, the main cause of the time differences must be in the algorithm specific learning part.



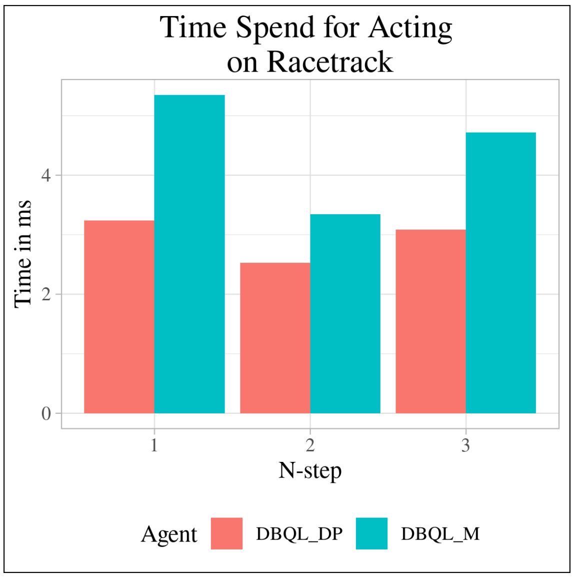
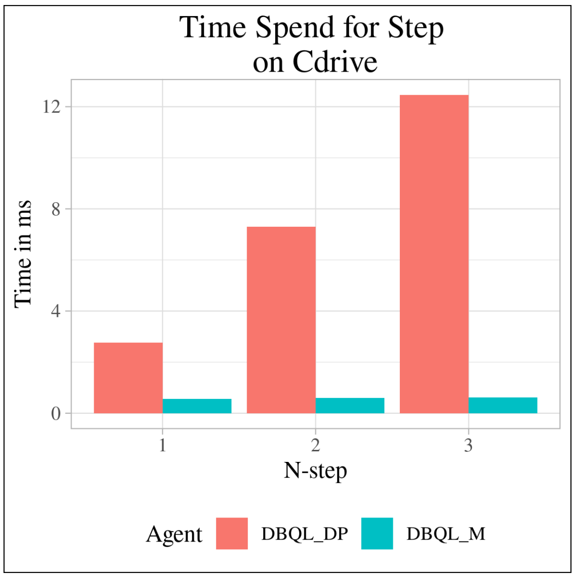


Figure 7.14: Bar charts plotting the average execution time of both n-step agents for choosing an action and interacting with the environment dependent on the n-step parameter and environment used.

A learning step includes updating any buffers and the replay memory, as well as training the policy network by computing targets, performing backpropagation and soft-updating target networks. The time for anything related to learning should increase for higher n-steps, as the target computation gets more complex for both agents. Figure 7.15 plots the average time spend for a learning step for each n-step agent for each environment. The time differences between the n-steps for each agent correlate well with the time differences for each episode shown in Figure 7.13. Again, the step times for the matrix agents are much lower than for the DP agent. To understand this, the learning step is broken down into comparable parts for both agents.



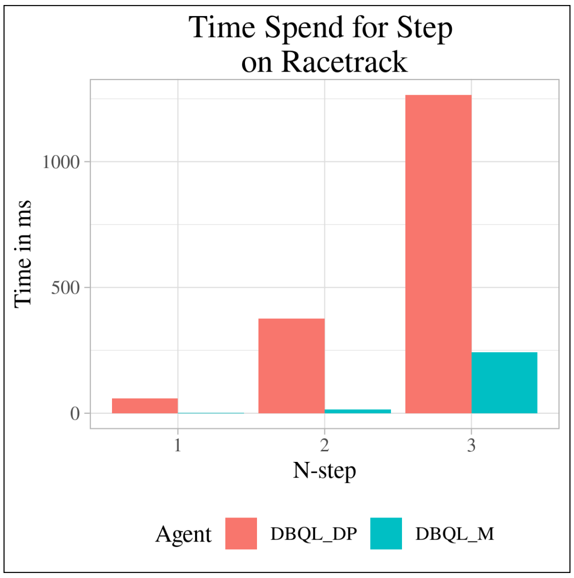


Figure 7.15: Bar charts plotting the average execution time of both n-step agents for agent specific learning steps, dependent on the n-step parameter and environment used.

Recall from Chapter 4 that each n-step agent computes a n-step target which is used to update the current policy by minimizing a loss function. One crucial difference between the n-step agents presented is the way this target is computed. Thus, the differences in execution time are likely caused by the target computation and the policy update. Figure 7.16 and 7.17 illustrate the time for learning, including the target computation, backpropagation and soft-updating the target networks for both agents on either Racetrack and Cdrive. The DP agent spends a significant amount of time in the learning procedure while the time spend by the matrix agent is negligible small. Note that the average learning time for DP is even higher than the corresponding average episode time. This is possible because both agents learn only every few episodes. However, for DP learning times per n-step seem to correlate well with the episode times, showing the same pattern on both environments. For the matrix agent, this is not the case as the learning time on Racetrack for the 3-step agent does not increase in contrast to the episode times (see Figure 7.13).

This is due to the difference in the way both agents compute their targets. The DP agent computes targets during learning by building a whole search tree with depth n and then traverses it recursively to compute a target. The deeper the tree (higher n-step) the more complex this operation and the more time is needed to compute the targets. This is done whenever the policy is updated, which is every few steps. In contrast, the Matrix agents target computation is rather easy by design, as it samples “the search-tree” from its buffer as a matrix of transitions of size n. It then just has to perform a simple max operation. The advantage of sampling transitions from a buffer is that sampling during learning is fast, especially as learning involves computing

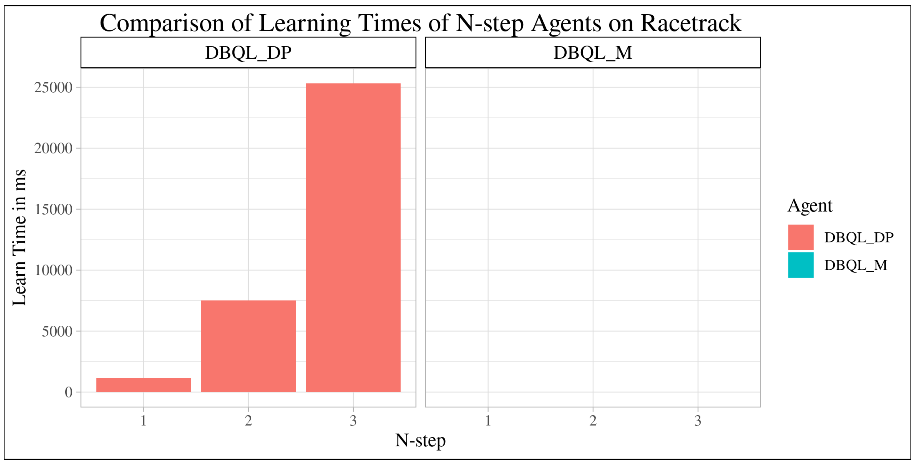


Figure 7.16: Bar charts plotting the average execution time of both n-step agents on Racetrack for target computation, backpropagation and soft-updating the target networks.

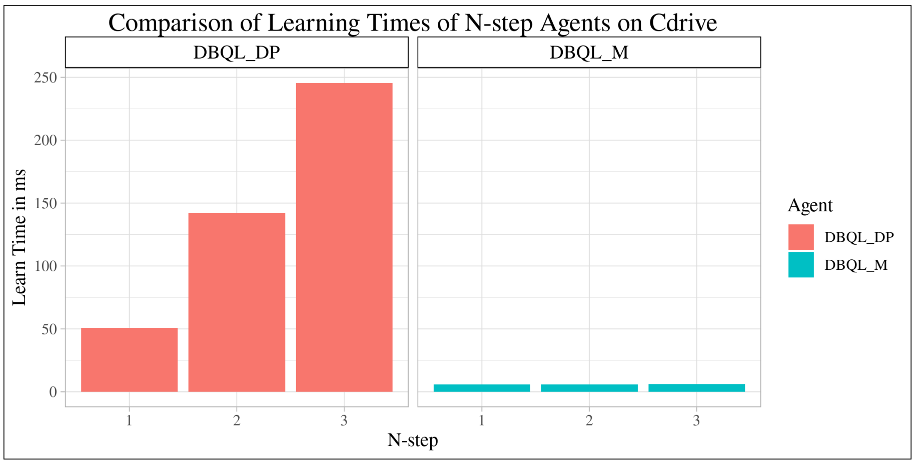


Figure 7.17: Bar charts plotting the average execution time of both n-step agents on Cdrive for target computation, backpropagation and soft-updating the target networks.

multiple targets. However, the transition matrices have to be computed outside the learning process in every single step. Although in every step only one transition matrix is build and saved to a buffer, this remains the most expensive operation. Since the most expensive part of the target computation for the matrix agent is outsourced from the learning process, it makes sense to compare the time needed for the target computation of the DP agent shown in Figure 7.18 with the time needed for building the n-transition matrices of the matrix agent shown in Figure 7.19.

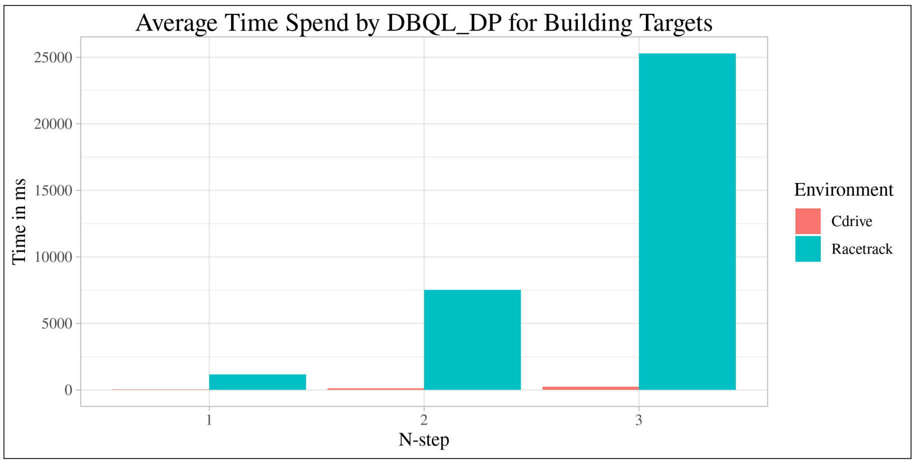


Figure 7.18: Bar chart plotting the average execution time of DP agent for target computation, dependent on n-step and environment used.

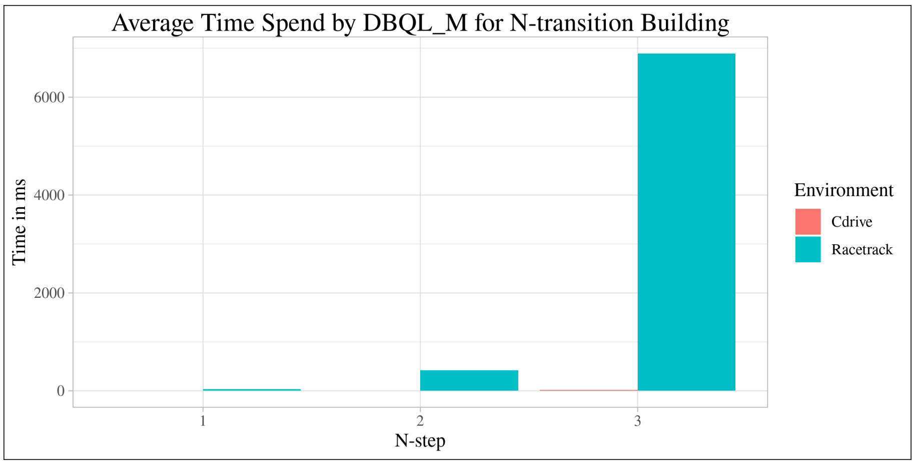


Figure 7.19: Bar chart plotting the average execution time of matrix agent needed to build the n-transition matrix in every step, dependent on n-step and environment used.

Note that the time needed for building the transition matrix correlates well to the episode time increase per n-step. While we established that the main cause for increased execution time is the target computation for the DP agent and the transition matrix building for the matrix agent, it remains unclear why the episode times for the latter on Racetrack with n-step three spikes while this is not the case on Cdrive. One possible explanation for this could be caching. Both agents make use of caches to speed up training. While the DP agent reduces forward passes of the policy network during search tree traversal by reusing already existing target estimations, the matrix agent uses a cache in order to eliminate the need to constantly re-compute the n-transition matrix when revisiting already seen states. Both buffers are referred to as *v-buffer* and its usage for each agent on each environment for each n-step is plotted in

Figure 7.20. Notice that the usage is only dependent on the environment and independent of the n-step parameter. Concluding, while the caches are important for the training speed-up, they cannot explain why the matrix agent's average training time for a single episode spikes on Racetrack with n-step three. This question is thus left for future work.

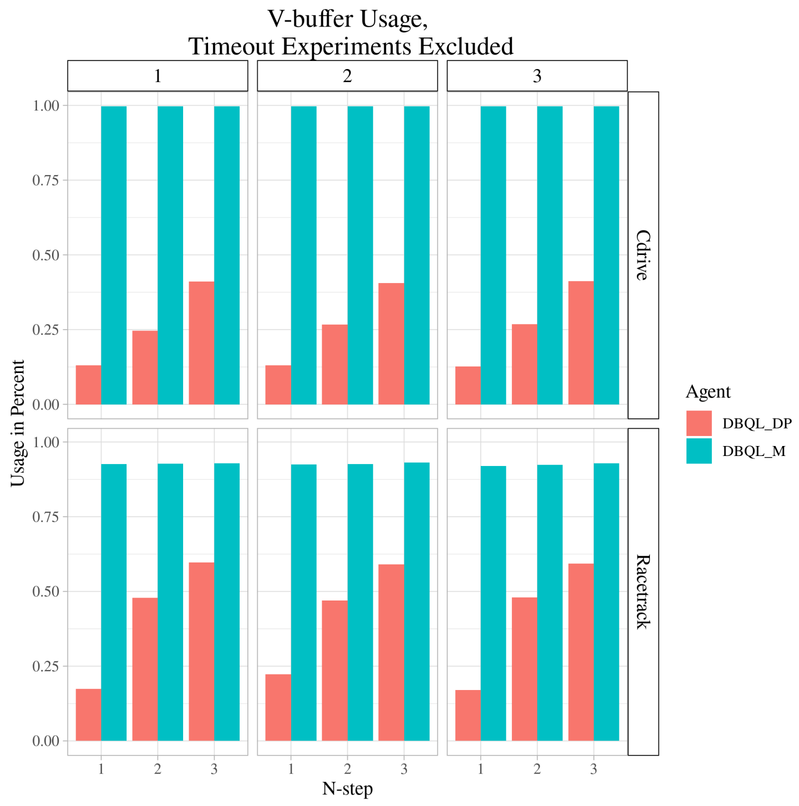


Figure 7.20: Average usage of the v-buffer used by both agents dependent on the agent, environment and n-step used.

7.3.1 Conclusion

To summarize, for each agent the average time needed to train each episode increases with each n-step. The effect is most noticeable for the DP agent which is much slower than the matrix agent due to its need to build and recursively traverse search trees of depth n for the target computation. The matrix agent, designed to have fast target computation due to loosening its theoretical foundations, shows to be faster while still being able to learn. However, being able to train faster does not generally induce that the matrix agent learns better policies, as the DP agent seems more sample efficient.

Chapter 8
Conclusion and Future Work

This chapter concludes the thesis by summarizing the most important findings in relation to the research aims and questions. It shortly outlines what the thesis contributes to the deep reinforcement learning community, discusses limitations of the study, and it finally provides some recommendations for future research.

The thesis aimed to find (1) reasons why some benchmarks of the QVBS are not learnable by the DQN agents like the one used in the paper introducing MoGym, thereby discovering MoGym specific challenges, and (2) it aimed to provide the DRL community with agents and solutions to overcome these challenges by improving on existing approaches. With the help of the experiments, the thesis provides evidence in favor of confirming the hypothesis that the main challenges for hard QVBS benchmarks loaded with MoGym are the action-space explosion, which can occur when loading models using MoGym, and lack of sophisticated exploration.

Within this thesis the use of DBQL is proposed, a DQN variant which is less affected by the action-space dimension and makes use of the special property of the environments created by MoGym, namely the known transition function. The experiment results indicate that DBQL with the same exploration technique used as a baseline DQN agent is capable of learning on environments where DQN is not, highlighting the effectiveness of the approach and the severity of the action-space explosion problem. The experiments further suggest that exploration remains an important challenge to solve, because multiple prominent exploration techniques in conjunction with DBQL are tested on hard QVBS benchmarks but only the newly proposed Stochastic Go-Explore framework enables learning on all tested environments. More specifically, the thesis includes and tests exploration techniques which either make use of random exploration (NoisyNet), intrinsic motivation (ICM, RND) or apply a planning like approach (Stochastic Go-Explore).

The thesis fulfills its aim to improve on existing solutions by improving on the Go-Explore framework with the newly introduced Stochastic Go-Explore technique, as well as improving on DBQL to use n-step learning by providing three new n-step DBQL agents. The original Go-Explore framework relies on the assumption that either the training environment used allows the agent to reset to any specified state during training or allows temporarily disabling stochasticity or alternatively, that a deterministic simulation environment is available for training. The introduced Stochastic Go-Explore framework drops this requirement, providing a more versatile framework which can be applied to a wider range of problems. The experiments indicate that Stochastic Go-Explore is not only a feasible extension to the original framework, but it also remains a very powerful exploration technique compared to the other ones tested. The newly proposed n-step DQBL agents, DBQL^n_M, DBQL^n_{DP} and DBQL^n_S, aim to improve the sample efficiency of the base agents. The study results confirm this for the agents except for DBQL^n_S, where the experiments were dropped because the base agent was not able to learn on the environments for the amount of episodes tested. While the feasibility of the n-step agents is demonstrated, the agents might not be the best choice when training time is a limiting factor.

Due to the time constraints and scope of the thesis, there are several ideas which are left for future work. With the main challenges of action-space explosion and need for sophisticated exploration known, the thesis has limited itself to DQN-like DRL architectures, or more broadly to value-based DRL algorithms. The effect of these challenges on policy gradient DRL algorithms, when training on hard QVBS problems using MoGym, remains unaddressed for now. Similarly, the proposed DBQL agents make use of the known transition function provided by MoGym environments. While this enables planning on the real-model and is a fundamental necessity for DBQL to be applicable, this is a rare feature in the world of DRL. Typically, DRL is applied to black box environments. Thus, it would be interesting to apply DRL algorithms successfully which do not abuse this advantage but are nevertheless designed to cope with high state and action spaces. One idea is to train an AlphaZero agent or any model-based reinforcement learning approaches which perform planning on a learned model and not the real one. More research can also be done on the improvements proposed in this thesis. Since Stochastic Go-Explore is shown to be a feasible and yet powerful derivation of the original framework, it would be interesting to quantify the difference in effectiveness of the new approach in sparse reward settings compared to the original Go-Explore framework and its existing alternatives, e.g. policy-based Go-Explore and DTSIL. In terms of the matrix n-step agent DBQL^n_M, while the experiments show that it can learn, its target computation is not following the Bellman optimality equation. A proof that this agent can or cannot learn optimal policies, or a proof that the matrix agent is guaranteed to learn optimal policies given some assumptions is not provided here and remains an interesting question for future work.

To summarize, the thesis has highlighted the challenge of large action-spaces and exploration for some instances of the QVBS loaded with MoGym and proposes to apply DBQL in conjunction with sophisticated exploration techniques to make these instances learnable. Furthermore, the thesis provides the DRL community with improvements of DBQL by proposing three new n-step DBQL agents. These agents are shown to increase sample efficiency by increasing the n-step parameter while this might also lead to higher, sometimes infeasible, training times. In addition, a new variant of the exploration framework Go-Explore is proposed in this thesis which is applicable to a wider range of DRL settings due to dropping some limiting assumptions of the original framework. The study found evidence that this new Stochastic Go-Explore remains a powerful exploration technique compared to all the other techniques tested. Concluding, the thesis provides a basis to solve hard problems of the QVBS using value-based DRL algorithms with the algorithms proposed. However, due to the limitations of the thesis, it remains for future work how policy-gradient and model-based reinforcement learning methods perform on the same tasks and if they face the same challenges.

Bibliography

1. Mnih, V. *et al.* Human-level control through deep reinforcement learning. *Nature 2015 518:7540* **518,** 529–533. ISSN: 1476-4687. https://www.nature.com/articles/nature14236 (7540 Feb. 2015).
2. Hessel, M. *et al.* Rainbow: Combining Improvements in Deep Reinforcement Learning. http://arxiv.org/abs/1710.02298 (Oct. 2017).
3. Silver, D. *et al.* A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* **362,** 1140–1144. ISSN: 10959203. https://www.science.org/doi/full/10.1126/science.aar6404 (6419 Dec. 2018).
4. Schrittwieser, J. *et al.* Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model. *Nature* **588,** 604–609. http://arxiv.org/abs/1911.08265 (7839 Nov. 2019).
5. Khan, M. A.-M. *et al.* A Systematic Review on Reinforcement Learning-Based Robotics Within the Last Decade. *IEEE Access* **8,** 176598–176623. https://ieeexplore.ieee.org/document/9206554 (2020).
6. Jumper, J. *et al.* Highly accurate protein structure prediction with AlphaFold. *Nature 2021 596:7873* **596,** 583–589. ISSN: 1476-4687. https://www.nature.com/articles/s41586-021-03819-2 (7873 July 2021).
7. Fawzi, A. *et al.* Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature 2022 610:7930* **610,** 47–53. ISSN: 1476-4687. https://www.nature.com/articles/s41586-022-05172-4 (7930 Oct. 2022).
8. Gros, T. P. *et al.* *MoGym: Using Formal Models for Training and Verifying Decision-making Agents* in *Computer Aided Verification* (eds Shoham, S. & Vizel, Y.) (Springer International Publishing, Cham, 2022), 430–443. ISBN: 978-3-031-13188-2.
9. Budde, C. E. *et al.* JANI: Quantitative Model and Tool Interaction. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* **10206 LNCS,** 151–168. ISSN: 16113349. https://link.springer.com/chapter/10.1007/978-3-662-54580-5_9 (2017).
10. Hartmanns, A., Klauck, M., Parker, D., Quatmann, T. & Ruijters, E. *The Quantitative Verification Benchmark Set* in *Tools and Algorithms for the Construction and Analysis of Systems* (eds Vojnar, T. & Zhang, L.) (Springer International Publishing, Cham, 2019), 344–350. ISBN: 978-3-030-17462-0.
11. Hartmanns, A., Klauck, M., Parker, D., Quatmann, T. & Ruijters, E. *Website of the Quantitative Verification Benchmark Set (QVBS)* https://qcomp.org/benchmarks/index.html (2022).
12. Moerland, T. M., Broekens, J., Plaat, A. & Jonker, C. M. A Unifying Framework for Reinforcement Learning and Planning. https://arxiv.org/abs/2006.15009v4 (June 2020).
13. Yi, F., Fu, W. & Liang, H. Model-based Reinforcement Learning: A Survey. *Proceedings of the International Conference on Electronic Business (ICEB)* **2018-December,** 421–429. ISSN: 16830040. https://arxiv.org/abs/2006.16712v4 (June 2020).
14. *Automated planning theory and practice* eng. ISBN: 9780080490519 (2004).
15. LaValle, S. M. *Planning Algorithms* (Cambridge University Press, 2006).
16. Sutton, R. S. & Barto, A. G. *Reinforcement Learning: An Introduction* Second (The MIT Press, 2018).

17. Kwiatkowska, M., Norman, G. & Parker, D. *PRISM 4.0: Verification of Probabilistic Real-time Systems* in *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)* (eds Gopalakrishnan, G. & Qadeer, S.) **6806** (Springer, 2011), 585–591.

18. Legay, A., Delahaye, B. & Bensalem, S. Statistical Model Checking: An Overview. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* **6418 LNCS,** 122–135. ISSN: 03029743. https://link.springer.com/chapter/10.1007/978-3-642-16612-9_11 (2010).

19. Gros, T. P., Hermanns, H., Hoffmann, J., Klauck, M. & Steinmetz, M. *Deep Statistical Model Checking* in *Formal Techniques for Distributed Objects, Components, and Systems* (eds Gotsman, A. & Sokolova, A.) (Springer International Publishing, Cham, 2020), 96–114. ISBN: 978-3-030-50086-3.

20. Lillicrap, T. P. *et al.* Google Deepmind 2016 Continuous control with deep reinforcement learning. *4th International Conference on Learning Representations (ICLR 2016).* http://arxiv.org/abs/1509.02971v6 (2016).

21. Gros, T. P., Groß, J., Hoffmann, J. & Wolf, V. *Description-based Q-learning* (in progress).

22. Ecoffet, A., Huizinga, J., Lehman, J., Stanley, K. O. & Clune, J. Go-Explore: a New Approach for Hard-Exploration Problems. https://arxiv.org/abs/1901.10995v4 (Jan. 2019).

23. Hernandez-Garcia, J. F. & Sutton, R. S. Understanding Multi-Step Deep Reinforcement Learning: A Systematic Study of the DQN Target. https://arxiv.org/abs/1901.07510v2 (Jan. 2019).

24. Sharma, S., J, G. R., Ramesh, S. & Ravindran, B. Learning to Mix n-Step Returns: Generalizing lambda-Returns for Deep Reinforcement Learning. https://arxiv.org/abs/1705.07445v2 (May 2017).

25. Daley, B. & Amato, C. Reconciling λ-Returns with Experience Replay. https://arxiv.org/abs/1810.09967v3 (Oct. 2018).

26. Mnih, V. *et al.* Asynchronous Methods for Deep Reinforcement Learning. *33rd International Conference on Machine Learning, ICML 2016* **4,** 2850–2869. https://arxiv.org/abs/1602.01783v2 (Feb. 2016).

27. Pathak, D., Agrawal, P., Efros, A. A. & Darrell, T. Curiosity-driven Exploration by Self-supervised Prediction. *34th International Conference on Machine Learning, ICML 2017* **6,** 4261–4270. https://arxiv.org/abs/1705.05363v1 (May 2017).

28. Barto, A. G., Bradtke, S. J. & Singh, S. P. Learning to act using real-time dynamic programming. *Artificial Intelligence* **72,** 81–138. ISSN: 0004-3702. https://www.sciencedirect.com/science/article/pii/000437029400011O (1995).

29. *JANI model format specification.* https://jani-spec.org/. Accessed: 2022-04-11.

30. Legay, A. *et al.* in *Computing and Software Science: State of the Art and Perspectives* (eds Steffen, B. & Woeginger, G.) 478–504 (Springer International Publishing, Cham, 2019). ISBN: 978-3-319-91908-9. https://doi.org/10.1007/978-3-319-91908-9_23.

31. Efroni, Y., Ghavamzadeh, M. & Mannor, S. Online Planning with Lookahead Policies. *Advances in Neural Information Processing Systems* **2020-December.** ISSN: 10495258. https://arxiv.org/abs/1909.04236v2 (Sept. 2019).

32. Achiam, J. & Sastry, S. Surprise-Based Intrinsic Motivation for Deep Reinforcement Learning. https://arxiv.org/abs/1703.01732v1 (Mar. 2017).

33. Tang, H. *et al.* Exploration: A Study of Count-Based Exploration for Deep Reinforcement Learning. *Advances in Neural Information Processing Systems* **2017-December,** 2754–2763. ISSN: 10495258. https://arxiv.org/abs/1611.04717v3 (Nov. 2016).

34. Bellemare, M. G. *et al.* Unifying Count-Based Exploration and Intrinsic Motivation. *Advances in Neural Information Processing Systems,* 1479–1487. ISSN: 10495258. https://arxiv.org/abs/1606.01868v2 (June 2016).

35. Bellemare, M., Veness, J. & Talvitie, E. *Skip Context Tree Switching* in *Proceedings of the 31st International Conference on Machine Learning* (eds Xing, E. P. & Jebara, T.) **32** (PMLR, Bejing, China, 22–24 Jun 2014), 1458–1466. https://proceedings.mlr.press/v32/bellemare14.html.

36. Taïga, A. A., Fedus, W., Machado, M. C., Courville, A. & Bellemare, M. G. *Benchmarking Bonus-Based Exploration Methods on the Arcade Learning Environment* 2019. https://arxiv.org/abs/1908.02388.

37. Burda, Y., Edwards, H., Storkey, A. & Klimov, O. *Exploration by Random Network Distillation* 2018. https://arxiv.org/abs/1810.12894.

38. Fortunato, M. *et al. Noisy Networks for Exploration* 2017. https://arxiv.org/abs/1706.10295.

39. Ecoffet, A., Huizinga, J., Lehman, J., Stanley, K. O. & Clune, J. First return, then explore. *Nature 2021 590:7847* **590,** 580–586. ISSN: 1476-4687. https://www.nature.com/articles/s41586-020-03157-9 (7847 Feb. 2021).

40. Guo, Y. *et al. Memory Based Trajectory-conditioned Policies for Learning from Sparse Rewards* 2019. https://arxiv.org/abs/1907.10247.

41. Park, C. *Jupyter notebook reference implementation for n-step buffer of RainbowDQN* June 2022. https://nbviewer.org/github/Curt-Park/rainbow-is-all-you-need/blob/master/07.n_step_learning.ipynb.

42. Salimans, T. & Chen, R. *Learning Montezuma's Revenge from a Single Demonstration* 2018. https://arxiv.org/abs/1812.03381.

43. Schaul, T., Quan, J., Antonoglou, I. & Silver, D. *Prioritized Experience Replay* 2015. https://arxiv.org/abs/1511.05952.

44. Henderson, P. *et al.* Deep Reinforcement Learning that Matters. *CoRR* **abs/1709.06560.** arXiv: 1709.06560. http://arxiv.org/abs/1709.06560 (2017).

45. Zhang, B. *et al. On the Importance of Hyperparameter Optimization for Model-based Reinforcement Learning* in *Proceedings of The 24th International Conference on Artificial Intelligence and Statistics* (eds Banerjee, A. & Fukumizu, K.) **130** (PMLR, 13–15 Apr 2021), 4015–4023. https://proceedings.mlr.press/v130/zhang21n.html.

46. Jaderberg, M. *et al. Population Based Training of Neural Networks* 2017. https://arxiv.org/abs/1711.09846.

47. Kingma, D. P. & Ba, J. *Adam: A Method for Stochastic Optimization* 2014. https://arxiv.org/abs/1412.6980.

48. Riedmiller, M. *et al. Learning by Playing Solving Sparse Reward Tasks from Scratch* in *Proceedings of the 35th International Conference on Machine Learning* (eds Dy, J. & Krause, A.) **80** (PMLR, Oct. 2018), 4344–4353. https://proceedings.mlr.press/v80/riedmiller18a.html.

49. Memarian, F., Goo, W., Lioutikov, R., Niekum, S. & Topcu, U. *Self-Supervised Online Reward Shaping in Sparse-Reward Environments* in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2021), 2369–2375.

50. Wikipedia. *Local regression — Wikipedia, The Free Encyclopedia* http://en.wikipedia.org/w/index.php?title=Local%20regression&oldid=1114048038. [Online; accessed 19-October-2022]. 2022.

51. Wickham, H. *et al. Smoothed conditional means* Oct. 2022. https://ggplot2.tidyverse.org/reference/geom_smooth.html.

52. Oh, J., Guo, Y., Singh, S. & Lee, H. *Self-Imitation Learning* in *Proceedings of the 35th International Conference on Machine Learning* (eds Dy, J. & Krause, A.) **80** (PMLR, Oct. 2018), 3878–3887. https://proceedings.mlr.press/v80/oh18b.html.

53. Schulman, J., Wolski, F., Dhariwal, P., Radford, A. & Klimov, O. *Proximal Policy Optimization Algorithms* 2017. https://arxiv.org/abs/1707.06347.

Appendix

9.1 Hyperparameters

The following hyperparameters where used for the experiments. These are the parameters which resulted from tuning the hyperparameters. In addition, the random seeds $1, 2$ and 3 where used for each experiment to account for randomness and make the experiments replicable.

DQN/DBQL parameters

α	The learning rate for gradient descend optimization	0.005
τ	The soft-update coefficient for target networks	0.005
T	Maximal step-length of an episode	100
γ	Discount factor	0.99
	Number of steps between learning	20
	The replay buffer size	10^6
	The batch size used for learning	25

Epsilon-Greedy parameters

ϵ_{start}	The start value of ϵ	1
ϵ_{end}	The final value of ϵ	0.1
ϵ_{decay}	The decay factor applied to ϵ after each episode until ϵ_{end} is reached	0.999

ICM parameters

η	Factor which balances importance of policy loss against ICM loss	1.0
β	Factor which balances importance of forward module loss against inverse module loss when computing ICM intrinsic reward	0.25
$\alpha_{forward}$	Learning rate of the forward module in ICM	0.005
$\alpha_{inverse}$	Learning rate of the inverse module in ICM	0.005

RND parameters

α_{RND}	The learning rate used to distill the training network to the random network	0.005

Go-Explore parameters

(Stochastic) Go-Explore uses the same parameters for ICM or RND, dependent on what is used as heuristic as the pure ICM and RND agents above.

exp_bs	The size of the exploration archive	10^5
exp_st	Minimum amount of entries needed in archive before agent starts to sample from it.	50
exp_icm	If set, ICM is used as heuristic, else RND is used	-

9.2 Discussion: Stochastic Go-Explore vs Existing Approaches

This thesis introduces Stochastic Go-Explore in an effort to address the research gap of the original paper introducing the Go-Explore framework [22]. The originally proposed framework required a deterministic environment during the exploration phase or the ability to disable stochasticity or reset to a state for the return step of the exploration phase. This limits the applicability of this powerful approach to environments which fulfill at least one of these requirements. Stochastic Go-Explore drops these assumptions by accommodating for stochasticity during the return step which allows directly training agents in purely stochastic environments. Thereby, the research community is provided with a more widely applicable version of Go-Explore. However, we emphasize that other approaches emerged which address the same problem. The Go-Explore authors published an algorithm called policy-based Go-Explore in later revisions of their paper [39] and another team of researchers published an algorithm called Diverse Trajectory-conditioned Self-Imitation Learning (DTSIL) [40]. We think that our approach is still a valuable contribution to the DRL community despite this because it uses fundamentally different concepts and consequently might be easier to implement. In the following sections, both policy-based Go-Explore and DTSIL are outlined and the differences to Stochastic Go-Explore are highlighted. For an in-depth explanation of both algorithms we refer the interested reader to the original papers. First, the main problem with the return step of Go-Explore in stochastic environments is outlined. Then, an outline of policy-based Go-Explore is given which specifies how this algorithm addresses these issues as well as a short high-level overview of the algorithm concepts. The same is done for DTSIL and finally a short summary of the discussion is given.

Returning with Stochasticity

In a deterministic environment returning to a state sampled from the archive is easier because it suffices to apply the same actions which have lead to the discovery of the state. Thus, the set of actions is stored alongside the state in the archive. In a stochastic setting, applying a stored action might lead to the agent derailing from its path to the sampled state s.t. the agent cannot reach the interesting state. This violates the basic concept of Go-Explore which tries to mitigate derailment to keep exploring interesting frontiers.

Recall that in Stochastic Go-Explore the issue is addressed by storing a trajectory of states and actions instead of just actions. This allows the agent to check if it is still on the correct path after every step taken in the environment. If the agent has derailed from the trajectory it is supposed to imitate, the return phase is ended and the agent can start exploring from there on. While this does not solve the derailment problem it acknowledges its existence by applying a best effort strategy. The intuition is that derailment is bound to happen in stochastic environments, which is why it is useful to explore the state-space around the best trajectories found and consequently to learn a robust policy which can use the gathered data to learn what to do in these situations. This is possible because Stochastic Go-Explore trains a policy while exploring and returning. In contrast, the original framework delayed training a policy to the robustification phase using self-imitation learning which essentially follows the same principle: training a DRL agent to imitate the trajectory in a stochastic setting to learn how to react once derailment occurs.

Policy-based Go-Explore

In policy-based Go-Explore the problem of derailment during the return phase is also addressed with a best-effort method. However, while the Stochastic Go-Explore agent immediately switches to the exploration phase when a derailment is noticed, the policy-based Go-Explore agent first tries to get back to any state on the trajectory it is supposed to follow which would allow it to potentially reach the interesting state anyway. To do this, the archive now also provides a trajectory of states and actions instead of only actions to allow the agent to recognize derailment. The agent then learns a goal-conditioned policy $\pi_\theta(a|s, g)$ where a is an action available in the current state s and g is a goal state the agent is supposed to reach. While the goal-state could just be the sampled state from the archive which the agent should return to, this can be problematic if this state is far away from the current position [39]. Instead, the agent follows the stored trajectory by setting every state in the trajectory as a subgoal. If it reaches a subgoal or any other successor state of the subgoal state of the trajectory, it sets the immediate successor state as a new subgoal until the sampled interesting state is reached. Every time a goal is reached, the agent receives a small positive reward which encourages the agent to get back on the trajectory it is supposed to follow. The authors call this process of roughly following a trajectory, even if derailment occurs, following a soft-trajectory [39].

However, sometimes it is not possible for an agent to reach the specified goal either because there is no path back to the trajectory or because the policy needs more training. To prevent agents from getting stuck endlessly roaming the environment, the agents eventually start the exploration step by aborting the return phase if no progress was made for a specified number of steps. Thus, this approach can also be considered a best effort method. Once the exploration phase starts, the agent starts exploring randomly with 50% probability or else it chooses an exploration which goal it tries to reach by using its policy. If the agent has reached this goal and continues to find new states another exploration goal is chosen.

While goal-conditioned policies have the advantage that an agent can recover from derailment instead of just giving up the return phase immediately, it also introduces overhead. The agent must keep track of the goals for each sampled state during training of the policy and it must choose goals during exploration to use the policy in the exploration phase which is not straight forward. Additionally, there must be a notion of progress to decide whether returning still failed despite the goal-conditioned policy and the reward for reaching a goal or subgoal must be chosen carefully as this introduces some form of intrinsic reward. This is interesting as the original paper criticized intrinsic reward to lead to detachement and derailment. Another important change compared to Stochastic Go-Explore is that the agent still needs to rely on stored trajectories found during training to execute the trained policy during evaluation because the policy needs to be conditioned on (sub)goals.

Another major difference between policy-based Go-Explore and Stochastic Go-Explore is the use of a self-imitation learning (SIL) loss during training which should not be confused with the self-imitation learning algorithm used in the robustification phase. To train the goal-conditioned policy, e.g. for a DQN agent, the agent samples from a replay-buffer random trajectories $((s, g), a, r, (s', g'))$ to calculate the policy loss

$$L_{\text{policy}}(\theta_i) = \mathbb{E}_{\theta_i}[((r + \gamma \max_{a'} Q'_\theta(a', s', g') - Q_\theta(a, s, g))^2]$$

by computing a target using the current version of both the target and the policy network, where θ are the parameters of the q-function approximator and θ' are the target network parameters. Note that we insert both the state and the goal to the q-function since we use a goal-conditioned policy. Since the agents are trained to follow the best trajectories found, additionally the agents sample a few of the best trajectories and compute the SIL loss. Given a sampled trajectory $[((s_0, g_0), a_0, r_0, s_1), ..., ((s_T, g_T), a_T, r_T, s_{T+1})]$, the SIL loss

$$L_{\text{sil}}(\theta_i) = \frac{1}{2}\sum_{t=0}^{T-1} \parallel G_{t+1} - Q_{\theta_i}(a_t, s_t, g_t) \parallel^2,$$

is the MSE of the known discounted return and the estimation of the estimator at each time step of the trajectory [52]. Note that the policy-based Go-Explore paper [39] uses proximal policy optimization (PPO) [53] instead of DQN, but SIL can be extended to q-value based algorithms [52]. The agent then optimizes for a combined policy and SIL loss with the ability to weigh the importance of both terms using hyperparameters. The Stochastic Go-Explore algorithm, which is introduced in this thesis, does not include a SIL loss which further reduces the algorithm's complexity compared to policy-based Go-Explore (and DTSIL) while still remaining a powerful alternative as shown by the experiment results.

Other distinctions to Stochastic Go-Explore include the heuristic used for sampling states from the archive, as the policy-based Go-Explore algorithm proposed uses count-based exploration while our agent uses ICM or RND as a heuristic. Note that this can also be used for policy-based Go-Explore though and seems to be a design choice of the authors [39]. Furthermore, policy-based Go-Explore follows a batch learning patterns where multiple agents gather data to train a single policy. While this is not required, Stochastic Go-Explore is designed for a single-agent use, although it can be extended to use a batch learning pattern.

DTSIL

The DTSIL algorithm [40] uses the same basic concepts as policy-based Go-Explore because it also uses a goal-conditioned policy, computes a self-imitation learning loss (SIL loss) and makes use of soft-trajectories. Due to their similarities, the authors of policy-based Go-Explore included a discussion of the differences between both approaches [39]. One key difference is the use of a sequence-to-sequence model with an attention mechanism to represent the policy. Instead of single goal-states, the policy $\pi_\theta(a_t|e_{\leq t}, o_t, g)$ is conditioned on a vector of state-embeddings $e_{\leq t} = \{e_1, ..., e_t\}$ for the states seen so far, the current observation o_t and a vector of goal state-embeddings $g = \{e_1^g, ..., e_{|g|}^g\}$ which is the soft-trajectory the agent should follow [40]. Note that $|g|$ is the length of the trajectory g and the policy is also denoted as goal-conditioned policy $\pi_\theta(\cdot|g)$ [40]. By having access to $e_{\leq t}$ and g the model is able to determine which part of the demonstration was successfully followed and can choose actions accordingly. The goal of the agent is to find the optimal state-embedding sequences $g*$ and the optimal policy $\pi_{\theta*}(\cdot|g)$ which maximizes the discounted return, i.e. $\theta^* = \max_{g,\theta} \mathbb{E}_{\pi_\theta(\cdot|g)}[\sum_{t=0}^{T} \gamma^t r_t]$ [40].

Since trajectories are followed softly, the archive is filled with a diverse set of trajectories the agent can learn from which explore the space around promising trajectories. The main idea remains the same between Stochastic Go-Explore, policy-based Go-Explore and DTSIL. Figure 9.1 from the DTSIL paper [40] is a high-level overview of the algorithm. However, just like for policy-based Go-Explore the introduction of goal-conditioned policies, a self-imitation learning loss and the usage of a sequence-to-sequence model using an attention mechanism makes these alternatives more complex than Stochastic Go-Explore.

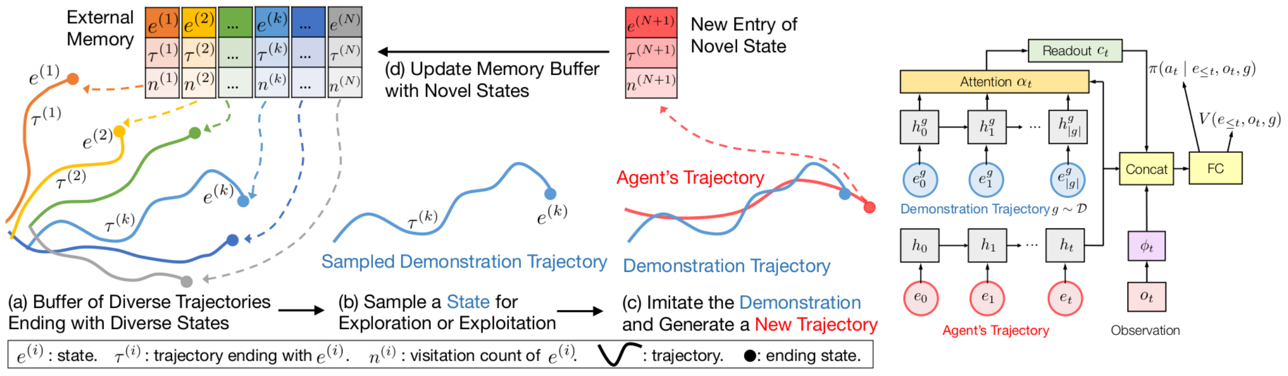

External Memory
(d) Update Memory Buffer with Novel States
New Entry of Novel State
Sampled Demonstration Trajectory
Agent's Trajectory
Demonstration Trajectory
(a) Buffer of Diverse Trajectories Ending with Diverse States
(b) Sample a State for Exploration or Exploitation
(c) Imitate the Demonstration and Generate a New Trajectory
$e^{(i)}$: state. $\tau^{(i)}$: trajectory ending with $e^{(i)}$. $n^{(i)}$: visitation count of $e^{(i)}$. : trajectory. ●: ending state.
Readout c_t
Attention α_t
$\pi(a_t \mid e_{\leq t}, o_t, g)$
$V(e_{\leq t}, o_t, g)$
Concat
FC
Demonstration Trajectory $g \sim \mathcal{D}$
Agent's Trajectory
Observation

Figure 9.1: Overview of DTSIL as shown in the DTSIL paper [40].

Summary

While policy-based Go-Explore, DTSIL and Stochastic Go-Explore all try to deal with stochasticity during the return step by applying a best-effort approach, the algorithms vary in the concepts used to do this. Both policy-based Go-Explore and DTSIL use a goal-conditioned policy in combination with soft-trajectories and add a SIL loss in addition to the DRL agents policy loss. Stochastic Go-Explore does not rely on any of these techniques resulting in less state-keeping and complexity overhead. The intuition, that learning from trajectories resulting from derailment of the promising trajectory is useful, is the same for all three algorithms. However, policy-based Go-Explore and DTSIL agents put more effort in trying to imitate the trajectory closely even with derailment. In contrast, the Stochastic Go-Explore agent always ends its imitation efforts as soon as derailment occurs. The performance difference of the agents on hard exploration environments is left for future work. Because Stochastic Go-Explore could train successfully on all training environments of the thesis we conclude that our approach is a powerful exploration technique despite its simpler structure than the here presented alternatives. Due to this, we think that the reduced complexity might make it easier to deploy on hard problems and thus provides an interesting alternative to policy-based Go-Explore and DTSIL. In addition, Stochastic Go-Explore relies on fewer hyperparameters than the other two variants which is beneficial for hyperparameter tuning.

9.3 DSMC Exploration Plots using GRP

The following plots serve as additional material to Section 7.1.1 and illustrate the mean goal-reaching probability (GRP) for each agent used in the exploration experiments for each agent on a specific environment. Note that these plots have a corresponding plot in Section 7.1.1 where expected return is used as DSMC criterion. The plots show the same results, s.t. often the only visible difference is the y-axis scale which indicates a high correlation between both GRP and expected return criteria for these specific experiments with the specific reward function used.

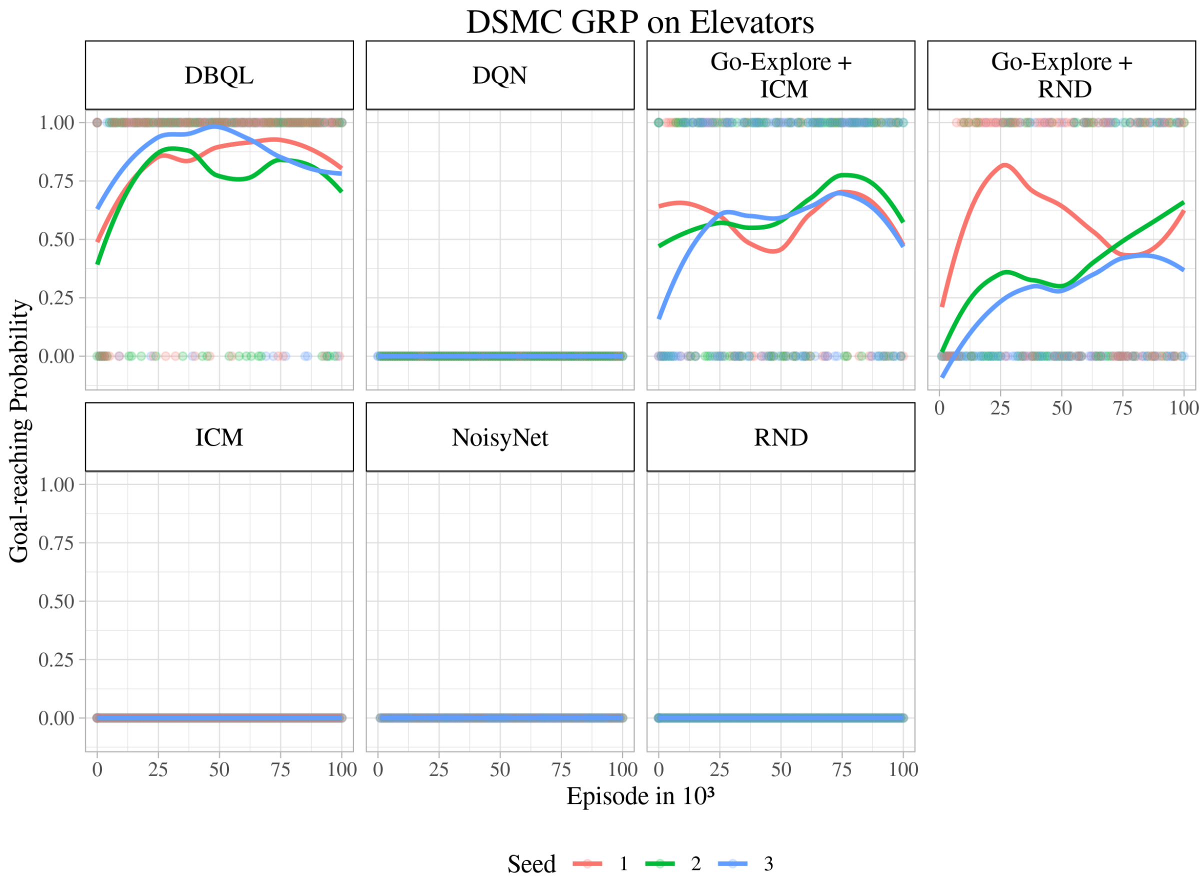


Figure 9.2: DSMC evaluation of experiments on Elevators using GRP as criterion.

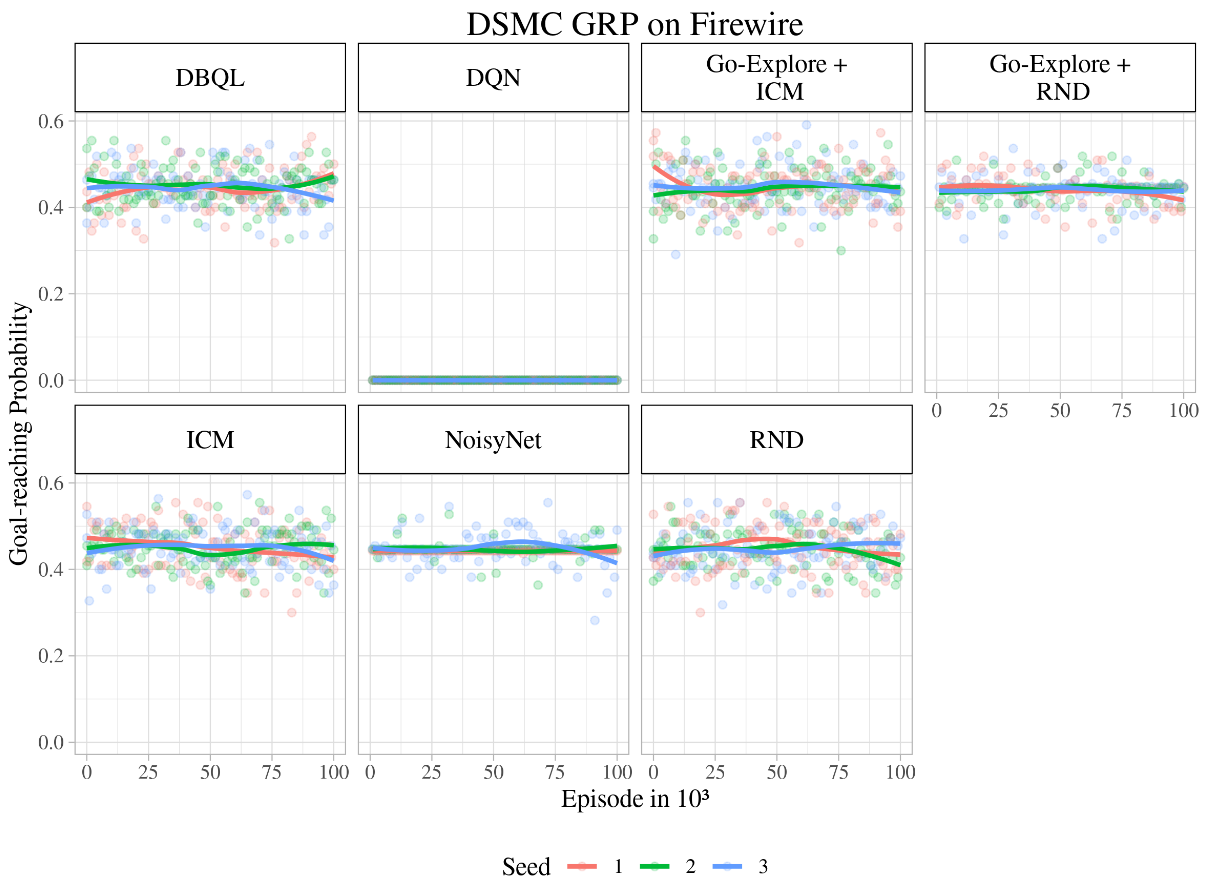


Figure 9.3: DSMC evaluation of experiments on Firewire using GRP as criterion.

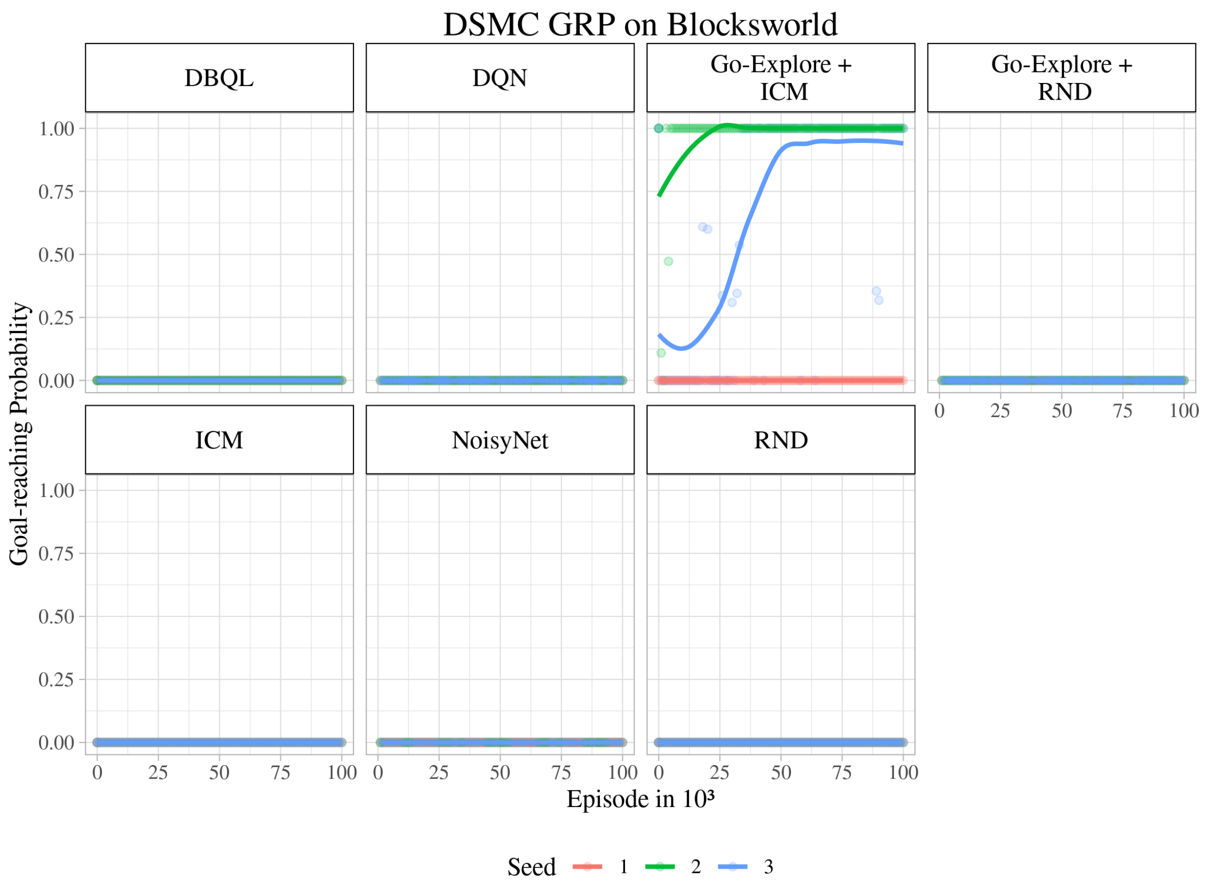


Figure 9.4: DSMC evaluation of experiments on Blocksworld using GRP as criterion.

9.4 DSMC N-step Plots using GRP

The following plots serve as additional material to Section 7.2.1 and plot the mean goal-reaching probability (GRP) for each agent used in the sample efficiency experiments for each agent for each n-step. Note that these plots have a corresponding plot in Section 7.2.1 where expected return is used as DSMC criterion. The plots show the same results, s.t. often the only visible difference is the y-axis scale which indicates a high correlation between both GRP and expected return criteria for these specific experiments with the specific reward function used.

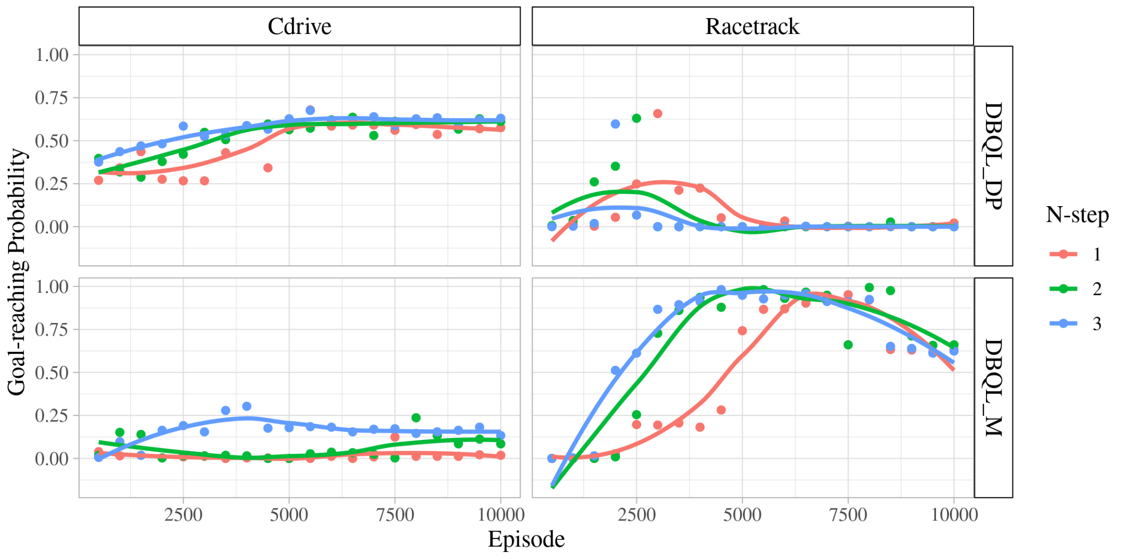


Figure 9.5: Mean goal-reaching probability of both n-step agents plotted by n-step on Racetrack and Cdrive, estimated using DSMC on the extracted policies. The agents trained for a fixed amount of time.

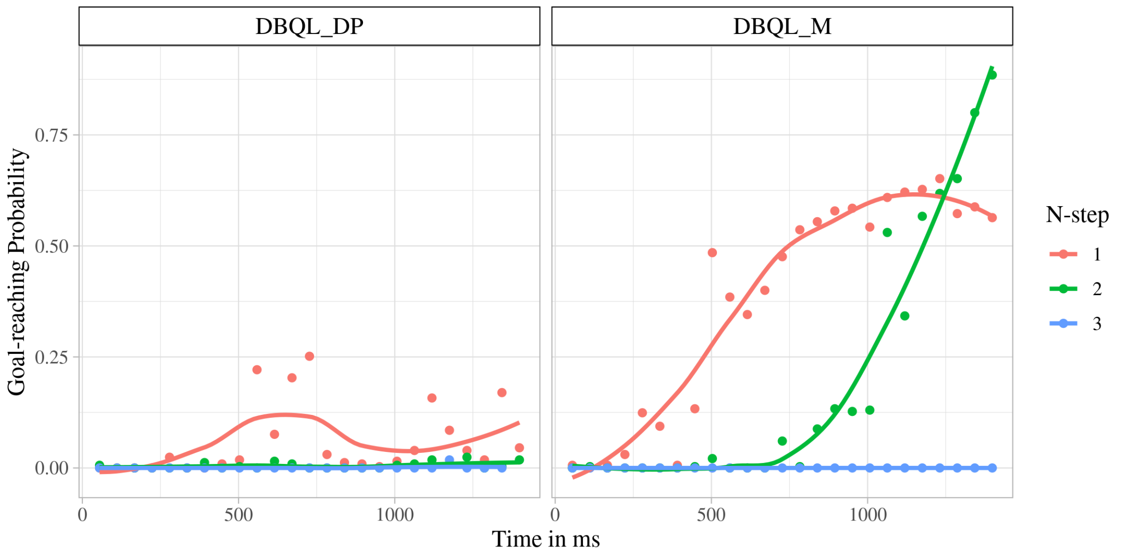


Figure 9.6: Mean goal-reaching probability of both n-step agents plotted by n-step on Racetrack, estimated using DSMC on the extracted policies. The agents trained for a fixed amount of time.

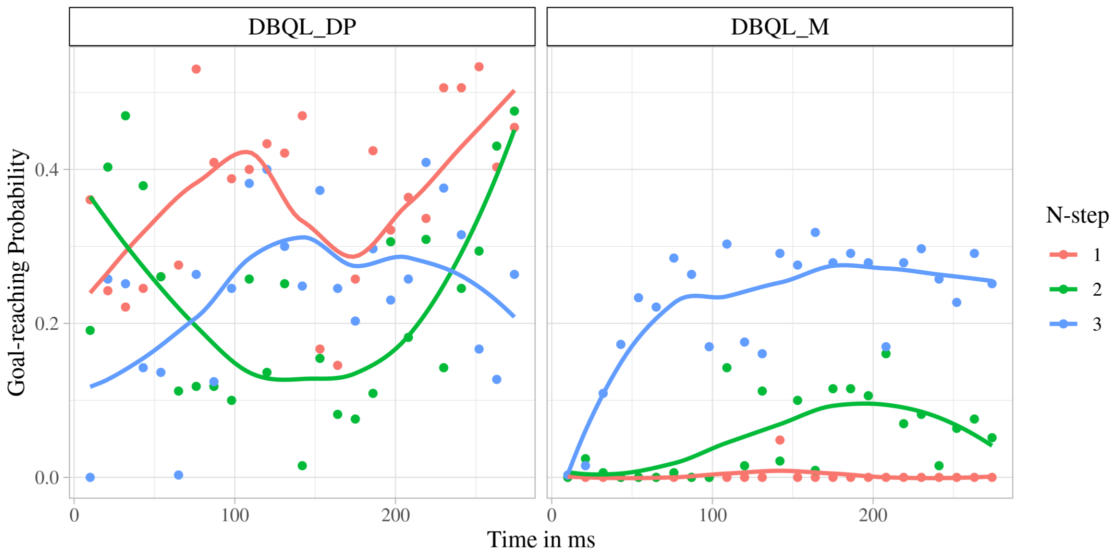
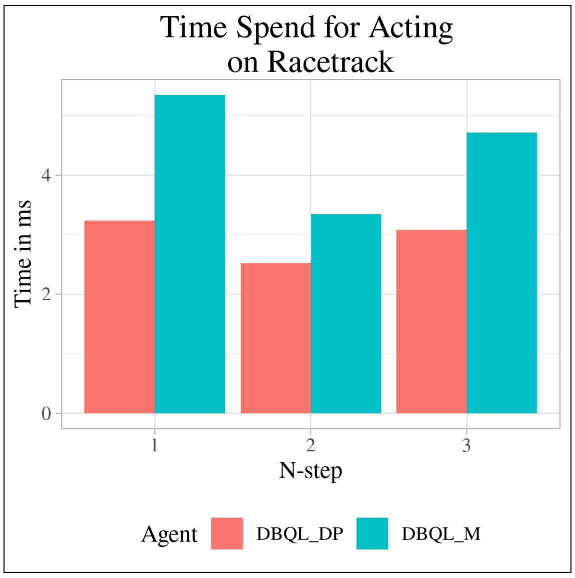


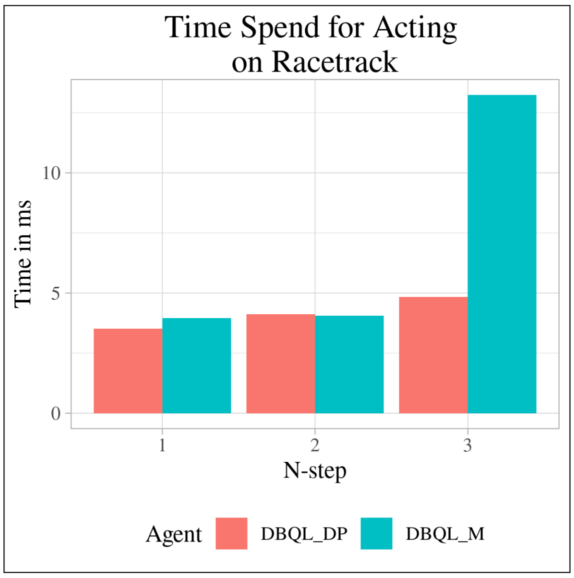
Figure 9.7: Mean goal-reaching probability of both n-step agents plotted by n-step on Cdrive, estimated using DSMC on the extracted policies. The agents trained for a fixed amount of time.

9.5 Timeout Data vs Non-Timeout Data

In Section 7.3, only the timer data from n-step experiments with a fixed amount of episodes to train on is used. The reason for this is that for the timeout experiments, the caches used by the agents cannot develop their full efficiency. Thus, including the data of these experiments would skew the results of the plots in a negative direction. To demonstrate this, compare the plots of Figure 9.8a which shows the average time for the agents for acting with the environment including the timeout experiment data to the plots of Figure 9.8b, which excludes the timeout experiment data. Due to the caches not being filled enough, the execution times including the timeout experiments are higher on average.



(a) Data excludes timeout experiments.



(b) Data includes timeout experiments.

Figure 9.8: Bar charts plotting the average execution time of both n-step agents for choosing an action and interacting with the environment dependent on the n-step parameter and environment used.