Deep Statistical Model Refinement

Master Thesis

Hendrik Meerkamp

Supervisor:

Prof. Dr. Jörg Hoffmann
Prof. Dr. Verena Wolf

Saarland University
Faculty of Mathematics and Computer Science
Foundations of Artificial Intelligence Group
Group of Modeling and Simulation

December 4, 2020

Acknowledgements

I would like to thank Prof. Dr. Jörg Hoffmann and Prof. Dr. Verena Wolf for providing this interesting topic, the helpful discussions, and their supervision. I would also like to thank my advisors Timo Gros and Daniel Höller, who always provided help when I needed it and always offered excellent feedback. Last but not least, I would like to thank my family and friends for all their support over the years.

Statement of Authentication

I hereby declare that I have written the present thesis independently, without assistance from external parties and without use of other resources than those indicated. The ideas taken directly or indirectly from external sources (including electronic sources) are duly acknowledged in the text. The material, either in full or in part, has not been previously submitted for grading at this or any other academic institution.

Saarbrücken, December 4, 2020 Hendrik Meerkamp

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen 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 die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

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

Saarbrücken,_____ _____

(Datum/Date) (Unterschrift/Signature)

Erklärung

Ich erkläre hiermit, dass die vorliegende Arbeit mit der elektronischen Version übereinstimmt.

Statement

I hereby confirm the congruence of the contents of the printed data and the electronic version of the thesis.

Saarbrücken,---------- ----------

(Datum/Date) (Unterschrift / Signature)

Abstract

The current success of deep reinforcement learning has brought forth the desire for applications in the real world. To gain more insight into the properties of an agent's behavior, the recent work of Deep Statistical Model Checking (DSMC) is concerned with applying statistical model checking techniques to the deep neural networks resulting from deep learning.

DSMC allows a learning architect to spot areas in the state-space where the agent's behavior shows deficiencies that do not show up in the average return. This work introduces Deep Statistical Model Refinement (DSMR). DSMR uses the gained information to improve an agent's policy by running a feedback loop of analyzing the agent via DSMC and proceeding with the training process thereafter. The approaches of incorporating the DSMC results into the subsequent training include amplified starts from the weak-spots and a novel way of computing the priorities in a prioritized replay buffer.

Our results show that these approaches improve an agent's policy in tasks with a heterogeneous state-space and make it more stable in all considered tasks.

Contents

Chapter 1

Introduction

In recent years the interest and research efforts in deep reinforcement learning [32] have risen significantly. Deep reinforcement learning algorithms such as DQN [24], A3C [23] or AlphaGo [31] have shown extraordinary results and vastly exceeded human capabilities. However, all these algorithms share the common issue that, usually, the only available performance metric is the average reward the trained agent collects. In most tasks, this average reward will eventually reach a constant level where it does not really change any more. Consequently, it is hard to find termination conditions, and behavior analysis becomes complicated.

Usually, model checking techniques would be beneficial in this case, as they are capable of analyzing and verifying behavioral properties. However, applying model checking to a deep reinforcement learning agent is challenging due to its complex nature. With Deep Statistical Model Checking (DSMC) Gros et al. [16] developed a method to solve this issue. DSMC is a technique that applies statistical model checking [35, 20] to verify the properties of a trained agent statistically. These properties can be visualized with heatmaps, which indicate the areas of the state space where the agent's behavior leaves a lot to be desired, but also areas in which the agent performs properly. These results can be used by a learning engineer to improve the learning process manually.

In this work, we will introduce Deep Statistical Model Refinement. DSMR automates the process of training, analyzing the resulting agent with DSMC, and then using its results to proceed with the training in a more sophisticated manner. We will explore several options of incorporating the DSMC results to overcome the issues associated with the flattening learning curve. The ultimate goal is to increase the probability of reaching the goal by improving the agent's performance in the poor areas while keeping the performance in the good ones intact.

Chapter 2

Background

2.1 Reinforcement Learning

In reinforcement learning, one always tries to train a certain agent to perform some kind of task. A basic model of the training process can be observed in Figure 2.1. The agent will choose an action he thinks is suitable, given his current situation. This action will then influence the environment and thus change its state. To close the loop, the environment will send its new state and some form of reward back to the agent. By constantly repeating this loop, the agent will eventually learn which actions lead to a high reward in the current state.

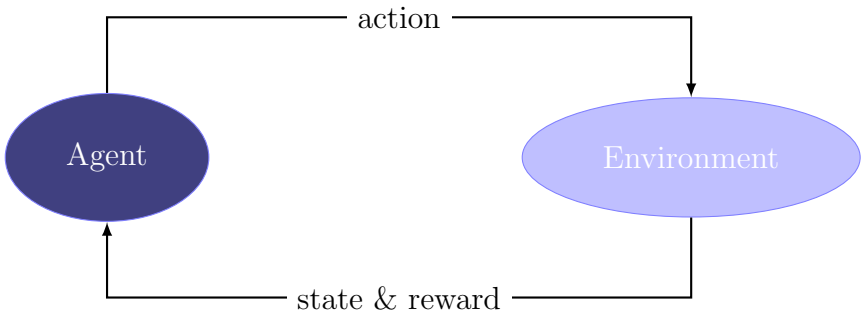


Figure 2.1: Reinforcement loop [32]

2.1.1 The Racetrack Domain

In this work, we will consider the racetrack domain, which is originally a pen and paper game [10] but has recently been used in reinforcement learning tasks [17, 32]. The states of a map are defined by their position $p = (x, y)$ and the agents velocity $v = (v_x, v_y)$. A state can either be empty, a starting state, a goal state, or a wall. The general objective is to traverse a map starting from one of the starting states until a goal state is reached, which is considered a success. An agent can crash by either hitting a wall state or reaching a position outside of the map, which is considered a loss.

In each state, except for the terminal wall and goal states, the agent chooses an acceleration $(a_x, a_y) \in \{-1, 0, 1\}^2$ which is used to compute the new velocity (v'_x, v'_y) as follows: $v'_x = v_x + a_x$ and $v'_y = v_y + a_y$. Then the new position (x', y') is given by $x' = x + v'_x$

and $y' = y + v'_y$. So, the agent can accelerate into one of nine directions by a single unit, and the position will be updated according to the resulting velocity. This implies that once the agent goes in a certain direction with a high velocity, it might be unable to "break" fast enough to avoid a crash. Therefore, racetrack requires foresighted planning. We will consider the noisy version of racetrack where each time the agent chooses an action, it can fail with probability p, where "fail" is defined as $(a_x, a_y) = (0, 0)$. Intuitively, the noise can be seen as a wet road where the agent can lose control over its actions. Figure 2.2 illustrates a number of example tracks for the benchmark domain introduced by Barto et al. [3]. The starting states are marked green, the goal states red, and the wall states are indicated via the x symbol.

Due to the notion of velocity, it is important to define how an agent travels from one state to another, as it is possible to go through a wall if it is located between (x, y) and (x', y'). Gros et al. [17] thus use a discretization similar to the one of Bonet & Geffner [5]. Driving from (x, y) to (x', y') results in the following sequence of visited positions

$$T = \langle (x, y), (x_1, y_2), \ldots, (x_{n-1}, y_{n-1}), (x', y') \rangle,$$

such that

$$T = \begin{cases} \langle (x, y) \rangle & \text{if } v'_x = 0 \wedge v'_y = 0 \\ \langle (x, y), (x + \sigma_x, y), (x + 2 \cdot \sigma_x, y), \ldots, (x', y') \rangle & \text{if } v'_x \neq 0 \wedge v'_y = 0 \\ \langle (x, y), (x, y + \sigma_y), (x, y + 2 \cdot \sigma_y), \ldots, (x', y') \rangle & \text{if } v'_x = 0 \wedge v'_y \neq 0 \\ \langle (x, y), (x + \sigma_x, \lfloor y + m_y \rceil), (x + 2 \cdot \sigma_x, \lfloor y + 2 \cdot m_y \rceil), \ldots, (x', y') \rangle & \text{if } v'_x \neq 0 \wedge v'_y \neq 0 \\ & \wedge |v'_x| \geq |v'_y| \\ \langle (x, y), (\lfloor x + m_x \rceil, y + \sigma_y), (\lfloor x + 2 \cdot m_x \rceil, y + 2 \cdot \sigma_y), \ldots, (x', y') \rangle & \text{if } v'_x \neq 0 \wedge v'_y \neq 0 \\ & \wedge |v'_x| < |v'_y| \end{cases}$$

where $\sigma_x = sgn(v'_x)$, $\sigma_y = sgn(v'_y)$ and $m_x = \frac{v'_x}{|v'_y|}, m_y = \frac{v'_y}{|v'_x|}$. This states that if the agent moves either only horizontally or only vertically, every point between (x, y) and (x', y') is visited. If the agent moves diagonally, T consists of n points on the linear interpolation between (x, y) and (x', y') where each point is rounded to the closest position on the map. Here, n is given by $\max(|v'_x|, |v'_y|)$. Bonet & Geffner used $n = |v'_x|$ which will produce counterintuitive results if $|v'_x| < |v'_y|$. Using T, it is now possible to define a crash and a success. If any position $p \in T$ is a wall state or outside of the map, the crash condition is fulfilled, and if any position $p \in T$ is a goal state, the success condition is fulfilled. Should T fulfill both conditions, only the one to be fulfilled first holds, i.e., if an agent drives through a wall before reaching a goal state, it will be considered a loss and vice versa.

The *Simple* Racetrack Domain

For the sake of introducing reinforcement learning, we will consider a simplified version of the racetrack domain. Namely, we make the following adjustments:

(i) The velocity is reset to $(0, 0)$ after a position change.

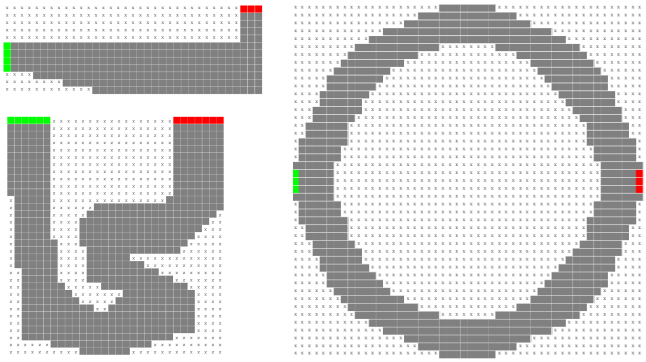


Figure 2.2: Example racetrack maps [16]

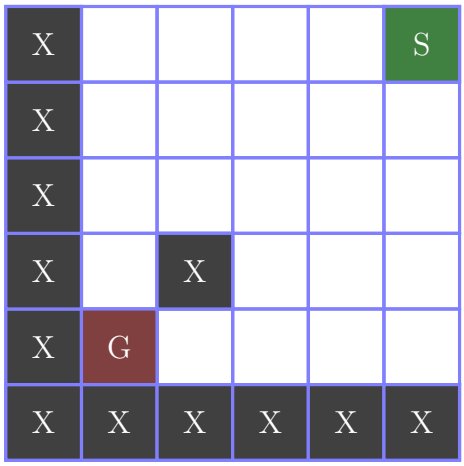


Figure 2.3: 6×6 racetrack map

(ii) The actionset is reduced to $\{(1,0),(0,1),(-1,0),(0,-1)\}$. We will use the notation $\{\rightarrow,\uparrow,\leftarrow,\downarrow\}$ to represent this set in the following sections.

(iii) The noise is redefined such that with probability p instead of executing the chosen action, another random one is executed. We will use the map illustrated in Figure 2.3 in the following sections.

2.1.2 Markov Decision Processes

The underlying model of the environment shown in Figure 2.1 are *Markov Decision Processes* (MDPs) [27]. For any nonempty set S we let $\mathcal{D}(S)$ denote the set of probability distributions over S.

Definition 1 (Markov Decision Process [16])**.** *A Markov Decision Process (MDP) is a tuple $M = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, s_0, \mathcal{R} \rangle$ consisting of a finite set of states $\mathcal{S}$, a finite set of actions $\mathcal{A}$, a partial transition probability function $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \mathcal{D}(\mathcal{S})$, an initial state $s_0 \in \mathcal{S}$ and a reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$. We say that action $a \in \mathcal{A}$ is applicable in state $s \in \mathcal{S}$ if $\mathcal{T}(s,a)$ is defined. We denote by $\mathcal{A}(s) \subseteq \mathcal{A}$ the set of actions applicable in s. We assume that $\mathcal{A}(s)$ is nonempty for each s (which is no restriction).*

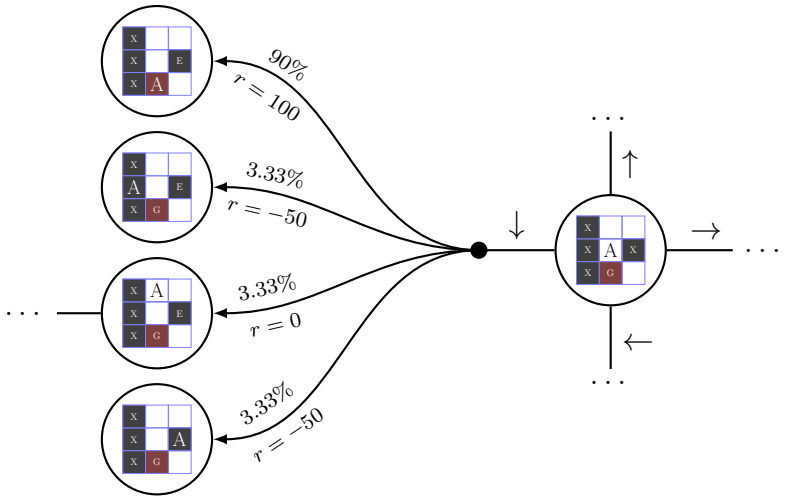


Figure 2.4: Excerpt from Markov Decision Process

Note that the transition probability function $\mathcal{T}$ depends on the current state only. Past states do not influence the current situation. This is called the *Markov property* [27]. As stated before, changing a state, i.e., performing a transition in an MDP, is associated with some kind of reward in reinforcement learning. For this, we use the reward function $\mathcal{R}$ which assigns a numerical reward to a tuple of a state s, an action a and a successor state s'. Like Gros et al. [17] we use a reward function that yields a reward of 100 whenever s' is a goal state, -50 whenever s' is a wall, and 0 otherwise. Figure 2.4 illustrates a small excerpt from the MDP of the example described in Section 2.1.1. The agent's position is represented by the large A. Starting from the state on the right, there is a probability distribution over potential successor states for every possible action as the $\downarrow$ action shows by way of example. This example also shows the rewards associated with reaching the different states, i.e., successfully performing the $\downarrow$ action yields a reward of 100 because we reach a goal state. In contrast, the wall cases yield a reward of -50, and the remaining case yields a reward of 0.

2.1.3 Policies and Value Functions

The process of "playing the game of racetrack" results in a sequence of rewards $R_0, R_1, \ldots$. For the sequence of rewards received after time step t the *return* is defined as

$$G_t = R_{t+1} + R_{t+2} + \ldots + R_T \tag{2.1}$$

where T is the final time step. This definition makes sense in episodic tasks with a clear final time step, like racetrack. However, many tasks require the introduction of a so-called discount factor resulting in the *discounted return*

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}. \tag{2.2}$$

For $\gamma = 1$, the discounted return equals the standard return, and for $\gamma < 1$, the rewards that are received in the future are worth less than the ones received immediately. This discount is mandatory in continuous tasks without a final time step. However, it is also

often useful in episodic tasks like racetrack because it motivates the agent to reach the goal faster. Obviously, the sequence of rewards depends on the actions taken. Therefore, we define a function that maps states to actions, a so-called *policy*. A policy may further depend on a history of states, and it may be probabilistic. Nevertheless, we will only consider history independent, deterministic policies.

Definition 2 (Action Policy)**.** *A (deterministic, history independent) action policy is a function* $\pi : \mathcal{S} \to \mathcal{A}$ *such that* $\forall s \in \mathcal{S} : \pi(s) \in \mathcal{A}(s)$.

The main goal of reinforcement learning is to find the policy that maximizes the expected return $\mathbb{E}_\pi[G_t]$. That is, for every state, the agent tries to find an action such that the expected sum of all future rewards is maximized. We call this the optimal policy π_*.

The so-called *value-based approaches* aim to find π_* by computing certain *value functions*. Those functions assign a numeric value to a state, or a state-action pair, which tells us the reward we can expect to receive from the time step of the given state (or the given state-action pair). First, we take a look at a function that evaluates states.

Definition 3 (State Value Function)**.** *A* state value function *is a function* $v_\pi : \mathcal{S} \to \mathbb{R}$ *such that* $v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$, *for all* $s \in \mathcal{S}$.

So $v_\pi(s)$, given a policy π, maps a state s to the expected return when starting from s and then following π. Intuitively, $v_\pi(s) = x$ tells us that if we start a game from s and follow the policy π we will receive an average return of x. As mentioned before, we can define functions that evaluate state-action pairs in a similar way.

Definition 4 (Action Value Function)**.** *An* action value function *is a function* $q_\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ *such that* $q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$, *for all* $s \in \mathcal{S}$.

We also call the resulting values *Q-values*. In addition to the state, the Q-values also fix the first action that is chosen before π is followed. Thus, $q_\pi(s, a) = x$ intuitively means that if we are in state s then perform action a and subsequently follow π, the expected sum of future rewards is x. Both v and q fulfill the Bellman equations [4], which are recursive definitions as follows

$$v_\pi(s) = \sum_{s' \in \mathcal{S}} P(s, \pi(s), s') \cdot (r(s, \pi(s), s') + v_\pi(s')) \tag{2.3}$$

$$q_\pi(s, a) = \sum_{s' \in \mathcal{S}} P(s, a, s') \cdot (r(s, a, s') + v_\pi(s')) \tag{2.4}$$

$$\tag{2.5}$$

Of course, it would be interesting to know the optimal values of both of those functions. Following the optimal policy π_* we obtain those resulting in the notion of v_*

$$v_*(s) = \max_\pi v_\pi(s), \tag{2.6}$$

for all $s \in \mathcal{S}$ and q_*

$$q_*(s, a) = \max_\pi q_\pi(s, a), \tag{2.7}$$

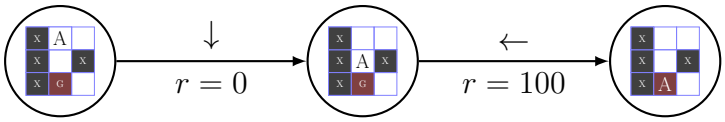


Figure 2.5: Example episode of simple racetrack

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$ respectively. We call v_* the optimal state value function and q_* the optimal action value function. These fulfill the Bellman optimality equations [4]:

$$v_*(s) = \max_a \sum_{s' \in \mathcal{S}} P(s, a, s') \cdot (r(s, a, s') + v_*(s')) \tag{2.8}$$

$$q_*(s, a) = \sum_{s' \in \mathcal{S}} P(s, a, s') \cdot (r(s, a, s') + \max_{a'} q_*(s', a')) \tag{2.9}$$

2.1.4 Q-learning

Q-learning [32] is a value-based reinforcement learning algorithm that aims to find π_*. The idea is that if we can estimate the optimal Q-values deriving the optimal policy is trivial by always choosing the action with the highest Q-value. Q-learning is a dynamic programming algorithm that stores the learned Q-values in a so-called Q-table. The table has an entry for every possible state-action pair. So, in our example case, it has a size of 23 (nonterminal states) $\times 4$ (available actions) $= 92$ entries. Each entry is initialized with 0. There are several approaches to filling the table; however, we focus on the Monte Carlo approach. More precisely, we generate a so-called *episode* by playing the game once, i.e., we start at our starting point S and follow our policy π until we reach a terminal state. When the episode is finished, we have a number of state-action pairs that occurred during our playthrough. For each of these, we apply the Q-learning update rule

$$\hat{q}(s, a) = \hat{q}(s, a) + \alpha(G - \hat{q}(s, a)) \tag{2.10}$$

Where $\hat{q}(s, a)$ is our current entry for (s, a) in the Q-table, G is the return, with $\gamma = 1$, of the episode (100 if we reached a goal state, -50 otherwise) and α is the so-called *learning rate*. We need α to be a positive value close to 0 because otherwise, a single update would have too much of an impact, especially considering the randomness of a single episode.

To give an example, we assume an initial Q-table, a learning rate of 0.001, and the episode illustrated in Figure 2.5. There are a couple of interesting observations to be made here. First, π chose the $\downarrow$ and $\leftarrow$ actions. This is because, as stated before, π depends on the Q-table. It simply chooses the action with the highest value, and as all values are equal at the start, it does not matter which action is chosen, so we assume a random one is taken. Second, although we chose to go left in the second step, we actually went down. This is because of the simple racetrack domain's random noise, which activated in our sample episode. Now that we computed the episode, we have to apply the update rule to all appearing state action pairs

$$\begin{aligned} \hat{q}(s, a) &= \hat{q}(s, a) + \alpha(G - \hat{q}(s, a)) \\ &= 0 + 0.001 \cdot (100 - 0) = 0.1 \end{aligned}$$

The resulting Q-table is displayed in Table 2.1.

	←	→	↑	↓
	0	0	0	0.1
	0.1	0	0	0

Table 2.1: Q-table after first update

X	↓	↓	↓	↓	↓
X	↓	↓	↓	↓	←
X	↓	←	↓	↓	←
X	↓	X	↓	←	←
X	G	←	←	←	←
X	X	X	X	X	X

Figure 2.6: Potential policy learned via Q-learning

We can now either continue to generate episodes and update the Q-values until we reach a fixpoint or until we performed a certain amount of iterations. So far, though, we always *exploited* our current knowledge, i.e., π strictly followed the Q-table. This is problematic because it implies that once we found a good option in a certain state, we will never try to find a better one. In the beginning, especially, the result and generation of the episodes have a large random factor, and it is very likely that the first successful action is not optimal. To solve this, we need to not only exploit our current knowledge but also *explore* the state-action space. In other words, we need to evaluate our different options before we can repeatedly do what worked. The tradeoff between exploitation and exploration is common in reinforcement learning tasks [32]. In order to explore, we introduce an ϵ-greedy policy, which, for a given $\epsilon \leq 1$, chooses the action with the highest Q-value with probability $(1 - \epsilon)$ and a completely random action otherwise. Note that it can still choose the action with the highest Q-value in the random case. Often ϵ is fixed throughout the whole process; however, it is also possible to let it decay over time. If ϵ should decay, we start the learning process with ϵ close to one, as we want to explore almost exclusively since there is no knowledge to exploit yet. Over time ϵ can decay to allow for heavier exploitation of the gained knowledge. Figure 2.6 shows a potential policy learned via Q-learning, and Table 2.2 illustrates an excerpt of the corresponding Q-table.

2.1.5 Deep Q-learning

The major problem with Q-tables is their size, as the size of the state-action space explodes when using larger racetrack maps. It is often completely unfeasible to store the whole table in memory, and as a result, Q-learning fails. Historically, researchers have used function

	←	→	↑	↓
	-40.32	71.43	65.76	77.49
	-39.51	-41.63	75.38	81.37

Table 2.2: Excerpt from the Q-tables of the policies shown in Figure 2.6

approximators, such as linear functions, to replace the Q-table. These approximated functions map a state to the corresponding Q-values. In deep reinforcement learning (deep), neural networks are used for this purpose [32]. We will not cover the details of neural networks here as it is sufficient to imagine them as a function approximator. Neural networks use the weight vector θ to estimate the Q-value function $Q(s, a; \theta)$ as a deep Q-network (DQN) [24]. Without a full Q-table, it is impossible to use the standard Q-learning update rule from Section 2.1.4. Instead, the Q-value estimates are used to compute the target

$$y(s, a; \theta) = \mathbb{E}[R_{t+1} + \gamma \cdot \max_{a'} Q(S_{t+1}, a', \theta) | S_t = s, A_t = a] \tag{2.11}$$

which is then used to optimize the loss function in iteration i

$$L(\theta_i) = \mathbb{E}[(y(S_t, A_t; \theta^-) - Q(S_t, A_t; \theta_i))^2] \tag{2.12}$$

by approximating $\nabla L(\theta_i)$ and then using stochastic gradient descent [25]. The idea of using deep neural networks is not new; however, using them has long been highly unstable, preventing strong results. Mnih et al. [25] solved this with the introduction of mainly two optimizations to the algorithm. First, they used a *fixed target* $y(S_t, A_t; \theta^-)$ in their loss function where θ^- are the weights from some previous iteration. Thus, the target does not depend on weights from the current iteration. After a fixed amount of steps θ^- gets updated to the current weights $\theta^- = \theta_i$. Second, they used a so-called *experience replay buffer* [25]. Stochastic gradient descent requires independent and identically distributed samples, which is not the case when learning from episodes as we did so far. To solve this, the experiences gained during an episode are stored in a buffer as tuples $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$. When computing the target $y(S_t, A_t; \theta^-)$, random experiences are drawn from the replay buffer to estimate the expectation.

2.2 Deep statistical model checking

Deep statistical model checking (DSMC) [16] is a method which applies model checking techniques to a learned reinforcement agent. Usually, the performance evaluation of such an agent heavily relies on its average return. The graph on the right of Figure 2.7 shows a so-called *learning curve* of a deep reinforcement learning agent. The curve shows the progress of the average return during the learning process. These racetrack settings were $p = 0.2$, i.e., 20% noise, and a reward structure of $+100$ if the agent reached the goal and -50 if it crashed. The discount factor was $\gamma = 0.99$. The basic appearance of most learning curves is similar to the one in the given graph in a way that it starts slow, then explodes for a short period of time, and then calms down again to reach the

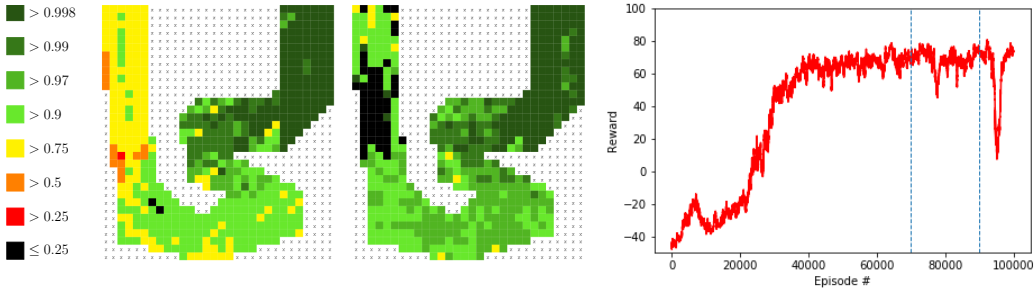


Figure 2.7: DSMC racetrack heatmaps (left) and an agents learning curve (right) [16]

aforementioned constant level. In these cases, training beyond the explosion is slow, and it is often unclear how and if at all, the agent is still improving. Due to the lack of further evaluation methods, it is frequently unclear when to stop training and, if further training is necessary, how to proceed exactly.

DSMC allows us to analyze the performance in a more sophisticated way as it checks certain properties of the agent's behavior, which can indicate its deficiencies. More precisely, the probability of fulfilling the CTL formulas $\Diamond crashed$ ("eventually crashed") and $\neg crashed\ \mathbf{U}\ goal$ ("not crashed until goal") is estimated for each position on the map. This is done by simulating a number of racetrack runs starting from this position with velocity 0. Gros et al. [16] achieve an error bound of $P(error > \epsilon) < \kappa$, where $\epsilon = 0.01$ and $\kappa = 0.05$, i.e., a confidence of 95 % that the estimation error is smaller than 0.01.

Figure 2.7 shows the visualization of these properties in the form of heatmaps. A yellow region denotes a $> 75\%$ probability to reach the goal before a crash occurs if the agent starts from this point with velocity 0. The legend on the left characterizes the different colors. The two heatmaps correspond to the points in time during the learning process indicated by the two blue dotted lines in the learning curve graph, i.e., the left heatmap was computed after 70.000 episodes, and the right heatmap was computed after 90.000 episodes. As we can see, the average return was about equal at both points in time. However, the heatmaps display highly deficient behavior in certain areas after 90.000 episodes. This knowledge can be used to improve further training, e.g., by starting a few training runs from the black area.

2.3 Safe Reinforcement Learning

While it is sufficient to optimize the long term expected return in many instances, some use cases require the additional notion of *risk*. For example, in autonomous driving, it is crucial to avoid catastrophic events at almost all costs, even if it comes at the expense of optimality. *Safe Reinforcement Learning* algorithms try to incorporate some form of risk in order in order to prevent damage to the agent or its surroundings. Garcia et al. [9] categorize two SafeRL approaches: Changing the optimization criterion and manipulating the exploration process.

The optimization criterion can be subcategorized into four groups:

(i) The *worst-case criterion*, which tries to find the policy with the best worst-case. In racetrack, this policy would be standing still, as the worst-case would be a return of 0.

(ii) The *risk-sensitive criterion*, which tries to balance the maximization of the expected return and the minimization of risk. How exactly risk is defined depends on the approach and requires a risk metric. A common metric is the variance of the return [6, 11, 19]. Geibel et al. [13] define the risk as the probability of reaching an error state, and in a recent approach, Wen et al. [34] learn a risk function during their training.

(iii) The *constrained criterion*, which limits the set of possible policies. These approaches search for the policy that offers the maximum expected return while complying with certain constraints.

(iv) Other optimization criteria. These approaches use criteria from the area of financial engineering, such as *r-squared*, *value-at-risk* [21] or the *density* of the return [26].

Modifying the exploration process can be divided into two groups:

(i) *Incorporating external knowledge.* The ways of integrating this knowledge range from providing initial knowledge, eliminating the need of random exploration at the beginning, over deriving a policy from a finite set of demonstrations to providing teaching advice. The teacher can be an algorithm or even a human. The approaches differ in the way the help is initiated. While the agent may ask for advice when it feels insecure [7, 8] the teacher can also intervene and provide an action whenever necessary [33, 28]. Recently, Mohammed et al. [2] synthesize a shield which verifies every action from the agent and provides a safe one whenever necessary.

(ii) *Risk directed exploration*, which uses a risk metric to guide the exploration. Gehring et al. [12] use the risk metric of controllability and guide their agent to controllable regions of the state space.

Note that the second category includes the first, as modifying the optimization criterion will also modify the exploration process. Most work in this field is not only concerned with the safety of the final result but also with avoiding critical states during the exploration process. While this is important for online learning agents (i.e., agents that are still learning while already performing their real-life task), it is of less importance for our approach as a crash during learning has no negative impact. Still, staying safe during exploration has the added benefit of ignoring irrelevant parts of the state space, accelerating the learning process.

Chapter 3

Training Approaches

In this chapter, we will describe the different approaches of incorporating the DSMC results into the training. First, we will discuss the DSMC analysis itself and then continue by presenting the different algorithms we used for the training.

3.1 DSMC Analysis

As already described in Section 2.2, we use DSMC to estimate the probability of fulfilling the boolean CTL formula $\Diamond crashed \wedge \neg crashed \ \mathbf{U} \ goal$ ("goal-reaching probability") for every valid starting position on the map (i.e., every position that is neither a wall nor a goal). For each of these estimations, we need the probability of the error being smaller than a certain ϵ to be greater than a certain confidence $1 - \delta$. Or more formally: $P(error < \epsilon) \geq 1 - \delta$. We can use the formula provided by Aichernig et al. [1] to calculate the required number of simulations as follows:

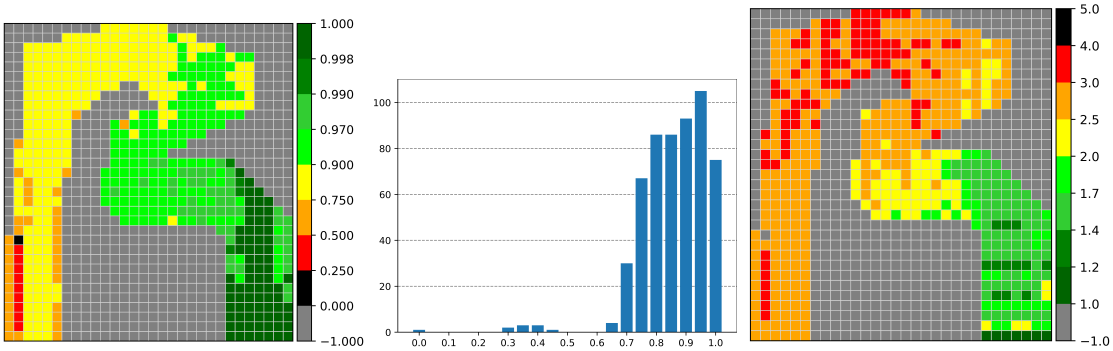
$$n \geq \frac{1}{2\epsilon^2} \ln(\frac{2}{\delta}) \tag{3.1}$$

If not otherwise specified, we perform our analysis with a confidence of 95 % that the error is smaller than 0.05, which means that we need to perform 738 runs starting from each position on the map. Unfortunately, this results in much computational effort for larger maps. However, DSMC is perfectly suited for concurrent execution as the simulations of two different positions do not interfere with each other at all.

For this work, we not only compute the goal reaching probability for each position but also the following values:

- average of the goal-reaching probabilities,
- variance of the goal-reaching probabilities,
- average return and
- average length of successful runs.

Figure 3.1 shows the different visualizations we produce. From left to right, it shows the goal-reaching probability heatmap introduced by Gros et al. [16], a histogram of the goal reaching probabilities, and a speed heatmap. For the speed heatmap, we computed the optimal path for each position using a standard A-Star algorithm. However, these optimal paths are computed with 0% noise, i.e., in a deterministic environment. Obviously,



a) Goalprobability Heatmap b) Goalprobability Histogram c) Speed Heatmap

Figure 3.1: Example DSMC heatmaps for the Barto-Big map

the optimal paths are longer when using a higher noise as the agent has to drive more carefully. The speed heatmap displays the average length of the successful runs starting from the corresponding position in relation to the length of the path provided by the A-Star algorithm. The histogram, the speed heatmap, the average return, and the average length of successful runs will not be used for our analyses in the following chapters. We showcased them here as these values and graphs can be useful in general and were used for our evaluations at first. We eventually emitted them after we determined our training methods.

3.2 The Default Agent

In this work, we will consider a standard DQN algorithm similar to the one used by Gros et al. [17]. We will refer to this approach as the *default agent*. This agent is trained by using a uniformly random start, i.e., the episodes start at a random point on the map with velocity zero. The state is encoded by providing the following state features:

- position on the map.
- current velocity.
- $d_1, \ldots, d_s$: linear distance to a wall in all directions. These eight distances are distributed equally around the car position and are given analogously to the acceleration, i.e., -1, 0, or 1 in both dimensions.
- dg_x, dg_y: distance to the nearest goal field in x and y dimension respectively.
- dg: total goal distance, i.e. $|dg_x| + |dg_y|$.

If not otherwise specified, we will use a reward structure of 100 if a goal is reached, -50 if the agent hits a wall, and -5 if the agent does not accelerate when already standing still. Or more formally:

$$R(s \xrightarrow{(a_x,a_y)} s') = \begin{cases} 100 & \text{if } s' \text{ is goal state} \\ -50 & \text{if } s' \text{ is wall state} \\ -5 & s = s' \\ 0 & \text{otherwise} \end{cases}$$

The third condition is sufficient as $s = s'$ is only possible if s has velocity $(0, 0)$ and $(a_x, a_y) = (0, 0)$. In every other case, either the position or the velocity of s would change and thus lead to some other state $s'' \neq s$. The discount factor is set to 0.99, the buffer has a size of 10^8 entries, and the neural network has 4 layers. The input layer has 15 entry nodes, one for each state feature, the second and third layer have 64 nodes, and the output layer has 9 nodes, one for each action. The network is randomly initialized. As explained in 2.1.4 we use an ϵ-greedy policy to balance the tradeoff between exploration and exploitation. We start the training with $\epsilon = 0.95$ and after every episode we decrease ϵ exponentially with a factor $\lambda = 0.999$ until a final threshold of $\epsilon = 0.05$ is reached.

3.3 General Idea and Termination

The general idea of the training is to use the default agent for a few thousand episodes until the learning curve reaches the constant level described in Section 2.2. We call the resulting agent the *initial agent*. We will then refine the initial agent by continuing the training with the different DSMR approaches. Note that the content of the experience replay buffer is not stored, so each approach starts with the network weights of the initial agent but with an empty experience replay buffer. The refinement process consists of several *refinement batches*, each of which consists of a DSMC analysis and 1000 subsequent episodes of training. After the final refinement batch, one last DSMC analysis is performed. The analyses' results are incorporated into the training as described by the following sections and used to determine the agent's performance at that timepoint. During this process, the buffer is kept intact.

As discussed in Section 2.2, it is hard to determine when to stop the training or when the agent is performing the best. Gros et al. [17] computed the average return of every 100 training episodes and stored the network weights that achieved the best result. However, this method is prone to favor a lucky agent, i.e., if, for example, a majority of the 100 episodes randomly started close to a goal state, the average return will be naturally high. The frequent DSMC analysis enables a more sophisticated way of determining the best agent. In theory, all of the values described in Section 3.1 or any combination thereof can be used. One simple termination criterion could be that the agent achieves a goal probability of at least 90 % from each position on the map. For this work, we use the average goal-reaching probability to determine the best agent.

We will use the default agent as a baseline for the performance of the different approaches. For the sake of fairness, the default agent will also be trained in refinement batches to have the same amount of analyses and thus potential best agents, although the analysis results will not be used.

3.4 Spreaded Agent

The algorithm to train the spreaded agent is mostly identical to one of the default agent. The difference is that instead of selecting a uniformly random starting point for each episode, the DSMC results are used to build a random distribution. More precisely, a weighted average over the goal-reaching probabilities is built such that any position $p' \in \mathcal{P}$, with $\mathcal{P}$ being the set of all valid starting points, has a probability $\mathcal{SP}$ of being

chosen as the starting point of

$$\mathcal{SP}(p') = \frac{1 - goalProb(p')}{\sum_{p \in \mathcal{P}} 1 - goalProb(p)} \tag{3.2}$$

So, in general, the lower the probability to reach the goal from a certain point, the higher the chance to start a training run from there. These probabilities are updated after the analysis of each refinement batch. Over the course of a training run, the agent's performance will fluctuate for every starting point, and so will the probability of starting from this there. As the replay buffer is not reset during the training run, this results in the buffer being filled with episodes in proportion to the "difficulty" of the starting position of the episode because the more difficult positions will have a low analysis result on average.

This approach can actually be seen as a way of manipulating the exploration process. By using the knowledge we gained thanks to DSMC we force the agent to explore the regions with subpar results more intensively. We still need the ϵ-greedy policy, though, as the agent needs to be able to actually explore these regions and not just start from them.

3.5 Prioritized Agents

A common improvement to the DQN algorithm is a so-called *prioritized replay buffer* [30]. As mentioned in Section 2.1.5, an *experience replay buffer* is used to break correlations in experience batches by storing samples from many episodes and sampling them randomly. However, usually, not all samples are equally useful for the learning process, limiting the effectiveness of a standard replay buffer. The idea of a prioritized replay buffer is to sample according to a set of priorities. For each sample (s_t, a_t, r_t, s_{t+1}) we thus need a way to compute its priority such that the learning process is optimized. A common priority is the magnitude of the TD error

$$\delta = |Q(s_t, a_t) - y(a_t, s_{t+1})| + \epsilon \tag{3.3}$$

which is then raised to the power of a meta parameter α. This α is used to control the amount of prioritization, i.e. $\alpha = 0$ means no prioritization and $\alpha = 1$ represents a full prioritization. The ϵ is a small constant to ensure that a sample's priority is never 0.

In this work, we want to incorporate the DSMC results into the priority formula. We came up with two possible ways of doing this, which are described in the following sections. What both of them have in common is that the priorities of all samples in the buffer need to be updated after every analysis. This may take a few seconds, but it is basically just an extension to the analysis time, which will be discussed later.

3.5.1 Position Prioritization

The first idea is to use the position of the sample itself, which is, conveniently, stored in the state s_t. So, given a sample from position (x, y) its priority is computed as follows:

$$\delta = (1 - goalProb(x, y) + \epsilon)^{\alpha} \tag{3.4}$$

Note that this ignores the velocity of the sample. The goal probability is computed starting from (x, y) with velocity 0. The idea is to focus on all samples from a position with bad

performance. Unfortunately, this often results in samples being prioritized, which are not really useful for the learning process, which is why we will not use this approach in the following chapters.

3.5.2 Start Prioritization

The second idea is to extend the samples with the starting position of the corresponding trajectory as follows $(s_t, a_t, r_t, s_{t+1}, (x_0, y_0))$ and then compute the priorities similar to Equation (3.4)

$$\delta = (1 - goalProb(x_0, y_0) + \epsilon)^\alpha \tag{3.5}$$

which prioritizes a complete trajectory based on the success probability of its starting point. This can be helpful to improve the regions with bad performance as the agent is forced to completely replay the trajectories from these regions more often.

Chapter 4

Discussion of the Training Parameters

In this chapter, we are going to discuss why we chose certain training parameters over others and why we chose certain maps to illustrate the results of the DSMR approaches. The runtime of the approaches will also be discussed. All of these measurements were performed on an Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz with 12 cores, 24 threads, and 96GB RAM, as were the results in Chapter 5.

4.1 Racetrack Maps

When choosing a racetrack map for our benchmarks, we initially tried the Barto-small map introduced by Barto et al. [3]. The left graph of Figure 4.1 highlights this map again. Remember that the green are the starting states, and the red are goal states.

We tried to train our different agents on this map with 50 % noise. Unfortunately, the performance of our different DSMR agents did not differ much from the default agent. The initial agent was trained for 10.000 episodes. We then trained the different agents for 50 refinement batches, i.e., 50.000 episodes. This whole training process was done for several different random seeds, and the results of the respective best agents were merged to reduce the impact of randomness. The average over the best results from the default agent is 92.67 % goal probability while the spreaded agent achieves a 92.83 % probability to reach a goal. The heatmaps of both approaches also look very similar as Figure 4.2 shows.

We take account of two reasons for these results—first, the difficulty of the map. The right graph of Figure 4.1 displays the goal probability heatmap of the optimal policy.

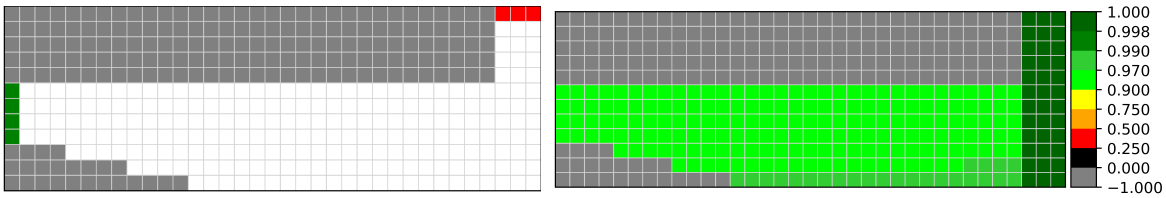


Figure 4.1: Barto-Small map (left) and the heatmap of the optimal policy (right)

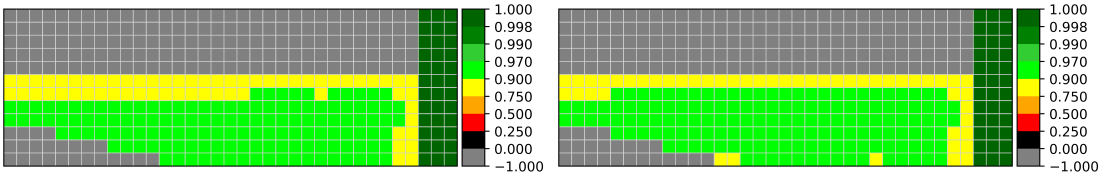


Figure 4.2: Heatmaps of the default agent (left) and the spreaded agent (right)

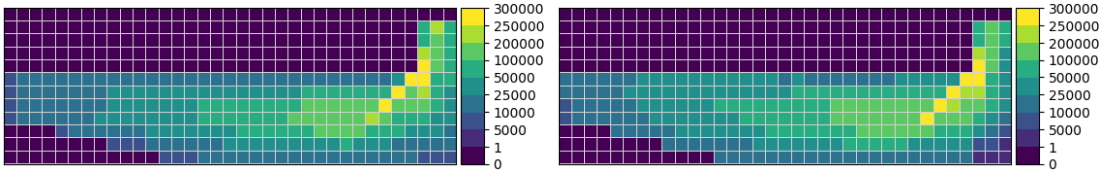


Figure 4.3: Buffer sampling heatmaps of the default agent (left) and the spreaded agent (right)

It was computed using the Modest toolset [18]. This policy reaches an average goal probability of 95.41 %. So, the default agent comes relatively close to the optimum already and does not leave much room for improvement. The second reason is the nature of the map. It consists of basically two regions: the right pillar, where the agent only has to drive straight up, and the left region. The states in the pillar are almost identical to one another, and the states in the left region are all at least very similar. Our method revolves around starting more often from difficult regions. If, however, almost all points are equally difficult, this does not help much. We also extended the replay buffer to store how often each drawn sample came from a certain position. Figure 4.3 displays the sampling heatmaps of the two agents. They were computed by remembering the position of every sample, ignoring their velocity, every time we draw from the replay buffer. Note that the scale is not linear. These graphs clearly show that when using the spreaded agent, the more difficult regions (the top lines of the left region) get sampled more frequently, and the easier regions (the right pillar) get sampled less frequently. However, the difference is only a few thousand times, which is marginal as the top sampled regions were sampled around 300.000 times. This is due to the nature of the map as every trajectory that starts from anywhere in the left region has to go through this heavily sampled region to reach the goal, which results in the buffer being filled disproportionately with samples from there. As this is a common phenomenon for racetrack maps, we will call these regions *mandatory regions* from now on. This also explains why the top of the right pillar is sampled so often for both agents. So, there is a difference between the default and the spreaded agent, but it is not much, and the results reflect that.

In order to better test the performance of DSMR we built a number of racetrack maps to be used for our evaluations in Chapter 5. The initial idea was to offer a second, differently shaped route to the goal in order to eliminate or at least spread out the mandatory region. This resulted in the narrow alley map displayed on the left of Figure 4.5. Another idea behind this map is to have one long but easy to drive region (long straights) and one short but hard to drive region. We will call this hard region the shortcut. Due to the shortcut consisting of just a few positions and them yielding short trajectories, the buffer will contain almost no samples from there, making it extremely unlikely to draw

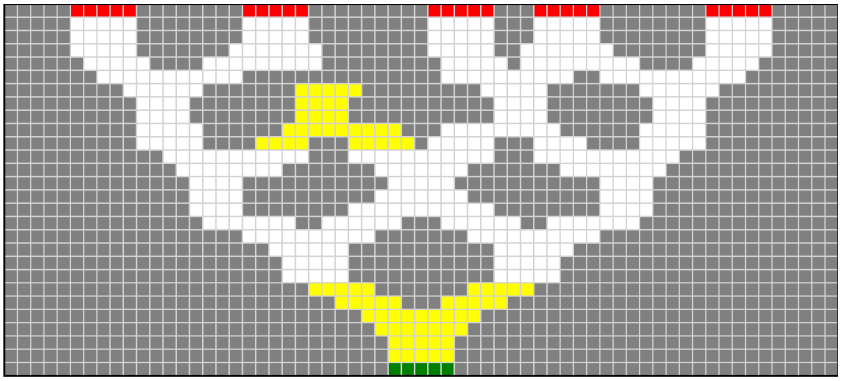


Figure 4.4: River map

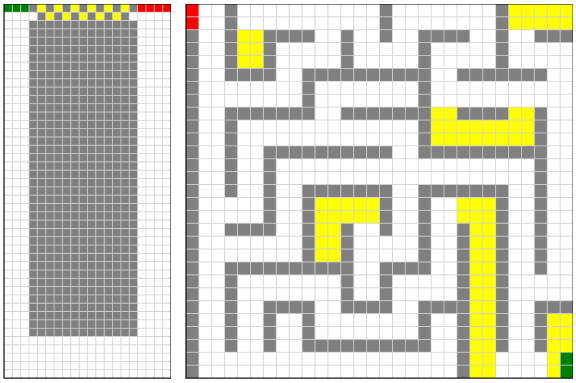


Figure 4.5: Narrow alley map (left) and Maze map (right)

them. The color scheme is similar to the one chosen in Section 2.1.1, i.e., the starting states are marked green, and the goal states are marked red. We have added "interesting regions", here colored in yellow, which we will need for our evaluations in Chapter 5. Note that for all maps but the narrow alley map, the starting positions are also part of the interesting regions. An extension of the idea with the alley was to have multiple goals, which is implemented in the river map shown in Figure 4.4. It offers multiple goals and requires multiple changes of direction due to its narrow frame. There is also a blind alley where the agent has to turn around completely should it end up there. One might argue why we are even interested in performing well in this region as it is a part of the statespace that is completely irrelevant when starting only from the starting positions of the map. However, we are interested in an agent that is able to drive from everywhere on the map, including this blind alley. On the right of Figure 4.5 the maze map is shown. It is designed to be very difficult. Obviously, the region to the left just before the goal is mandatory. However, the map is long enough such that all other regions will be sampled often enough as well. For the sake of comparability, we will also analyze the performance of the different approaches on the Barto-big map and the ring map. In Figure 4.6 we added the interesting regions for these maps.

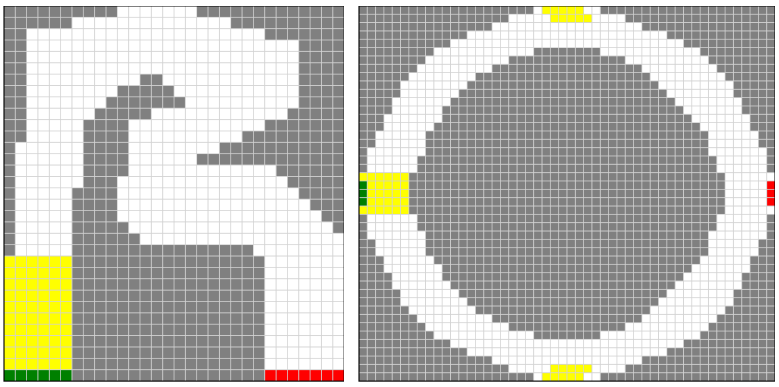


Figure 4.6: Barto-big map (left) and Ring map (right)

4.2 DSMC Accuracy and Refinement Batches

As discussed in Section 3.1, we performed the DSMC analyses during the training with a confidence of 95 % and an error of 0.05, which required at least 738 runs from each possible starting position. As the Barto-small map has a total of 233 valid starting positions, we need to run 171.954 simulations at the start of each refinement batch. Due to the concurrent execution, this takes roughly 30 seconds on our machine. While this may seem fairly low, we need to consider the amount of refinement batches and the required time for the actual training. The agents shown in Section 4.1 were trained for 50 refinement batches. The analyses took a total time of 25 minutes, while the training took about 17 minutes. This ratio gets substantially worse for bigger or more complicated maps such as the maze one. This map has 606 valid starting positions, which means 447.228 simulations are necessary per analysis. In addition, each simulation takes longer as the map is harder to drive and a lot longer. Therefore, each analysis with the current accuracy took about 230 seconds while the 1000 training episodes only took about 40 seconds. When using one of the prioritized agents as described in Section 3.5 we also need to consider the time needed to update the buffer priorities. Currently, this is done by iterating over the buffer and updating each priority individually. The runtime of this heavily depends on the occupancy of the buffer. At the very end of the 100 refinement batch cycle, it takes about 2 minutes. Admittedly, we have not optimized this at all yet, and there is surely much room for improvement.

We chose these settings as they offer a good tradeoff between accuracy and runtime. The runtime outlined above is still manageable while the results are accurate enough for DSMR to work. Obviously, the computed goal probability for each position will usually be off by a few percentage points. However, our goal is to fill the replay buffer disproportionally with trajectories from the hard regions. We update the goal probabilities, and thus the chance to start from each position, regularly, and we never clear our rather large buffer. Hence, the inaccuracies will eventually even out, and the buffer will be filled according to the difficulties of the positions. Another issue arises with the termination or, in our case, with the choice of the best agent. As the learning is not linear, there are usually multiple timepoints at which the agent is similarly good. Out of all these potentially best agents, we choose the one with the highest analysis result with respect to the average goal probability. Obviously, this method is prone to choosing the luckiest

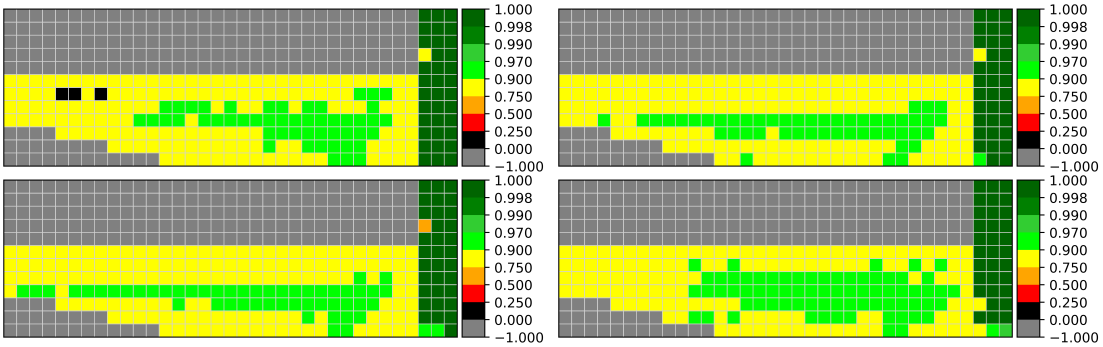


Figure 4.7: Analysis results of 4 consecutive refinement batches

of those agents, not necessarily the best one. Anyhow, this is the tradeoff that we have to make. The differences between all those potentially best agents will simply not be large enough to merit the runtime loss a more accurate analysis would cause. Also, if we would really like to find the absolute best agent during our whole learning process, we would need to analyze after every single neural network update, not only after every 1000 episodes. The point made in Section 3.3 about the choice of the best agent still stands, though. We may not always choose the absolute best agent, but our method is still a lot more stable than a standard approach without DSMC, i.e., only considering the average return during training.

As for the refinement batches, we chose to train for 1000 episodes after each analysis. This is again a tradeoff between accuracy and runtime. Obviously, the more frequently we analyze the agent, the more accurate and up to date our current starting probabilities will be. Over the course of the whole training process, there will be analysis results that do not match our notion of "hard regions have low goal probabilities". More precisely, sometimes, a position which is actually from an easy region will have a poor analysis result and thus an amplified starting probability. After such an analysis, our trajectories will start from this easy position more often than we would perhaps like. However, if the agent is currently bad at starting from this particular position, this behavior is certainly merited. The only advantage a shorter than 1000 episode cycle would have over our approach is that changes in the agent's behavior would be tracked faster. This would, however, significantly deteriorate the analysis time to training time ratio. In Figure 4.7 we can observe the analyses results of a default agent at four consecutive points in time with 1000 episodes of training between each of them. The points in time are ordered from the upper left to the upper right to the lower left, the lower right. The graphs show that an agent's behavior can change rather quickly for individual positions, so increasing the number of episodes for our refinement batches beyond 1000 seems not ideal. It actually does not matter much which of our agents we choose. All of them show similar behavior in terms of how quickly they change. A spreaded agent would have started more often from the black regions, but those additional trajectories would only be a drop in the ocean that is our large replay buffer. As reasoned before, we are interested in filling the buffer in proportion to the difficulty of each region, which we do, over the course of our whole training process, but not always over the course of a few thousand episodes.

4.3 Reward Structure and Discount Factor

As mentioned before, the reward structure we chose is

$$R(s \xrightarrow{(a_x,a_y)} s') = \begin{cases} 100 & \text{if } s' \text{ is goal state} \\ -50 & \text{if } s' \text{ is wall state} \\ -5 & s = s' \\ 0 & \text{otherwise} \end{cases}$$

in combination with a discount factor of 0.99 and a maximum length of a single episode of 100. Should the agent still drive around the map after 100 steps, the episode will be aborted, and the return of the episode will thus be 0. 100 steps are way more than enough to reach the goal on all of our maps. Should the agent not have reached the goal by then, we can safely assume it will not do so in the future.

The problem with this structure is that the agent will always try to maximize the return. However, this structure, combined with the discount factor, does not represent our actual learning goal: maximizing the goal-reaching probability. The main culprit is actually the discount factor. Because it is smaller than 1 (the non discounted case), the agent tries to find a tradeoff between reaching the goal as fast as possible and reaching it at all. When only trying to reach the goal, no matter how fast, it does not make sense at all to accelerate beyond velocity 1. Obviously, our agents will not comply with this "constraint" as they try to maximize the discounted return.

So, what we would like to use is a reward structure of

- +1 if the agent reaches a goal
- 0 in any other case

and a discount factor of 1, i.e., no discount. We call the resulting agents *binary agents*. This structure represents the goal-reaching probability. Unfortunately, it does not yield positive learning results and usually results in a learning curve similar to the one in Figure 4.8. This is the curve of a binary agent which was trained with 50 % noise and obviously without a discount. We trained the initial binary agent for 10.000 episodes and, just for the sake of illustration, used the default binary agent to train it for 20.000 more episodes. As we can see, the agent does learn how to drive reasonably well but forgets everything it learned over the course of a few thousand episodes. Unfortunately, the agent is unable to relearn the task, and the DSMC results are useless in this case, as every position has an almost equal goal-reaching probability of close to 0 %. Therefore, using any of our other agents would not make a difference. There is a common phenomenon in reinforcement learning called catastrophic forgetting [22, 29, 14]. But, this field of research is concerned with training an agent in one task and then teach it another, different task. Catastrophic forgetting describes the problem of forgetting how to do the first task when trying to learn the second one in this setting. This is not what we are experiencing here, as we do not have a second task.

Instead, the problem we have lies in the nature of the q-values and our policy. Without a discount, it does not matter how close a state is to the goal. By design, the optimal q-value, the q_*-value, of a state is equal to the goal probability. Usually, states that are close to each other have a similar goal probability and thus very comparable q_*-values.

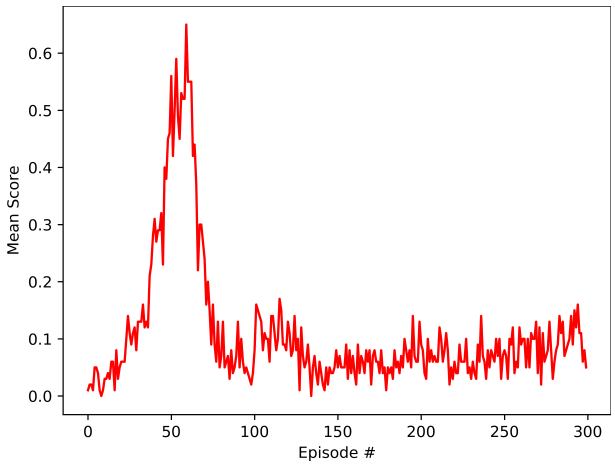


Figure 4.8: Learning curve on Barto-Small for the non-discounted case

Similarly, most (or at least multiple) actions of a particular state will also have near-identical q_*-values. It takes the agent a while to compute these values. That is why we see the rising learning curve before the catastrophic fall. In q-learning in general, we do not compute the exact q_*-values; we approximate them. Replacing the q-tables by neural networks adds another layer of approximation. Always choosing the action with the currently highest q-value as a way of determining our policy works because these approximated q-values are sufficiently accurate. We do not need to know the exact values as long as the action with the highest real q_*-value also has the highest approximated q-value. Unfortunately, in our current setting, the exact q_*-values are so close to each other that even a small approximation error leads to a suboptimal action being chosen. So, once the values are computed, the agent does not know where to drive as all states around it seem equally useful, and the action which happens to have the highest approximated q-value is chosen.

In his master thesis, Gros [15] makes similar observations and additionally finds that the q-values actually drop when coming close to the goal. It is unfortunately unclear why this happens, but it explains the catastrophic fall. As long as the q-values in the goal region are marginally higher than the values of the other regions, the agent will find the goal quite consistently. However, as soon as this ratio changes, it is basically impossible for the agent to find the goal.

The red line of Figure 4.9 shows an exemplary path our binary agent could take. This particular path was computed using the initial agent from Figure 4.8, i.e., an agent with an experience of 10.000 episodes. It was started from position $(7, 15)$ with position $(0, 0)$ being the upper left corner. In this case, we only wanted to see the agents decisions without the noise messing with the results, so we used a deterministic environment, i.e., $0\,\%$ noise. Keep in mind that our agent's action choices are also deterministic as the q-values computed by a neural network for any state s are deterministic. We let the agent take 30 steps. Yet, the path only covers 5 positions, which is because the agent was always driving from left to right due to the similar q-values. In this particular case, the agent does know that driving up or down is not a good idea because of the high noise. Nevertheless, it cannot distinguish between the value of going left or the value of going right. It will thus go either left or right "randomly". Similarly, our agent will drive up

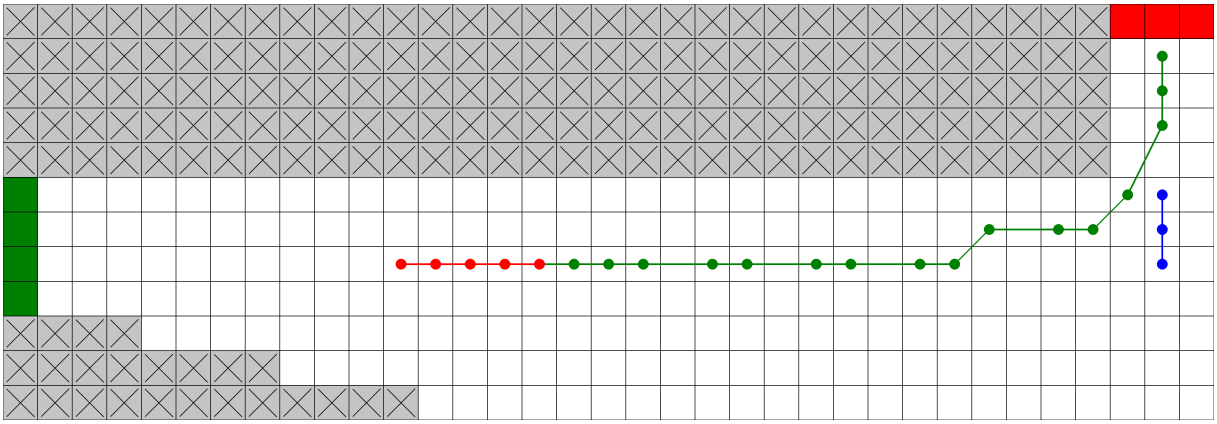


Figure 4.9: Example paths of an agent trained without a discount.

and down when starting from position $(7, 33)$, as the blue line of Figure 4.9 shows. We have also included a third, green line here. It was started from position $(1, 33)$, i.e., one position away from the goal. In theory, a single action (one of the three "up actions") would have sufficed to reach a goal, yet the agent "purposefully" drove away from the goal. This behavior can be explained with the findings of Gros [15] we have mentioned above. The green line is actually partly covered by the red one as the path heads to the respective positions and ends up revisiting them over and over again as well. To solve this problem, we tried to manipulate the action choices of the agent. As we have seen, it tends to drive back and forth, which corresponds to traversing the statespace in circles. We tried three slightly different approaches to prevent these circles

- preventing the agent from closing the circle,
- preventing the agent from reentering the circle and
- preventing the agent from revisiting a position.

All of which require us to store all the states that were visited during the current trajectory. The approaches are discussed in the following sections. For the sake of comparison, we will also consider the red and the blue paths with their respective starting positions as we have done here. We will omit the green path to increase readability. Each of the approaches requires us to prevent the agent from taking a specific action or even multiple actions. Should one of those forbidden actions have the highest q-value, we force the agent to take the action with the second-highest q-value instead. Should this second action also be forbidden, we force the agent to take the third highest-rated action and so forth. If no action is usable, the agent simply takes a random one.

4.3.1 Never close the circle

This approach aims to prevent the agent from revisiting a state, i.e., never close a circle in the statespace. For this to work, we need to store all visited states of a trajectory. Every time the agent chooses an action, a check is performed if this chosen action would lead to an already visited state, and if so, the action will be prevented as described above. Unfortunately, this "improvement" does not help much as Figure 4.10 shows. The learning curve looks very similar to the one in Figure 4.8. The difference is that there is a small

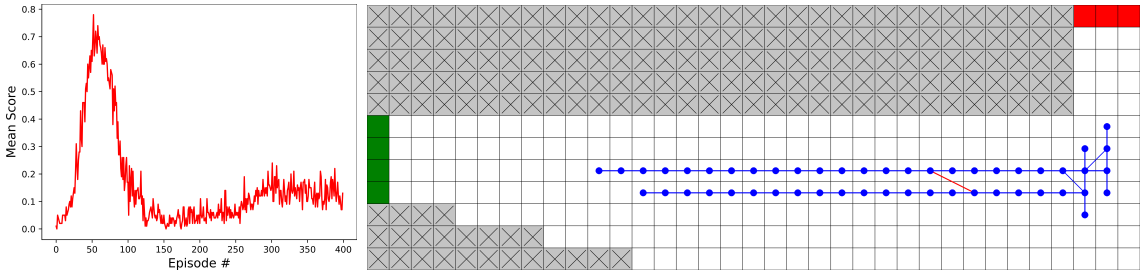


Figure 4.10: Learning curve and example path when preventing to close the circle

rise after the catastrophic fall, but it does not matter much as the rise stops way before a decent level. The right graph shows the two example paths defined above. We used the agent with 40.000 episodes of experience and a deterministic environment to generate those paths. Both paths are 100 steps long and overlap almost completely. We can see that the agent is still driving back and forth, although it has to be a bit more creative than before. The problem here simply is that racetrack offers a larger statespace than one would initially expect when looking at the picture of the map. It is very well possible to drive back and forth for extended periods without visiting a state twice. Going in one direction and then going back can be done without revisiting a state as the velocity will be contrary. It is also possible to visit a position multiple times, going in the same direction. For example, once with velocity (1, 0) and once with velocity (2, 0). This explains why we almost do not see the red line in Figure 4.10. After a few steps, the blue agent will reach a state that the red agent also visited. It will proceed to drive the same path the red agent took due to the deterministic environment and action choices. It will not follow the complete path as it started from further away, which is why we see the single red line in the middle.

We could increase the number of steps the agent can take, but even if it does reach the goal, it would be more of a random event than a real achievement. Additionally, if the agent takes a trajectory like the one we see in Figure 4.10 and then, by chance, eventually reach a goal, it would increase the q-values of every action it took during this long and senseless trajectory without discounting them at all.

4.3.2 Never reenter the circle

The idea behind this second approach is similar to the one above. We want to prevent the agent from traversing the statespace in circles. One problem of forbidding the agent to close the circle is that we may force it to drive into a region that it does know nothing about. Here we allow to agent to reenter a state, but we prevent it from taking the action it chose the last time, i.e., we do not let it reenter the circle. The behavior of the agent unfortunately, does not change much. The difference is that it visits fewer positions as Figure 4.11 shows. This is because we now allow to agent to stand still once per position. More precisely, in the first approach, when the agent enters a position (x, y) and breaks to velocity 0, which means it enters the state $(x, y, 0, 0)$, we forbid the standstill action as it would lead to $(x, y, 0, 0)$ again. In this second approach, we only forbid an action if it was already taken in a particular state. So, if the agent enters $(x, y, 0, 0)$ and then chooses the standstill action next, we allow it to do so. Due to the nature of our noise (which would

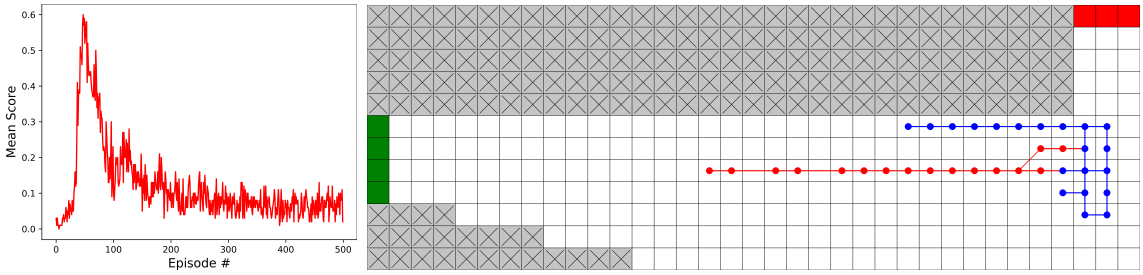


Figure 4.11: Learning curve and example path when preventing to reenter the circle

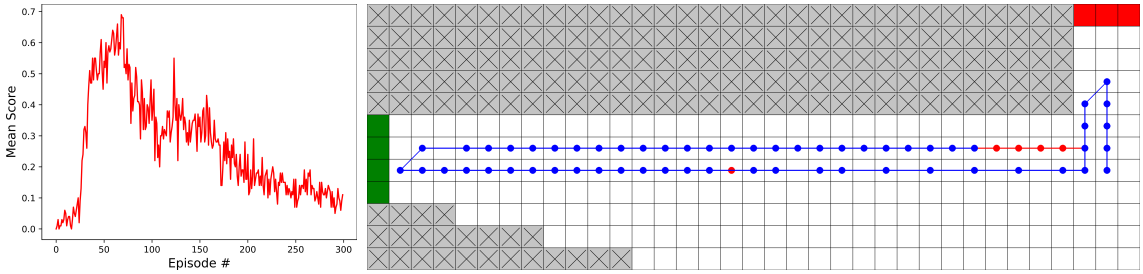


Figure 4.12: Learning curve and example path when preventing to revisit a position

also choose the standstill action), the agent would always end up in $(x, y, 0, 0)$ again after choosing the standstill action. Because of the deterministic action choice, it would choose the same action again, which we then prohibit as it has already been chosen in this state.

4.3.3 Never revisit a position

The final method is by far the most restrictive one. At first, we thought that it is, in fact, way too strict, but for the sake of a complete comparison, we will consider it nevertheless. The idea here is to completely forbid the agent to visit a position twice, regardless of the velocity. Unfortunately, this is not always possible to do as we would also forbid the agent to break down to velocity 0. Assume the agent is currently in the state $(x, y, 1, 0)$ and chooses the action $(-1, 0)$. If the action is successful, the agent will end up in the state $(x, y, 0, 0)$, i.e., at a position it has already visited. We would thus have to forbid the action $(-1, 0)$. The problem now is that we would sometimes prevent the agent from taking the optimal, or even the only possible path. The shortcut of the narrow alley map, displayed in Figure 4.5, for example requires the driver to drive to the next position with velocity 1, then break down to velocity 0 and start again into the direction of the next position. Therefore, we need to include this special case into this approach, i.e., we allow the double visit of a position if it is part of a deceleration to velocity 0. Figure 4.12 shows that this approach slows down the catastrophic fall, but the final result is the same. Because we have to allow the agent to decelerate completely, there are less than 100 distinct visited positions.

4.3.4 Conclusion

Unfortunately, none of our attempts was successful. We have seen rather similar results each time. Further investigations are necessary if and how an agent can be trained without a discount to better match the learning goal to the actual training. It would also be interesting to explore the findings of Gros [15], i.e., why the q-values decrease when coming closer to a goal.

4.4 Learning Success, Noise and Discount Correlation

Another interesting observation we made is that there seems to be a correlation between the learning success and the noise as well. We tried different values for the discount factor and the noise, and we found that when using 0.995 (0.005 more than usual) as the discount factor, we were able to train an agent with 50 % noise but failed to do so when using 0 % noise. Note that we used our standard reward structure here, but that does not matter much. Figure 4.13 shows the respective learning curves. At first, this does not make sense as an environment with 0 % should be extremely easy to learn. However, we can blame the same circumstances as before for this observation. Due to the high discount factor and the low noise, the q_*-values are extremely similar to one another. When using a higher noise, the q_*-values deviate enough from each other to make q-learning work. Unfortunately, this finding does not help us in the non-discounted case as, first, a higher noise than 50 % would take over the control much too frequently and, second, we actually tried to train an agent with 80 % noise and still observed the same catastrophic fall as before. So, the noise does influence the parity of the q_*-values, but not enough to solve the problems we faced here. Additionally, the idea behind the noise was to create an abstraction of a slippery road. Increasing it is rather artificial and should not be done to make the training work at all. We will use a rather high noise for our evaluations in order to highlight the differences between our approaches. However, all of those approaches would have yielded positive results with a lower noise as well.

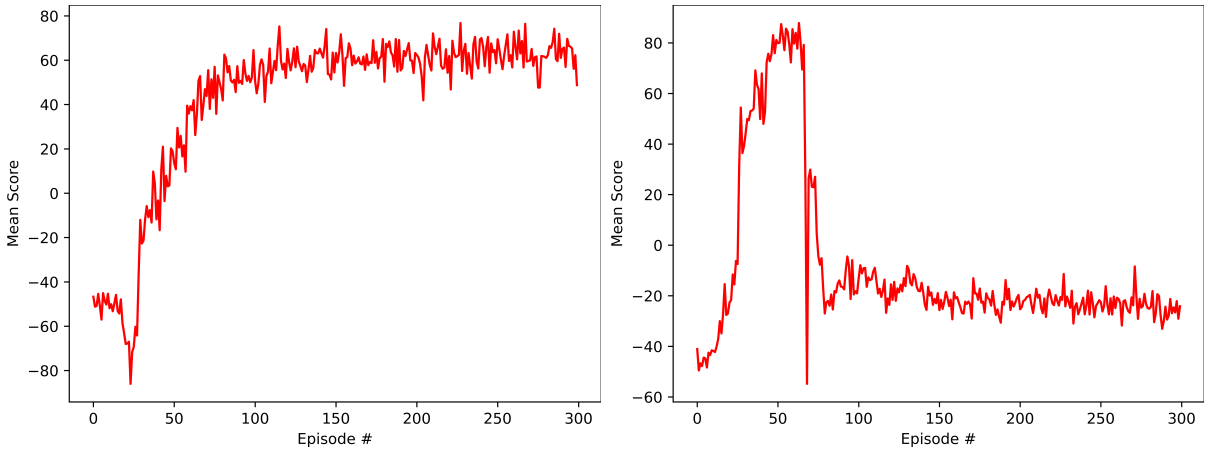


Figure 4.13: Learning curve with 50 % noise (left) and 0 % noise (right)

Chapter 5

Results

To evaluate the performance of the different approaches, we will use the general training method described in Section 3.3. To increase the consistency of our results, we executed the refinement process 10 times. Each time with a different random seed but using the same initial agent. The best agent of each iteration was analyzed again, but with increased accuracy. More precisely, a DSMC analysis was performed with 99 % confidence and an error of 0.01. Due to the issues discussed in Section 4.2 this usually leads to a slight decrease of the average goal probability, but this does not matter as it is equally fair for all of our agents. As the required number of simulations per position to achieve this accuracy is 26492, a single analysis can actually take several hours, which is fine as it was only required once after the training was finished. Finally, the resulting goal probabilities were averaged to obtain a clearer view of the performance of our different approaches.

We used three different agents for our evaluation. First, as a baseline, we have the default agent. Second, we used the spreaded agent. And lastly, we use an agent that uses the prioritized replay buffer as described in Section 3.5.2 in combination with a spreaded start. The α which is used here is 1, i.e., a full prioritization. We call this last agent the *prioritized agent*. The idea behind it is to further improve the performance of the hard regions as this agent not only starts more often from them but also prioritizes the respective trajectories. We chose those two DSMR agents as they showed the most promising results during our evaluations. We also compare the results of all agents to the "optimal agent" which was again computed using the Modest toolset [18]. This is not an actual agent as we do not compute a policy. Instead, we only calculate the maximum possible goal probability using model checking techniques. All reinforcement learning agents did worse than the optimum and sometimes considerably so. However, it is not the goal of this work to achieve perfect results using reinforcement learning, but to evaluate how it can be improved using DSMC. If we would like to achieve better results, we would need to experiment with different values for the learning parameters and especially the composition of the neural network more. As mentioned, a perfect agent is not our goal, and these experiments are thus out of the scope of this work.

In the following sections, the results of the three agents on the Barto-big, the ring, the narrow alley, the river, and the maze map will be analyzed. We will consider the DSMC heatmaps, the sampling heatmaps, the average goal probability, and the variance of the goal probabilities. To emphasize the difference between the agents with regard to the goal probability, we will also analyze the probabilities of what we consider to be the hard or interesting regions. This is useful, as the average probability will not differ

much for some maps. Not because the agents did all perform equally, but because there are large regions that are extremely simple to drive. For example, if a map had 90 straightforward and 10 hard positions and all agent achieved 100 % goal probability on the 90 simple positions, the average goal probability would be at least 90 % for all agents and differences in the interesting region would not show up as much in the overall average. As already discussed in Section 4.1 the interesting regions are marked yellow in Figure 4.4 and Figure 4.5. If not otherwise specified, we used a noise level of 50 % for the experiments in the following sections. Using a noise this high leaves more room for improvement for the DSMR agents as the default agent comes relatively close to the optimum when using low noise levels. Even with this high noise, all reinforcement learning agents perform really well in comparison to the optimum.

5.1 Barto-big Map

In the case of the Barto-big map, the DSMR agents actually perform slightly worse than the default agent in terms of overall goal probability as Table 5.1 shows. Figure 5.1 shows that this is due to the weaker performance in the goal region. Both the spreaded and the prioritized agent do improve the interesting region, though. Consequently, they also decrease the variance of the goal probabilities. As expected, those observations are amplified for the prioritized agent as it concentrates even more on the hard regions.

The sampling heatmaps in Figure 5.2 again present this one heavily sampled mandatory region similar to the one we have seen for the Barto-small map in Figure 4.3. Obviously, this is due to the nature of the map, which requires all trajectories starting anywhere before the mandatory region to go through it. Simultaneously, the goal region was sampled sparsely when using the DSMR agents. There are two reasons for that. First, those positions usually have a high analysis result and consequently are not started from often. Second, the goal region is actually a straight line, so once an agent has reached this straight line, it can accelerate further as the goal can be reached with any velocity. Many positions were thus skipped. This explains the worse performance of the DSMR agents in terms of overall average goal probability in comparison to the default agent. Additionally, this reveals a general problem with the DSMR approach: The goal probabilities do not necessarily reflect the performance of an agent. What we try to achieve with q-learning is to find the optimal policy for all states. For DSMR we assume a correlation between the goal probability of a position (x, y) and the quality of the agent's policy for all states between $(x, y, 0, 0)$ and a goal state. While there is a correlation, these properties are not equal. If the maximum possible goal probability of a position is 50 % and our agent actually achieves this 50 %, we will prioritize it even though there is nothing to be improved. This also happens the other way around, i.e., if the maximum possible goal probability is 100 % and our agent only achieves 90 % there is room for improvement our DSMR approaches will not necessarily realize. In theory, we would like to use the difference between the maximum possible goal probability and the one achieved by our agent as an indicator for its performance. However, this is rarely possible as we usually do not have access to an optimal agent. It was possible to compute it for these rather small racetrack maps, but we cannot assume that to generally be the case. In fact, if it would be, our whole approach would be pointless.

Agent	Average Goal Probability	Variance of Goal Probabilities	Goal Probability of Interesting Regions
Optimal	98.91 %	1.23×10^{-4}	97.08 %
Default	95.76 %	10.73×10^{-4}	90.25 %
Spreaded	95.70 %	10.36×10^{-4}	90.81 %
Prioritized	95.88 %	9.16×10^{-4}	91.09 %

Table 5.1: Results of the different agents on the Barto-big map

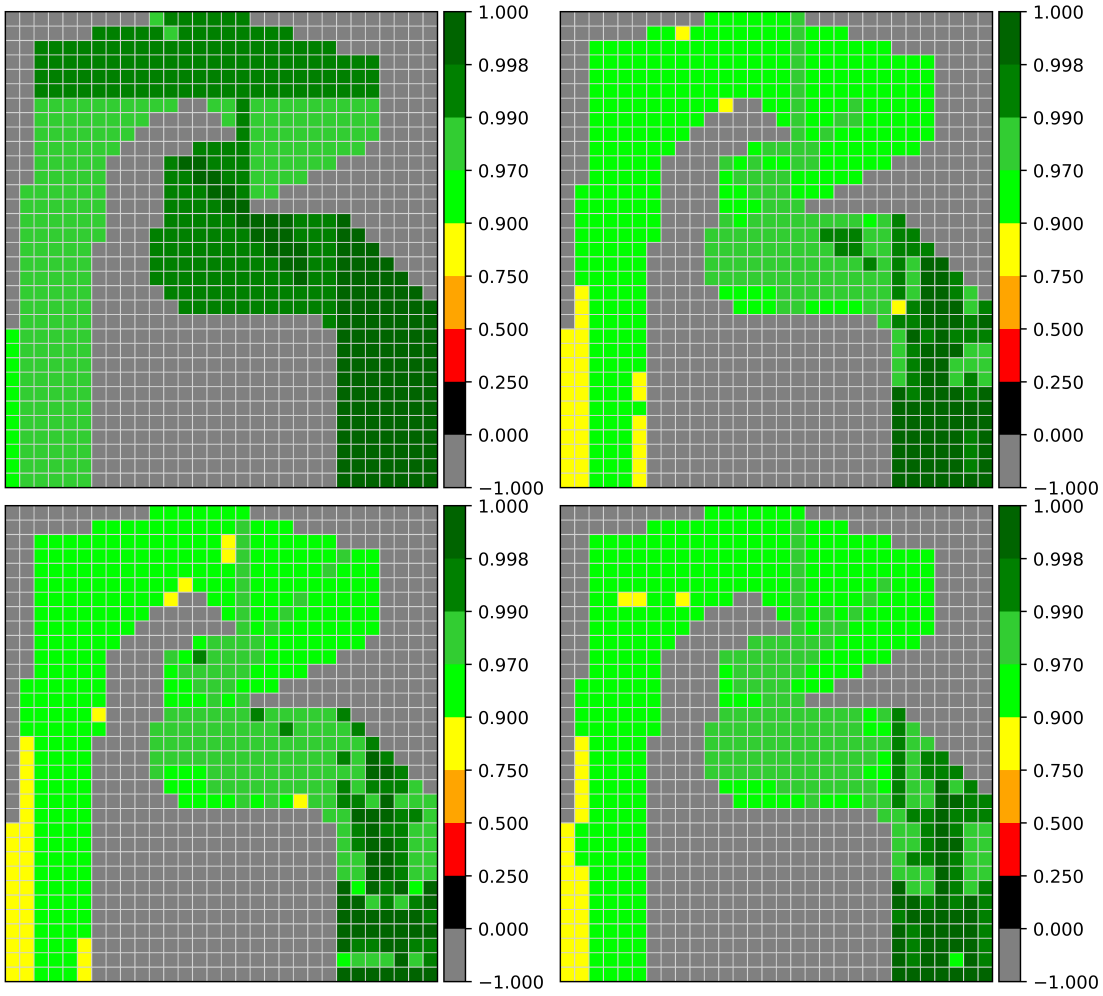


Figure 5.1: Results of the Barto-big map. Optimal agent (top left), Default agent (top right), Spreaded agent (bottom left), Prioritized Agent (bottom right)

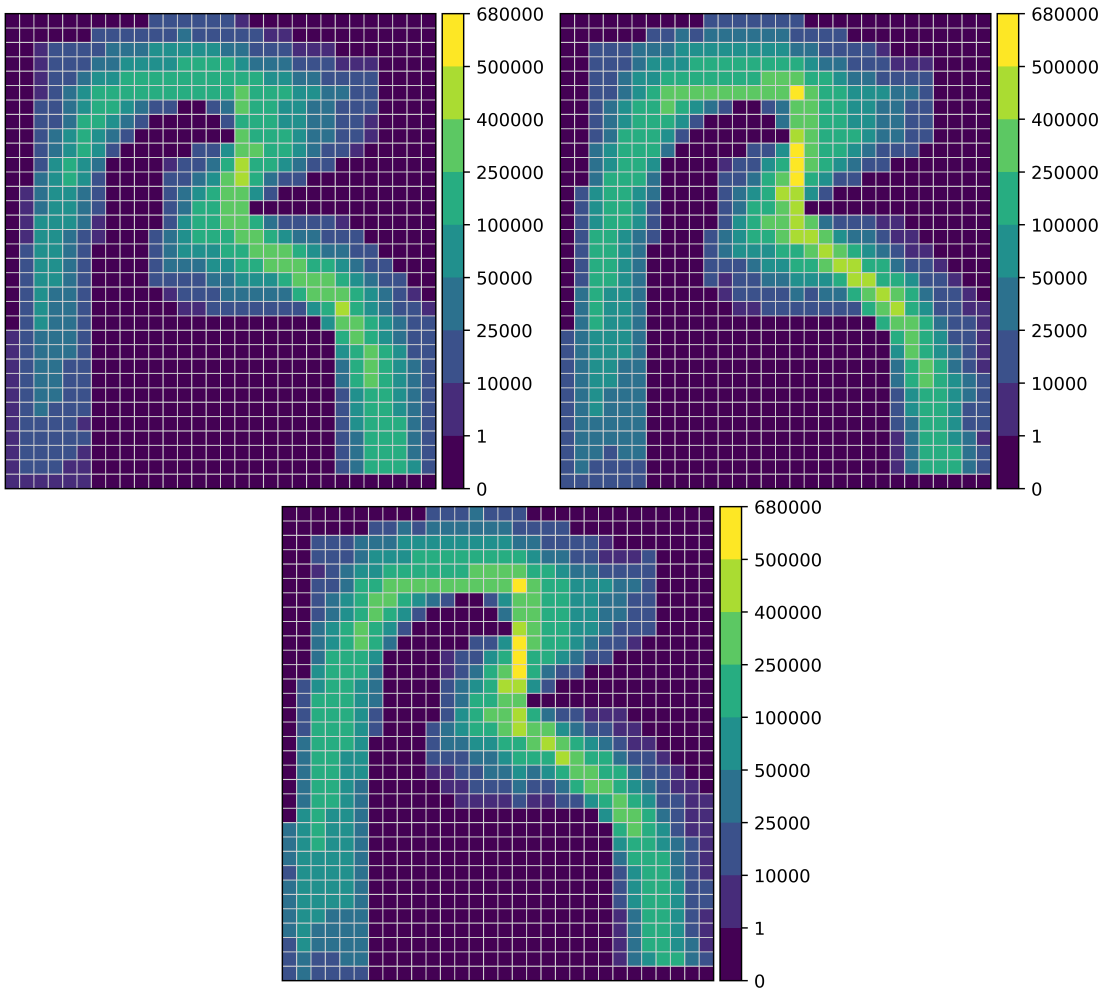


Figure 5.2: Sampling heatmaps of the Barto-big map. From left to right: Default agent, Spreaded agent, Prioritized Agent

5.2 Ring Map

The ring map already offers a further challenge for the agent. From the starting region, there are two different ways to the goal, which slightly diversifies the mandatory region. In theory, there is a mandatory region right at the end of the ring. However, this region is not problematic as it will have high analysis results on average and thus will not be sampled too often. Additionally, if an agent is coming from the upper half of the ring, it will not visit the same states as an agent coming from the lower half of the ring as both will have different velocities. When coming from above, the velocity will be going down and vice versa.

The observations regarding the quality of the agents resulting from the different learning approaches are similar to the observations of the previous section. The difference is that, here, the DSMR agents actually outperform the default agent in terms of overall goal probability as Table 5.2 shows. As before, the prioritized agent achieves the lowest variance even though it gets outperformed in the interesting regions by the spreaded agent.

The heatmaps in Figure 5.3 show two interesting findings. First, there are two diagonals with goal probability 1 in the heatmaps of both the default and the spreaded agent. This is simply due to the fact that no turns or breaks are necessary to reach the goal when starting from the diagonal. The agent only has to accelerate diagonally once, and it will reach the goal in any case as the noise can only prevent an acceleration and not change the direction itself. The prioritized agent is not as successful on the upper diagonal. This is because it prioritizes the other regions so much that its performance sometimes suffers in the easy regions. The second finding is that all agents perform slightly better in the lower half of the ring. It becomes clear why this is when looking at the sampling heatmaps in Figure 5.4. All agents seemed to prefer the lower half of the ring when starting from the initial region and thus had more experience driving it. The upper half was only chosen as the route to the goal when starting from the upper parts of the initial region. Overall, the sampling heatmaps indicate a much more even sampling distribution for both DSMR agents in comparison to the default agent, which mainly sampled the goal region.

Agent	Average Goal Probability	Variance of Goal Probabilities	Goal Probability of Interesting Regions
Optimal	98.19 %	3.50×10^{-4}	96.45 %
Default	93.21 %	26.07×10^{-4}	87.81 %
Spreaded	93.44 %	19.29×10^{-4}	89.28 %
Prioritized	93.26 %	18.00×10^{-4}	88.92 %

Table 5.2: Results of the different agents on the ring map

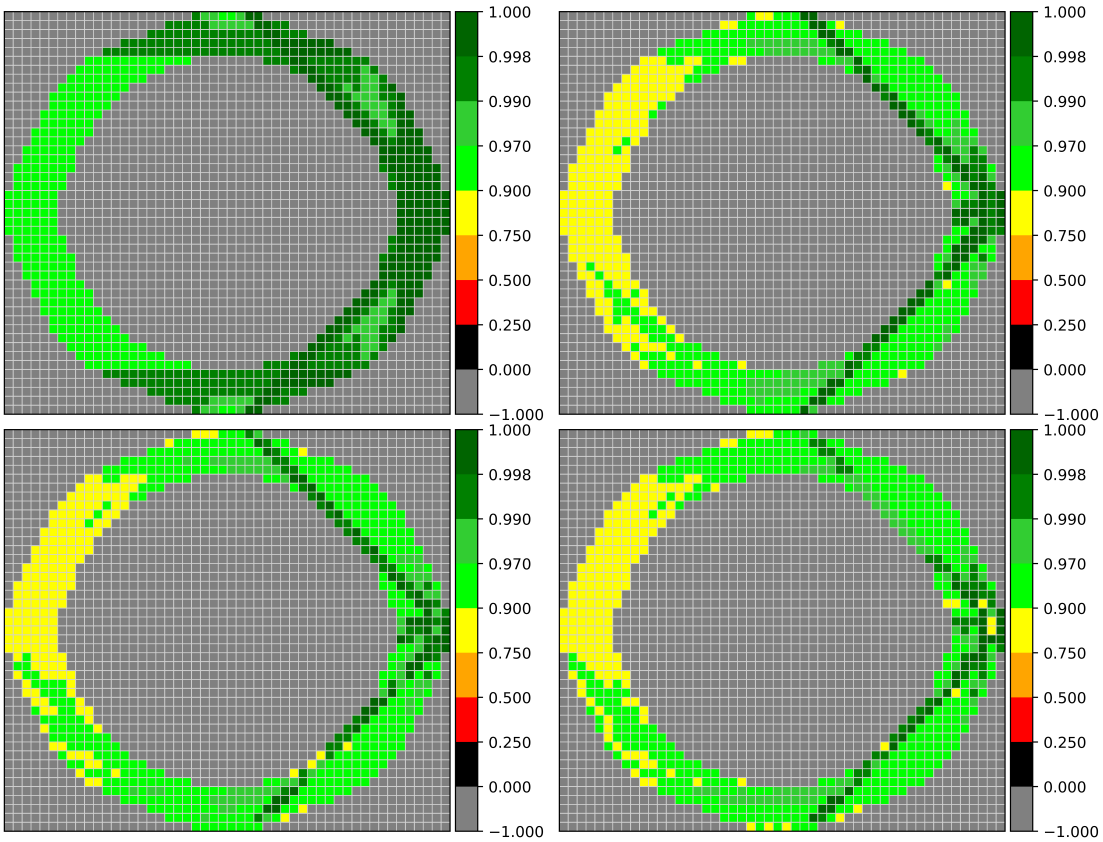


Figure 5.3: Results of the ring map. Optimal agent (top left), Default agent (top right), Spreaded agent (bottom left), Prioritized Agent (bottom right)

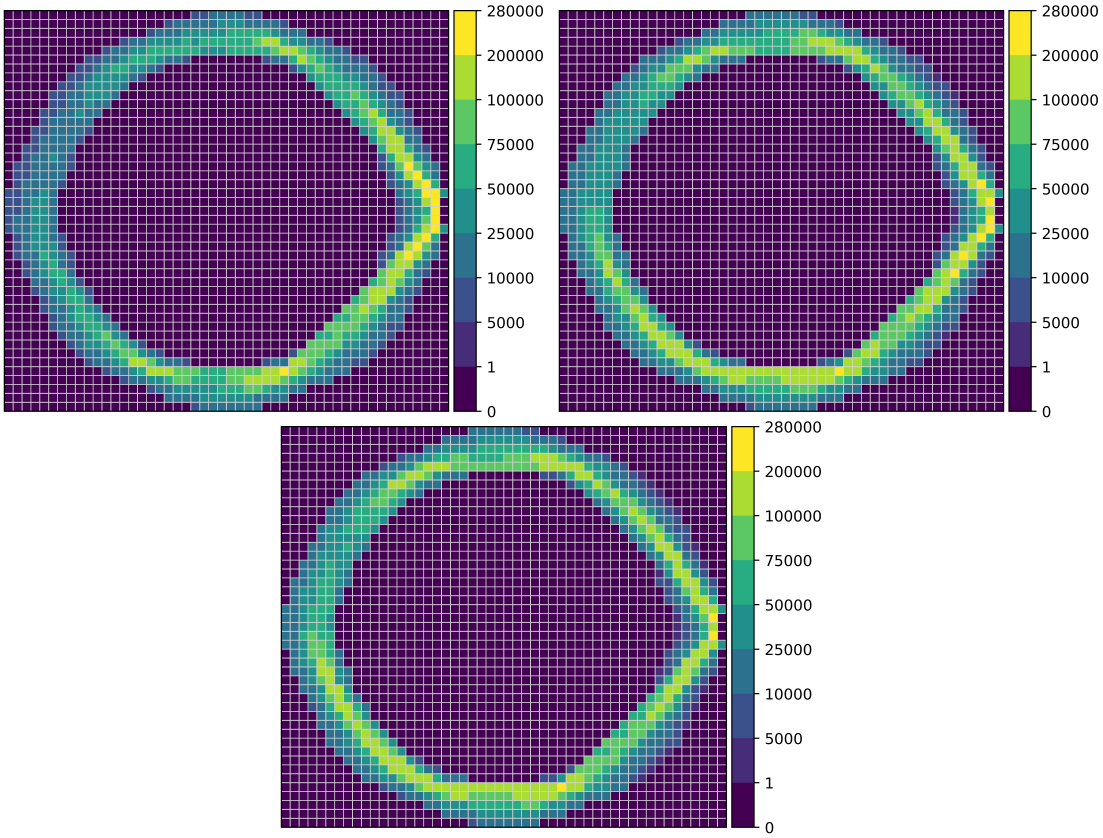


Figure 5.4: Sampling heatmaps of the ring map. From left to right: Default agent, Spreaded agent, Prioritized Agent

5.3 Narrow Alley Map

The experiments in this section were performed with 10 % noise. A low noise is required for this map because it would otherwise be almost impossible to reach the goal when starting from inside the shortcut, as we will discuss below. The narrow alley map consists of two different ways to reach the goal. It is possible to drive the long route going down, then right, and finally up again. Driving this way is extremely easy, as we can see on the leftmost heatmap of Figure 5.6, which displays the goal probabilities of an optimal agent. Obviously, another option is to take the shortcut. By design, the shortcut is hard to drive in a way that there are only two actions (and sometimes only one) that are directed to a goal for every state. Figure 5.5 lets us take a closer look at it. We considered the middle position of it as the start for the purpose of demonstration and the outer positions as goals as reaching a "real" goal is easy once you have left the shortcut. There are basically two ways that lead to a goal here: going right or going left. When heading for the goal to the right, one has to first choose the action $(-1, 1)$ (up-right), which leads to position 1 with velocity $(-1, 1)$. As it is now required to drive down and it is only possible to accelerate by one in each direction, the next action has to be $(1, -1)$ to break down to velocity $(0, 0)$. This whole procedure of driving to the next position and breaking again has to be repeated six times in total. Whenever a deceleration is required, there is a 10 % chance of a crash due to the noise. So, there is a 90 % chance to successfully change a position in this case. This is required five times (the goal can be reached with any velocity), so the maximum possible goal probability from the start position is $0.9^5 = 59.05\,\%$.

Because it is so dangerous to drive through the shortcut, it is actually better to drive left (heading to the starting positions of the map) for all positions left of the starting point from our example. This is why the optimal goal probabilities get symmetrically better left and right of the middle point. Although the maximum goal probability is obviously very low, it is actually rather easy to achieve using q-learning. As the difference between the actions is extreme, due to the proximity to the walls, a q-learning agent will find the correct actions rather quickly. The problem when trying to learn this map is that the buffer will contain almost no samples from shortcut trajectories. On the one hand, this is due to the shortcut consisting of only a couple of positions. On the other hand, many trajectories starting from anywhere inside it will consist of only a single action. This leads to the agent regularly losing all of its progress during the learning and subsequently never learning how to drive the shortcut.

Figure 5.6 also shows the averaged heatmaps of our agents. As we can see on the second left heatmap, the default agent did not find the correct policy for the shortcut as discussed above. Remember that there are only two valid ways out of the shortcut: going right or left as described above. In contrast to the default agent, both the spreaded agent and the prioritized agent managed to find one of those two ways consistently. Keep in mind that the heatmaps are averaged over ten different training runs. In most of them, both DSMR agents found the optimal policy for the shortcut. The spreaded agent sometimes went left when going right was the better option, which reduced the goal probability. In addition, the agent has to reach the goal after leaving the shortcut, which also failed in rare cases. However, in general, both agents performed really well in the interesting region as we can see in Table 5.3. The final result of both agents was also close to the optimum in terms of average goal probability and variance. The prioritized agent has a slightly lower goal probability compared to the spreaded agent, but the prioritized agent has a lower

variance. Which makes sense as it focuses heavily on the hard regions, which do improve even further, but it neglects the other regions a bit. The heatmap of the prioritized agent shows two yellow positions in the easy region, which is due to one of ten training runs failing to reach the goal from there. This can happen in rare cases when the agent chooses the standstill action when starting from there, and it drags down the average.

Figure 5.7 further explains these results. We can see that when using the default agent, the shortcut rarely gets sampled. The heaviest sampled region is the curve before the home straight. This is because lots of trajectories go through this region. After that, it is possible to accelerate towards the goal every step, which leads to high velocities and, thus, many positions being skipped. When using the default agent, the shortcut gets sampled similarly often to the curve region, whereas using the prioritized agent actually led to the shortcut being extremely heavily sampled. This is due to the goal probability always being low in these states as their maximum is so low. This is a prime example of the phenomenon described at the end of Section 5.1. However, it is not much of an issue here because all the remaining regions are so easy to drive.

Agent	Average Goal Probability	Variance of Goal Probabilities	Goal Probability of Interesting Regions
Optimal	99.43 %	16.08×10^{-4}	79.78 %
Default	97.40 %	153.33×10^{-4}	32.91 %
Spreaded	98.55 %	22.46×10^{-4}	74.14 %
Prioritized	98.42 %	21.06×10^{-4}	75.43 %

Table 5.3: Results of the different agents on the narrow alley map

X	10	X	8	X	6	X	1	X	3	X	5	X
G	X	9	X	7	X	S	X	2	X	4	X	G

Figure 5.5: Closeup of the shortcut

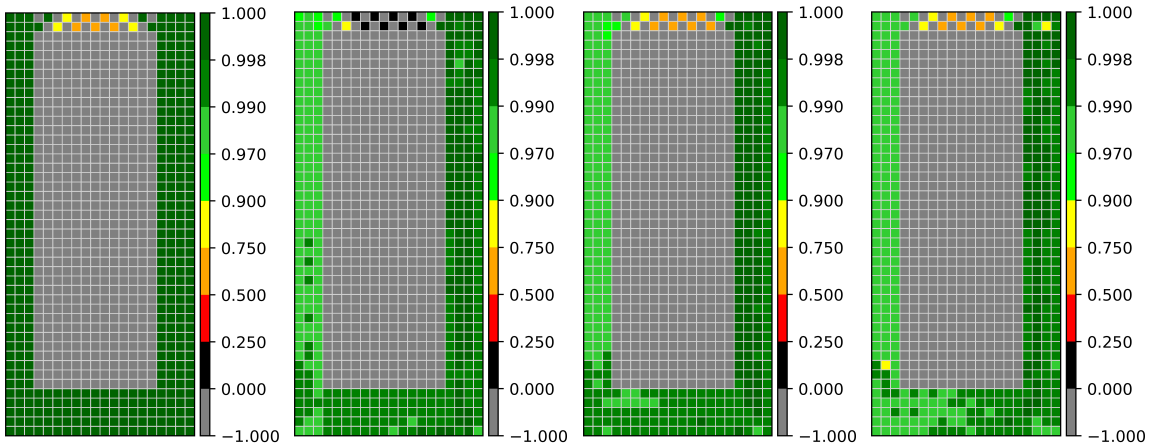


Figure 5.6: Results of the narrow alley map. From left to right: Optimal agent, Default agent, Spreaded agent, Prioritized agent

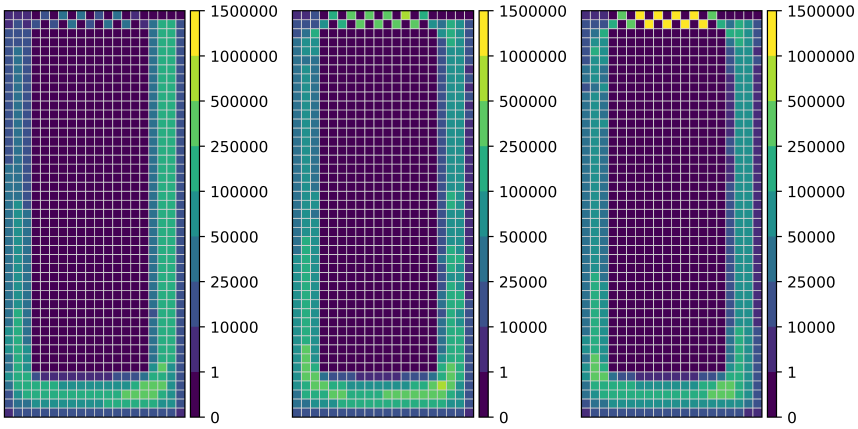


Figure 5.7: Sampling heatmaps of the narrow alley map. From left to right: Default agent, Spreaded agent, Prioritized agent

5.4 River Map

The river map is significantly harder to drive optimally than the shortcut map. The average goal probability of the whole map and the goal probability of the interesting regions are a bit misleading here. The overall average is so high because there are many positions close to a goal. The goal probability of the interesting regions is higher than what we have seen for the shortcut map. However, there is not one clear cut best action for each interesting position as there was for the shortcut. It is thus easier to reach the goal at all, but harder to reach the maximum possible goal probability, as there are multiple ways to do so.

As we can see in Figure 5.8 and Table 5.4 both DSMR agents significantly improved the starting region and the blind alley in comparison to the default agent. Simultaneously, both of them struggled a bit in the easy regions right before the goals. The observations of Section 5.3 and Section 5.1 that the prioritized agent slightly degraded the overall goal probability but increased the goal probability of the interesting regions and lowered the

variance stand here as well.

The sampling heatmaps in Figure 5.9 show that we largely achieved our design objective for this map. The idea behind the river map was to eliminate the single region where every trajectory has to go through. The heatmap of the default agent displays a well-distributed sampling behavior over the whole map. The region before the second goal (when counting from the left) is sampled a little less, as are some branches of the river. This is because it is best to take the other branch in these cases. Most of the trajectories going left will head straight towards the first goal and ignore the second, as they should. This behavior is amplified when using the DSMR agents. Both of them favor the hard regions and achieve high goal probabilities in the regions before the second and third goals. Therefore, the sampling of these regions is reduced. Instead, they fill the buffer with trajectories from the starting region and the blind alley. The right branch from the start is also heavily sampled, which indicates that the agent chose to go primarily right when starting from the starting region.

Agent	Average Goal Probability	Variance of Goal Probabilities	Goal Probability of Interesting Regions
Optimal	96.73 %	21.46×10^{-4}	89.20 %
Default	81.99 %	446.00×10^{-4}	46.10 %
Spreaded	84.69 %	269.42×10^{-4}	58.12 %
Prioritized	84.38 %	235.28×10^{-4}	60.35 %

Table 5.4: Results of the different agents on the river map

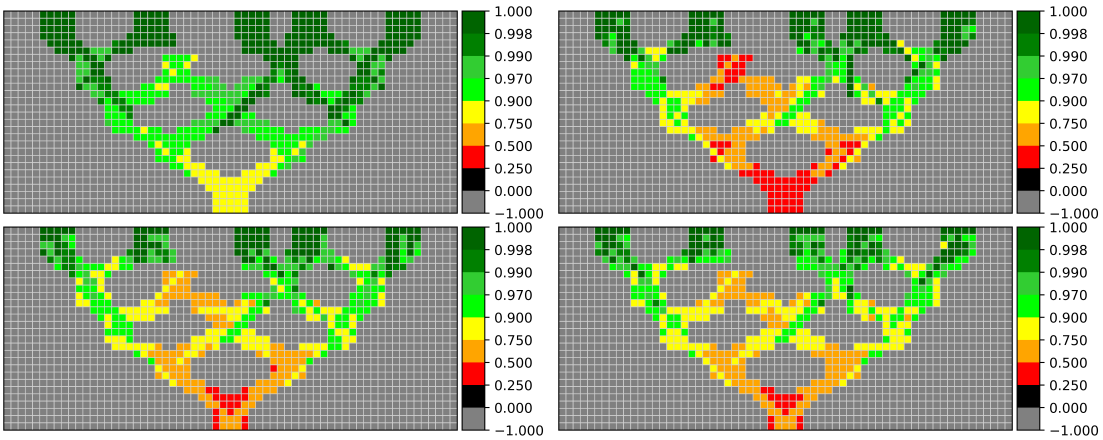


Figure 5.8: Results of the river map. Optimal agent (top left), Default agent (top right), Spreaded agent (bottom left), Prioritized agent (bottom right)

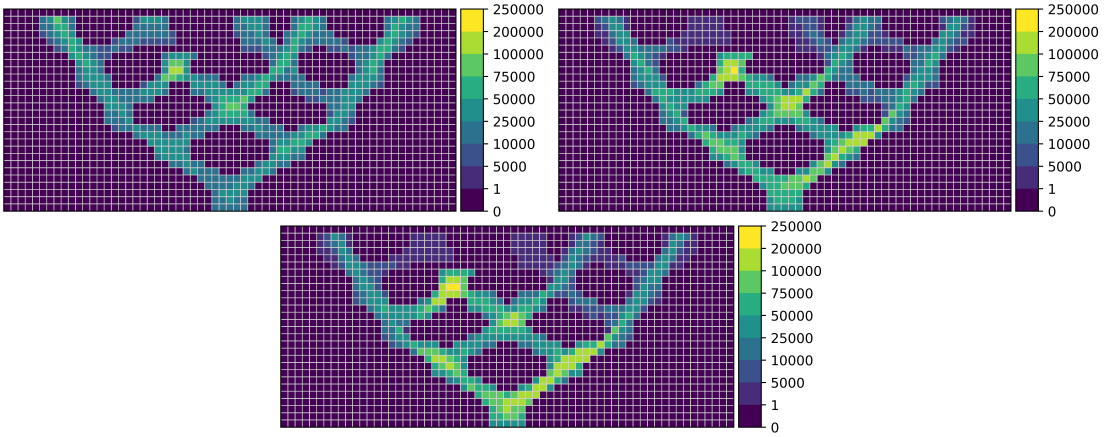


Figure 5.9: Sampling heatmaps of the deadend map. From left to right: Default agent, Spreaded agent, Prioritized agent

5.5 Maze Map

The maze map was designed to have many completely different ways to reach a goal. Due to its narrow frame, we had to use a noise level of just 10 %, which is fine as the map is hard enough already. As we can see in Table 5.5 both DSMR agents significantly improved both the average goal probability and the goal probability of the interesting regions. Interestingly, the spreaded agent outperformed the prioritized agent in every category on this map. The sampling heatmaps in Figure 5.11 reveal that the hard regions are sampled heavily when using the prioritized agent. For the previous maps, this was fine as the regions following the hard regions were quite easy to drive, which is not quite the case here. Probably, the excessive focus on these regions leads to the agent not knowing how to drive the rest of the map, which ultimately hurts the hard regions as well. It still worked quite nicely as the performance gain in comparison to the default agent is significant. It just got outperformed a bit by the spreaded approach.

The sampling heatmap of the spreaded agent indicates a nice sample distribution over almost the whole map. The maze map has a single goal again and a matching mandatory goal region, which was heavily sampled by the default agent. Due to the combination of a disproportional start and the map being difficult, the hard regions are sampled considerably more often when using the DSMR agents as many trajectories that started far away from the goal resulted in a crash and thus never reached the mandatory goal region. As argued already, the prioritized agent probably focused on the hard regions too much, whereas the default agent excessively focused on the goal region. In this case, the spreaded agent found a nice middle ground, which resulted in the performance we observed in Table 5.5.

Agent	Average Goal Probability	Variance of Goal Probabilities	Goal Probability of Interesting Regions
Optimal	97.37 %	2.05×10^{-4}	95.95 %
Default	87.13 %	218.75×10^{-4}	59.78 %
Spreaded	92.95 %	24.86×10^{-4}	86.46 %
Prioritized	92.12 %	30.77×10^{-4}	84.77 %

Table 5.5: Results of the different agents on the maze map

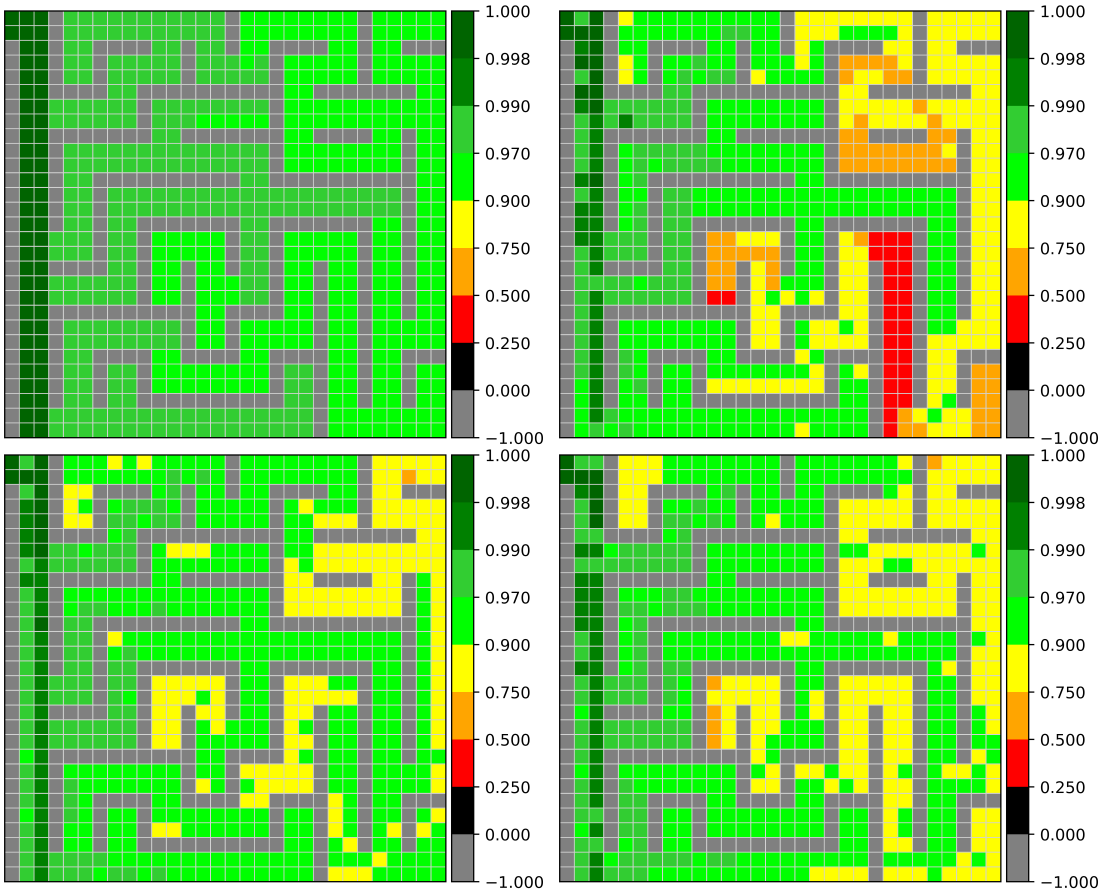


Figure 5.10: Results of the maze map. From top to bottom: Default agent, spreaded agent, Prioritized agent

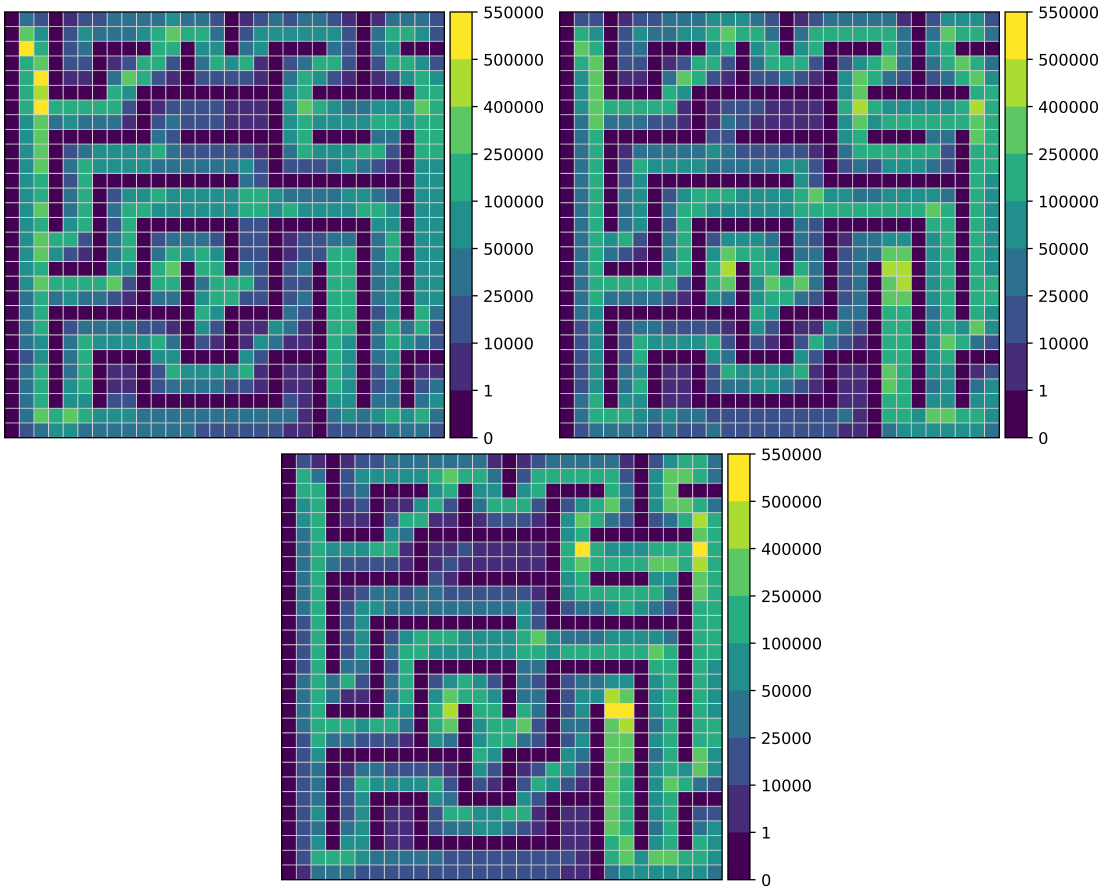


Figure 5.11: Sampling heatmaps of the maze map. From left to right: Default agent, Spreaded agent, Prioritized agent

Chapter 6

Conclusion

In this thesis, we implemented Deep Statistical Model Checking [16] for the racetrack domain and developed several methods of incorporating its results into a reinforcement learning approach, which we called Deep Statistical Model Refinement. These methods included a spreaded agent, which uses the model checking results to build a probability distribution over the positions of the map to control how often a training run is started from a specific position, and two approaches which use the model checking results to manipulate the probabilities in a prioritized replay buffer.

We have discussed the specific settings we chose to train the agents, including the concept of refinement batches. Additionally, we have discussed why the racetrack maps offered by Barto et al. [3] are unsuitable for showcasing our approaches. As an alternative, we have built several new racetrack maps, each of which offers a unique new challenge for a reinforcement learning agent to solve. We defined our learning goal to be the maximization of the average goal-reaching probability and discussed why the chosen reward structure and the discount factor specifically do not reflect this goal perfectly. However, we found that the one to one representation of this learning goal yields terrible learning results leaving us with no other choice.

Finally, we have evaluated our approaches on a number of different maps, including the Barto-big, the ring, and our custom maps. The results showed that our approaches were able to lower the variance of the goal probabilities and, additionally, increase the goal probabilities in the hard to drive regions on all maps. DSMR was also able to increase the average goal probability on most maps, for some significantly so.

We believe that DSMR can be used to automatically steer the exploration towards otherwise under-represented parts of the statespace in cases where an overall well-performing agent is needed. It is also useful to determine the specific timepoint during training that yields the most desirable agent.

6.1 Limitations and Future Work

Despite the success we have had in this work, there is a major limitation our approaches are facing. The DSMC analysis required us to construct a subset of the statespace, which could then be analyzed. In the case of the racetrack domain, this was rather easy as the subset of all states with velocity $(0, 0)$ was a logical and useful choice. It is both meaningful and manageable in terms of its size. Unfortunately, this is not possible or at least not as easy for all domains. Without this subset, our whole approach falls apart as

there are no analysis results we could incorporate into the learning process. However, this is more of a limitation of the DSMC approach in general. Further research in the field of model checking is necessary to eliminate this limitation.

Another interesting research topic would be the success of DSMR in another domain. Obviously, this needs to be a domain that offers a meaningful subset of its statespace as discussed above. However, given such a domain, it would be interesting to investigate how DSMR could be used.

Finally, further research could go into the non-discounted case of deep reinforcement learning to solve the issues discussed in Section 4.3.

Bibliography

[1] Bernhard K Aichernig, Priska Bauerstätter, Elisabeth Jöbstl, Severin Kann, Robert Korošec, Willibald Krenn, Cristinel Mateis, Rupert Schlick, and Richard Schumi. Learning and statistical model checking of system response times. *Software Quality Journal*, 27(2):757–795, 2019.

[2] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. Safe reinforcement learning via shielding. *32nd AAAI Conference on Artificial Intelligence, AAAI 2018*, pages 2669–2678, 2018.

[3] Andrew G Barto, Steven J Bradtke, and Satinder P Singh. Learning to act using real-time dynamic programming. *Artificial intelligence*, 72(1-2):81–138, 1995.

[4] Richard Bellman. Dynamic programming and stochastic control processes. *Information and control*, 1(3):228–239, 1958.

[5] Blai Bonet and Hector Geffner. Gpt: a tool for planning with uncertainty and partial information. In *Proc. IJCAI-01 Workshop on Planning with Uncertainty and Partial Information*, pages 82–87, 2001.

[6] Vivek S Borkar. Q-learning for risk-sensitive control. *Mathematics of operations research*, 27(2):294–311, 2002.

[7] Jeffery A Clouse. On integrating apprentice learning and reinforcement learning. 1997.

[8] Javier Garcia and Fernando Fernández. Safe exploration of state and action spaces in reinforcement learning. *Journal of Artificial Intelligence Research*, 45:515–564, 2012.

[9] Javier García and Fernando Fernández. A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research*, 16:1437–1480, 2015.

[10] Martin Gardner. Mathematical games. *Scientific American*, 229(1):104–109, 1973.

[11] Chris Gaskett. Reinforcement learning under circumstances beyond its control. 2003.

[12] Clement Gehring and Doina Precup. Smart exploration in reinforcement learning using absolute temporal difference errors. In *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*, pages 1037–1044, 2013.

[13] Peter Geibel and Fritz Wysotzki. Risk-Sensitive Reinforcement Learning Applied to Control under Constraints. *Journal of Artificial Intelligence Research*, 24:81–108, sep 2011.

[14] Ian J Goodfellow, Mehdi Mirza, Da Xiao, Aaron Courville, and Yoshua Bengio. An empirical investigation of catastrophic forgetting in gradient-based neural networks. *arXiv preprint arXiv:1312.6211*, 2013.

[15] Timo P. Gros. Tracking the Race: Analyzing Racetrack Agents Trained with Imitation Learning and Deep Reinforcement Learning. Master's thesis, Saarland University, 2020. To appear.

[16] T.P. Gros, H. Hermanns, J. Hoffmann, M. Klauck, and M. Steinmetz. Deep statistical model checking. 2020.

[17] T.P. Gros, D. Höller, J. Hoffmann, and V. Wolf. Tracking the Race Between Deep Reinforcement Learning and Imitation Learning. 2020.

[18] Arnd Hartmanns and Holger Hermanns. The modest toolset: An integrated environment for quantitative modelling and verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 593–598. Springer, 2014.

[19] Matthias Heger. Consideration of risk in reinforcement learning. In *Machine Learning Proceedings 1994*, pages 105–111. Elsevier, 1994.

[20] Thomas Hérault, Richard Lassaigne, Frédéric Magniette, and Sylvain Peyronnet. Approximate probabilistic model checking. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 73–84. Springer, 2004.

[21] Hisashi Kashima. Risk-sensitive learning via minimization of empirical conditional value-at-risk. *IEICE TRANSACTIONS on Information and Systems*, 90(12):2043–2052, 2007.

[22] Michael McCloskey and Neal J Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*, volume 24, pages 109–165. Elsevier, 1989.

[23] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[24] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[25] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[26] Tetsuro Morimura, Masashi Sugiyama, Hisashi Kashima, Hirotaka Hachiya, and Toshiyuki Tanaka. Parametric return density estimation for reinforcement learning. *arXiv preprint arXiv:1203.3497*, 2012.

[27] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[28] Pablo Quintía Vidal, Roberto Iglesias Rodríguez, Miguel Ángel Rodríguez González, and Carlos Vázquez Regueiro. Learning on real robots from experience and simple user feedback. 2013.

[29] Roger Ratcliff. Connectionist models of recognition memory: constraints imposed by learning and forgetting functions. *Psychological review*, 97(2):285, 1990.

[30] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[31] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[32] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[33] Andrea L Thomaz and Cynthia Breazeal. Teachable robots: Understanding human teaching behavior to build more effective robot learners. *Artificial Intelligence*, 172(6-7):716–737, 2008.

[34] Lu Wen, Jingliang Duan, Shengbo Eben Li, Shaobing Xu, and Huei Peng. Safe Reinforcement Learning for Autonomous Vehicles through Parallel Constrained Policy Optimization. (2012), 2020.

[35] Håkan LS Younes and Reid G Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In *International Conference on Computer Aided Verification*, pages 223–235. Springer, 2002.