Saarland University

Master's Thesis

Tracking the Race: Analyzing Racetrack Agents Trained with Imitation Learning and Deep Reinforcement Learning

Submitted by:

Timo P. Gros

Submitted on:

May 4, 2021

Reviewers:

Univ.-Prof. Dr. Verena Wolf

Univ.-Prof. Dr. Jörg Hoffmann

Erklärung

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

Teilweise ist diese Arbeit, insbesondere Teile der Kapitel $1-3$, 5, und 10, bereits publiziert worden [10, 11].

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.

Some parts of this thesis, in particular parts of the Chapters $1-3$, 5, and 10, have already been published [10, 11].

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, May 4, 2021

Timo P. Gros

Abstract

Learning-based approaches for solving large sequential decision-making problems have become popular in recent years. The resulting agents perform differently and their characteristics depend on those of the underlying learning approach.

Here, we consider a benchmark planning problem from the reinforcement learning domain, the Racetrack, to investigate the properties of agents derived from different deep (reinforcement) learning approaches.

This thesis examines three different aspects.

We first compare the performance of deep supervised learning, particularly imitation learning, versus reinforcement learning for the Racetrack model. We find that imitation learning yields agents that follow more risky paths. In contrast, deep reinforcement learning decisions are more foresighted, i.e., avoid states where fatal decisions are more likely. Our evaluations show that for this sequential decision-making problem, deep reinforcement learning performs best in many aspects, even though for imitation learning, optimal decisions are considered.

Secondly, we focus on the more promising approach, namely deep Q-Learning. We demonstrate the exploration-exploitation dilemma and especially the problem that occurs when training with sparse reward. We propose a simple yet effective method to solve a task when rewards are observed rarely. Additionally, we investigate different hyperparameters of deep Q-learning and analyze their influence on the resulting policies. We find some interesting insights about the influence of the designed reward function when the goal is given in words, as well as about the discount factor.

Thirdly, we take a step towards general policy training by solving the Racetrack (i) purely on image-based representation without manually created features, and (ii) for arbitrary Racetrack goal lines. We do so by making use of convolutional neural networks. The policy obtained has a performance close to such policies supported by the manually crafted features and limited to a fixed goal line.

Acknowledgments

First of all, I would like to thank Prof. Dr. Jörg Hoffmann and Prof. Dr. Verena Wolf for supervising and reviewing this thesis.

Further thanks go to all those who worked on or with the Racetrack and therefore pointed out things that could be improved, mainly Joschka Groß, Hendrik Meerkamp, and Michaela Klauck.

The most important shall be emphasized in the end. All my thanks belong to my family. My sister Julia, who always tried to make me a better person. My mother Kirsten, who showed me the wonders of this world. My father Markus, whom I have always looked up to. My fiancée Christina, who always had my back and supported me wherever she could. Without you all, none of this would have been possible.

Contents

1 Introduction

In recent years, deep learning (DL) and especially deep reinforcement learning (DRL) have been applied with great successes to the task of learning near-optimal policies for sequential decision-making problems. DRL has been used for various applications such as Atari games [18, 19], Go and Chess [28, 29, 30], or Rubic's cube [2]. It relies on a feedback loop between self-play and improving the current strategy by reinforcing decisions that lead to good performance.

Passive imitation learning (PIL) is another well-known approach to solve sequential decision-making problems, where a policy is learned based on training data labeled by an expert [26]. An extension of this approach is active imitation learning (AIL), where after an initial phase of passive learning, additional data is iteratively generated by exploring the state space based on the current strategy and subsequent expert labeling [14, 25]. AIL has successfully been applied to common reinforcement learning benchmarks such as cart-pole or bicycle-balancing [14].

Sequential decision-making problems are typically described by Markov decision processes (MDPs). During the simulation of an MDP, the set of those states that will be visited in the future depend on current decisions. In PIL, the agent, which represents a policy, is trained by iterating over the given expert data set, whose distribution does not generally resemble this dependence. AIL extends the data with sequentially generated experiences. Hence, the data is more biased towards sequentially taken decisions. In contrast, DRL does not rely on expert data but alternates between the exploitation of former experiences and exploration of the state space. It is apriori not obvious which method achieves the best result for a particular sequential decision-making problem.

Thus, this thesis first aims at an in-depth study of empirical learning agent behavior for a range of different learning frameworks. Specifically, we are interested in differences due to the sequential nature of action decisions inherent in reinforcement learning but not in imitation learning. To study and understand algorithm behavior in detail, we conduct our investigation in a simple benchmark problem, namely the Racetrack.

The Racetrack is originally a pen and paper game, adopted as a benchmark in AI sequential decision-making to evaluate of MDP solution algorithms [3, 5, 24, 31]. A map with obstacles is given, and a policy for reaching a goal region from an initial position has to be found. Decisions for two-dimensional accelerations are taken sequentially, which requires foresighted planning. Ignoring traffic, changing weather conditions, fuel consumption, and technical details, Racetrack can be considered a simplified model of autonomous driving control [8]. The Racetrack is ideally suited for comparing different learning approaches because not only the performance of different agents but also their "driving characteristics" can be analyzed. Moreover, for small maps, expert data describing optimal policies can be obtained.

We train different agents for Racetrack using DRL, PIL, and AIL and study their characteristics. We first apply PIL and train agents represented by linear functions and artificial neural networks. As expert labeling, we apply the A^* algorithm to find optimal actions for states in the Racetrack. We suggest different variants of data generation to obtain more appropriate sample distributions. For AIL, we use the DAGGER approach [25] to train agents represented by neural networks. We use the same network architecture when we apply deep reinforcement learning. More specifically, we train deep Q-networks [19] to solve the Racetrack benchmark. We compare the resulting agents considering three aspects: the success rate, the quality of the resulting action sequences, and the relative number of optimal and fatal decisions.

Amongst other things, we find that, *even though it is based on optimal training data, imitation learning leads to unsafe policies, much riskier than those found by RL.* Upon closer inspection, it turns out that this apparent contradiction actually has an intuitive explanation in terms of the nature of the application and the different learning methods: *to minimize time to goal, optimal decisions navigate very closely to dangerous states.* This works well when making optimal decisions throughout – but is brittle to (and thus fatal in the presence of) even small divergences as are to be expected from a learned policy. We believe that this characteristic might carry over many other applications beyond Racetrack.

Using DRL as the algorithm of choice, we secondly aim at gaining further insights into so trained agents. We demonstrate the exploration-exploitation dilemma, in particular, the problem when training with sparse reward. We propose a method that helps us solving the task, i.e., finding a route from the start to the goal, where the standard RL algorithm is not capable of doing so. We demonstrate that this method can help solving tasks even from states that have not been visited while learning due to the generalization of neural networks.

Additionally, we are interested in how the chosen hyperparameters influence the resulting policy. We perform extensive investigations to model the influence of the reward function's design and of the discount factor. While we do not claim that all of our findings carry over to generally every domain, we can at least make some general conclusions about how to correctly set/find the best hyperparameters for a given purpose.

Lastly, we use the gained insights to make a step towards a more general framework for sequential decision-making: solving the Racetrack purely based on image representation without feature generation and additionally with arbitrary start and goal positions. Among others, we are able to create a Racetrack policy with a performance not equal but close to policies that were trained for a fixed goal with manually created features.

The outline of this thesis is the following: We first introduce the Racetrack domain ands our implementation and the options associated with them (Chapter 2). We then introduce the used algorithms of deep learning, particularly the DAGGER framework

and deep Q-learning (Chapter 3), before we describe our application to the Racetrack domain (Chapter 4). In Chapter 5, we present our comparison between the different deep learning algorithms. Afterward, we present our proposed method to solve the exploration-exploitation dilemma in Chapter 6. It then follows the investigations for the design of the reward function (Chapter 7) and of the discount factor (Chapter 8). Our attempt for general policy learning is given in Chapter 9. We finally draw a conclusion and present future work in Chapter 10.

2 Racetrack

The Racetrack has been used as a benchmark in the context of planning [5, 24] and reinforcement learning [3, 31]. It can be played on different maps. The examples used throughout this thesis are displayed in Figure 2.1.

2.1 The Racetrack – a Pen and Paper Game

At the beginning of the game, a car is placed randomly at one of the discrete positions on the start line (in purple) with zero velocity. In every step, it can speed up, hold the velocity or slow down in x and/or y dimension. Then, the car moves in a straight line with the new velocity from the old position to a new one, where we discretize the maps into cells. The game is lost when it crashes, which is the case when either (1) the new position itself is a wall position or outside the map, or (2) the straight line between the old and new position intersects with a wall, i.e., the car drives through a wall on its way to the new position. The game is won when the car either stops at or drives through the goal line (in green).

2.2 The Racetrack – a Markov Decision Process

Given a Racetrack map, the game can be modeled as a Markov decision process.

States S . The current state is uniquely defined by the position $p = (x, y)$ and the velocity $v = (v_x, v_y)$.

Actions A. Actions represent the acceleration a. As the car can be accelerated with values $\{-1, 0, 1\}$ in the x and in the y dimension, there are exactly $3^2 = 9$ different actions available in every state.

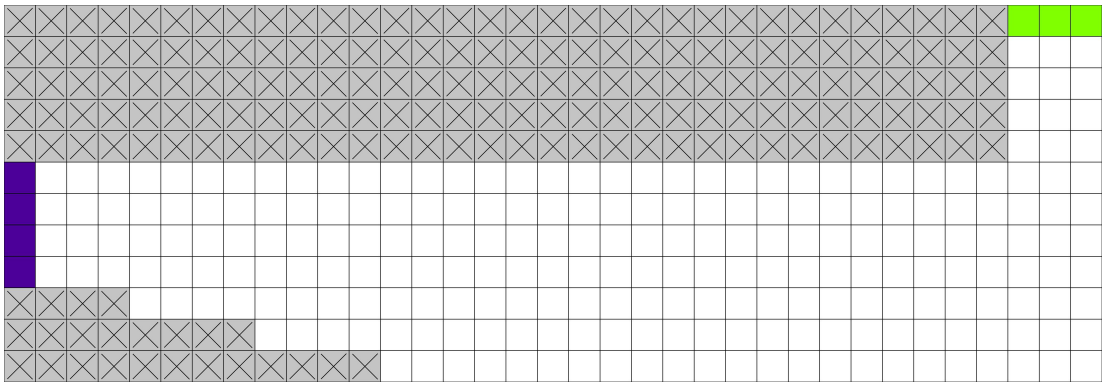
Transitions. We assume a wet road, so with a chance of 0.1, the acceleration cannot be applied, i.e., $a = (0, 0)$. Otherwise, with a probability of 0.9, the acceleration is given by the chosen action. The new velocity $v' = (v_x', v_y')$ is given by the sum of the

acceleration $a = (a_x, a_y)$ and the current velocity. The new position $p' = (x', y')$ is given by adding v' to the current position, i.e.,

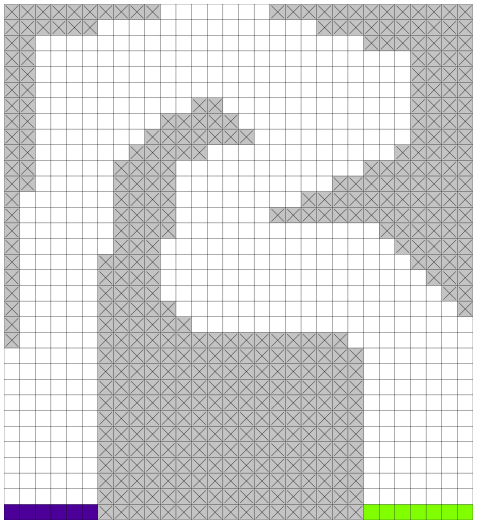
$$\begin{aligned} {v_x}' &= v_x + a_x, & x' &= x + {v_x}', \\ {v_y}' &= v_y + a_y, & y' &= y + {v_y}'. \end{aligned}$$

To define several properties, we use a discretization of transitions of the MDP similar to the one of Bonet & Geffner [5]. The corresponding driving trajectory T is a sequence of visited positions

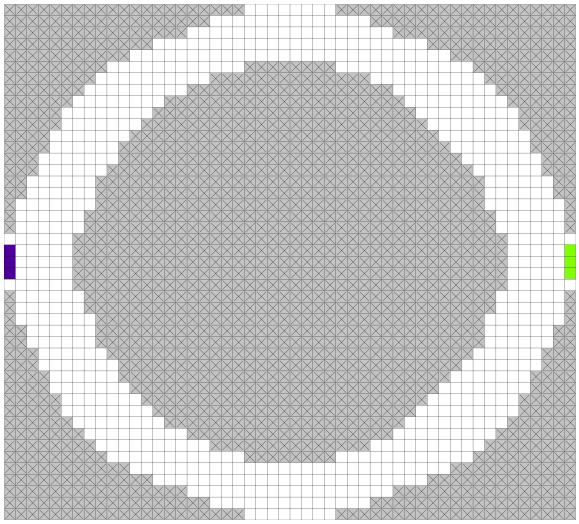
$$T = \langle (x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1}), (x_n, y_n) \rangle ,$$



(a) Barto-small



(b) Barto-big



(c) Ring

Figure 2.1: Examples of Racetrack maps: goal line is green, start line is purple.

such that

$$T = \begin{cases} \langle (x,y) \rangle & \text{if } {v_x}' = 0 \text{ and } {v_y}' = 0 \quad (1) \\ \langle (x,y), (x+\sigma_x, y), (x+2\cdot\sigma_x, y) \ldots, (x',y') \rangle & \text{if } {v_x}' \neq 0 \text{ and } {v_y}' = 0 \quad (2) \\ \langle (x,y), (x, y+\sigma_y), (x, y+2\cdot\sigma_y) \ldots, (x',y') \rangle & \text{if } {v_x}' = 0 \text{ and } {v_y}' \neq 0 \quad (3) \\ \langle (x,y), (x+\sigma_x, \lfloor y+m_y \rceil), (x+2\cdot\sigma_x, \lfloor y+2\cdot m_y \rceil) \ldots, (x',y') \rangle & \text{if } {v_x}' \neq 0 \text{ and } {v_y}' \neq 0 \text{ and } |{v_x}'| \geq |{v_y}'| \quad (4) \\ \langle (x,y), (\lfloor x+m_x \rceil, y+\sigma_y), (\lfloor x+2\cdot m_x \rceil, y+2\cdot\sigma_y) \ldots, (x',y') \rangle & \text{if } {v_x}' \neq 0 \text{ and } {v_y}' \neq 0 \text{ and } |{v_x}'| < |{v_y}'| \quad (5), \end{cases}$$

where $\sigma_x = sgn({v_x}')$, $\sigma_y = sgn({v_y}')$ and $m_x = \frac{{v_x}'}{|{v_y}'|}$, $m_y = \frac{{v_y}'}{|{v_x}'|}$ [9].

If either the vertical or horizontal speed is 0, exactly all grid coordinates between $p = (x, y)$ and $p' = (x', y')$ are contained in the trajectory. Otherwise, we consider n equidistant points on the linear interpolation between the two positions and each round to the map's closest position. While in the original discretization $n = |{v_x}'|$ [5], in this model it is given by $\max(|v_x|, |v_y|)$. The former is problematic when having a velocity that moves less into the x- than into the y- direction, as then only a few points will be contained in the trajectory and counterintuitive results may be produced.

We consider a transition to be *valid* if and only if it does not crash, i.e., no position $p \in T$ is either a wall or outside of the map. A transition is said to *reach the goal* if and only if one of the positions $p \in T$ is on the goal line. Additionally, a transition cannot be invalid *and* reach the goal. If a transition fulfills the conditions for both, only the one that was fulfilled first holds. In other words: if a car has already reached the goal, it cannot crash anymore and vice versa.

A transition leads to a state with a new position p' if it is valid and does not reach the goal. If it is invalid, it leads to a bottom state $\bot$ that has no further transitions. Otherwise, i.e., it is reaching the goal, it leads to the goal state $\top$.

Rewards/Costs. As we consider both planning and learning approaches, we define the following two cost functions: For planning, we consider a uniform cost function, such that an optimal planner will find the shortest path to reach the goal line. For reinforcement learning, we consider a reward function that is positive if the step reaches the goal, negative if the step is invalid, and 0 otherwise. More concretely, we chose

$$R\left(s \xrightarrow{(a_x,a_y)} s'\right) = \begin{cases} 100 & \text{if } s' = \top \\ -50 & \text{if } s' = \bot \\ 0 & \text{otherwise} \end{cases}$$

for the reward of a transition from any state s with action (a_x, a_y) to state s'.

2.3 The Racetrack Tool – a Python Simulation

Next to being a pen and paper game, Racetrack is a well-known benchmark problem: In the context of planning, it has first been considered by Bonet & Geffner [5]. It has further been used as a benchmark for reinforcement learning [31]. Based on this benchmark domain, we have created a tool to

- simulate games with arbitrary decision-making agents,
- use planning algorithms to find an optimal policy,
- create datasets for imitation learning,
- train linear functions or specified neural networks for a given dataset,
- use DAGGER [25] to train neural networks,
- train agents with reinforcement learning,
- visualize the map and traveled trajectories, and
- evaluate specified agents.

The tool is available at GitHub. Due to ongoing research, it is kept private.

Simulation. For a specified map, we provide several variants of the simulation, which can be activated or deactivated within our tool:

1. Normal start (NS) versus random start (RS): Usually, a game starts on the start line, but our tool also enables a start at a random (valid) position on the map.
2. Zero velocity (ZV) versus random velocity (RV): Usually, a game starts with velocity $(0, 0)$, but our tool also enables starts with a random velocity (v_x, v_y) within 0 and a given upper bound.
3. Deterministic (D) versus noise (N): Usually, the chosen acceleration is applied with the rules given above. When the noise option is set, the chosen acceleration is only applied with a chance of 0.9 and an acceleration of $(0, 0)$ otherwise.

Data Set Creation. Although a state in our model is uniquely given through its position and velocity, we provide several other features that can be used as state encoding.

- $d_1, \ldots, d_8$: linear distance to a wall in all directions. These eight distances are spread equally around the car and are given analogously to the acceleration, i.e., positive, neutral, or negative (by not considering the combination negative-negative), 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., $|dg_x| + |dg_y|$.

Together with the position and the velocity, this gives us a total of 15 features per state.

Data set creation can create labeled datasets, i.e., tuples of the given state features and an optimal action to take in that state. We find the optimal solution by applying a Racetrack-tailored version of the A^* algorithm to find an optimal plan from the current state. In the base case, we randomly sample states and velocities and add the optimal actions (there might be more than one such action), i.e., the optimal acceleration, to the state. However, our tool provides the following options:

- Complete trajectory (T): If this option is set, all states on the way to the goal are added to the dataset instead of only the current state.

- Exhaustive (E): If the exhaustive option is set, the tool provides *all* optimal solutions for the specified state and adds them to the dataset.

- Unique (U): If this option is activated, only states having a unique optimal acceleration are added to the dataset.

Option E excludes option T due to runtime constraints as the number of optimal trajectories increases exponentially with the trajectory's length.

Training. Our tool can be used to train the following functions on a given dataset

- linear functions using logistic regression,

- linear functions using linear discriminant analysis,

- neural networks with a specified structure, and

- neural networks with a specified structure using DAGGER.

Further, neural networks can be trained using deep Q-learning [19]. This deep reinforcement learning approach uses the same features used for supervised learning/dataset creation.

For all training procedures, there are several hyperparameters to be set.

Visualization. For a specified map file and given agents, our tool provides the option to illustrate the taken trajectories graphically. This can be used to examine the behavior of the agents. Several examples can be found from Chapter 5 to Chapter 8.

Evaluator. We provide the option to evaluate specified agents. For the given hyper-parameters, the evaluator tests the agents on (the same) random simulations. It then provides three statistics.

1. Win-loose statistics: Provides statistics on how many of the runs reached the goal, how many crashed, and how many timed out for the specified number of maximal steps.

2. Reward-step statistics: Provide statistics about both objectives: what average reward was received and how many average steps were taken to reach the goal.

3. Optimal-fatal statistics: Provides statistic on how many of the selected actions were optimal, non-optimal but save, and fatal (crashed or lead to a state where a crash was unavoidable).

3 Deep Learning

3.1 Imitation Learning

We consider both passive and active imitation learning. For passive imitation learning, we use (1) logistic regression (LR) and linear discriminant analysis (LDA) to train linear functions, and (2) stochastic gradient descent to train neural networks. To represent the class of active imitation learning algorithms, we consider DAGGER [25].

DAGGER. *Dataset Aggregation* (DAGGER) is a meta-algorithm for active imitation learning. The main idea of DAGGER is to mitigate the problem related to the disruption of the independently identical distributed assumption in passive imitation learning for sequential decision-making. The trained agent is then used to iteratively sample more labeled data to train a new neural network. The algorithm starts with a pre-trained neural network using the following steps:

(i) It follows the current action policy to explore the state space.

(ii) For every visited state, it uses the expert to find the action that shall be imitated.

(iii) It adds the pairs of state and action to the training set, and

(iv) trains a new policy on the enlarged data set.

Step (i) can be varied via a hyper-parameter $\beta \in [0, 1]$ that sets the ratio of following the current policy or the expert for exploration. With $\beta = 0$, it follows the current policy only. Step (iii) can be done with any thinkable expert and step (iv) can be done with any training procedure.

3.2 Reinforcement Learning

Deep Q-learning. Given an MDP, we train an agent representing a policy such that the expected cumulative reward of the MDP's episodes is maximized. As (potentially) a race can last forever, the task is a continuing one [31] and the accumulated future reward, the so-called *return*, of step t is therefore given by $G_t = \sum_{i=t}^{\infty} \gamma^i \cdot R_{i+1}$, where

γ is a discount factor with $\gamma \in [0, 1]$ and we assume that R_{i+1} is the reward obtained during the transition from the state S_i to state S_{i+1} for $i \in \{0, 1, \ldots\}$ [31].

For a fixed state s, an action a, and a policy π, the *action-value* $q_\pi(s, a)$ gives the expected return that is achieved by taking action a in state s and following the policy π afterward, i.e.,

$$q_\pi(s, a) = \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \;\middle|\; S_t = s, A_t = a \right].$$

We write $q_*(s, a)$ for the *optimal action-value* function that maximizes the expected return. The idea of *value-based* reinforcement learning methods is to find an estimate $Q(s, a)$ of the optimal action-value function. Artificial neural networks can express complex non-linear relationships and are able to generalize. Hence, they have become popular for function approximation. We estimate the Q-value function using a neural network with weights θ, a so-called deep Q-network (DQN) [18]. We denote the DQN by $Q(s, a; \theta)$ and optimize it w.r.t. the target

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

Thus, in iteration i, the corresponding loss function is

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

where θ^- refers to the parameters from some previous iteration, with the so-called *fixed target* [19] $y(S_t, A_t; \theta^-)$. We optimize the loss function by stochastic gradient descent using an approximation of $\nabla L(\theta_i)$ [19].

Furthermore, we apply the idea of *experience replay* [19]. Instead of directly learning from observations, we store all experience tuples in a data set and sample uniformly from that set.

We generate our experience tuples by exploring the state space epsilon-greedily, that is, with a chance of $1 - \epsilon$ during the Monte Carlo simulation, we follow the policy that is implied by the current network weights and otherwise uniformly choose a random action [19].

In the following, we will use the terms reinforcement learning (RL) and deep reinforcement learning (DRL) interchangeably.

4 Training Racetrack Agents

In this Chapter, we describe the training process of agents based on active and passive imitation learning as well as deep reinforcement learning.

Objective Function. The learning methods that we consider rely on two different objective functions: DRL uses the reward function and imitation learning uses data sets that were optimized w.r.t. the number of steps until the goal is reached. As DRL makes use of discounting (see Chapter 3.2), the accumulated reward is higher if fewer steps are taken. Thus, both objective functions serve the same purpose, even though they are not completely equivalent. Note that a direct mapping from the costs used in the planning procedure to the reward structure is not possible. We tested different reward structures for DRL and found that a negative reward for every single step combined with a positive reward for the goal and a negative reward for invalid states led to inferior convergence properties of the training procedure. No well-performing alternative was found to the reward structure defined in Chapter 2.2 up to scaling.

4.1 Imitation Learning

We want to train agents for all simulation scenarios, including those where the car starts at an arbitrary position on the map and visits future positions on the map with different velocities. Usually, all learning methods are based on the assumption that the data is i.i.d.. Data that is generated via simulation of the Racetrack greatly disrupts this assumption. Thus, we propose different approaches for data generation to encounter this problem.

Data Sets. We combine the different options to create data sets as given in Chapter 2.3 to a total of 6 different combinations as displayed in Table 4.1.

The first data set contains uniformly sampled (valid) positions and velocities and combines them with a single optimal action. This explores the whole state space equally. The data sets (2) and (3) differ in their starting points. For (2), the car is positioned on the start line; for (3), it might be anywhere on the map. Both sets contain the optimal acceleration (i) for this starting state, and (ii) for every state visited on the trajectory from the starting state to the goal. To do both, uniformly sample through the state space and take into account the trajectories, (4) starts with a random position and a random velocity but still collects the whole trace. The data set (5) includes all optimal

Table 4.1: Racetrack configurations used to create our data sets.

No	ID	Description	RS	RV	T	E	U
(1)	RS-RV	Uniform sample from all positions on map and all possible velocities.	✓	✓	✗	✗	✗
(2)	NS-ZV-T	Uniform sample from all positions on the start line; combined with zero velocity. All states that were visited on the optimal trajectory to the goal line are included in the data set.	✗	✗	✓	✗	✗
(3)	RS-ZV-T	Uniform sample from all positions on the map; combined with zero velocity. All states that were visited on the optimal trajectory to the goal line are included in the data set.	✓	✗	✓	✗	✗
(4)	RS-RV-T	Uniform sample from all positions on the map and all possible velocities. All states visited on the optimal trajectory to the goal line are included in the data set.	✓	✓	✓	✗	✗
(5)	RS-RV-E	Uniform sample from all positions on the map and all possible velocities. All optimal actions for that state are included in the data set.	✓	✓	✗	✓	✗
(6)	RS-RV-U	Uniform sample from all positions on the map and all possible velocities. Only such states that have a unique optimal action are included in the data set.	✓	✓	✗	✗	✓

solutions instead of just one. Apart from that, (5) is similar to set (1). (6) only includes entries that have a unique optimal next action.

For each learning method, we train several instances; at least one on each data set. Each data set consists of approximately 10^5 entries.

4.1.1 Passive Imitation Learning

Linear Predictors. While deep learning is more powerful than linear learning, linear classifiers have the advantage that their decisions are more transparent.

We use the package `scikit-learn` [23] to apply both *Linear Discriminant Analysis* (LDA) and *Logistic Regression* (LR). Together with the six combinations of data sets, this gives 12 different agents.

Neural Networks. We use the `PyTorch` [15] package to train neural networks by repeatedly iterating over the labeled training data. The MSE is used as a loss function. As neural networks tend to overfit when the number of iterations is too high, we store the neural network after every iteration. We experimentally found that a maximum iteration number of 20 is more than sufficient. As we again use every 6 data sets, this gives us a total of 120 agents.

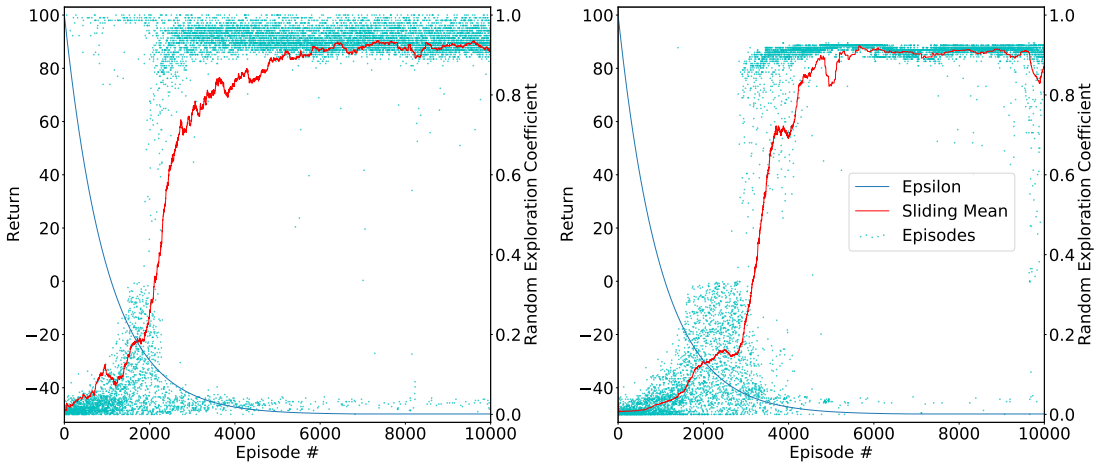
As explained in Chapter 2, a state is represented by 15 features, which gives us the input size of the network. There are nine possible actions. As we do not process the game via an image, but through predefined features, we only use fully connected layers. More sophisticated network structures are only needed for complex inputs such as images. For a fair comparison, we use the same network size for all methods. We use two hidden layers of size 64, resulting in a network structure of $15 \times 64 \times 64 \times 9$.

4.1.2 Active Imitation Learning

DAGGER. In the case of active imitation learning, we applied DAGGER using $\beta = 0$ for all iterations, i.e., after the pre-training, we follow the trained agent without listening to the expert for exploration. To have a fair comparison, DAGGER has the same number of samples as PIL, i.e., 10^5. Still, the pre-training is important for sampling within an iteration of the algorithm, but the main idea is to generate further entries that are more important for the agent's training. Thus, we pre-trained the agent on each of our data sets and then additionally allowed DAGGER to add 10^5 samples. We split these 10^5 samples into 20 iterations. The neural network was trained by performing eight iterations over the data set. Our experiments with the networks described in Chapter 4.1.1 show that this is the best trade-off between over- and under-fitting. Again, we store the trained agents after every iteration, giving us a total of 120 agents for the DAGGER method.

4.2 Deep Reinforcement Learning

Deep Q-learning. In contrast to imitation learning, reinforcement learning is not based on data sets and thus is not applied to any of the data sets given in Table 4.1. Training is done by self-play only; the Racetrack agent chooses its actions using a neural network and applies them to the environment. After every move, (i) the next state (given by the features), (ii) the reward as defined in Chapter 2 that was achieved for the move, and (iii) the information whether the episode is terminated are returned to the agent. The agent then uses the loss (MSE) between the accumulated reward and the expected return to correct the weights of the network.



(a) Training in RS-N mode.

(b) Training in NS-D mode.

Figure 4.1: Training progress of the RL agent.

All imitation learning agents were trained with 10^5 (new) samples using the same network structure. Therefore, we here restrict the agents to 10^5 entries in the replay buffer, i.e., the maximal number of entries an agent can learn from at the same time. The neural network is not pre-trained but initialized randomly.

To have a trade-off between exploration and exploitation, our agent acts ϵ-greedy, i.e., with a probability of ϵ it chooses a random acceleration instead of the best acceleration to explore the state space. As our DRL agent is initialized randomly – and thus starts without any experience about what good actions are – at the beginning of the training phase, the focus lies on exploration. We therefore begin our training with $\epsilon = 1$, i.e., always choosing a random action. After every episode i, we decrease ϵ exponentially with a factor $\lambda = 0.999$ to shift the focus from exploration to exploitation during training, until a threshold of $\epsilon_{\text{end}} = 0.0001$ is reached, i.e., $\epsilon_{i+1} = \max(\epsilon_i \cdot \lambda, \epsilon_{end})$.

Table 4.2: Racetrack configurations used to train Racetrack agents with deep reinforcement learning.

No	ID	Description	RS	D
1	NS-D	Starting on a random position on the start line using the deterministic simulation.	✗	✓
2	NS-N	Starting on a random position on the start line using the noisy simulation.	✗	✗
3	RS-D	Starting on a random position on the map using the deterministic simulation.	✓	✓
4	RS-N	Starting on a random position on the map using the noisy simulation	✓	✗

To train the agents not just to reach the goal but to minimize the number of steps, we make use of a discount factor $\gamma = 0.99$.

Besides the given options of either starting on the start line (NS) or anywhere on the map (RS), DRL can benefit from learning while the noisy (N) version of Racetrack is simulated instead of the deterministic (D) one. This gives us four different training modes listed in Table 4.2.

To determine the best network weights, the average return over the last 100 episodes during the training process is used. We save the network weights that achieve the best result. Additionally, we save the network weights after running all training episodes, independent of the received average return. This results in total in 8 different DRL agents. The training progress is displayed in Figure 4.1. For both depicted training variants, one can observe that the sliding mean of the received return increases over the number of trained episodes. The mean increases from a value around -50, i.e., a range where nearly all training episodes crashed, to a value greater than 80, i.e., nearly all episodes succeeded. In Figure 4.1a, the single score dots with a value close to 100 show that the goal is reached already within the first training episodes. Therefore, an increase in the sliding mean can be observed after already 2000 episodes. For the NS variant, Figure 4.1b depicts that the goal is reached frequently only after about 3000 episodes, which is why the training curve increases later.

5 Tracking the Race Between Deep Reinforcement Learning and Imitation Learning

For evaluation, we consider all possible combinations given by the simulation parameters as described in Chapter 2.3. In total, it results in 6 different simulation settings on which we compare the trained agents. These settings are given in Table 5.1. The combinations with NS and RV are not considered, as they include more starting states where a crash is inevitable than solvable ones.

In the sequel, for each learning method, we present the best-performing parameter combination of all those that we tested. We investigate three aspects of the resulting agents' behavior: the success rate, the quality of the resulting action sequences, and the relative number of optimal and fatal decisions.

Table 5.1: Configurations on which we evaluate the agents.

No	ID	Description	RS	RV	D
1	NS-ZV-D	Starting on a random position on the start line with zero velocity using the deterministic simulation	✗	✗	✓
2	NS-ZV-N	Starting on a random position on the start line with zero velocity using the noisy simulation	✗	✗	✗
3	RS-ZV-D	Starting on a random position on the map with zero velocity using the deterministic simulation	✓	✗	✓
4	RS-ZV-N	Starting on a random position on the map with zero velocity using the noisy simulation	✓	✗	✗
5	RS-RV-D	Starting on a random position on the map with a random velocity using the deterministic simulation	✓	✓	✓
6	RS-RV-N	Starting on a random position on the map with a random velocity using the noisy simulation	✓	✓	✗

5.1 Success Rate

We first compare how often the agents win a game, i.e. reach the goal, or lose, i.e., crash into a wall. We limit the game to 1000 steps; if an agent then neither succeeded nor failed, we count the episode as timed out. We compare the agents on 10^4 simulation runs. For every single run of the simulations, all agents start in the same initial state. The results can be found in Figure 5.1.

We omit the plot for NS-ZV-D, as all of the agents have a 100% winning rate. The linear agents perform worst. Especially with random starting points and velocities, they fail to reach the goal. DAGGER outperforms the passive imitation learning agents, as it has been designed to cope with sequential decision-making.

Throughout all settings, the DRL agents perform best. They clearly outperform DAGGER, reaching the goal more than 1.5 times more often in the NS-ZV-N setting.

5.2 Quality of Action Sequences

We illustrate results for the quality of the chosen action sequences in Figure 5.2.

The left plot gives the cumulative reward reached by the agents averaged over all runs (also over those that are not successful). DRL clearly achieves the highest cumulative reward. We remark that the optimal policies computed via A* give higher cumulative rewards as the goal is reached faster. However, imitation learning achieves lower results on average as it fails more often.

We show results for the number of steps in the right plot in Figure 5.2. When a car crashes, we are not interested in the number of steps taken. Therefore – in this specific analysis – we only report on successful runs. They show that – while reinforcement learning has the most wins and is the best agent considering the reward objective – it is consuming the highest number of steps when reaching the goal. It even takes more steps than linear classifiers.

5.3 Quality of Single Action Choices

Next, we examine whether the agents choose the optimal acceleration, i.e., the acceleration that does not crash and leads to the goal with as few steps as possible for different positions and velocities. We distinguish between (1) optimal actions, (2) fatal actions that unavoidably lead to a crash, and (3) secure actions that are neither of the former.

Figure 5.1: Success rate results for all classes of examined agents.

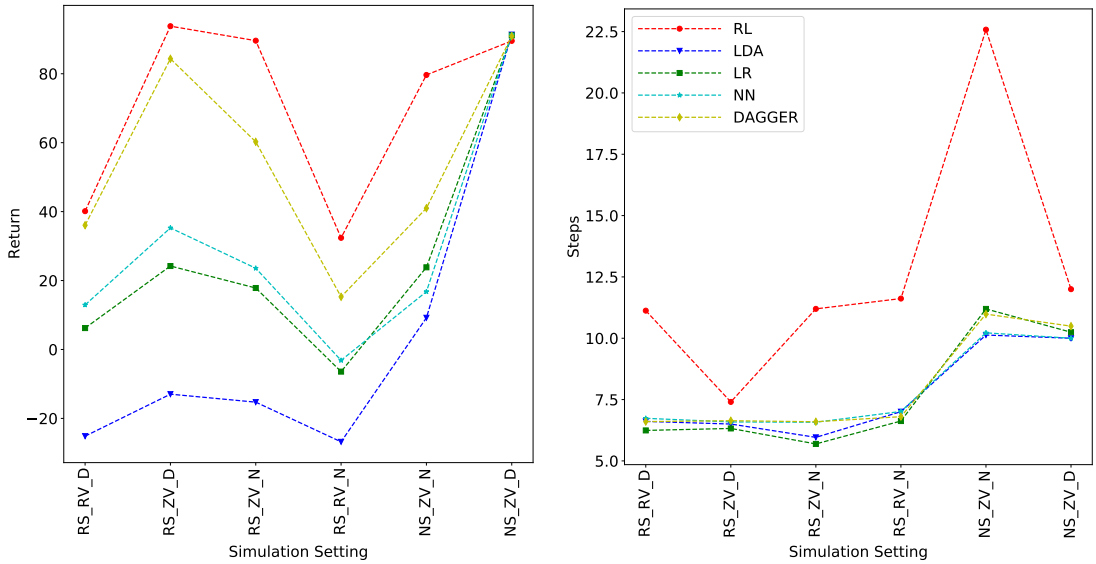


Figure 5.2: Average reward (left) and average number of needed steps (right) for all classes of agents.

We use the same settings as before, except for the ones with noise, which does not make sense when considering optimal actions: NS-ZV, RS-ZV, and RS-RV.

The results are given in Figure 5.3. Especially when we start from a random position on the map, we see that (independent from the setting) passive imitation learning with neural networks selects optimal actions more often than active imitation learning or deep reinforcement learning. Interestingly, it selects both optimal *and* fatal choices very often.

5.4 Discussion

We found that passive imitation learning agents perform poorly (see Figure 5.1) even though they select optimal actions most often. One reason for this is that the data sets from which they learn contain samples that have not been generated by iteratively improving the current policy. Hence, it is unbiased towards sequences of dependent decisions leading to good performance. We have observed that DAGGER and DRL (even stronger) sometimes do not select optimal actions, but those with lower risk of hitting a wall. As a result, they need more steps than other approaches before reaching the goal, but the trajectories they use are more secure and they crash less often.

This is an interesting insight, as all approaches (including PIL) try to optimize the same objective: reach the goal as soon as possible without hitting a wall.

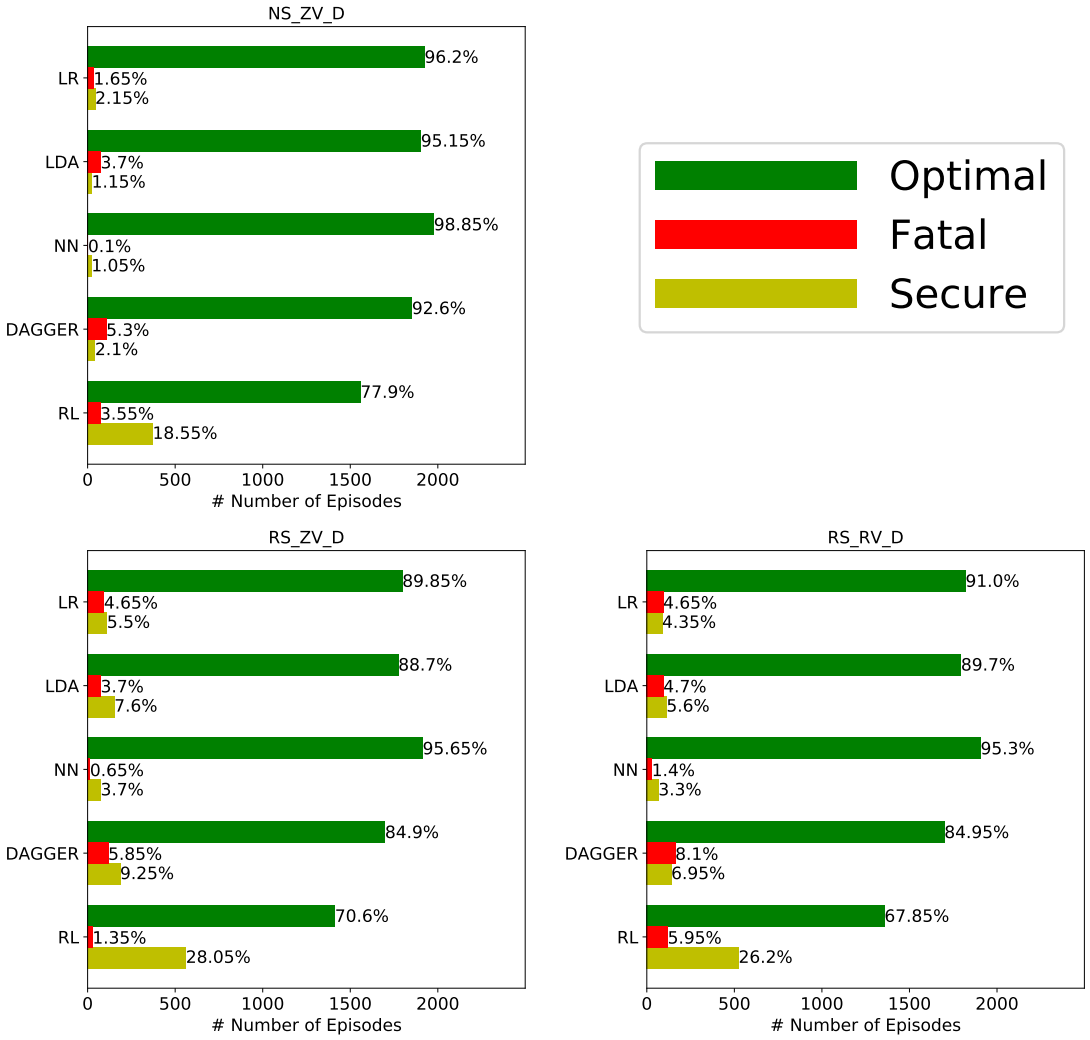
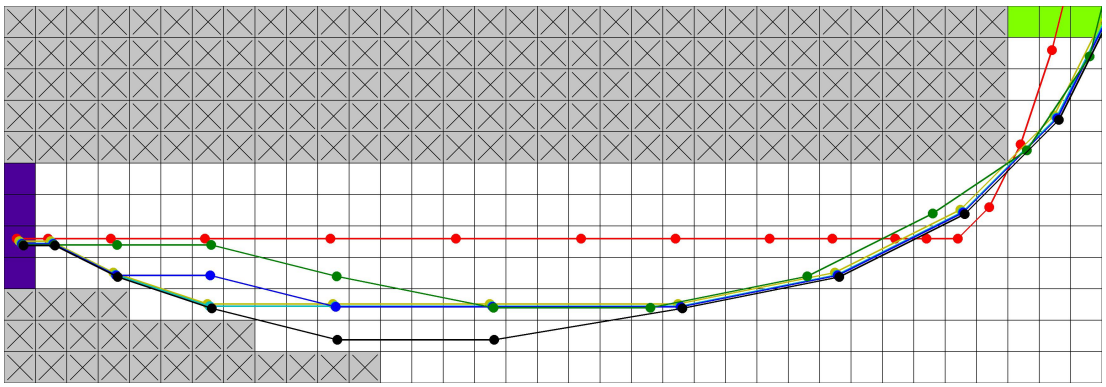


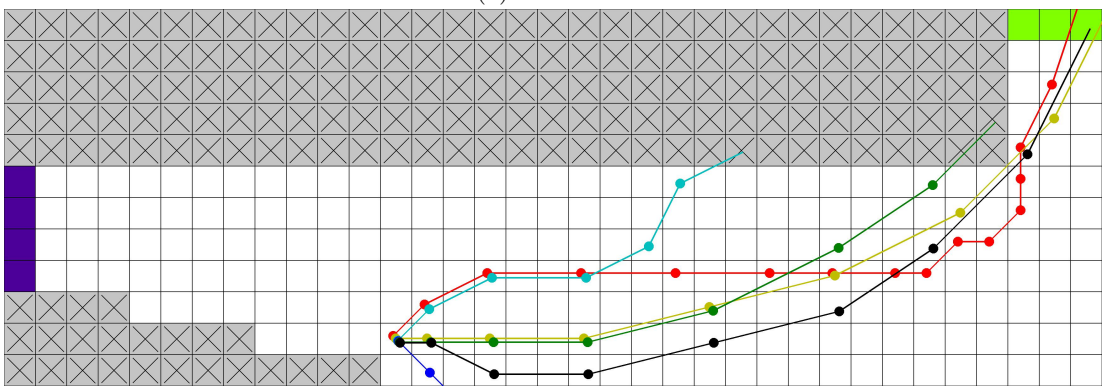
Figure 5.3: Quality of selected actions.

The fact that both DAGGER and RL have a relatively high number of fatal actions but not an increased number of losses leads us to the assumption that these agents avoid states where they might make fatal decisions, even though these states could help to reach the goal faster.

Figure 5.4 illustrates the paths taken by the different agents for the easiest case (NS-ZV-D) where all policies reach their goal. DRL differs the most from the optimal (black) trajectory, which describes one of the shortest paths to the goal and obtains the maximum cumulative reward. For the harder setting where a starting point is chosen randomly (RS-ZV-D), only DAGGER and DRL make it to the goal, with DRL using significantly more steps than the optimal agent.



(a) NS-ZV-D



(b) RS-ZV-D

Figure 5.4: Traces of different Racetrack agents. The black trajectories are optimal. The other colors are chosen as in Figure 5.2.

In summary, DRL performs surprisingly well. In some aspects, it performs even better than active imitation learning, which is not only considered a state-of-the-art for sequential decision-making [14] but – in contrast to DRL – even has the chance to benefit from expert knowledge.

6 Tracking Initial States

So far, we compared reinforcement learning and imitation learning. We concluded that DRL is best suited for a sequential decision-making problem such as the Racetrack. With this Chapter, we will no further consider imitation learning but start to examine different aspects of deep reinforcement learning.

One of the most challenging aspects of RL is the exploration-exploitation dilemma [31]. While the RL agents need to *exploit* former observations to maximize the accumulated reward, at the same time, they also has to *explore* to improve the decision-making. This problem even becomes harder when considering environments with sparse/delayed rewards. If rewards are obtained only rarely, the agent needs to explore persistently to see a positive reward at all. Still, it may be problematic to obtain any non-zero (or non-negative) reward by basically taking random choices only.

There have been several completely different approaches to handle this dilemma. One of the most prominent amongst them is reward shaping, i.e., redefining the reward function s.t. rewards are less sparse. While more dense rewards improve the exploration-exploitation dilemma, the reward shaping comes with its very own issues: spreading rewards throughout the state space may alter the optimal policy. The introduction of potential-based reward shaping [20] was shown to maintain the optimal policy. This has later been proven wrong [12] and was then fixed [13] by Grzes. Potential-based reward shaping gives the chance to incorporate domain knowledge into the reward function, basically guiding the agent with the obtained rewards to the desired goal. The remaining challenge is to create domain-specific potentials suited to guide the agent. In some domains, that still is a challenging task.

Another well-known approach to ensure enough exploitation is the optimistic initialization [31]. Hereby, the Q-values are initialized higher than or equal to the highest observable reward. Therefore, the Q-values are biased towards an over-approximation. Each (or almost each) of the actions taken will thus lower the corresponding Q-value at the beginning of the training. Therefore, the agent may think to exploit while it is actually exploring. This approach has recently been lifted to general function approximation such as neural networks [17].

The underlying concept of optimistic initialization is the so-called intrinsic motivation [27, 22], i.e., the motivation to explore unknown parts of the state space (and not making the same decision over and over again) is included in the agents without explicit exploration. Another approach that is also based on intrinsic motivation is the recently published work by Bellemare et. al. [4] that successfully applies count-based exploration with an underlying density model and argues why this basically is another way of introducing intrinsic motivation.

Go-Explore [7] is a new approach that stores former seen but not fully explored states. It uses imitation learning to guide the agent back to such states and start further exploration from there.

In this Chapter, we will also focus on the exploration-exploitation dilemma. We will first demonstrate the problem of sparse rewards with the Racetrack, as the map-based game is perfectly suited to demonstrate the problem. We will secondly show that the adaption of neural networks and their ability to generalize brought great benefits to the solution of the exploration-exploitation dilemma of reinforcement learning. Further, we will propose a simple but useful idea to tackle the exploration-exploitation dilemma and analyze it with the Racetrack tool.

6.1 Training with Sparse Rewards

We consider the standard version of the Racetrack game. When training Racetrack agents with RL, we act ϵ-greedy, i.e., with a chance of ϵ we take a random decision and otherwise follow the current NN controller. In the beginning, we start with $\epsilon = 1$, i.e., we only take random decisions to explore the state space. During the training progress, we decay the ϵ and therefore increase the number of decisions to follow the NN controller.

When considering the NS-D version of the racetrack game, the exploration-exploitation dilemma already becomes visible when looking at the training curves only. As displayed in Figure 6.1, the training curves for the Barto-small map are quite different. This especially holds for the number of training episodes needed until (1) the first time the goal is reached, and (2) the average return plot is increasing. So for this example, the most successful training run reached the goal already after 825 episodes, while the worst one was only reaching the goal for the first time after 3282 training runs, i.e., needed almost four times as many tries to reach the goal at all. The same tendency can be observed when checking for the average return to become greater than 0, i.e., the agent really starts learning how to reach the goal. The earliest occurrence is after 2824 episodes, the latest one after 7577. So already for this relatively small map, the problem of random exploration becomes clear.

This observation is even more highlighted when considering larger maps. As can be seen in Figure 6.2a, the training on the Barto-big map is failing. Even with the doubled number of training episodes, the agent is not reaching the goal for a single time. Therefore, the agent is not able to learn how to navigate to the goal, but only how to avoid crashes. This especially comes clear when considering Figure 6.2b that depicts the exploration of the agent on the Barto-big map. While the area around the start line is explored extensively, the exploration decreases sharply while moving towards the goal line. To successfully train a Racetrack agent on this map, the vanilla deep Q-learning approach is not working - at least not with a reasonable number of training episodes.

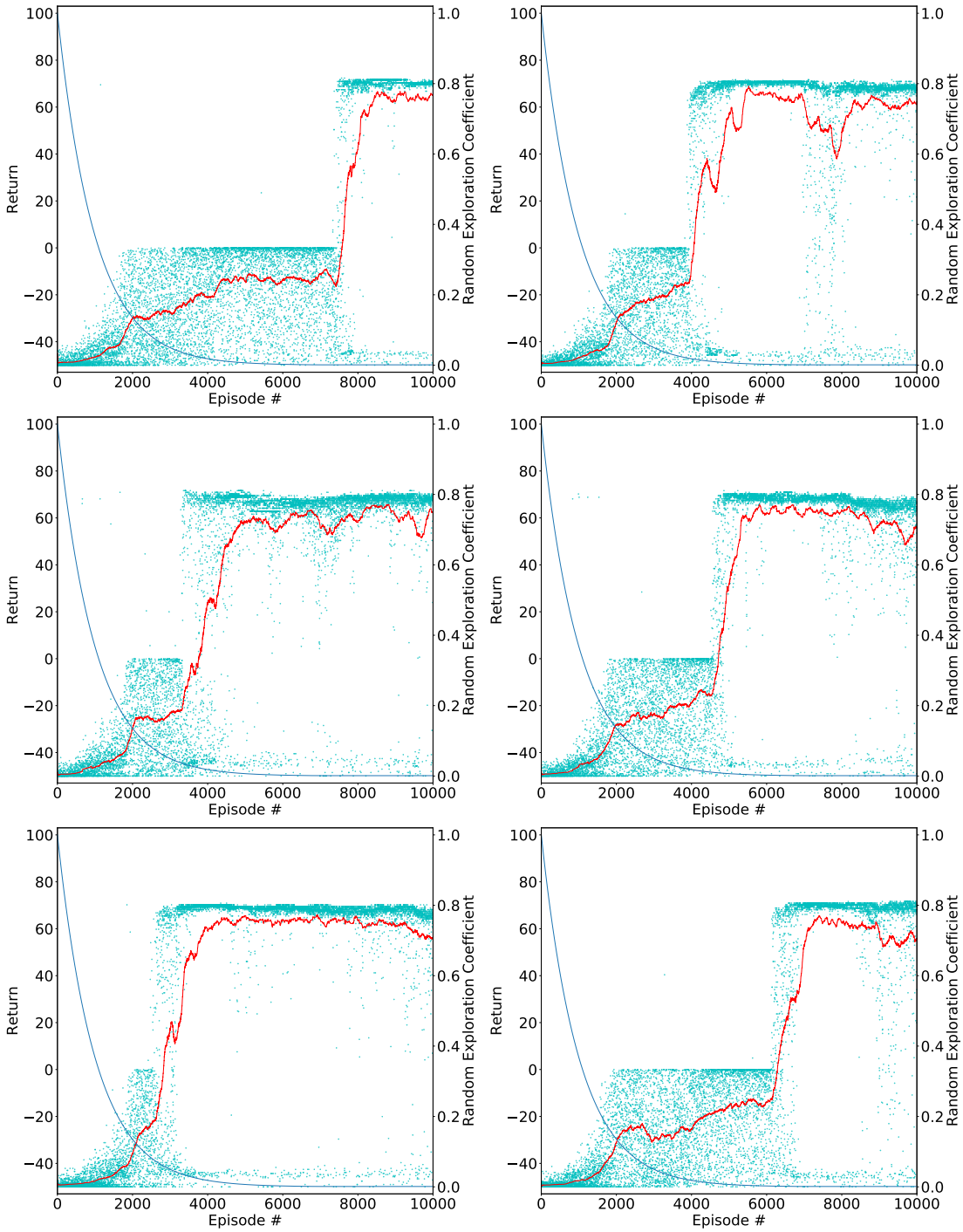
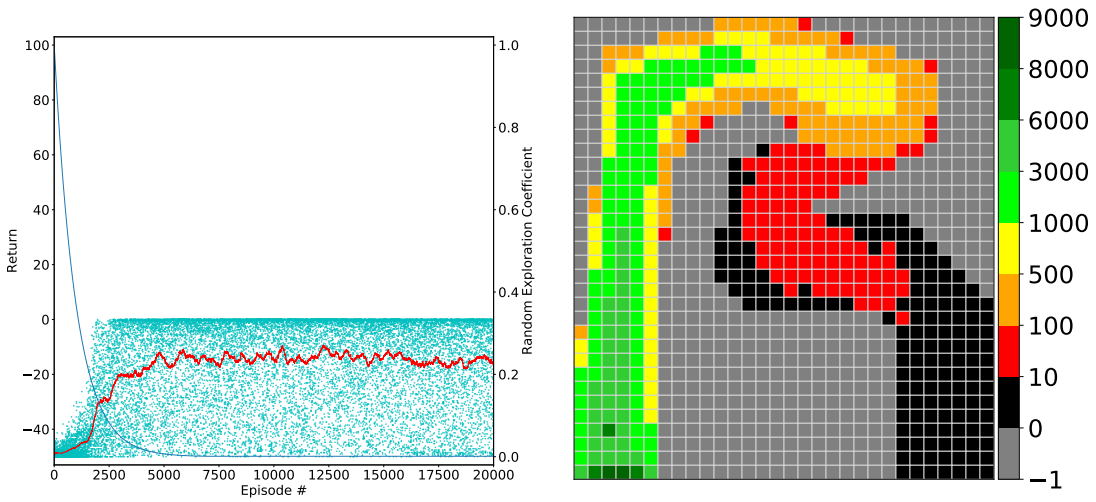


Figure 6.1: Training plots of the Barto-small map in the NS-D version with 6 different random seeds.



(a) Trainings plot.

(b) Exploration map.

Figure 6.2: Training plots of the Barto-big map in the NS-D version

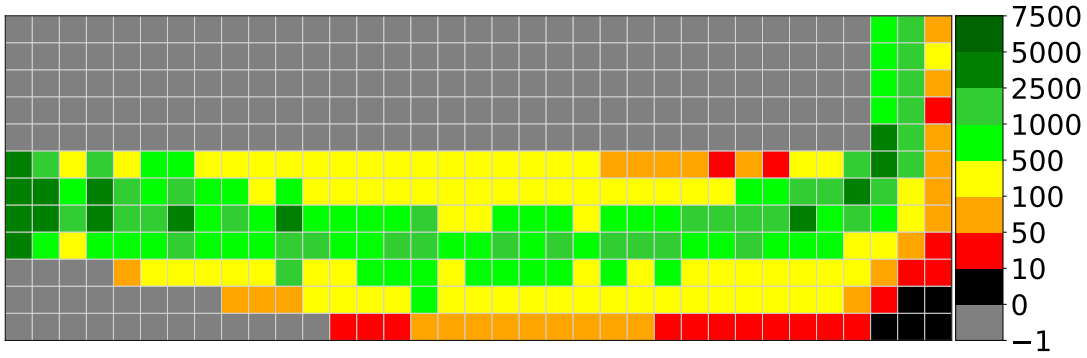
6.2 Generalization Effect of Neural Networks

When considering tabular RL, Q-learning is only working with sufficient training runs, such that the Q-values of every state can be adjusted. Recall that we are often presenting the state space via the two-dimensional map, but in fact, the state space is four-dimensional. In a tabular approach, every single state must be visited repeatedly to learn how to act from there.

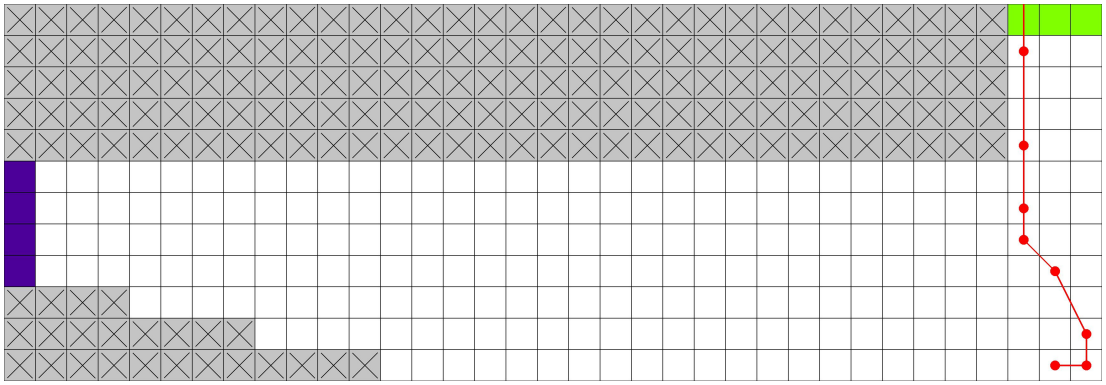
To show that the usage of (deep) neural networks brings benefit to the exploration-exploitation dilemma, we consider an agent trained in the NS-D variant on the Barto-small map. To stress test it, we perform evaluation runs in the RS-D variant. As displayed in Figure 6.3a, during training, the agent just rarely visited the lower right corner. In particular, the position on the second most right on the bottom line was visited only four times during the 10^5 training runs. More on, the state with that position and zero velocity was never visited at all.

Surprisingly, when considering an evaluation run in the RS-D variant starting in that position, i.e., starting in a state the agent has not seen a single time during training, it is able to solve the problem and make it to the goal. This can be seen in Figure 6.3b.

When considering tabular approaches, this would not have been possible at all, as all states have to be learned for themselves. In contrast, when using the neural network, the ability to generalize makes it possible for the agent to also solve the problem from that state on. This observation demonstrates how neural networks help solve the exploration-exploitation dilemma, as not all states have to be visited (repeatedly) to solve the problem.



(a) Training heatmap of the NS-D variant.



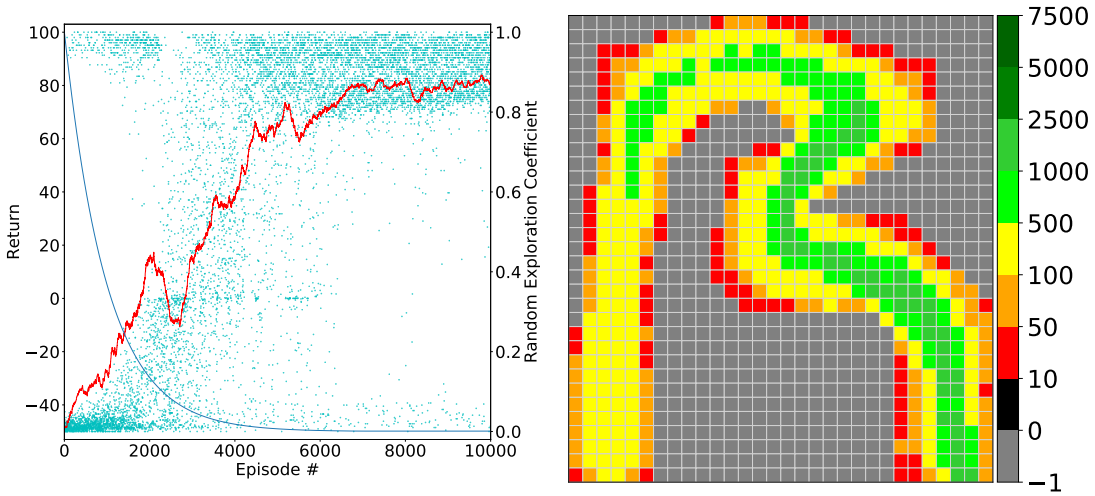
(b) Evaluation run of the policy in the RS-D variant.

Figure 6.3: Stress test: training a policy in the NS-D variant and testing it in the RS-D variant.

6.3 Changing Initial States

So far, in this Chapter, we have seen that the exploration-exploitation dilemma is present in the Racetrack. This holds especially for larger maps, but already for smaller ones such as the Barto-small map. Moreover, we have seen that the usage of neural networks can benefit the solution of the exploration-exploitation dilemma, as their ability to generalize can help solve problems even if that particular state has only been visited rarely, if so at all. Still, the problem presented in Figure 6.2 stays unsolved: the neural networks cannot generalize if the goal was not reached for a single time, i.e., no positive reward has been observed.

We, therefore, propose a simple yet effective way of exploring the state space: instead of just starting from the start line, which was the original task and to which we referred to as NS so far, we randomly start from multiple different states sampled throughout the state space, which we have so far called RS.



(a) Trainings plot.

(b) Exploration map.

Figure 6.4: Training plots of the Barto-big map in the RS-D version.

In contrast to supervised learning and particularly imitation learning, Reinforcement learning is not based on labeled data but simulation-based self-play. This makes it possible to simply change the selected initial state. In general, these initial states can be reached by sampling either from the whole state space or sampling some of the state variables from their domain. We have implicitly already considered the former as RS-RV and the latter as RS-ZV. In this Chapter, we focus on RS-ZV, i.e., we uniformly sample the position out of all non-wall positions from the map but leave the initial velocity unchanged, s.t. the agents start with zero velocity.

While one's first intuition may be that having several starting points is harder than having just a few ones, we observe quite the opposite: even though more states have to be learned for, the agent's quality improves. In particular, in some instances, the training only becomes possible throughout that method. By randomly sampling through the state space, some selected states are easier to solve, i.e., in the Racetrack, those placed closer to the goal line. When reaching the goal from there, the neural network's ability to generalize helps us solve the problem also from other initial states - even if the trajectory from there does not visit the same particular states.

Figure 6.4a shows us that the training for the RS-D setting succeeded. More on, we can observe from Figure 6.4b that the evaluation of the map was significantly more balanced than with the usual training method. When evaluating the game with the original task, i.e., starting on the start line (NS-D), all possible runs lead to the goal without crashing into a wall. The corresponding runs can be found in Figure 6.5. Thus, the original task was solved even though the initial states were varied during the training phase.

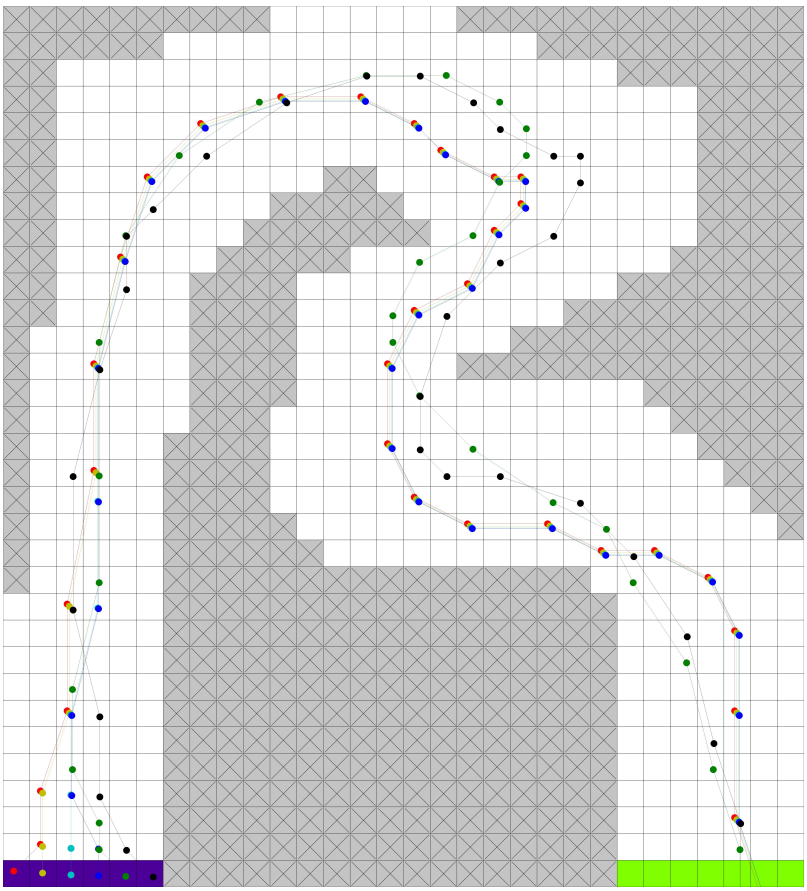


Figure 6.5: Evaluation runs of the RS-D trained policy in the NS-D variant.

6.4 Evaluating Resulting Policies

We have seen that starting spread throughout the state space may help solve tasks that would not be solvable otherwise. Still, there remains the question about how the changed training procedure infects the resulting policy if both training methods are successful. Therefore, in this section, we consider policies trained in all four of the settings described in Table 4.2 and afterward compare them again in all of these four settings on the Barto-small map.

We can observe in Figure 6.6 that two of our findings are just as expected:

- The agent that was trained in the exact same setting that the evaluation uses receives the highest average return, i.e., performs best.
- When noise is included in the evaluation, the agents trained in noisy variants perform better than those trained in the deterministic ones.

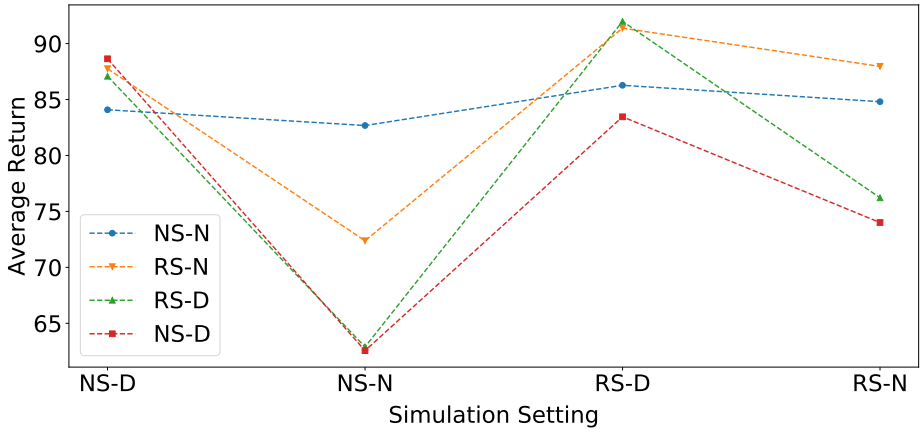


Figure 6.6: Average return of agents trained in four simulation variants and evaluated in the same four variants.

What is worth mentioning is that when considering the evaluation of the NS-D variant, the agent that is performing the second best is the one trained in the RS-N setting, followed by the one trained in the RS-D setting. This means that two agents trained to start anywhere on the map perform better than an agent that was specifically trained to start from that point on. Also surprising is that the agent trained in the NS-N setting is quite stable in its performance, independent from the given setting.

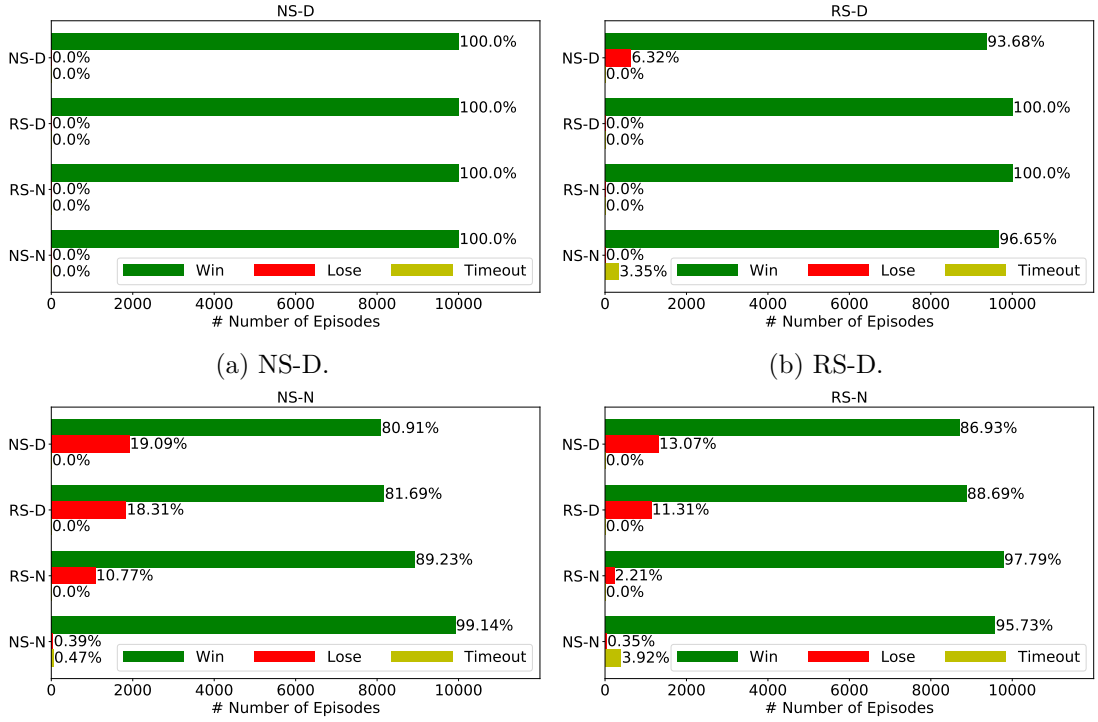
What can be well observed is that the agent that was trained in the RS-N setting is performing best overall. While it performs best when evaluated on the very own variant, it performs second-best on all other settings. The explanation is quite simple:

1. The agent that was specifically trained for the given setting performs (slightly) better but is, therefore, (very) limited in the other evaluations.

2. Due to the better exploration achieved by the variation of the initial state, the agent is able to react in almost all circumstances. This makes it (1) sometimes act more carefully, thus needing more steps to reach the goal, and (2) in total reaching the goal more frequently.

Additional evidence is provided in Figure 6.7. For the deterministic variants, all RS-trained agents have a 100% success rate. For the NS-N setting, the RS-N agent has the second-highest success rate.

Therefore one can say that the agent trained in the RS-N setting is more stable than the others.



(a) NS-D. (b) RS-D.

(c) NS-N. (d) RS-N.

Figure 6.7: Win-Lose-Timeout evaluation of all four considered agents.

6.5 Summarizing Initial States

This Chapter provided an introduction to the exploration-exploitation dilemma. With the help of the Racetrack, we provided a short example of why this dilemma sometimes makes it impossible to learn a task if rewards are only observed rarely. More on, we provided a simple yet useful method to make learning possible at all: instead of solving each episode from scratch, we just randomly start anywhere. Due to the neural network's capability of generalization, this can help the agent to learn how to solve the task at all.

When evaluating on benchmarks where both the standard training and our approach of changed initial states are able to solve the task, we find that the original agent performs slightly better, even though our agent seems to be more robust. As our approach on some benchmarks makes training the agents possible at all, one can easily come up with combinations, e.g., starting the training with our approach until the goal is reached repeatedly and then continuing the training with the original task or more complex combinations. This leaves room for future work.

Even if these findings are promising, they are so far limited to the Racetrack. To make general conclusions, further experiments on other benchmarks would be necessary.

7 Tracking Reward Functions

Designing the reward function is a critical part of each reinforcement learning task. The idea of optimizing a reward function gives an advantage over supervised learning because the reward function, indicating what the goal is, can be optimized without the knowledge of optimal actions/decisions within the single states. Still, the design of the reward function is crucial for the success of the RL agent [31].

While there are examples, such as game-like environments, where the task itself is to maximize a given score [6], in other environments, the translation of the original goal into a reward function is not trivial [31].

Considering the Racetrack, two apparent differences need to be considered when engineering a reward function: winning the game and losing it.

Our foreign goal is primary to reach the goal without crashing into a wall and secondary to do this as fast as possible. In this Chapter, we will consider different reward functions, based on the distinction of whether the chosen action leads to a win, to a loss, or nothing of both, and analyze which one serves our purpose the most.

7.1 Reward Function

In contrast to the reward function described in Chapter 2.2, we will here consider a reward function of the form

$$R\left(s \xrightarrow{(a_x,a_y)} s'\right) = \begin{cases} 100 & \text{if } s' = \top, \\ c \cdot (-100) & \text{if } s' = \bot, \\ 0 & \text{otherwise} \end{cases}$$

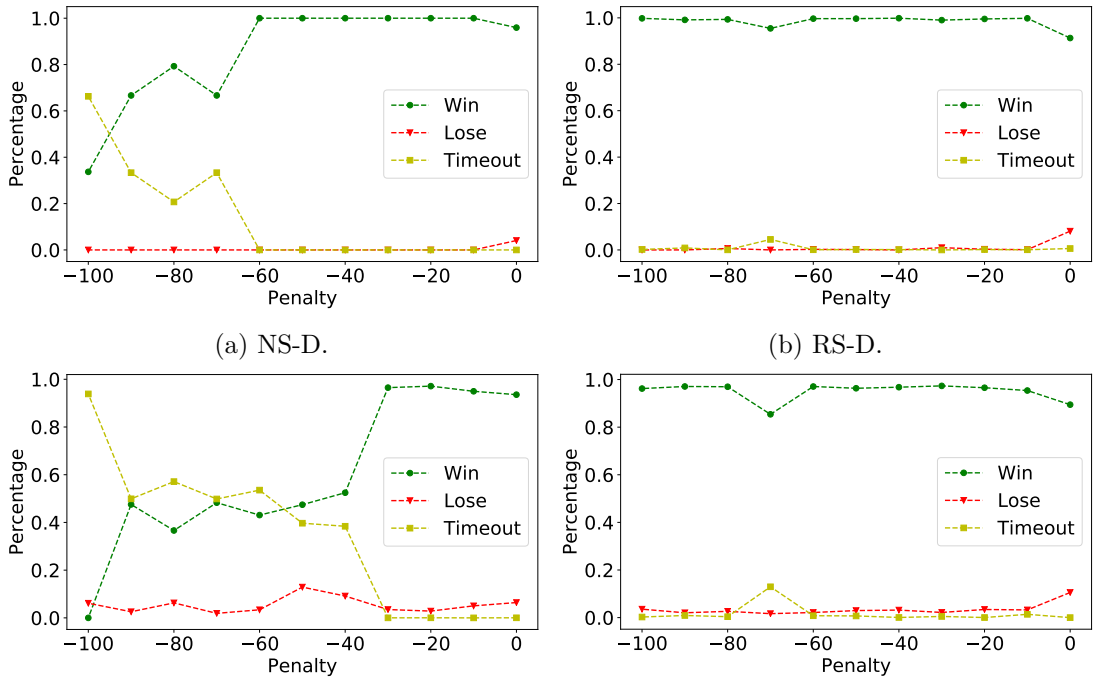
with $c \in [0, 1]$.

To check the reward function's influence, we vary the penalty $p = c \cdot (-100)$ that is applied when crashing into the wall. Therefore, we inspect the relation between rewarding a win and penalizing a loss for this general kind of problem (up to scaling).

7.2 Grid Search

To investigate the influence of the penalty, we consider a grid-search. We train several policies on the Barto-small and the ring map and then perform an analysis similar to Chapter 5.1, i.e., checking whether they reach the goal, crash into the wall, or time out. As can be seen in Figure 6.1, the random seed has a major influence on the training procedure. To minimize the random seed's influence, we train six different agents with different random seeds for each setting and report the average results.

7.2.1 Barto-small



(a) NS-D.

(b) RS-D.

(c) NS-N.

(d) RS-N.

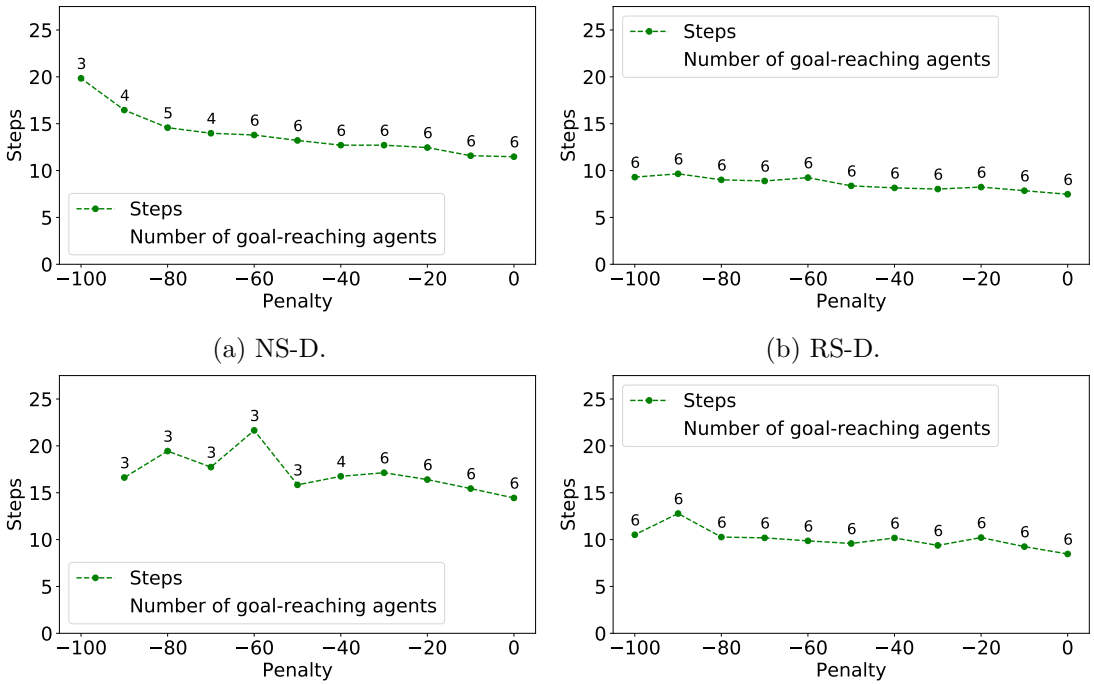
Figure 7.1: Win-lose-timeout evaluation of different penalties on the Barto-small map.

Figure 7.1 displays the evaluation for the Barto-small map. We find that for the RS evaluation (7.1b, 7.1d), i.e., starting anywhere on the map, the height of the penalty has just little influence on reaching the goal. We find a slight decrease of successful runs when having a penalty that is -20 or greater. Note that there is a singularity for both the deterministic and the noisy version when $p = -70$, for which we have no explanation. This singularity can also be observed weakened when considering the NS version (7.1a, 7.1c) of the Racetrack. More on, in the NS setting, there is the observation that if the penalty is too high, the agent is reaching the goal less often/struggling to learn how to

reach the goal. For the NS-D setting (7.1a) the penalty should at least be -60, for the NS-N setting (7.1c) at least -30.

Overall, we find for the Barto-small map that although it is important to have a penalty, it must not be too high in order to master the phrased task successfully. For all settings, the optimal penalty seems to be within $[-30, -20]$, i.e., the best relation between rewarding the goal and penalizing a loss is within $[0.2, 0.3]$.

Considering our secondary goal, reaching the goal line as fast as possible, Figure 7.2 shows that with an increasing penalty (i.e., decreasing goal rewarding-loss penalizing relation), the number of average steps to reach the goal is decreasing. As this is our secondary goal only, for Barto-small, we conclude to choose the relation near 0.2, i.e., $p \approx -20$.



(a) NS-D.

(b) RS-D.

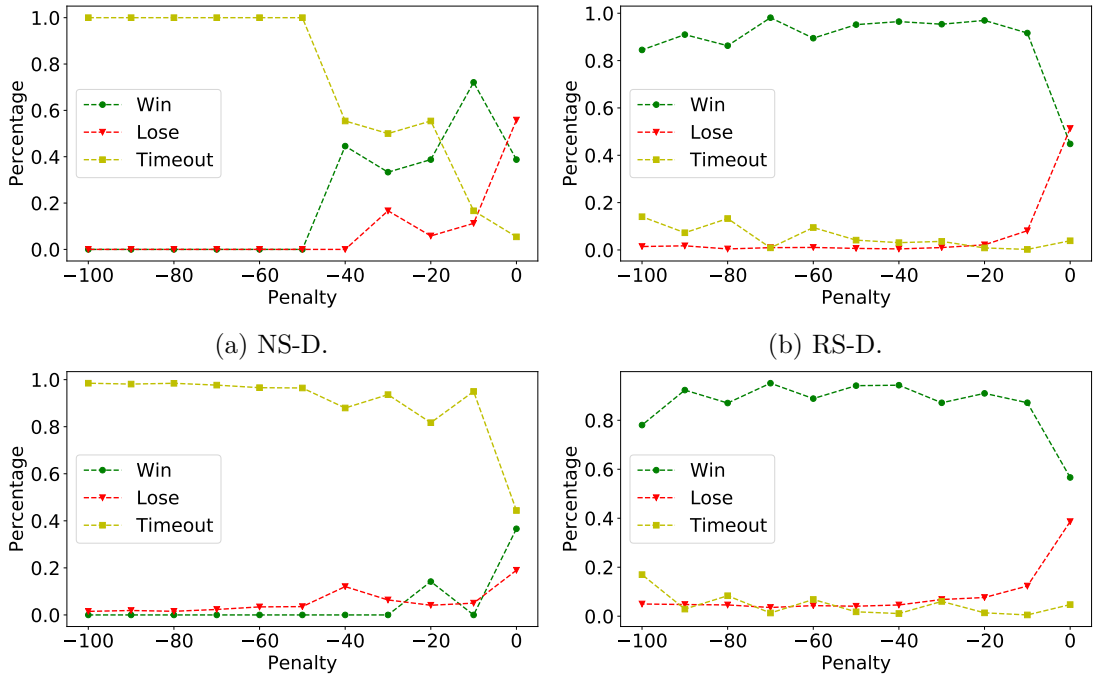
(c) NS-N.

(d) RS-N.

Figure 7.2: Average number of taken steps in the successful runs. Digits give the number of agents able to solve the task at all (of six in total).

7.2.2 Ring

When considering the same evaluation that we did on the ring map, Figure 7.3 depicts that the observations are not as clear as for Barto-small. For the RS setting (7.3b, 7.3d), a penalty $p = -70$ seems to be the best one. For RS-D (7.3b, $p \in [-50, -20]$ follows up quite close, while for RS-N (7.3d) $p \in [-50, -40]$ follows closely with $p \in [-30, -20]$ having a slightly decreased number of wins.



(a) NS-D.

(b) RS-D.

(c) NS-N.

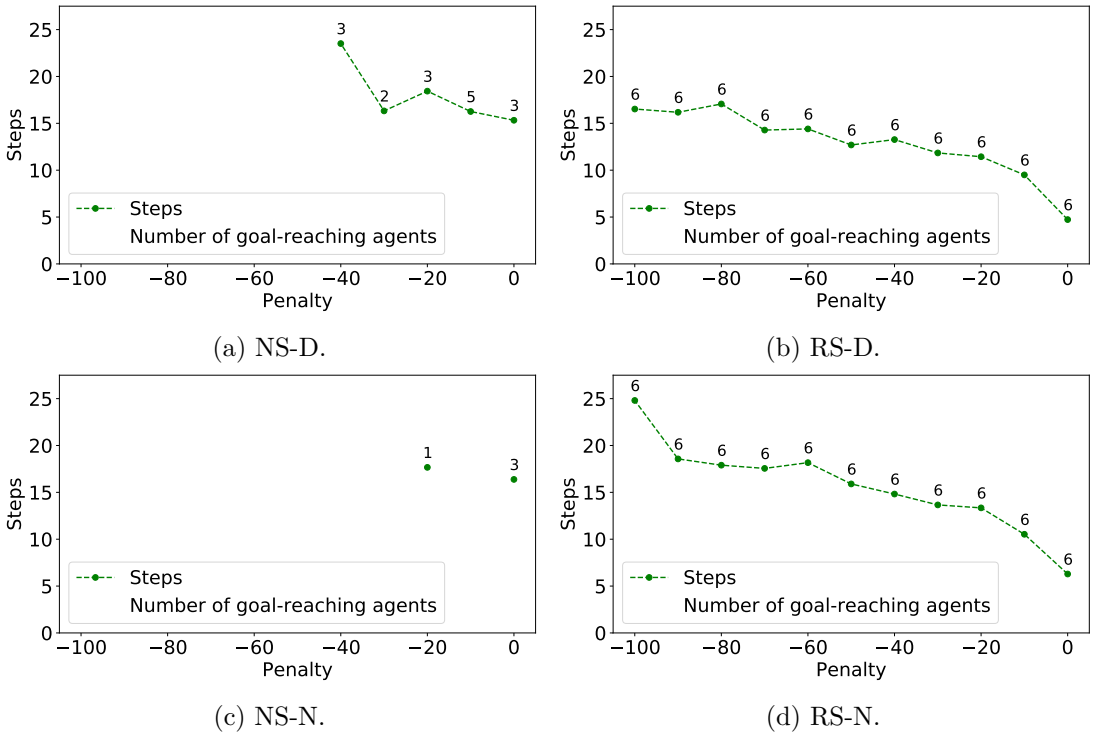
(d) RS-N.

Figure 7.3: Win-lose-timeout evaluation of different penalties on the ring map.

The NS settings (7.3a, 7.3c) reveal that the agent's general capability of solving the task depends on the chosen penalty. This can, for the ring map, even be observed better than for the Barto-small one. For the NS setting, many agents fail to solve the task at all. Roughly speaking, the number of different seeds that are able to solve the task is increasing with an increasing penalty in the NS setting. (While for the RS setting all training runs are able to find the goal.) This tendency can be seen in Figure 7.4.

The same Figure strengthens our findings from the Barto-small map for the secondary goal: the higher the penalty, the fewer steps are taken when driving to the goal, i.e., the riskier are the resulting policies.

Overall, the evaluation of the ring map is not giving us such a clear conclusion as the Barto-small map did: the optimal penalty here clearly depends on the setting. While for RS $p = -70$ seems to be optimal, as it leads to the highest percentage of runs making it to the goal, this penalty leads to a training failure in the NS setting. The conclusion of having no penalty is never a good option can be affirmed.



(a) NS-D.

(b) RS-D.

(c) NS-N.

(d) RS-N.

Figure 7.4: Average number of taken steps in the successful runs. Digits give the number of agents able to solve the task at all (of six in total).

7.3 Summarizing Reward Functions

Obviously, engineering a reward function for a phrased goal is not easy. Even if considering this special case of problems (win-lose-timeout), it remains a challenging task due to the multiple intentions we aim for: primarily optimizing the number of wins and secondary the number of taken steps.

This Chapter provided an extensive comparison of the relation between reward for a win and penalty for a loss. Our conclusion can be divided into three parts.

1. If the penalty is too low, the agent will fail to learn anything.

2. If there is no penalty, i.e., $p = 0$, the agent will have problems learning to avoid crashes.

3. The higher the penalty, the riskier are the resulting policies.

Even though we considered only the Racetrack, a game with different variants, and not any other games, we found that the optimal ratio between reward and penalty differs depending on the given map. For the ring map, which is rather narrow and where the agent is always close to the wall, we have different optimal penalties for different settings. For the Barto-small map, which can be considered spacious and only has a few positions where it is necessary to be close to the wall, an optimal penalty of $p = -20$ can be determined.

Unfortunately, this game-specific observation gives us the chance to make a general conclusion: if already for two different maps of the same game with the same rules, it is not possible to make a general conclusion about the optimal penalty-reward ratio, then there cannot be a general conclusion for this ratio that generalizes over several games or even this specific type of games. The design of the reward function for phrased objectives remains an open challenge. The success of the agents is brittle to even small changes in the reward structure.

This leaves room for future work. Systematic experiments that help to identify the optimal reward function for a phrased objective are interesting for the future. While a pure grid search as performed here may be too expensive, one could think about other optimization approaches to find the optimal reward function.

8 Tracking Discount Factors

In the former Chapter, we ran experiments to optimize the reward structure for the worded objective of the Racetrack: mainly reaching the goal without crashing into the wall and secondary to do that as fast as possible. While the main purpose can be expressed explicitly through the design of the reward function, the secondary objective is only given through the discount factor γ.

With this Chapter, we want to examine the discount factor's influence on both the primary and secondary goals. Similar to our former experiments, we consider a grid search about possible values of the discount factor and report the average of 6 agents trained with different random seeds.

8.1 Discount Factor

The goal of reinforcement learning is to maximize the return

$$G_t = \sum_{i=t}^{\infty} \gamma^i \cdot R_{i+1},$$

i.e., the (discounted) accumulation of rewards.

If the discount factor $\gamma = 1$, all rewards, independent of when that reward is received, have the same value. For a discount factor of $\gamma = 0$, only the reward received with the next action is considered.

By tweaking the discount factor, one can balance the importance of current rewards and future rewards. In the case of the Racetrack and other win-lose-games, one can influence how important it is to reach the goal as fast as possible.

8.2 Grid Search

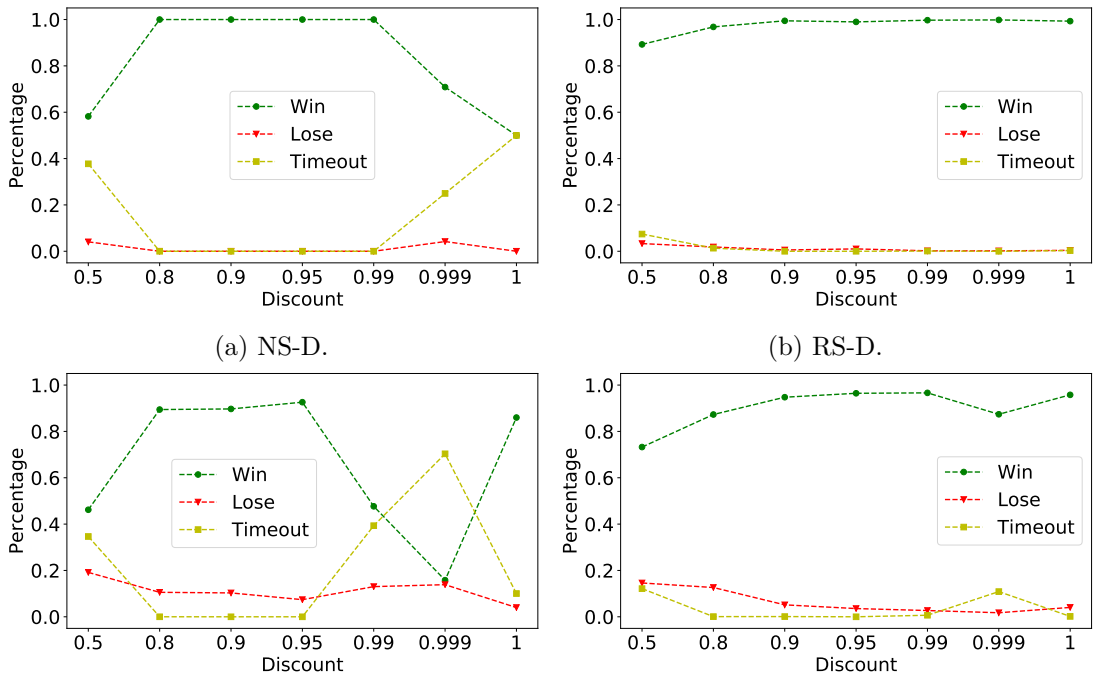
While for our former experiments we used $\gamma = 0.99$, we now consider several values within $[0.5, 1]$. While the upper border is given through the definition of the discount factor, the lower border was found experimentally.

8.2.1 Barto-small

The evaluation considering the statistical win-lose-timeout analysis can be seen in Figure 8.1. While the RS variants (8.1b, 8.1d) are rather stable and only show a slight decrease of wins when having a discount factor that is too low (with a minor exception in RS-N when $\gamma = 0.999$), a clear tendency can be observed when considering the NS variants (8.1a, 8.1c). While a too small discount factor (here 0.5) leads to the major goal being forgotten, as future rewards are rather unimportant, a too high discount factor also disturbs learning. This can also be seen when considering Figure 8.2, as the number of agents solving the task at all partially drops when the discount factor is too high.

The number of agents that are able to reach the goal at all decreases when the discount factor is too close to 1. Surprisingly, for the NS-N setting, the results of $\gamma = 1$ are noticeably better than $\gamma = 0.999$.

Considering some of the training curves of the NS-D setting for $\gamma = 0.999$ (Figure 8.3) and $\gamma = 1$ (Figure 8.4), we can see that the training seems to be successful at first sight. For some of the $\gamma = 0.999$ training runs and for of the $\gamma = 1$ training runs, the return drops after the task already seemed to be learned.



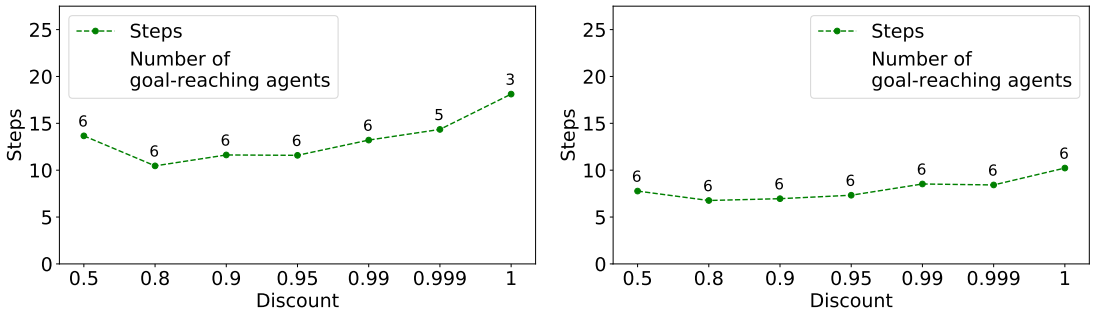
(a) NS-D.

(b) RS-D.

(c) NS-N.

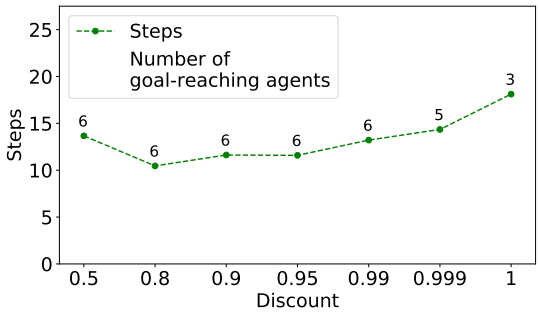
(d) RS-N.

Figure 8.1: Win-lose-timeout evaluation of different discount factors on the Barto-small map.

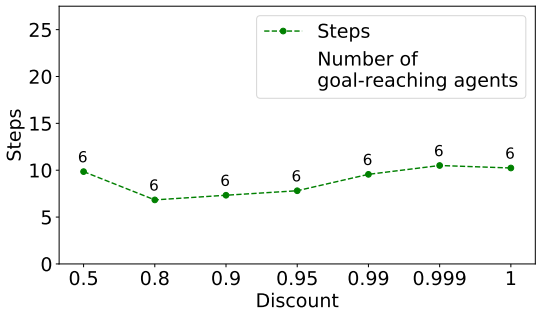


(a) NS-D.

(b) RS-D.



(c) NS-N.



(d) RS-N.

Figure 8.2: Average number of taken steps in the successful runs. Digits give the number of agents able to solve the task at all (of six attempts in total).

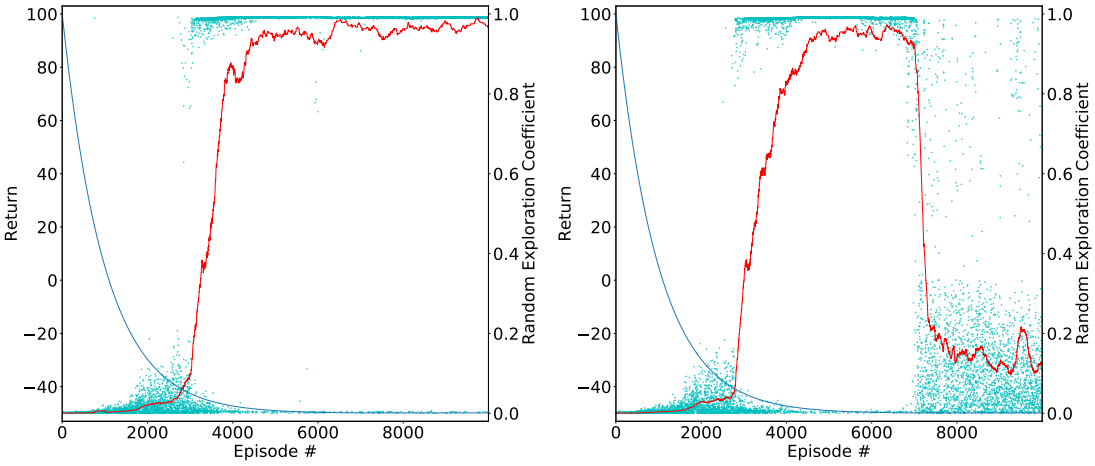


Figure 8.3: Training curves in the NS-D setting on the Barto-small map with $\gamma = 0.999$ with two different random seeds.

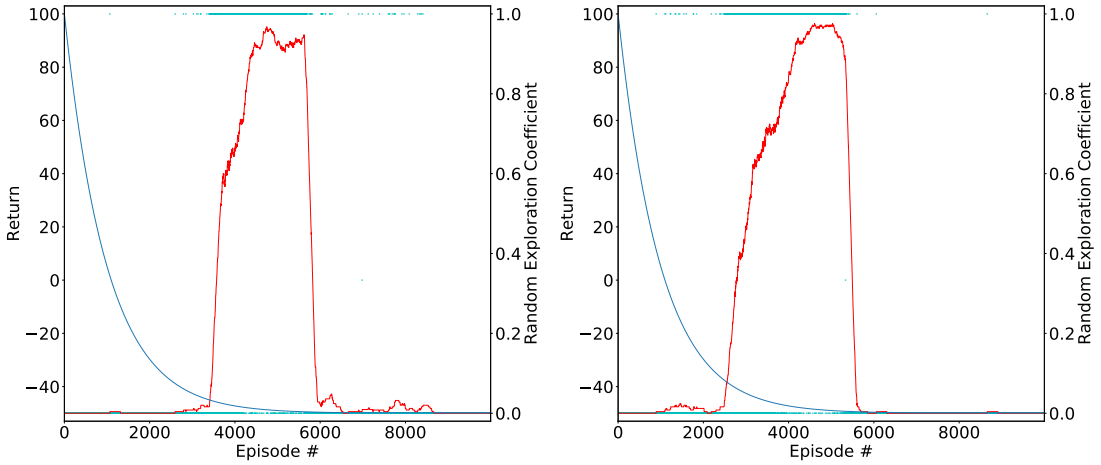


Figure 8.4: Training curves in the NS-D setting on the Barto-small map with $\gamma = 1$ with two different random seeds.

To check why this catastrophic forgetting occurs, i.e., why the agent's performance is dropping so suddenly, we examine the maximal Q-value that can be observed in the different states.

The neural network is updated using of the maximal Q-value that the network estimates for the successor states (see Equation I). Thus, (if we restrict ourselves to reasonable velocities,) we expect the values to increase with decreasing goal distance. As the Racetrack has a four-dimensional state space, we cannot simply illustrate $\max q_\theta(s, a)$ for all states $s \in S$. Therefore, Figure 8.5 is restricted to zero velocity. Although this is only a small part of the state space, our investigations show that the findings are similar for all reasonable velocities.

What catches the eye is that the opposite of our expectation is the case: with decreasing goal distance, the Q-values drop. Thus, it is of no wonder that the agent is not able to find the goal with the current network: If the estimations of the Q-values near the start line are better than the ones near the goal line, the agent will never navigate into goal direction.

This behavior can be explained with the underlying update rule. At the beginning of the training, the agent is completely uninformed, taking random actions. When the goal is reached for the first time, the Q-values of the states close to the goal will become better, i.e., their value will increase. When updating the agent according to Equation I with $\gamma = 1$ (or values close to 1), the network will finally reach an approximation, where the q-value of all states must be equal: when the discount is (close to) one, then it does not matter at all whether the goal is reached with a few or with potentially infinitely many steps. Thus, all Q-values will equalize. After that happened, the agent is once

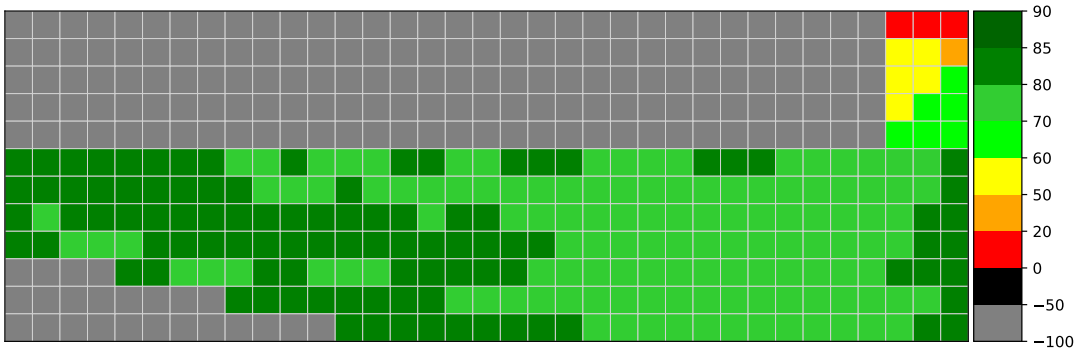


Figure 8.5: Heatmap for the estimated maximal Q-values with zero velocity at position $p = (x, y)$, i.e., $\max q_\theta(s, a)$ with $s = (x, y, 0, 0)$.

more uninformed: all of the states have approximately the same Q-values. It is unable to distinguish good states from bad ones. As we exponentially decay the exploration coefficient, the agent is unable to escape from that bad estimation, which can, e.g., be observed in Figure 8.4.

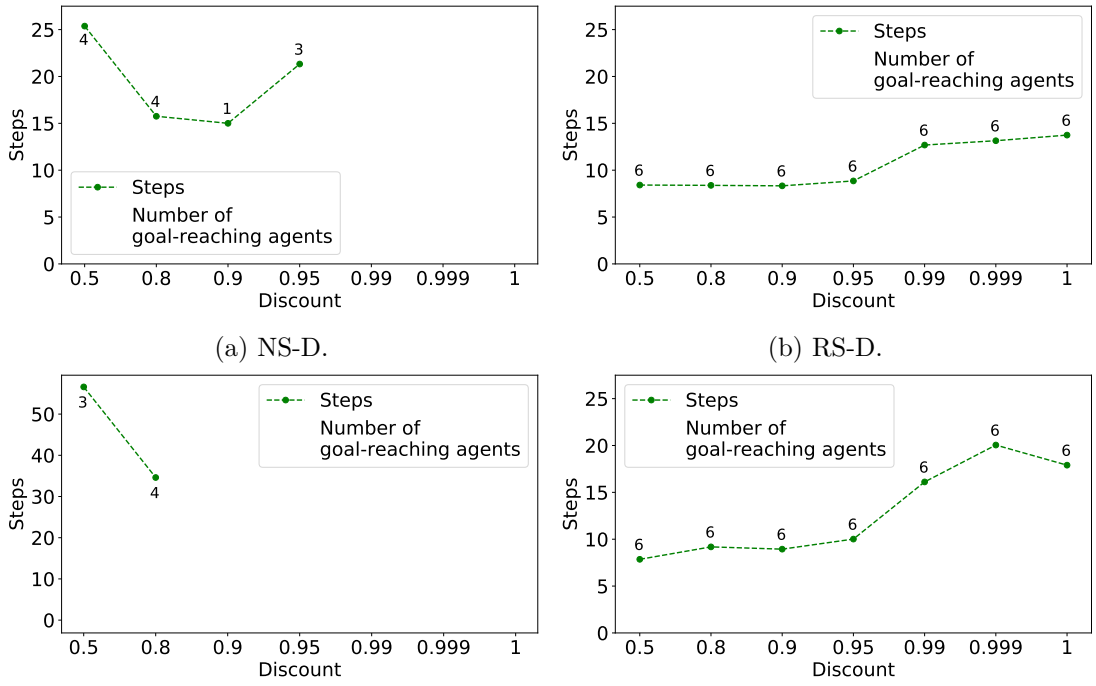
When considering the successful range ($[0.8, 0.99]$) of the discount factor only, Figure 8.2 provides a further surprise: the fewest number of steps occurred for $\gamma = 0.8$ and not, as expected, for $\gamma = 0.5$. We find that using a too-small discount factor can lead to taking more actions, so the opposite of what was intended.

Overall, the best number of wins occurred for $\gamma = 0.95$ and the fewest number of steps was taken for $\gamma = 0.8$, which is why we conclude to use values within that range.

8.2.2 Ring

We repeat our investigations on the ring map to check whether the conclusion of the Barto-small map remains unchanged.

The ring map is more narrow than the Barto-small one. Especially for the NS-N setting, this makes it harder to solve it. For the NS-N variant, only agents with $\gamma < 0.8$ were able to learn the task, for the NS-D setting, only agents with $\gamma \in [0.5, 0.95]$. Additionally, for both of these settings, there is no value of the discount factor for that the problem could be solved for all six of the random seeds. This especially becomes vivid considering Figure 8.6. Note that it might be possible that an increase in the number of training episodes improves the agents or their quality, respectively, but this is not our scope here.



(a) NS-D.

(b) RS-D.

(c) NS-N.

(d) RS-N.

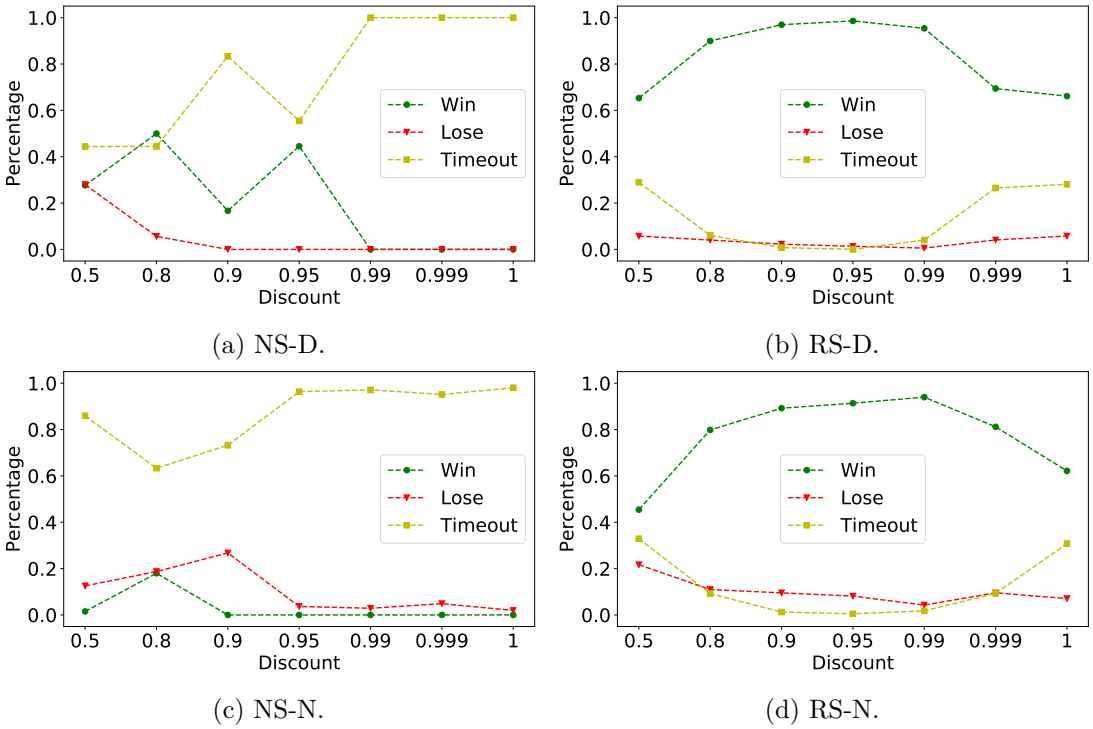
Figure 8.6: Average number of taken steps in the successful runs. Digits give the number of agents able to solve the task at all (of six in total).

We first consider the primary goal. In the NS setting, there are only a few agents able to solve the task. The highest number of wins in Figure 8.7 can be observed for $\gamma = 0.8$ for both NS-N and NS-D.

Considering the RS version, all values within $\{0.9, 0.95, 0.99\}$ perform equally. For RS-D $\gamma = 0.95$ and RS-N $\gamma = 0.99$ have the highest number of wins. $\gamma = 0.8$ performs slightly worse with having only an insignificant decreased number of wins but, especially in RS-N, a significant increase in the number of timeouts.

When we look upon the secondary goal, we generally see that as the discount factor decreases, the number of steps decreases. The exception holds for $\gamma = 0.5$, where the NS-N and the NS-D variants record an increased number of steps when reaching the goal. For all settings except NS-N, the smallest number of steps was taken for $\gamma = 0.9$. For NS-N, no agent was able to solve the task for that specific value.

Overall, the best performance of our primary goal was achieved for $\gamma \in \{0.95, 0.99\}$ and our secondary goal with $\gamma = 0.9$.



(a) NS-D.

(b) RS-D.

(c) NS-N.

(d) RS-N.

Figure 8.7: Win-lose-timeout evaluation of different discount factors on the ring map.

8.3 Summarizing Discount Factors

This Chapter provided an extensive comparison for different discount factors as well as an insight into problems when considering such close to the value one. We can once again divide our conclusion into three parts.

1. If the discount factor is too close to one, the agent will catastrophically forget what he has learned. In our case, this phenomenon occurred for $\gamma \geq 0.999$.

2. If the discount factor is too low, the primary goal will be forgotten. The agent will mainly focus on a few steps and therefore the overall performance decreases. In our case, this occurred for $\gamma \leq 0.5$.

3. The optimal value for the primary goal and the secondary goal differ.

Especially the last point must be phrased: depending on whether it is more important to reach the goal either frequently and safely or quickly, the discount factor should be chosen.

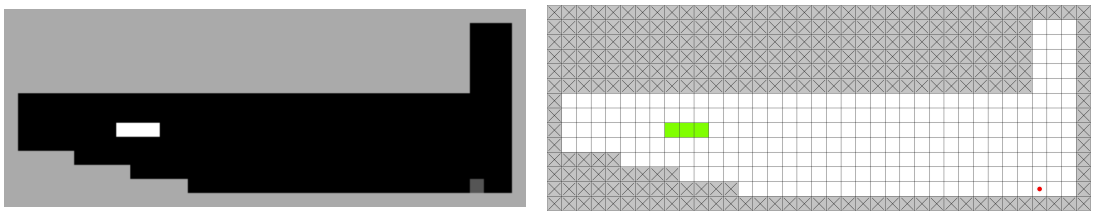
For the Racetrack, the discount factor's optimal value was similar for both maps, regardless of which goal to consider. For general conclusion, especially for the optimal value of γ, further experiments on other benchmarks are necessary. However, it is of our belief that the phenomenon of catastrophic forgetting for too high discount factors generally holds. Future work acquiring more knowledge about why this happens and how it could be circumvented clearly is of interest.

9 Generalized Policy Learning

Now that we investigated how different hyperparameters inflect the learning process and the resulting policy, we take another step towards generally solving the Racetrack domain: instead of finding a policy that is coped to find the fixed goal from the start line (or, as discussed in Chapter 6, from anywhere on the map), we would also like to be able to find an arbitrary goal line without training a new policy. Considering a fixed map size, this is the most straight-forward and simplest way when thinking about learning generalized policies to solve the Racetrack.

9.1 State Representation

The state representation used so far (15 features: position, velocity, wall distances, and goal distances) cannot be used for this purpose. One can easily come up with several counterexamples, where the exact same features describe even the same map with the goal line placed differently and thus having the need to choose different accelerations, i.e., actions. Therefore, we come up with a more general way to represent the state by using its graphical representation. Instead of having a fully connected network, this Chapter will introduce the use of concurrent neural networks that will extract all needed information from a grayscale image.



(a) Grayscale representation.

(b) RGB representation.

Figure 9.1: Barto-small map.

An example of this new state representation is given in Figure 9.1. While Figure 9.1b displays the already examined Barto-small map, Figure 9.1a represents the grayscale representation we will use in the following. Each of the different tiles (wall, free tile, goal), as well as the position of the agent are represented with another, arbitrarily chosen grayscale. In contrast, in our former representation, the displayed state would be given through the array $[11, 22, 0, 0, 6.0, 14.0, 2.0, 6.0, 2.0, 6.0, 14.0, 2.0, -4, -17, 21]$ which represents the position, the velocity, the wall distances, and the goal distances.

Note that, in contrast to the maps displayed in former Chapters, here the outer wall, which was assumed to surround the map, is explicitly displayed. During the learning process, which will be handled in the next section, we found that graphically adding the surrounding wall to the state representation increases the agent's quality and reduces the number of episodes needed for training. This is plausible, as the neural network then receives explicit information about the surrounding walls without the need to learn it. More on, the former state representation also contained this information due to the linear wall distance measures. As this is not the focus of this thesis, we will not report any more details about this.

The grayscale image contains all information that was formerly presented to the neural network, except for the velocity. While in our first experiments we tried to encode the velocity in the graphical representation by adding another grayscale and highlighting the point on the grid where the velocity points to; this representation was dropped, as it gives no option to present the velocity whenever it aims out of the map. Even with the added explicit representation of the surrounding walls, there is no option of representing whether the car would aim just a little or excessively outside of the map. Therefore, the state is represented in a combined manner: through the grayscale image, containing all information about the map and the position of the car, and additionally through the velocity vector $v = (v_x, v_y)$.

9.2 Convolutional Neural Network

To process the new state representation, there is the need to have a suited convolutional neural network [21]. Figure 9.2 depicts the structure of the neural network.

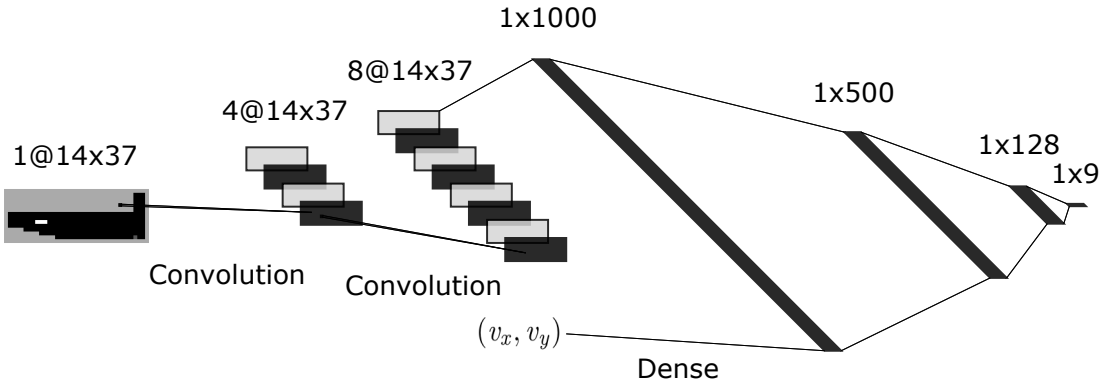


Figure 9.2: Structure of the used convolutional neural network [1].

The grayscale image is processed throughttwo convolutional layers. For both those layers, the kernel and input stride are set to 1. While often convolutional layers need to make use of kernel sizes and strides greater than that, we can here use this small

value due to the fact that we represent every position on the grid through one single pixel. Thus, there is no need to combine several pixels in the state representation, but the pixels can directly be used for feature generation. Additionally, to enable the construction of more general features, a dropout layer with a probability of 0.05 is applied after each convolutional layer. After the two convolutional layers, the eight matrices are linearized and the velocity v is added. Afterward, there follow four fully connected layers, successively decreasing the layer size to nine, which remains the output size.

9.2.1 Feature Prediction

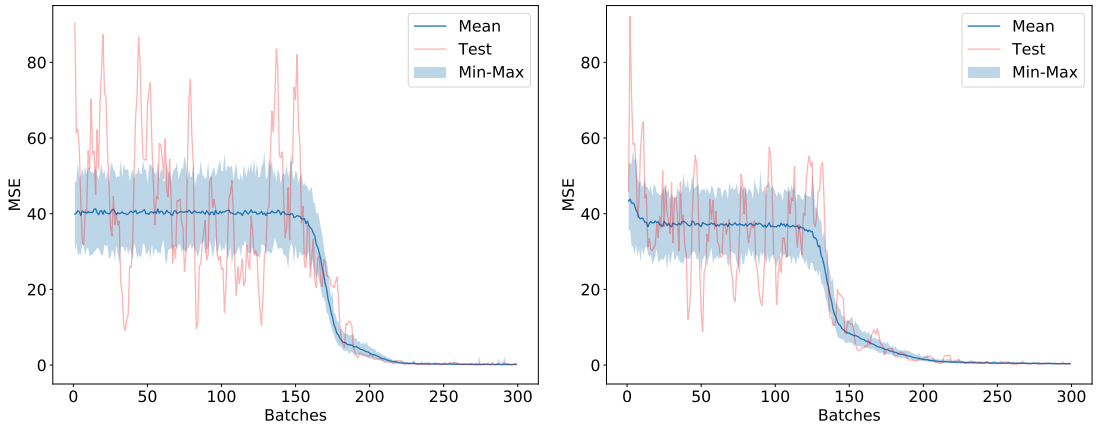
The network structure was found experimentally. The systematic optimization of the network structure is beyond the scope of this thesis. Still, to verify that this network is in principle capable of solving the Racetrack, we make experiments to use the convolutional layers to predict the distance features that were used throughout this thesis. If the network is capable of processing the image and approximate the features, this is an indication that the network is capable of solving the Racetrack. Thus, we make use of a network that consists of two convolutional layers (just like the network used as a policy) and two fully connected layers (thus not as deep as the network used as a policy). The network can process the grayscale image and outputs a vector of size 13, which matches the former state representation without the velocity values. We iteratively proceed in the following manner:

1. generate a random state on the map,
2. provide the grayscale image to the network and predict the vectorial state representation,
3. get the vectorial state representation,
4. compute the MSE of the vectorial state representation and its approximation, and
5. make a gradient descent step to optimize the network.

In detail, we sample a mini-batch of 64 states (1 - 3) and then compute the MSE and update the network (4 - 5) for all of these states simultaneously.

We report about the progress of learning the features in Figure 9.3. The Figure displays batches of 100 iterations (of 64 samples each). Next to the average loss, it shows the minimum and maximum loss that was received when predicting the batch. Additionally, it provides values of test predictions. After every batch, we test the current network to predict the vectorial state features without using this computation to optimize the network. While the features are integer numbers, the network predicts real numbers.

For this test computation, we therefore round the values predicted to the neural network to the next integer value and then compute the MSE. Figure 9.3a displays the prediction progress for the Barto-small map as presented in Figure 2.1 with the goal line in the upper right corner, while Figure 9.3b was obtained when placing the goal line randomly somewhere on the map.



(a) Fixed goal line.

(b) Randomly placed goal line.

Figure 9.3: Feature fitting.

We find that about the first 150 batches, (i.e., 150·100 samples = 150·100·64 predictions,) the network is just roughly learning anything, but afterward, the prediction improves quickly following to a loss close to 0. The provided test values indicate that the state approximation, when rounded to integer values, can be used to train a neural network policy.

As we used (i) a neural network with two fully connected layers based on the vectorial state representation to train an action policy, and (ii) a convolutional neural network based on the grayscale image representation to approximate the state features, we conclude that a combination, i.e., a convolutional neural network with additional fully connected layers, may be suited for training neural network action policies.

9.3 Training

We make use of deep Q-learning, just as we did for the feature-based policy training in Chapter 4. To underline the concept, this Chapter only focus on the setting that received the best results so far, i.e., D-RS-ZV. Further, we chose the hyperparameters from the set of optimal values determined in Chapters 7 and 8; namely, we use a penalty of -20 when crashing into a wall and a discount factor γ of 0.99.

Additionally to uniformly sampling the starting position from the grid, the goal line is also placed 4.1a for every episode. The goal line always consists of three tiles, just as in the original Barto-small map, placed either horizontally or vertically, but never diagonally.

To train the agent to steer to this arbitrary placed goal line, we place it randomly for every training episode. Thus, the replay buffer is filled with states including several different positions of the goal line and the network weights are therefore adapted to be capable of handling these different positions.

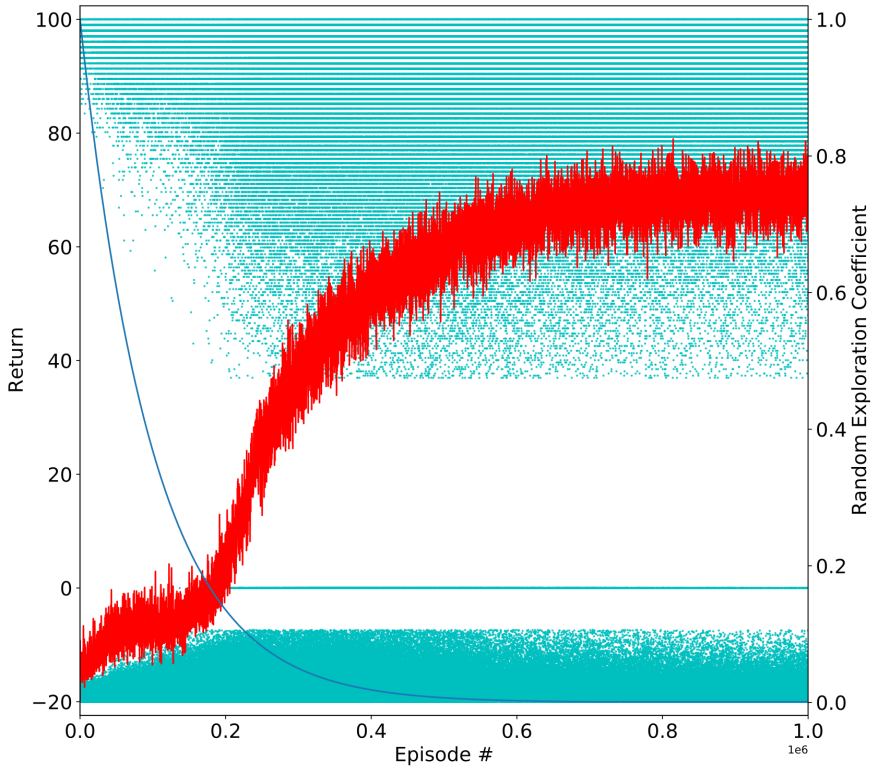


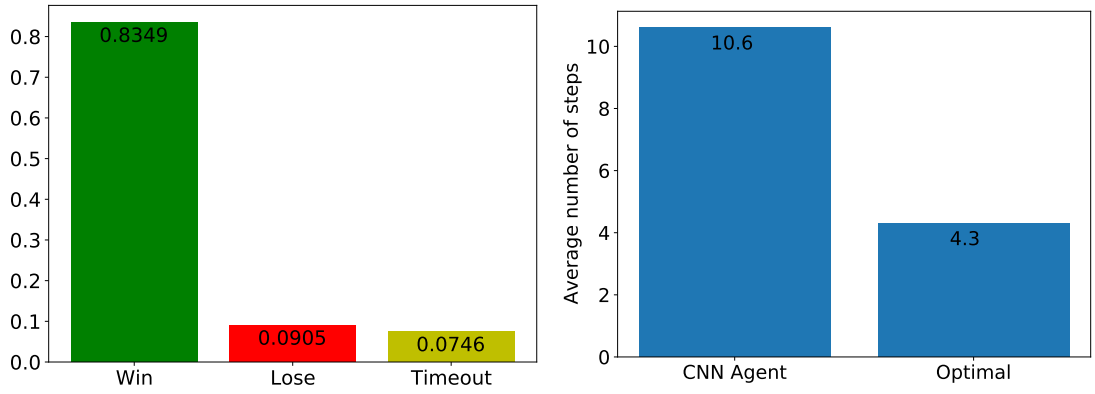
Figure 9.4: Training of the used convolutional neural network.

Figure 9.4 depicts the training procedure. What leaps out is that compared to the former training curves that display feature-based training, e.g., Figure 4.1, the training curve does not sharply grow as soon as the goal was reached frequently but increases slowly over time. More on, there is the need for significantly more training episodes, namely one to three million episodes compared to (mostly) 10000.

The training was conducted on a machine with a single Tesla V100 GPU and lasted approximately 44.85 hours.

9.4 Evaluation

To evaluate the performance of the trained convolutional agent, we again compare the number of wins, losses, and timeouts. We make use of 10000 evaluation runs, thereby randomly placing the goal line for every episode. As Figure 9.5a displays, the agent is successful in a vast majority of cases, but with about 9% of losing still has some potential for improvement. The same can be said about the number of steps taken, depicted in Figure 9.5b. On average, the agent takes more than twice as many steps as needed optimally to reach the goal.



(a) Win-lose-timeout.

(b) Average number of steps.

Figure 9.5: Evaluation.

The reason becomes vivid when looking at some of the evaluation runs. Most runs are comparable to the run pictured in Figure 9.6. The agent is either acting optimal, as depicted here, or at least near-optimal.

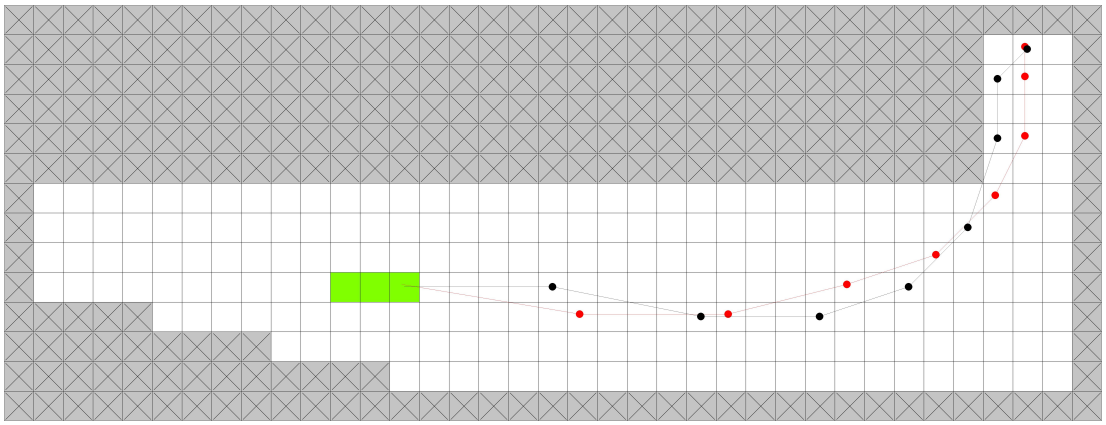
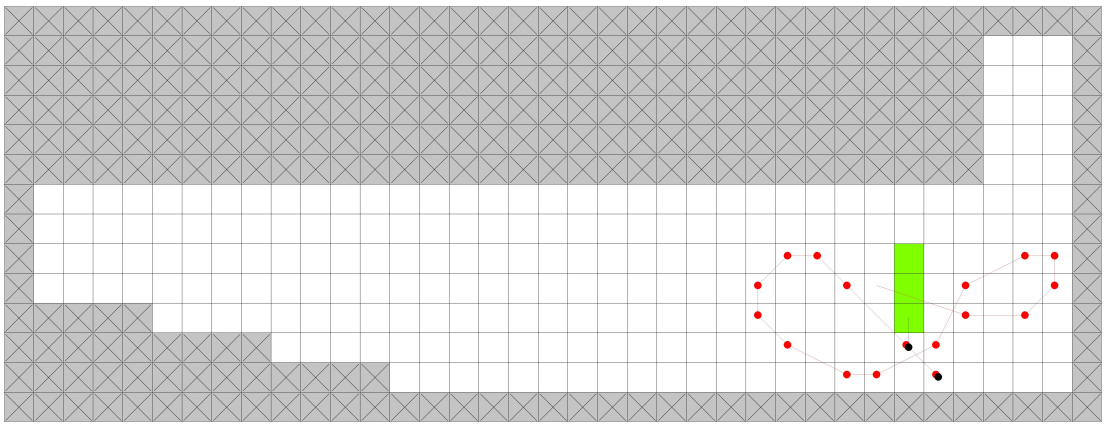
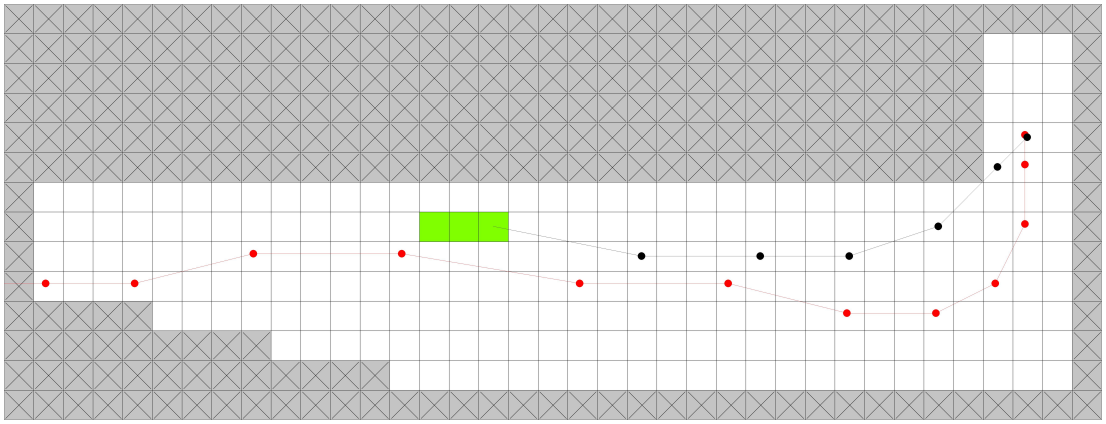


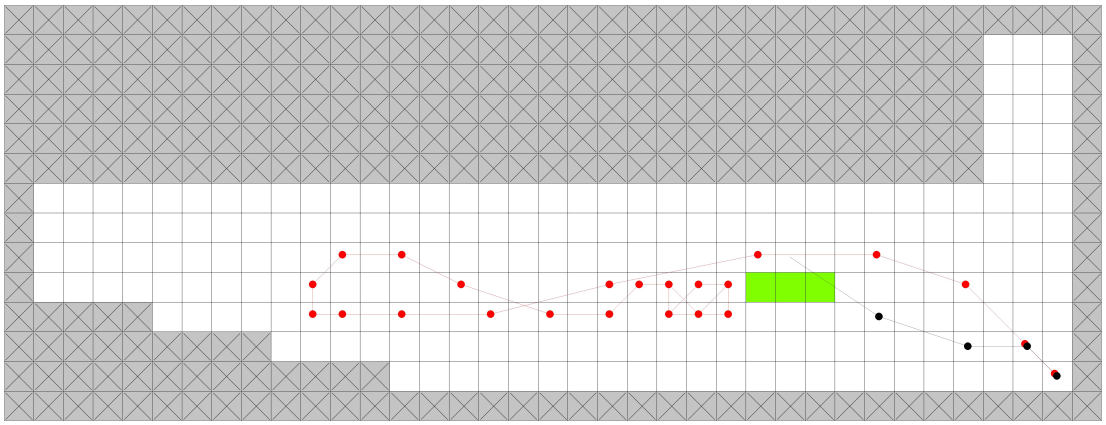
Figure 9.6: Successful evaluation run. The agent's path is colored in red, an optimal path is colored in black.



(a) Saved.



(b) Crashed.



(c) Timed Out.

Figure 9.7: Problematic evaluation runs. The agent's path is colored in red, an optimal path is colored in black.

In contrast, Figure 9.7 reveals a problem: the agent is repeatedly missing the goal line by one tile. In Figure 9.7a, the agent is still capable of saving himself. After missing the goal line even twice, the agent turns each time and finally can make it to the goal. The number of steps taken is vastly increased compared to the optimal agent. While the agent was still able to solve the task, Figure 9.7b depicts that this is not always possible: after the agent just closely missed the goal line, it is steering towards the wall with a very high velocity. There is no way of saving the situation; thus, a crash is unavoidable. Figure 9.7c depicts an evaluation that timed out. The agent once again missed the goal line by one tile, but here again, is able to make the turn. Unfortunately, it will then turn circles just one tile before making it to the goal until the time out is met.

We can conclude that there is still room for improving the convolution, as missing the goal line just by one tile seems to be the reason for both the crashes and timeouts as well as the increased number of steps when making it to the goal. As the agent reaches a performance close to the ones based on the vectorial state representation and further training a single agent lasts for days, this is out of the scope of this thesis.

9.5 Conclusion

This Chapter came one step closer to generally solving the Racetrack. We trained an agent based on a graphical state representation instead of a vectorial one. We first showed that this network can use the graphical input to approximate the so far used state features, which indicates that the network can be used to train action policies. The action policies need a strongly increased number of training runs. We receive a policy, which achieves a satisfying number of wins independent from the position of both the start and the goal line but also discover weak spots that leave room for improvement.

10 Conclusion and Future Work

10.1 Conclusion

This thesis kept track of several different aspects of deep learning.

The investigations and illustrations were made using the Racetrack, a benchmark known from the planning and learning community. We created a simulation-based tool of the Racetrack game that can be used to train and test different agents by using numerous different algorithms in several different variants of the Racetrack.

We have presented an extensive comparison between different learning approaches to solve the Racetrack benchmark. Even though we provided optimal decisions during imitation learning, the agents based on deep reinforcement learning outperform those of imitation learning in many aspects. Therefore, we afterward focused on deep reinforcement learning, in particular, deep Q-learning.

We have provided insights into the exploration-exploitation dilemma and depicted the problem of deep reinforcement learning agents with sparse rewards. We proposed a simple yet effective way to deal with that problem. By allowing the agent to start uniformly throughout the state space, the neural network is able to generalize and therefore to solve the task also from states that have been visited rarely, if so at all.

Further, we have gained insights into the influence of hyperparameters of deep Q-learning. We presented an extensive comparison for different reward structures for our worded goal (to mainly reach the goal without crashing and to secondly do it as fast as possible) as well as an extensive comparison for different discount factors. We found that even for small changes within the Racetrack, such as the map or the game variant we consider, different reward structures are optimal. Therefore we concluded that the design of the reward structure is a crucial part of every task and must be considered for each domain independently. For the influence of the discount factor, our results are somewhat more general. We found that for the Racetrack there is a certain range we can choose the discount factor from for the agent to solve our task. We showed that the properties of the resulting agents differ depending on the chosen discount factor. More on, we provided insights into the phenomenon of catastrophic forgetting when the discount factor is equal to/close to the value one.

Making use of our insights, we made a step towards general policy learning. We first showed that our selected convolutional neural network is capable of approximating our vectorial state representation, thus indicating that the same/a similar structure can be used for action policy learning. We then used such a CNN to create a policy that

is able to steer the car to an arbitrary placed goal line, based purely on the grayscale representation of the Racetrack. This resulted in a policy with a quality close to the more specific policies we had seen in this thesis so far.

10.2 Future Work

We believe that our observations about the characteristics of the different learning techniques carry over to other applications, particularly to more complex autonomous vehicle control algorithms. We plan to consider extensions of the Racetrack problem, which include further real-world characteristics of autonomous driving. We believe that to address the difficulties we observed with imitation learning, further investigations are necessary into the combination of expert data sets and reinforcement learning agents.

Additionally, other methods of guiding the agents to more promising solutions during training, such as reward shaping [16], will be examined as well as their influence on the characteristics of the final agent. Another interesting question for future work is whether multi-objective reinforcement learning can be used to adjust the agents' behavior in a fine-grained manner.

The influence of the chosen hyperparameters seems to be an under-researched problem. Many publications list the chosen hyperparameters without any justification about why or how these were chosen. To strengthen and generalize our investigations, future work needs to include investigations on other benchmarks.

Our attempt to generally solve the Racetrack resulted in a policy which has a performance close to the ones trained for a specific goal line based on the vectorial state representation. However, we determined the policy's weak spot: sometimes missing the goal closely and therefore either crashing, timing out, or having the need to turn and therefore take more steps to reach the goal. This suggests that there are improvements to make for the CNN structure, such that the performance of the general policy can compete with the ones trained for more specific settings.

Bibliography

[1] CNN Visualization Tool, 2021. http://alexlenail.me/NN-SVG/LeNet.html attended 17.03.2021.

[2] Forest Agostinelli, Stephen McAleer, Alexander Shmakov, and Pierre Baldi. Solving the Rubik's Cube with Deep Reinforcement Learning and Search. *Nature Machine Intelligence*, 1(8):356–363, 2019.

[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] Marc Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Remi Munos. Unifying Count-based Exploration and Intrinsic Motivation. In *Advances in neural information processing systems*, pages 1471–1479, 2016.

[5] Blai Bonet and Hector Geffner. GPT: A Tool for Planning with Uncertainty and Partial Information. In *Proceedings of the IJCAI Workshop on Planning with Uncertainty and Incomplete Information*, pages 82–87, 2001.

[6] Karl Cobbe, Christopher Hesse, Jacob Hilton, and John Schulman. Leveraging Procedural Generation to Benchmark Reinforcement Learning. *arXiv preprint arXiv:1912.01588*, 2019.

[7] Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Go-explore: a New Approach for Hard-exploration Problems. *arXiv preprint arXiv:1901.10995*, 2019.

[8] Timo P. Gros, Holger Hermanns, Jörg Hoffmann, Michaela Klauck, and Marcel Steinmetz. Deep Statistical Model Checking. In *Proceedings of the 40th International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE)*, pages 96–114. Springer, 2020.

[9] Timo P. Gros, Holger Hermanns, Jörg Hoffmann, Michaela Klauck, and Marcel Steinmetz. Models and Infrastructure used in "Deep Statistical Model Checking", 2020. to appear at `http://doi.org/10.5281/zenodo.3760098`.

[10] Timo P Gros, Daniel Höller, Jörg Hoffmann, and Verena Wolf. Tracking the Race Between Deep Reinforcement Learning and Imitation Learning. In *International Conference on Quantitative Evaluation of Systems*, pages 11–17. Springer, 2020.

[11] Timo P. Gros, Daniel Höller, Jörg Hoffmann, and Verena Wolf. Tracking the Race

Between Deep Reinforcement Learning and Imitation Learning – Extended Version. *arXiv preprint arXiv:2008.00766*, 2020.

[12] Marek Grzes. *Improving Exploration in Reinforcement Learning through Domain Knowledge and Parameter Analysis*. PhD thesis, University of York, 03 2010.

[13] Marek Grzes. Reward shaping in episodic reinforcement learning. 2017.

[14] Kshitij Judah, Alan P. Fern, Thomas G. Dietterich, and Prasad Tadepalli. Active Imitation Learning: Formal and Practical Reductions to I.I.D. Learning. *Journal of Machine Learning Research*, 15(120):4105–4143, 2014.

[15] Nikhil Ketkar. Introduction to pytorch. In *Deep Learning with Python*, pages 195–208. Springer, 2017.

[16] Adam Daniel Laud. Theory and Application of Reward Shaping in Reinforcement Learning. Technical report, 2004.

[17] Marlos C Machado, Sriram Srinivasan, and Michael Bowling. Domain-independent Optimistic Initialization for Reinforcement Learning. *arXiv preprint arXiv:1410.4604*, 2014.

[18] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. *arXiv preprint arXiv:1312.5602*, 2013.

[19] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level Control Through Deep Reinforcement Learning. *Nature*, 518(7540):529–533, 2015.

[20] Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy Invariance Under Reward Transformations: Theory and Application to Reward Shaping. In *ICML*, volume 99, pages 278–287, 1999.

[21] Keiron O'Shea and Ryan Nash. An Introduction to Convolutional Neural Networks. *CoRR*, abs/1511.08458, 2015.

[22] Pierre-Yves Oudeyer, Frdric Kaplan, and Verena V Hafner. Intrinsic Motivation Systems for Autonomous Mental Development. *IEEE transactions on evolutionary computation*, 11(2):265–286, 2007.

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel,

M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[24] Luis Enrique Pineda and Shlomo Zilberstein. Planning Under Uncertainty Using Reduced Models: Revisiting Determinization. In *Proceedings of the 24th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 217–225. AAAI Press, 2014.

[25] Stéphane Ross, Geoffrey J. Gordon, and Drew Bagnell. A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 15 of *JMLR Proceedings*, pages 627–635. JMLR.org, 2011.

[26] Stefan Schaal. Is Imitation Learning the Route to Humanoid Robots? *Trends in cognitive sciences*, 3(6):233–242, 1999.

[27] Jürgen Schmidhuber. A Possibility For Implementing Curiosity and Boredom in Model-building Neural Controllers. In *Proc. of the international conference on simulation of adaptive behavior: From animals to animats*, pages 222–227, 1991.

[28] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[29] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters Chess, Shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018.

[30] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the Game of Go Without Human Knowledge. *Nature*, 550:354–359, 2017.

[31] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning. The MIT Press, second edition, 2018.