Wissenschaftliche Arbeit im Rahmen des Studiums für das Lehramt an weiterführenden Schulen (Sekundarstufe 1 & 2) im Fach Informatik in der Fachrichtung Informatik der Fakultät MI der Universität des Saarlandes

Towards Undiscounted Q-Learning

Name, Vorname	Rasper, Giuliano Ricardo
E-Mail	giuliano.rasper@uni-saarland.de

Datum 25.10.2022

Erstgutachter	Univ.-Prof. Dr. Verena Wolf
Zweitgutachter	Univ.-Prof. Dr. Holger Hermanns

Selbstständigkeitserklärung:

Hiermit versichere ich, dass ich die vorliegende Wissenschaftliche Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen sind, sind unter Angabe der Quellen als Entlehnung kenntlich gemacht. Bei Zeichnungen, Skizzen oder Plänen sowie bildlichen und grafischen Darstellungen ist angegeben, wenn sie nach eigenen Angaben durch andere ausgeführt oder übernommen worden sind. Sollte ich Teile dieser Arbeit bereits für andere Prüfungen eingereicht haben, habe ich dies ebenfalls kenntlich gemacht. Die eingereichte elektronische Version der Arbeit stimmt mit der vorliegenden schriftlichen überein.

eingereicht am: 25.10.2022 Unterschrift: _____

Abstract

In reinforcement learning, discount factors weigh rewards, such that rewards received in the future are valued less than rewards received immediately. However, discounted methods fail when we aim to maximise the total reward received as well as consistently reach a defined goal state.

We formally define undiscounted performance measures and investigate why Q-learning and the undiscounted algorithms R-learning and H-learning fail at the task of learning a policy which is optimal in regard to such a performance measure.

Based on Q-learning we propose an approach called tuple learning which we show to be capable of learning a policy optimising an undiscounted performance measure.

We discover a problem we call Q-value explosion, which is closely related to undiscounted learning. In order to account for this, we investigate how to deal with the problem.

Contents

1 Introduction

In 1988, Sutton laid the foundation of a class of reinforcement learning algorithms, called temporal difference learning [20]. Three years later, Watkins et al. proposed a temporal difference method called Q-learning [24].

Reinforcement learning regained popularity by combining classical methods with neural networks, resulting in algorithms such as deep Q-learning [12]. A variation of games have been mastered using deep reinforcement learning methods [17, 16, 18, 2, 13], thus showing its potential.

In reinforcement learning, we want to maximise the rewards we receive when interacting with an environment [21]. The technique of discounting weighs rewards received in the future exponentially less than rewards received immediately. Discounting prevents infinite sums of rewards from diverging [21, 15] and encourages an agent to seek to reach a goal state quickly [7]. The compromise is that maximising the discounted reward does not necessarily coincide with the original reward structure, thus leading to a potential loss of rewards [15].

We explore why undiscounted algorithms (algorithms not using discounting) fail in solving environments with the objective of reaching a goal state. We propose an approach called tuple learning which optimises both, the total reward and reaching the goal at the same time. Throughout this thesis, we investigate Q-learning, the undiscounted methods R-learning and H-learning [15, 22], and the tuple learning approach. First, we evaluate these algorithms using regular reinforcement learning. Then, we transition into the deep reinforcement learning setting.

2 Reinforcement Learning

In reinforcement learning, we consider a learner called an *agent* which sequentially interacts with an environment.

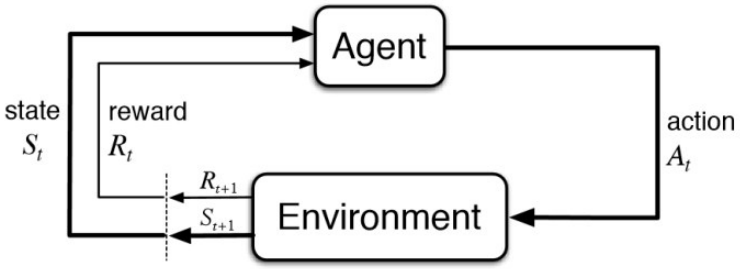


Figure 2.1: Agent-environment interactions [21]

Each interaction with the environment yields a numerical reward. The goal of an agent is to maximise these rewards according to some *performance measure* [8]. We call the information about the environment which might change while interacting with it the *state of the environment*. When interacting with an environment by performing *actions*, the agent can observe changes in the state of the environment [21].

2.1 Formalisation

We will now formalise the properties mentioned above so that we have a proper model to work with.

Definition 1 (Finite Markov Decision Process)**.** *Let $\langle S, A, T, R, F\rangle$ be a tuple consisting of a finite set of states S, a finite set of actions A, a transition function $T : S \times A \to S$, a reward function $R : S \times A \to \mathbb{R}$ and a set of final states F.*
Then we call $\langle S, A, T, R, F\rangle$ a finite deterministic markov decision process (MDP).

If we replace T by a function $P : S \times A \times S \to [0, 1] : (s, a, s') \mapsto p$ which yields the probability of resulting in state s' when performing an action a in state s, then $\langle S, A, P, R, F\rangle$ is called a nondeterministic MDP. If not all actions are applicable in each state, we also write $A(s)$ to denote the actions applicable in the state $s \in S$.

If not otherwise stated, when we use the component symbols from definition 1 without explicitly referring to an MDP, we assume they correspond to the same arbitrary MDP. Additionally, we denote the state, action and reward at time step t as random variables S_t, A_t, and R_t [21].

The behaviour of an agent is defined by a *policy*.

Definition 2 (Policy)**.** *A policy is a function $\pi : S \times A \to [0, 1]$ which maps a given state s and action a to the probability that a is taken in s. If for every $s \in S$ there is an $a \in A$ such that $\pi(s, a) = 1$, we call π deterministic. Otherwise, we call π stochastic [21].*

With

$$\mathbb{E}_\pi[\cdot] \tag{2.1}$$

we then denote the value of a random variable when the policy π is followed.

2.2 Discounting

The term discounting describes a method of assigning exponentially decaying value to the rewards received later on in a given sequence of rewards.

Definition 3 (Return)**.** *Let $R_t, R_{t+1}, R_{t+2}, ...$ be the series of rewards received after the t-th interaction with the environment. If there are only $i \in \mathbb{N}$ interactions with the environment, we set $R_j = 0$ for each $j > i$. The discounted sum of the rewards received since the t-th interaction*

$$G_t = \sum_{i=0}^{\infty} \gamma^i R_{t+i}$$

is called the return, where $\gamma \in (0, 1]$ is called discount factor.
We also write G_t^γ to denote the return for a given discount $\gamma \in (0, 1]$ [21].

We also say that a return is *undiscounted* if $\gamma = 1$. In this case, the return is equal to the sum of received rewards. With value functions, we describe the expected reward when following a given policy.

Definition 4 (Value Functions)**.** *Let $s \in S$ be a state and π be a policy for the corresponding finite MDP. [21].*
Then we call

$$v_\pi^\gamma : S \to \mathbb{R}, s \mapsto \mathbb{E}_\pi[G_t^\gamma \mid S_t = s]$$

the state-value function and

$$q_\pi^\gamma : S \times A \to \mathbb{R}, (s,a) \mapsto \mathbb{E}_\pi[G_t^\gamma \mid S_t = s, A_t = a]$$

the action-value function for the policy π.

The state-value function evaluates to the expected return when following a policy π from a given state $s \in S$. Meanwhile, the action-value function evaluates to the expected return when taking action $a \in A$ in a given state $s \in S$ and then following a policy π afterwards. Now we want to define what it means for a value function, and therefore its corresponding policy, to be good.

Definition 5 (Optimal Value Functions and Policies)**.** *We call state-value functions $v_*^\gamma(s)$ and action-value functions $q_*^\gamma(s,a)$ with the properties*

$$v_*^\gamma(s) \doteq \max_\pi v_\pi^\gamma(s)$$

$$q_*^\gamma(s,a) \doteq \max_\pi q_\pi^\gamma(s)$$

the optimal state-value function and optimal action-value function, respectively. Policies π with $v_*^\gamma = v_\pi^\gamma$ are called optimal policies [21].

Additionally, we consider the expected path-length as follows.

Definition 6 (Expected-Path-Length)**.** *Let π be a policy for some finite MDP. We call*

$$\#_\pi : S \to \mathbb{R}, s \mapsto \begin{cases} 1 + \sum_{a \in A} \pi(s,a) \cdot \sum_{s' \in S} P(s,a,s') \cdot \#_\pi(s') & , s \notin F \\ 0 & , s \in F \end{cases} \tag{2.2}$$

the expected path length from state s when following the policy π.

However, instead of optimising state-value functions, we take a more general approach by introducing *performance measures* [8] for policies, which allows us to define precise optimisation goals.

Definition 7 (Performance Measure)**.** *A performance measure $m : \pi \times S \to \mathbb{R}$ is any function which assigns a performance score to a policy π when starting from state s. We also write $m_\pi(s) := m(\pi, s)$.*

Note that a state-value function is a performance measure. Training an agent optimising a performance measure is equivalent to training an agent optimising a policy that is optimal according to a given performance measure.

Definition 8 (Optimal Policy)**.** *Given a performance measure f, a policy π is optimal in regard to f if and only if for every state $s \in S$,*

$$f_\pi(s) := \max_\pi f(s).$$

This definition is a generalisation for optimal policies in regard to the state-value function [21]. For our further work, we are interested in a special group of performance measures called *undiscounted performance measures.*

Definition 9 (Undiscounted Performance Measure)**.** *Let m be a performance measure. We say that m is an undiscounted performance measure if for any policy which is optimal in regard to m, it is also optimal regarding the state-value function v^1.*

This means that whenever we optimise an undiscounted performance measure f, we also optimise the undiscounted value function v, i.e., the set of optimal policies according to f is a subset of policies which are optimal according to v^1.

3 Racetrack Benchmark

For the evaluation of reinforcement learning algorithms, we will use the racetrack benchmark [7] which we introduce here.

The task of an agent is to navigate a car from the purple to the green finishing line on a given map (see Figure 3.1). If the car crashes into a wall (depicted in grey in the Figure), the task fails. If an agent can reach the goal, we also say that the agent can *solve* the racetrack.

Figure 3.1 shows the Barto-small and Barto-big map, both being solved by an agent.

The car is controlled by picking actions which accelerate the car in any of the directions shown in Figure 3.2. Alternatively, the agent may decide not to accelerate, leaving the velocity as is. When assigning a number to each of these actions we have an action space of $A = \{0, ..., 8\}$ with a total of nine actions.

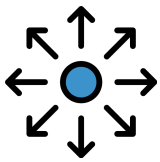


Figure 3.2: Directions in which the car may accelerate

By using the location and current velocity of the car, we can define the state of the environment as $(x, y, v_x, v_y) \in \mathbb{N}^4$ where x, y are the coordinates of the car and v_x, v_y are the velocities in the respective directions. Note that the position of the car as well as the velocities of the car are discrete. Therefore, the state space is finite. The location is restricted by the map itself, whereas the velocities are restricted because overly high velocities unavoidably lead to a crash, making it impossible to exceed a certain speed.

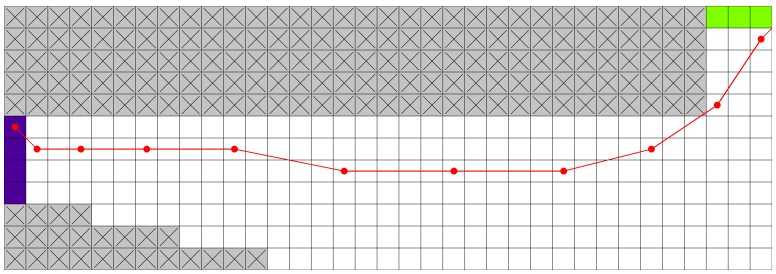
For deep learning, eleven additional features providing information about wall distances and goal distances are given [7]. However, the additional features do not increase the size of the state space.

The transition function T is given by the dynamics of the simulation of the environment, as described in the work of Gros [7].

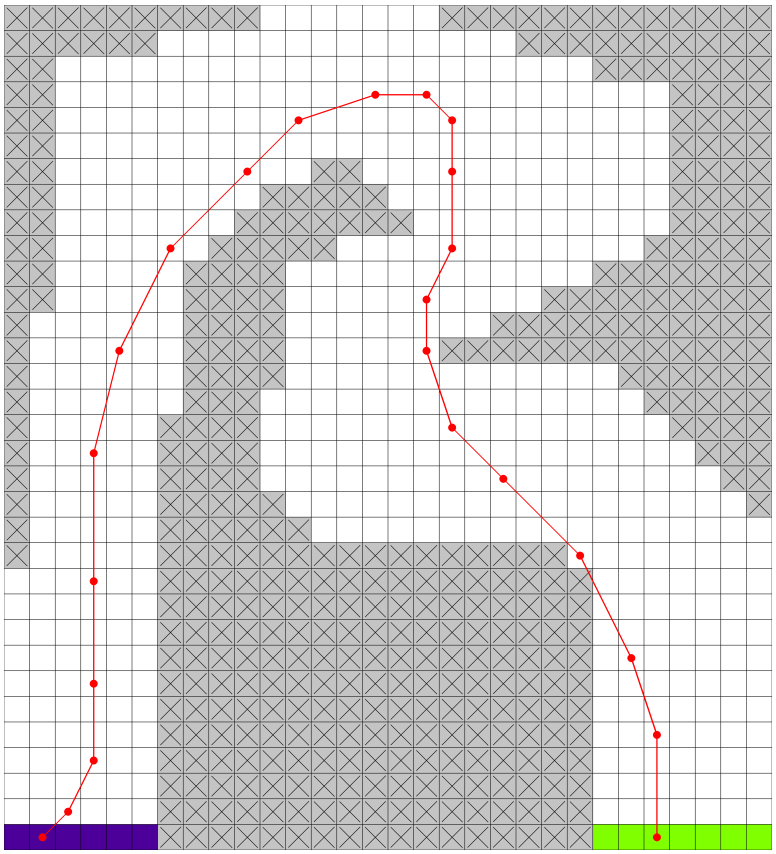
Non zero-rewards are only received when reaching a final state $f \in F = \{\bot, \top\}$ such that the rewards are defined by the following function

$$R(s, a, s') = \begin{cases} 100, & s' = \top \\ -20, & s' = \bot \\ 0, & \text{otherwise} \end{cases} \tag{3.1}$$

which yields the reward received when applying action a in state s and resulting in state s'. The state $\top$ marks a task as successful, whereas $\bot$ marks a task as failed. Then, the state space is $S \subset \mathbb{N}^4 \cup \{\bot, \top\}$. Thus, the MDP is given by $RT = \langle S, A, T, R, F \rangle$.



(a) barto-small



(b) barto-big

Figure 3.1: Racetrack benchmark: an agent (red line) solving the racetrack

4 Motivation and Problem Statement

In this chapter, we discuss problems of discounted and undiscounted methods. We then formally define desirable properties of a policy we want to learn and with that, we formulate the goal of this thesis. Afterwards, we explain the algorithm on which we base our approach.

4.1 The Problem of Discounted and Undiscounted Methods

The environment shown in the Figure (4.1) illustrates problems with both discounted and undiscounted learning.

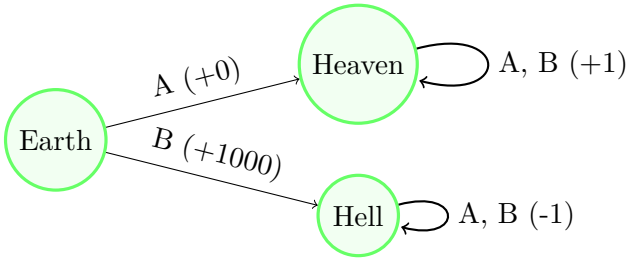


Figure 4.1: heaven-hell environment [15]

What we would want our agent to learn is to take action a resulting in the "heaven" state which yields a positive reward, no matter which action is chosen afterwards.
Now consider the policies $\pi_{Heaven}(s) = a$ taking action a and $\pi_{Hell}(s) = b$ taking action b regardless of the current state. We can observe the following:

- $v^{0.99}_{\pi_{Heaven}}(Earth) = 99 < 901 = v^{0.99}_{\pi_{Hell}}(Earth)$
- $v^{1}_{\pi_{Heaven}}(Earth) = +\infty > -\infty = v^{1}_{\pi_{Hell}}(Earth)$

It is notable that using $v^{0.99}$ as performance measure yields π_{Hell} as optimal performance measure, which is not desirable. The policy π_{Heaven} is optimal in regard to v^1. However, this performance measure is diverging on the given environment, which is also not a desirable property.

4.2 Problem Statement

In the following, we want to further investigate algorithms for environments with finite rewards and a non-empty set of final states.

Definition 10 (Finite Reward Goal Environment)**.** *We say that an environment is a finite reward environment if there is a $c \in \mathbb{R}$ such that for any policy π and $t \in \mathbb{N}$, we have $\mathbb{E}_\pi[G_t] \leq c \in \mathbb{R}$. We call an environment a finite reward goal environment if the set of final states is not empty.*

The racetrack environment satisfies the criteria of being a finite reward goal environment.

Even when restricting the environment to be a finite reward goal environment, we expect undiscounted methods to not yield a desirable policy. This expectation is based on the work of Gros we discuss in the next chapter. We want a policy maximising an undiscounted performance as well as the probability of reaching a final state with a low amount of steps. This intuition is defined by what we call undiscounted goal reaching performance measures.

Definition 11 (Undiscounted Goal Reaching Performance Measure)**.** *Let f be an undiscounted performance measure. We say that f is goal reaching exactly if it satisfies the property*

$$\#_\pi(s) > \#_\pi(s') \tag{4.1}$$

for each $s, s' \in S$ with

$$g_\pi(s) = g_\pi(s') \tag{4.2}$$

for some undiscounted performance measure g.

In case f is goal reaching, we call f an undiscounted goal reaching performance measure.

The Goal

The goal of this thesis is to construct an algorithms which optimises an undiscounted goal reaching performance measure and to then shift the algorithm to the deep reinforcement learning setting. That is, we want to (implicitly) approximate the performance score of said algorithm using a neural network.

Our approach is based on the Q-learning algorithm introduced in the next section.

4.3 Q-learning

The idea of Q-learning is to learn the optimal action-value function. The policy resulting from training a Q-learning agent takes those actions in a state which maximise the value of the approximated optimal action-value function for that state. The policy is given in its procedural form in Figure 4.2 [24].

```
def act(Q, state):
    return argmax(Q[state])
```

Figure 4.2: QL act method: yields optimal action according to the current policy

Here (Figure 4.2), Q is a table containing the values of the current estimation of q_*, also called Q-values. The function *argmax* returns a pseudo-random action for the given state for which the Q-value is maximal.

The agent is trained by improving Q using the update rule seen in Figure 4.3 [24] with observations obtained by exploring the environment.

```
def reinforce(Q, state, action, next_state, reward):
    q_val = reward
          + max(self.Q[next_state])
          * discount
    Q[state][action] = (1 - lr) * Q[state][action] + lr * q_val
```

Figure 4.3: QL reinforce method: training the agent with observations

The update rule minimises the temporal difference error (TD-error)

$$Q(s,a) - \gamma \cdot \max_{a* \in A}(Q(s', a^*)) \tag{4.3}$$

of the approximation $Q : S \times A \to \mathbb{R}$ with discount $\gamma \in (0, 1]$. The optimisation step is further controlled by the learning rate (lr) parameter, also denoted as α.

5 Related Work

In chapter 4, we state that even when restricting ourselves to finite reward goal environments, using undiscounted methods does not guarantee yielding a policy that follows an undiscounted goal reaching performance measure. We first discuss the work of Gros, leading us to this statement. Then we state two methods worth considering alongside with the approach we develop in the next chapter.

5.1 Deep Q-Learning on Racetrack

In Deep Q-Learning (DQL) we approximate Q-Values with a neural network instead of learning these explicitly [12].

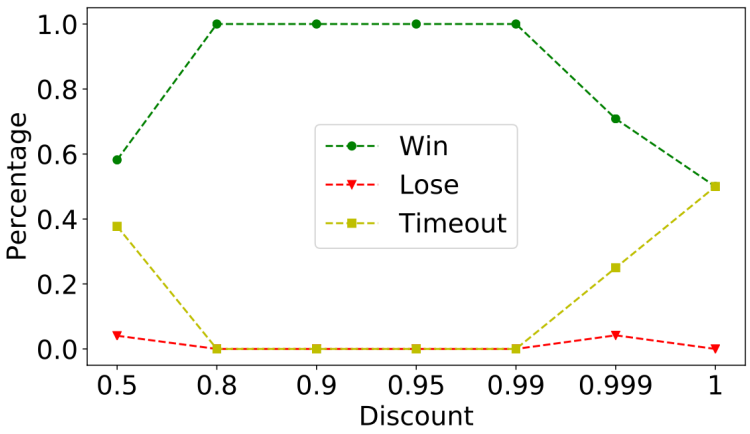


Figure 5.1: Win-lose-timeout evaluation of different discount factors on the Barto-small map [7]

In the work of Gros [7], it was experimentally shown (see 5.1) that an undiscounted DQL agent has a lower probability of reaching the goal of the racetrack environment than a discounted DQL agent.

It was reported that the Q-values for each state became approximately equal, as shown in the Figure (5.2) for velocities of zero. For other reasonable velocities, similar observations were reported.
That the maximal Q-Values for each state become equal can be explained by the fact

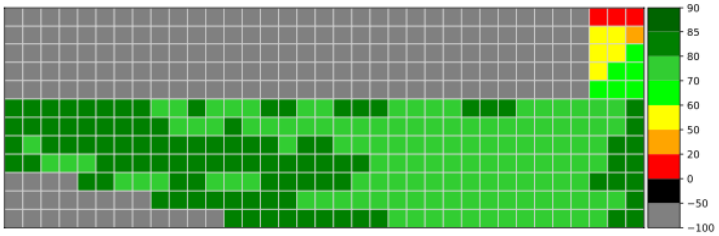


Figure 5.2: Heatmap for the estimated maximal Q-values with zero velocity [7]

that an agent may do anything as long as it does not lead to a crash. Due to not using discounting, the agent can still obtain the full reward.

As to be seen in Figure 5.2, it was reported that the Q-values of states further away from the goal were still higher than Q-values of states close to the goal. This suggests that the agent does not navigate the car randomly across the map, but rather develops a tendency to navigate the car to states with increased goal distance. Since this behaviour seems counterintuitive, a reinvestigation of the DQL algorithm is required.

5.2 H-learning and R-learning

The H-learning and R-learning algorithms learn without discounting [15, 22]. Both have been able to solve the environment presented in section 4.1 and similar environments.

Tadepalli and Ok describe R-learning as an undiscounted method that is model-free and can be considered a mixture of R-learning and ARTDP [22].

This begs the question if these algorithms optimise an undiscounted goal reaching performance measure. Therefore, we will include both algorithms in the evaluation of tabular methods.

6 Tuple Learning

In the following, we compose a technique which we call tuple learning (TL). In the next chapter, we will then use TL for constructing two algorithms which optimise an undiscounted goal reaching performance measure. The core idea of TL is to learn two action-value functions, whereas the importance of the value functions is implied by their position in the tuple.

6.1 Formalism

The formalism introduced in 2.1 assumes that the return is a singular real number, which is not suited for the idea of learning tuples. We therefore extend the definition of the return, the state-value function and the action-value function.

We define the return in the t-th time step with discount rates $\gamma_1, \gamma_2 \in \mathbb{R}$ as

$$G_t^{\langle\gamma_1,\gamma_2\rangle} \doteq \sum_{k=0}^{\infty} diag(\gamma_1, \gamma_2) \cdot (R_{t+k+1}, R_{t+k+1})^T \tag{6.1}$$

$$= (R_{t+1}, R_{t+1}) + diag(\gamma_1, \gamma_2) \cdot (G_{t+1}^{\langle\gamma_1,\gamma_2\rangle})^T \tag{6.2}$$

with $G_{t+1}^{\langle\gamma_1,\gamma_2\rangle} = 0$ in case of termination in the given time step t.

The state-value function and action-value function are defined analogous to before

$$v_\pi^{\langle\gamma_1,\gamma_2\rangle} : S \to \mathbb{R}^2, v \mapsto \mathbb{E}_\pi[G_t^{\langle\gamma_1,\gamma_2\rangle} \mid S_t = s] \tag{6.3}$$

$$q_\pi^{\langle\gamma_1,\gamma_2\rangle} : S \times A \to \mathbb{R}^2, (s, a) \mapsto \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] \tag{6.4}$$

by replacing G_t with $G_t^{\langle\gamma_1,\gamma_2\rangle}$.

Note that the former definitions of these terms are the special case of $\gamma_2 = 0$ in the notation just introduced.

6.2 Ordering Tuples

Using the state-value function for tuples, we can denote the expected value when following a policy. However, we are not able to compare those expectations with each

other since we have not defined a total ordering [5] for tuples.

6.2.1 Strict Tuple Order

Intuitively, one would strictly prioritise the primary component and only use the secondary component as a tiebreaker. The strict total ordering is given by the relation

$$R_L = \{((a,b),(c,d)) \in \mathbb{R}^2 \times \mathbb{R}^2 \mid a \leq c \wedge d \leq d \vee a < c\} \tag{6.5}$$

which is the the lexicographical ordering on the components. We also denote $(a,b) \in R_L$ as $a \leq b$.

For a set $S = \mathbb{R}^2 \times \mathbb{R}^2$, if there is an $m \in S$ such that for all $s \in S$ we have $s \leq m$, we denote $\max(S) \doteq m$.

6.2.2 Tolerant Tuple Order

In chapter 8, we will consider methods which do not learn state-action values explicitly, but approximate them instead. Thus, a strict ordering is not suitable here. We want to account for minor approximation errors by tolerating slight differences in the primary components, i.e., we want to consider primary components as equal when their relative difference is smaller than a tolerance $\delta \in [0, \infty)$.

A tolerance range

$$T_\delta^S = [(1-\delta) \cdot m_1, m_1] \times \mathbb{R} \subseteq \mathbb{R}^2 \tag{6.6}$$

with $(m_1, m_2) = \max(S)$ and a tolerance $\delta \in \mathbb{R}$ defines an error range in the primary component we are willing to accept.

We can construct a total order on a given set S by maintaining a strict ordering for elements in each of the two partitions obtained as follows. The two partitions are constructed by separating elements depending on whether they are within or not within the tolerance range T_δ^S, i.e.,

$$P_\delta^S = \{(a,b) \in \mathbb{R}^2 \times \mathbb{R}^2 \mid a \notin T_\delta^S \wedge b \in T_\delta^S \wedge a \leq b\} \tag{6.7}$$

$$Q_\delta^S = \{a, b \in \mathbb{R}^2 \times \mathbb{R}^2 \mid (a,b) \notin P_\delta^S \wedge (b,a) \notin P_\delta^S \wedge a \leq b\} \tag{6.8}$$

where $(m_1, m_2) = \max(S)$.

P_δ^S is a partial order with $(a,b) \in P_\delta^S$ if and only if b is within the tolerance range of $\max(S)$ but a is not. Q_δ^S is a partial order with $(a,b) \in Q_\delta^S$ if and only if both a and b are either within or not within the tolerance range of $\max(S)$ and $a \leq b$.

The union $R_\delta^S = P_\delta^S \cup Q_\delta^S$ is a total ordering on S where $(a, b) \in R_\delta^S$ is denoted as $a \leq_\alpha^S b$. For a set $S \subseteq \mathbb{R}^2 \times \mathbb{R}^2$, if there is an $m \in S$ such that for all $s \in S$ we have $s \leq_\delta^S m$, we denote $\max_\delta(S) \doteq m$.

We also write

$$\max_{i \in I}{}_\delta f := \max_\delta \{f(i) \mid i \in I\} \tag{6.9}$$

for a function $f : I \to \mathbb{R}^2$.

Note that for $\delta = 0$ both orderings are equivalent, i.e., $R = R_\delta^S$ for any set $S \subseteq \mathbb{R}^2$ and therefore $\max(S) = \max_\delta(S)$.

6.3 Optimal State-value Functions and Action-value Functions

With the introduced notation, we can define the optimal TL state-value function and TL action-value function

$$v_\delta^{\langle \gamma_1, \gamma_2 \rangle}(s) \doteq \max_\pi{}_\delta v_\pi^{\langle \gamma_1, \gamma_2 \rangle}(s) \tag{6.10}$$

$$q_\delta^{\langle \gamma_1, \gamma_2 \rangle}(s, a) \doteq \max_\pi{}_\delta q_\pi^{\langle \gamma_1, \gamma_2 \rangle}(s, a) \tag{6.11}$$

as the state-value function and action-value function of the policy maximising the return. With π_δ we denote an optimal policy, i.e., $v_\delta^{\langle \gamma_1, \gamma_2 \rangle} = \mathbb{E}_{\pi_\delta}[G_t^{\langle \gamma_1, \gamma_2 \rangle} \mid S_t = s]$.

Note that we can rewrite

$$q_\delta^{\langle \gamma_1, \gamma_2 \rangle}(s, a) = \mathbb{E}[R_{t+1} + (\gamma_1, \gamma_2) \cdot (v_\delta^{\langle \gamma_1, \gamma_2 \rangle}(s, a))^T \mid S_t = s, A_t = a] \tag{6.12}$$

by applying the definitions of the action-value function, the return and the state-value function.

6.4 Deducing an Update Rule

In order to iteratively find the optimal action-value function, we need to find a function which

1. has $q_\delta^{\langle \gamma_1, \gamma_2 \rangle}$ as a fixpoint

2. is a contraction

such that we can use a fixpoint iteration to find $q_\delta^{\langle \gamma_1, \gamma_2 \rangle}$ [25].

Regular state-value functions and action-value functions can be rewritten in such a way that they recursively depend on themselves without referring to any specific policy. Those equations are called the Bellman optimality equations and directly imply the Bellman operator which has $q_*(s)$ as a fixpoint [21].

For TL state-value functions, we can also deduce a recursive definition for TL state-value functions without referring to any policy

$$
\begin{aligned}
v_\delta^{\langle\gamma_1,\gamma_2\rangle}(s) &= \max_{a\in A(s)} q_{\pi_\delta}^{\langle\gamma_1,\gamma_2\rangle}(s,a) \\
&= \max_{a\in A(s)} \mathbb{E}_{\pi_\delta}[(G_t^{\gamma_1}, G_t^{\gamma_2}) \mid S_t = s, A_t = a] \\
&= \max_{a\in A(s)} \mathbb{E}_{\pi_\delta}[(R_{t+1} + \gamma_1 G_{t+1}^{\gamma_1}, R_{t+1} + \gamma_2 G_{t+1}^{\gamma_2}) \mid S_t = s, A_t = a] \\
&= \max_{a\in A(s)} \mathbb{E}_{\pi_\delta}[(R_{t+1} + \gamma_1 G_{t+1}^{\gamma_1}, R_{t+1} + \gamma_2 G_{t+1}^{\gamma_2}) \mid S_t = s, A_t = a] \\
&= \max_{a\in A(s)} \mathbb{E}_{\pi_\delta}[(R_{t+1}, R_{t+1}) + diag(\gamma_1,\gamma_2)\cdot(G_{t+1}^{\gamma_1}, G_{t+1}^{\gamma_2})^T \mid S_t = s, A_t = a] \\
&= \max_{a\in A(s)} \mathbb{E}_{\pi_\delta}[(R_{t+1}, R_{t+1}) + diag(\gamma_1,\gamma_2)\cdot v_{\pi_\delta}^{\langle\gamma_1,\gamma_2\rangle}(S_{t+1})^T \mid S_t = s, A_t = a] \qquad (6.13) \\
&= \max_{a\in A(s)} \sum_{s',r} p(s', r \mid s, a)[r + diag(\gamma_1,\gamma_2)\cdot v_\delta^{\langle\gamma_1,\gamma_2\rangle}(s')^T] \qquad (6.14)
\end{aligned}
$$

such that we obtain an optimality equation for TL state-value functions with two alternate forms (6.13, 6.14).

By applying the identity 6.12, we find a recursive definition for TL action-value functions

$$
\begin{aligned}
q_\delta^{\langle\gamma_1,\gamma_2\rangle}(s,a) &= \mathbb{E}[R_{t+1} + diag(\gamma_1,\gamma_2)\cdot \max_{a'}{}_\delta\, q_\delta^{\langle\gamma_1,\gamma_2\rangle}(S_{t+1}, a')^T \mid S_t = s, A_t = a] \qquad (6.15) \\
&= \sum_{s',r} p(s', r \mid s, a)[r + diag(\gamma_1,\gamma_2)\cdot \max_{a'\in A(s')_\delta} q_\delta^{\langle\gamma_1,\gamma_2\rangle}(s', a')^T] \qquad (6.16)
\end{aligned}
$$

which we call the TL action-value function optimality equations.

We define the right-hand side of the equation (6.16)

$$
(\mathbf{T}q_\delta)(s,a) = \sum_{s',r} p(s', r \mid s, a)[r + diag(\gamma_1,\gamma_2)\cdot \max_{a'\in A(s')_\delta} q_\delta(s', a')^T] \qquad (6.17)
$$

as the TL operator $\mathbf{T}$, mapping a action-value function to another action-value function. Note that according to equation 6.16, the operator $\mathbf{T}$ has $q_*^{\langle\gamma_1,\gamma_2\rangle}$ as a fixpoint.

Thus, an iterative update rule for obtaining the TL action-value function is given by

$$q_{\pi_{k+1}}^{\langle\gamma_1,\gamma_2\rangle}(s,a) \leftarrow \mathbf{T}q_{\pi_k}^{\langle\gamma_1,\gamma_2\rangle}(s,a), \tag{6.18}$$

which we call the TL update rule.

6.4.1 Convergence

Establishing general convergence guarantees for TL is a task which is not easily done when having to consider tolerance factors, if possible at all. We now focus on a property which gives us a convergence guarantee if met. If the first component of the tuple converges, the convergence of the second component follows.

The core idea of the following proof is to reduce the convergence of TL to the convergence of Q-learning (theorem 1) that was shown by Watkins [24].

Theorem 1. *Given bounded rewards* $|r_n| \leq \mathcal{R}$*, learning rates* $0 \leq \alpha_n < 1$*, and*

$$\sum_{i=1}^{\infty} \alpha_{n^i(x,a)} = \infty, \sum_{i=1}^{\infty} [\alpha_{n^i(x,a)}]^2 < \infty, \forall x, a$$

then $Q_n(x,a) \to Q^*(x.a)$ *as* $n \to \infty$*,* $\forall x, a$*, with probability 1.*

Essentially, the theorem states that the convergence of the Q-learning algorithm is guaranteed if the rewards are bounded and the learning rate converges towards zero neither too fast nor too slowly.

We now deduce said convergence property of TL.

Lemma 1. *Let*

1. $\gamma_2 \in (0,1)$.
2. $(a_k, b_k) = q_{\pi_k}^{\langle\gamma_1,\gamma_2\rangle}$ *the action-value function after the k-th iteration of the TL algorithm.*
3. *the prerequisites of theorem 1 be satisfied.*

If every $(s,a) \in S \times A$ *in the TL algorithm is sampled infinitely often and* $\lim_{k\to\infty} a_k = a_*$*, then* $\lim_{k\to\infty} b_k = b_*$ *with probability 1.*

Proof. Note that for each state $s \in S$, a_k induces a set $A_k(s) \subseteq A(s)$ of actions which yield expected values within the tolerance range when taken. Since $\lim_{k\to\infty} a_k = a_*$, there must be $n_0 \in \mathbb{N}$ such that there is an $A'(s) \subseteq A_k(s) = A'(s)$ for all $s \in S$ and $k \geq n_0$. It follows that for $k \geq n_0$ the TL update rule in the second component degenerates to

$$\begin{aligned}\kappa_2(q_{\pi_{k+1}}^{\langle\gamma_1,\gamma_2\rangle})(s,a) &\leftarrow \kappa_2(\mathbf{T}q_{\pi_k}^{\langle\gamma_1,\gamma_2\rangle})(s,a) \\ &\overset{k\geq n_0}{=} \sum_{s',r} p(s',r \mid s,a)[r + \gamma_2 \max_{a'\in A'(s')} \kappa_2(q_{\pi_k})(s',a')])(s,a),\end{aligned}$$

i.e., the second component is updated using the update rule of Q-learning, which according to theorem 1 converges towards an optimal state-value function with probability 1. Here κ_i is the function which maps to the i-th value of a tuple or the i-th component function, respectively.

Since for $k \geq n_0$ we have A' fixed, by definition of R_δ it follows that $q_* = \kappa_2(q_*^{\langle\gamma_1,\gamma_2\rangle})$, where q_* is the optimal action-value function on the reduced MDP with action space A'. Therefore, $\lim_{k\to\infty} b_k = b_*$.

■

Lemma 1 also holds for $\gamma_2 = 1$ if the underlying MDP has absorbing goal states. The additional precondition of absorbing goal states is required to guarantee Q-learning convergence when $\gamma_2 = 1$ [24].

Note that we do not make a statement regarding the convergence when using an approximator for the action-value function instead (see chapter 8).

We conclude that if we can guarantee convergence of the first component, TL is guaranteed to converge with sufficiently many samples for each state-action pair.

6.5 Algorithms

When designing an algorithm utilising the TL update rule 6.18, we want to keep the algorithm as close to the theoretical foundation as possible. The most notable practical restriction is that we cannot sample indefinitely often. Instead, we have to train for a finite number of episodes. Therefore, we also cannot use a series of learning rates. We will fix the learning rate instead.

Due to these practical limitations, the probability of converging towards the optimal

policy is not 1 in practice. However, with adjusting the learning rates and increasing the number of training episodes, we can achieve a convergence probability which is arbitrarily close to 1 (lemma 1).

6.5.1 Tuple Learning Core Algorithm

The key requisite we want to ensure for applying lemma 1 is to guarantee the convergence of the first component. We ensure this by learning the value of each component with a separate agent. Learning the first component with a **primary agent** using Q-learning guarantees convergence in the first component. The second component is learned by a **secondary agent** which has a reduced action space that is induced by the primary agent. Convergence of the learning process is then guaranteed according to lemma 1.

The optimal action according to the current policy is then implied by tolerant tuple order and given in its procedural form in 6.1.

```
def act(primary_agent, secondary_agent, state, tolerance):
    action = None
    best_actions = primary_agent.best_actions(tolerance)
    if len(best_actions) == 1:
        action = primary_agent.act(state)
    else:
        action = secondary_agent.act(state, restriction: best_actions)
    return action
```

Figure 6.1: TL act method: yields optimal action according to the current policy

Both subagents are trained with the same observations as shown in 6.2.

```
def reinforce(Q, discount, state, action, next_state, reward, tolerance)
    :
  target_action = act(primary_agent, secondary_agent, next_next,
    tolerance)
    q_val = reward
          + max(self.Q[next_state][target_action]
          * discount
    Q[state][action] = (1 - lr) * Q[state][action] + lr * q_val
```

Figure 6.2: TL reinforce method: training the primary agent / secondary agent with observations

Here, the target value is induced by the action which is optimal in regard to the currently learned policy.

6.5.2 Goal Reaching Tuple Learning Algorithms

In the following chapters of the thesis, we investigate two variants which are special cases of the core TL algorithm. Since we are interested in algorithms which optimise an undiscounted goal reaching performance measure, both variants will have a Q-learning agent with $\gamma = 1$ as primary agent. Even though TL is a broader term, when referring to TL, from now on we refer to the two methods introduced here.

QQ-Learning describes the approach where the secondary agent is a discounted Q-learning agent with a discount factor of $\gamma \in (0, 1)$.

QG-Learning describes the approach where the secondary agent learns the expected goal step distance. To stick with maximising a performance measure, we optimise the sign-reversed #-function, which is an equivalent task.

Learning the goal step distance is equivalent to learning a transformed reward function such that $v^*(s) = -\#_{\pi^*}(s)$, in an undiscounted setting.

This transformation is given by

$$R'(s, a, s') = \begin{cases} -\infty, & s' \in F \wedge \mathit{Goal}(s) \\ -1, & \text{otherwise} \end{cases} \tag{6.19}$$

which introduces a step cost of 1. We consider a final state as goal state if a goal condition $\mathit{Goal} : S \to \{\top, \bot\}$ is satisfied. For a final state which is not a goal state, there is no way that a goal state can be reached. Thus the goal distance being infinite. In practice, we will use a sufficiently high punishment instead of $-\infty$.

Since the optimal state-value of the transformed reward function equals the sign-reversed goal distance, minimising the goal distance is equivalent to choosing the action a with $q_*(s, a) = v_*(s)$.

6.6 Alignment of the Algorithms with the Goal

Now, we shortly argue that both methods described in section 6.5.2 align with the goal of the thesis to optimise an undiscounted goal reaching performance measure. In the following, let $m_Q(s)$ be a performance measure for QQL and $m_G(s)$ a performance measure for QGL. Here we consider tolerance factors δ which are sufficiently close to 0, since we cannot make any guarantees otherwise.

By construction of both variants, we learn the undiscounted state-value function with

the primary agent. Thus, for δ sufficiently close to 0, we only take actions which are optimal with respect to v^1, thus by definition we optimise an undiscounted performance measure.

It remains to be shown that QGL and QQL are both goal reaching. Since the secondary agent of QGL maximises $-\#_\pi(s)$, it follows directly that for every $s, s' \in S$ with $v^1(s) = v^1(s')$, we have

$$m_G(s) < m_G(s') \Leftrightarrow -\#_\pi(s) < -\#_\pi(s') \Leftrightarrow \#_\pi(s) > \#_\pi(s'),$$

i.e., $m_G(s)$ is goal reaching.

The secondary agent of QQL maximises $v^1(s) \cdot {\gamma_2}^{\#(s)}$. Therefore, if there are s, s' such that $v^1(s) = v^1(s')$ and $\#(s) > \#(s')$, it follows that ${\gamma_2}^{\#(s)} < {\gamma_2}^{\#(s')}$ and thus $m_Q(s) < m_Q(s')$, i.e., m_Q is goal reaching.

7 Learning in the Tabular Setting

As seen in chapter 5.1, the problem of learning from undiscounted rewards was observed in the deep reinforcement learning setting, namely using deep Q-Learning.

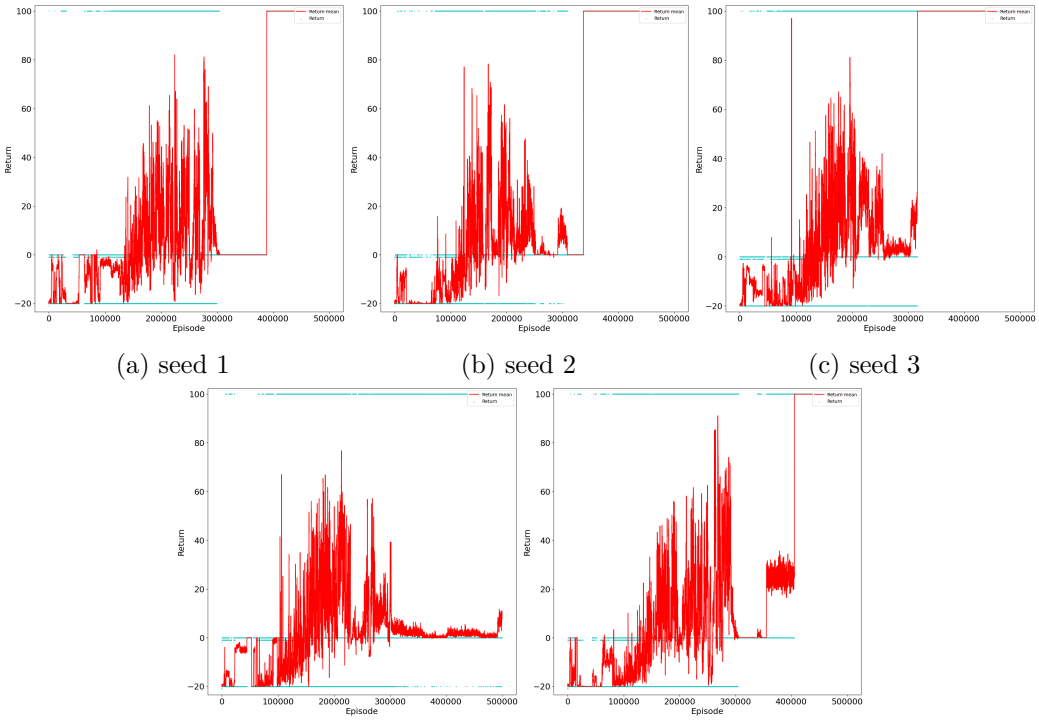
Here, we investigate the performance of Q-learning and undiscounted learning methods in the tabular setting. That means we explicitly learn performance measures by saving them into a table [21] instead of using a neural network as approximator.
For that, we consider the H-learning, R-learning and Q-learning algorithm as well as one of our composed approaches, namely QQL.

7.1 Results

The algorithms were trained and evaluated using the racetrack benchmark on the Barto-small map. Note that training and evaluation were completely separated, i.e., after each training episode an evaluation was performed based on the currently learned policy. Exploration was done ϵ-greedy [21] with exponential decay. We used the same hyperparameter configuration for ϵ-greedy exploration and the learning rate parameter as described in the upcoming chapter 9.

7.1.1 H-learning and R-learning

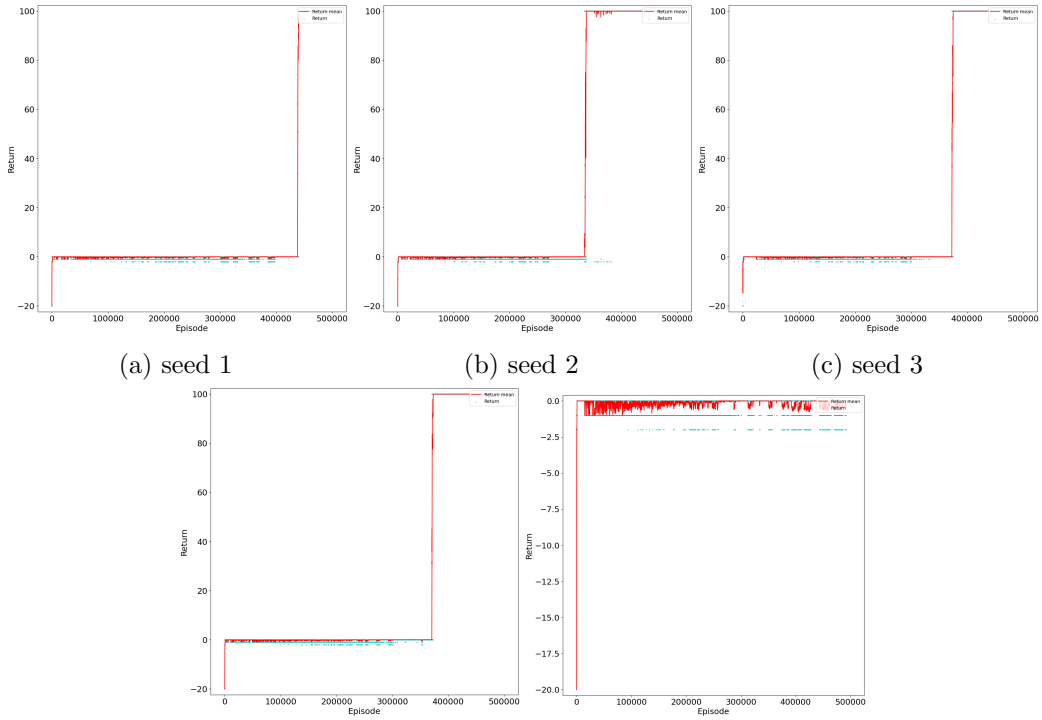
Both the learning curves of R-learning and H-learning show that 4 out of 5 agents were able to learn a policy which was able to reach the goal, whereas the 5th agent was able to learn crash avoiding behaviour.



(a) seed 1 (b) seed 2 (c) seed 3

(d) seed 4 (e) seed 5

Figure 7.1: Learning curves of R-learning for the Barto-small map

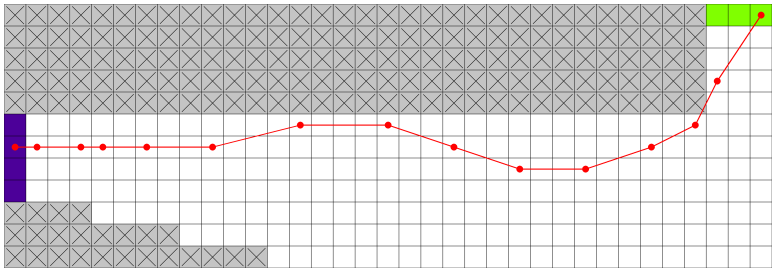


(a) seed 1 (b) seed 2 (c) seed 3

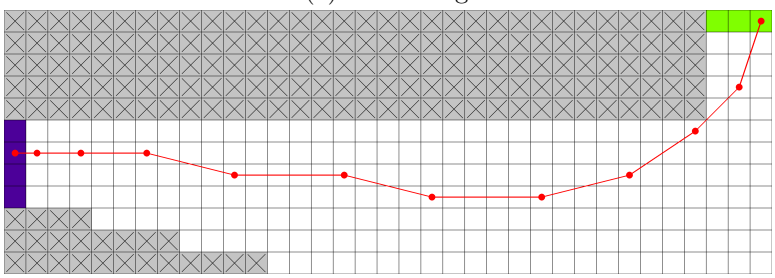
(d) seed 4 (e) seed 5

Figure 7.2: Learning curves of H-learning for the Barto-small map

The observation is verified by taking a look at the final paths (7.3) taken by agents of each algorithm.



(a) R-learning



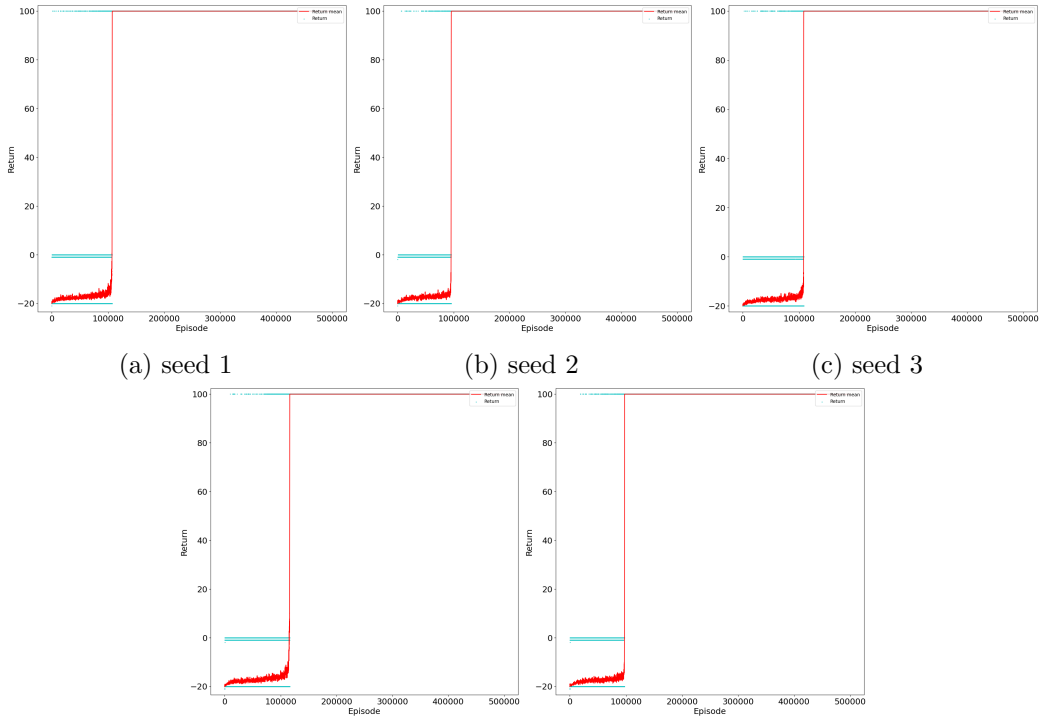
(b) H-learning

Figure 7.3: Paths after the final learning step on the Barto-small map

This begs the question on why both methods are successful despite being undiscounted methods. Here the answer lies in the details. Even though Schwartz calls R-learning an undiscounted method and H-learning builds on that idea, the optimisation goal does not coincide with what we call undiscounted performance measures. R-learning and H-learning both optimise the average undiscounted return per step [15, 22], thus taking longer paths is punished. However, the value of rewards received immediately does not differ from rewards received in the future. But since optimising the average discounted reward does not result in an undiscounted goal reaching policy, we will no longer consider both of these methods for the goal of this thesis.

7.1.2 Q-learning and Tuple Learning

For Q-learning, we observed that the agent was able to learn to solve the racetrack. In contrast to R-learning and H-learning, the agent learned to reach the goal much earlier.



(a) seed 1 (b) seed 2 (c) seed 3

(d) seed 4 (e) seed 5

Figure 7.4: Learning curves of Q-learning for the Barto-small

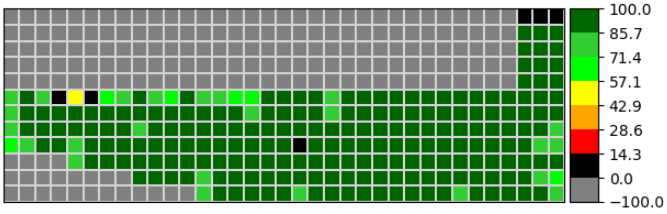
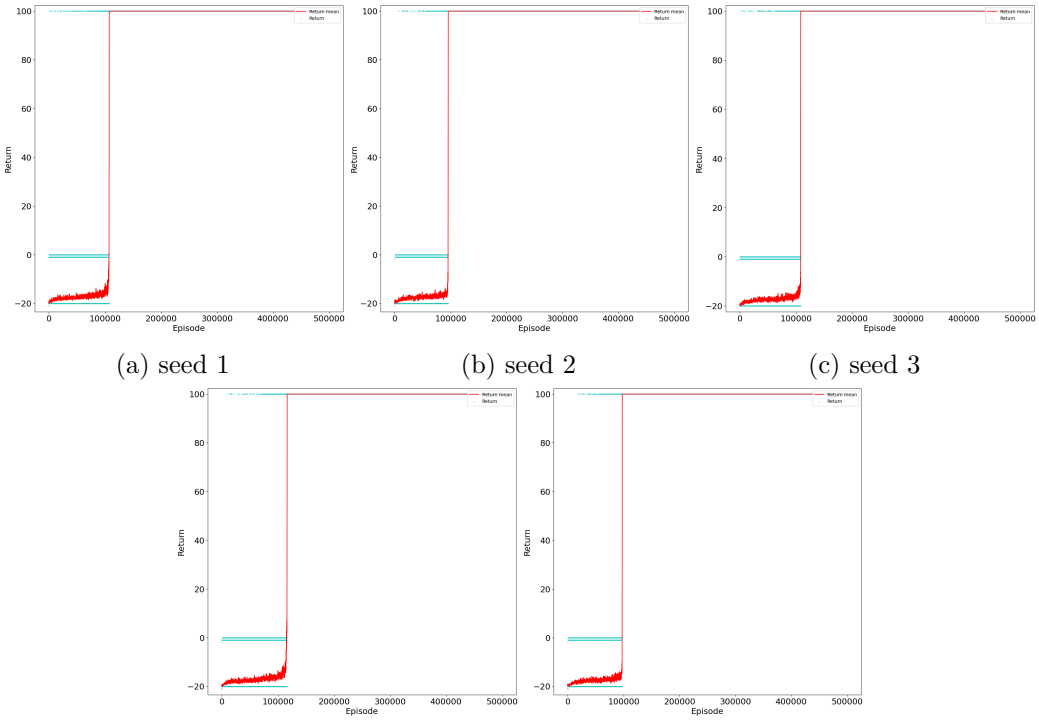


Figure 7.5: Heatmap of Q-values after the final learning step, weighted by visits for all velocities

Looking at the Q-values learned, it seems like they verify the expectation that the Q-values become equal (7.5).



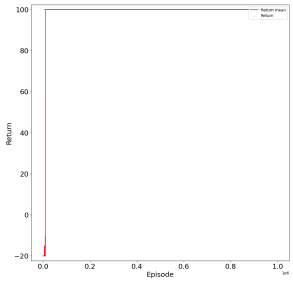
(a) seed 1 (b) seed 2 (c) seed 3

(d) seed 4 (e) seed 5

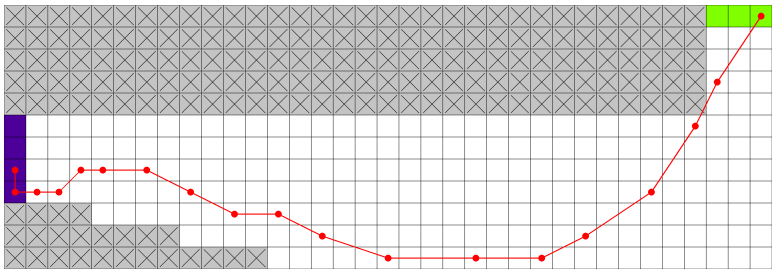
Figure 7.6: Learning curves of QQL for Barto-small map

However, when looking at the exact Q-values learned, it becomes obvious why the agents were successful despite our expectation that they would fail. Due to the way the update rule works 4.3, it is true that the Q-values become equal in the limit, as proven by Watkins [24]. But we are not sampling indefinitely often. We observe that the difference between the Q-values and the true expected return is smaller when moving closer to the goal. Therefore, the agent does not become uninformed since with finite samples, Q-values might only become equal due to limited accuracy of floating-point operations [11].

Consequentially, it is not surprising that TL is successful in the tabular case, since the secondary agent is not needed to achieve an undiscounted goal reaching policy. We can verify this by looking at the learning curves of QQL (7.7) and therefore refrain from evaluating QGL in the tabular setting.



(a) evaluation return



(b) final path

Figure 7.7: Learning curves of QQL and the path taken after the final learning step

7.2 Summarising Tabular Methods

In this chapter, we determined that there are discrepancies between what is considered an undiscounted method by Schwartz and what we consider undiscounted performance measures. We therefore concluded that R-learning and H-learning do not fulfil our criteria of being an undiscounted performance measure and are therefore not considered from now on.

We found that Q-learning was able to solve the racetrack and consequentially the TL method QQL also succeeded. However, since this success is strictly bound to the way tabular learning works, we cannot draw direct conclusions on the performance of corresponding methods in the deep reinforcement learning setting, which we will consider next.

8 Deep Tuple Learning

Now we transfer the tuple learning algorithms we composed to the deep reinforcement learning setting. We call the induced deep tuple learning (DTL) variants deep QQ-learning (DQQL) and deep QG-learning (DQGL), respectively. Instead of learning the optimal action-values explicitly, as in the tabular setting, we approximate the optimal action-values by training a neural network. By doing so, we utilise the ability of neural networks to generalise [19]. That is, even for unseen states, an approximation of the Q-value can be given and therefore can be used to determine the actions to perform.

Since TL requires two agents, we train two neural networks, each approximating the value function of one agent. Given the weights Θ of a neural network, we call

$$N_\Theta : S \times A, (s, a) \to \mathbb{R} \tag{8.1}$$

the corresponding approximated action-value function.

Since we want to minimise the TD-error in each state, we will define a loss function [4] as the metric to be optimised. We use stochastic optimisation using the Adam algorithm [9] as proposed by Andrychowicz [3], paired with experience replay as proposed by Mnih et al. [12], for training the neural networks. As in the deep Q-learning (DQL) paper, we use a target network for estimating the TD-error. Originally, the weights of the target network in the i-th iteration Θ_i^- were fixed to the weights Θ_{i-1} of the previous iteration [12]. However, instead of using the weights from a former iteration, we perform a soft update [10] on the target weights, i.e.,

$$\Theta_i^- = \begin{cases} (1-\alpha)\Theta_{i-1}^- + \alpha\Theta_i & , i > 0 \\ \Theta_I^- & , i = 0, \end{cases} \tag{8.2}$$

where Θ_I^- are the initialisation weights of the target network.

For the implementation of these concepts, we mainly use the libraries Pytorch [14] and RLMate [6].

8.1 Network Architecture

For the primary agent, we use a standard DQL network architecture with identical structure for the network and target network as seen in Figure 8.1.

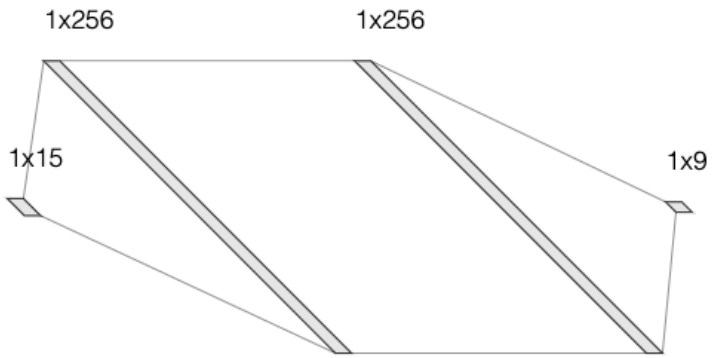


Figure 8.1: Network structure of the network approximating the action-value function [1]

Here we use an input layer of the same size as our actions space, two hidden layers as well as an output layer [26] representing the Q-value of each action.

For the secondary agent, we use the same network structure. Note that one could investigate the option of using shared weights for the primary agent and secondary agent to improve training time. However, for this proof of concept, we decided against investigating different possible network structures.

8.2 Formalising the Loss

Using the $argmax_\delta$ function we define as

$$\underset{a \in A}{argmax_\delta} : \{f \mid dom(f) = A \wedge im(f) \subseteq \mathbb{R}\} \to A, f \mapsto \{a \in A \mid f(a) \in T_\delta^{Im(f)}\}, \quad (8.3)$$

we can describe the actions from which the secondary agent can choose. $\delta \in \mathbb{R}$ is the tolerance from which a value may deviate and still be considered as optimal. This prevents the exclusion of actions due to an approximation error of the neural network.

Given the weights Θ, we denote

$$A_\delta^*(s, \Theta) \doteq \underset{a}{argmax_\delta}\, \mathbb{E}[N_\Theta(S_{t+1}, A_t) \mid A_t = a \wedge S_t = s] \quad (8.4)$$

for the set of actions which maximise the expected future reward in state s with tolerance δ.

According to our network architecture, we define two loss functions: one for the DQN used for the primary agent, one for the secondary agent.

By construction of the TL core algorithm (6.5.1), the mean squared error (MSE) loss [4] of the primary agent's network is equivalent to the MSE loss of a DQN and is given as

$$L(\Theta) = \mathbb{E}[(N_\Theta(S_t, A_t) - y(S_t, A_t; \Theta^-))^2] \tag{8.5}$$

with the target y given as

$$y(s, a; \Theta) = \mathbb{E}[R_{t+1} + \max_{a^* \in A} N_\Theta(S_{t+1}, a^*) \mid S_t = s, A_t = a] \tag{8.6}$$

for a given set Θ of weights.

For calculating the target values with discount $\gamma \in (0, 1]$ of the secondary agent

$$y_S(s, a; \Theta_1) = \mathbb{E}[R_t + \gamma \cdot \max_{a^* \in A^*_\delta(s, \Theta_1)} N_{\Theta_2^-}(S_{t+1}, a^*) \mid A_t = a \wedge S_t = s], \tag{8.7}$$

we must only consider actions $a \in A^*_\delta(s, \Theta_1)$, i.e., the corresponding undiscounted expected return is contained in the tolerance range induced by the state and the primary agent (see 6.5.1).

Therefore, for a set of weights Θ,

$$L(\Theta) = \mathbb{E}[(N_\Theta(S_t, A_t) - y_S(S_t, a_t, \Theta^-))^2] \tag{8.8}$$

is the MSE loss of the secondary agent.

9 Deep Methods Evaluation

In this chapter, we reevaluate DQL as well as the TL methods DQQL and DQGL on the racetrack benchmark. For each algorithm, we consider the agents with the hyperparameter configuration which had the most amount of successful seeds, i.e., the agents are able to reach the goal from the starting line. For those agents, we will also evaluate the performance when starting at a random location of the map, a total of 100 locations per seed. The average win, lose and timeout percentages (goal reachability evaluation) are then reported. DQQL and DQGL are additionally evaluated on a racetrack variant called magic button racetrack which is designed to empirically show that an undiscounted goal reaching performance measure is learned.

The evaluation of the algorithms is done on the Barto-small map (20k episodes) and Barto-big map (30k episodes) if successful on the former one. Exploration is done ϵ-greedy [21] with exponential decay. Reported learning curves are generated by testing the agents' performance in a separate evaluation step after each learning step.

Table 9.1: Shared hyperparameters

Parameter	Value
discount (γ_1)	1.0
learning rate (α)	0.0003
target smoothening coefficient (τ)	0.0001
seeds	{1,2,3,4,5}
step limitation	150
replay buffer size	$2 \cdot 10^4$
minibatch size	256
epsilon-start	1.0
epsilon-end	0.01

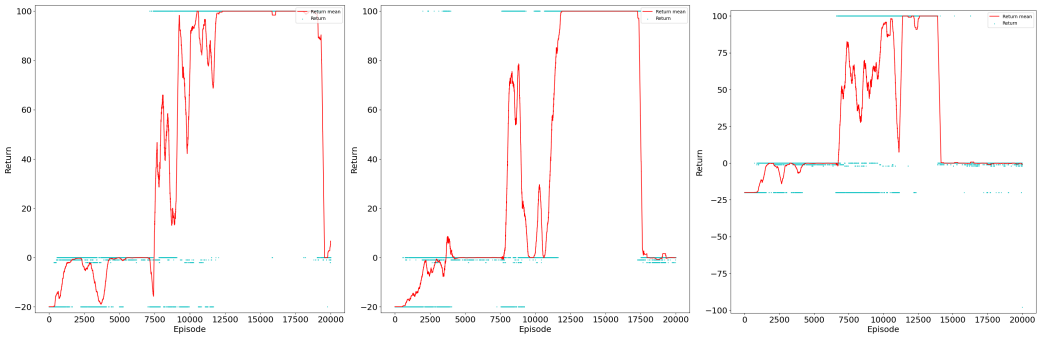
Table 9.2: TL-specific hyperparameters

Parameter	Value
tolerance (δ)	{0.01, 0.02, 0.05}
step cost (DQGL)	1
discount (γ_2)	0.9

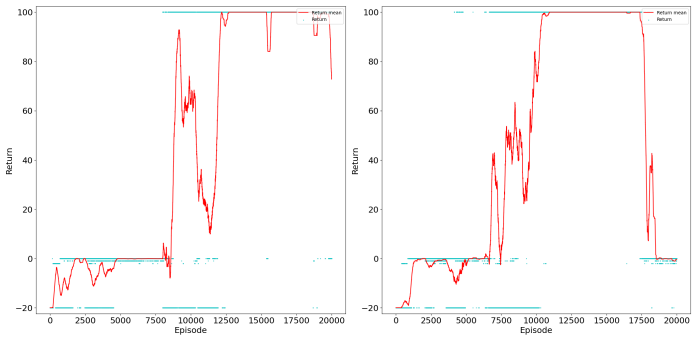
In Table 9.1, the hyperparameters used for all algorithms are listed. As proposed by Andrychowicz et al. we use a learning rate of $\alpha = 0.0003$ as a starting point and adjust it if necessary [3]. The Table 9.2 shows hyperparameters specific to the TL algorithms.

9.1 DQL Reinvestigation

A first look at the training curves (9.1) reveals that the task fails due to catastrophic forgetting, as discussed in chapter 5.1.



(a) seed 1 (b) seed 2 (c) seed 3



(d) seed 4 (e) seed 5

Figure 9.1: Learning curves of DQL for the Barto-small map

Taking a look at the heatmap visualising the Q-values (9.2), we do not observe the same pattern as discussed before in chapter 5.1. Q-values were not higher than Q-values of states closer to the start.

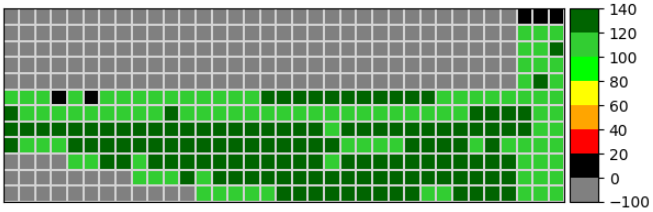
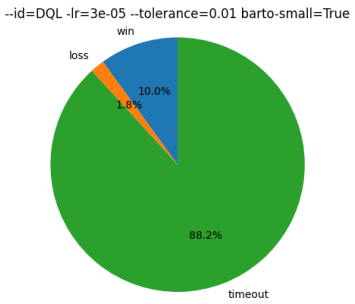


Figure 9.2: Heatmap of Q-values after the final learning step, weighted by visits for all velocities

Upon inspecting the exact values, we discover another reason why the task fails that the heatmap does not capture. Alongside Q-values becoming equal and therefore rendering the agent uninformed, we also observe that many state-actions pairs do have a Q-value larger (>20%) than the maximum possible reward of 100. We call this phenomenon Q-value explosion.

When evaluating the goal reachability for random starting locations, (9.3) we observe a higher timeout probability than in the experiments of Gros [7].



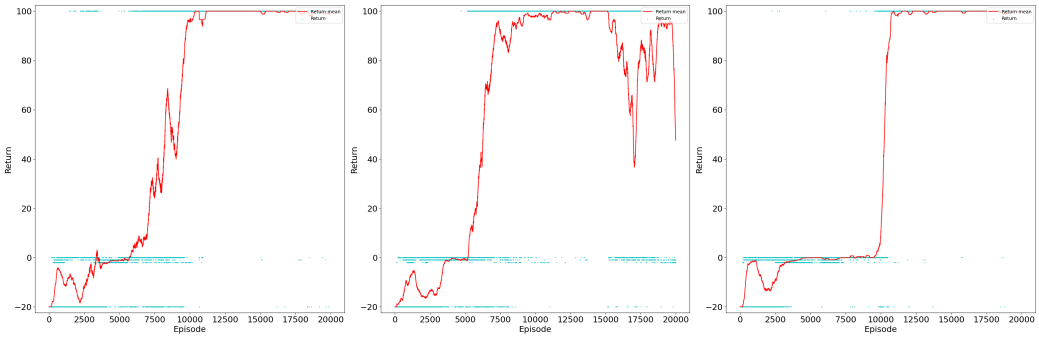
(a) $\alpha = 0.00003$

Figure 9.3: Win-Lose-Timeout evaluation of all DQL agents on Barto-small map (random start)

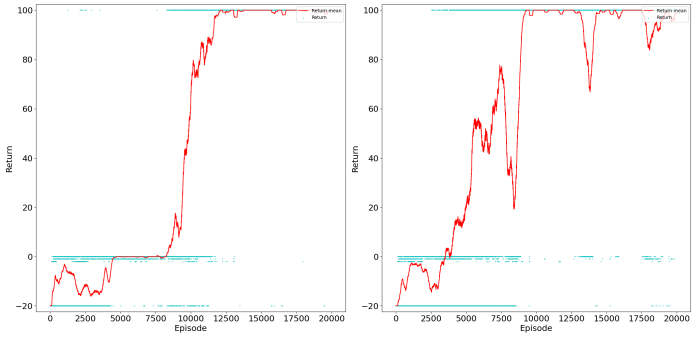
9.1.1 Effects of Reducing Overestimation Bias on Q-value Explosion

While it is true that Q-values becoming equal must cause a goal reachability drop, we suspect that Q-value explosion is a more significant problem, since the issue will carry over to TL methods. We suspect that this problem is a consequence of pairing the overestimation bias of DQL [23] with using a discount factor of $\gamma = 1$. This allows circular paths to push Q-values above achievable return values. Several techniques have been proposed to reduce overestimation bias [23], but we are not aware of a method preventing it. When applying double DQN as proposed by Guez et al. we observe better results.

Only one seed (9.4b) showed signs of catastrophic forgetting when looking at the learning curves (9.4).



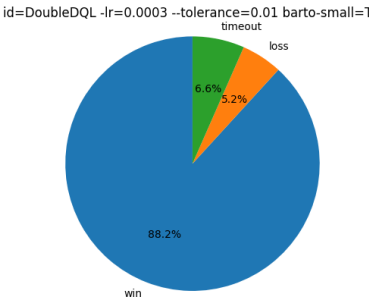
(a) seed 1 (b) seed 2 (c) seed 3



(d) seed 4 (e) seed 5

Figure 9.4: Learning curves of double DQL for the Barto-small map

The goal reachability evaluation (9.4b) shows that double DQN achieves a better win probability. The low timeout rate can be explained by the observation that the Q-values are updating at a slower rate than it is the case for DQL, therefore delaying the issue of Q-values becoming equal or overestimated.



(a) $\alpha = 0.0003$, $\delta = 0.01$

Figure 9.5: Win-Lose-Timeout evaluation of double DQL agents on Barto-small map (random start)

The heatmap 9.6 shows a Q-value pattern more similar to the pattern seen in Figure 5.2. We suspect that overestimation pushes Q-value explosion, especially near the starting states.

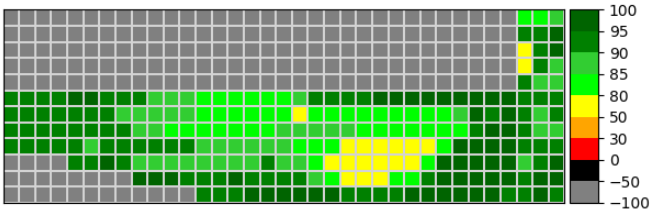


Figure 9.6: Heatmap for the estimated maximal Q-values with zero velocity

Even if we observe an improvement, we do not consider Q-value explosion to be fixed. The issue here is that double DQN only reduces the probability of Q-value overestimation, but does not prevent it. We predicted that the DTL algorithm will not be successful as it is right now, due to Q-value explosion interfering with the undiscounted part. In fact, piloting experiments for DTL suggested that Q-value explosion is not only an issue when using a DQN as network structure for the components, but also when double DQN is used. Therefore, we need to find a solution for dealing with Q-value explosion before considering DTL.

9.1.2 Dealing with Q-value Explosion

We present an idea for preventing Q-value explosion in the first place in chapter 10. For now, we make use of the fact that we want to show functionality of DTL on an environment with binary rewards. Therefore, capping the Q-values such that the future reward estimation is lower than the reward when reaching a goal state eliminates the problem of Q-value explosion. Whereas this does not help in case of DTL itself, since we define the overestimated Q-values to be equal, this does not harm a DTL agent due to still having its secondary component.

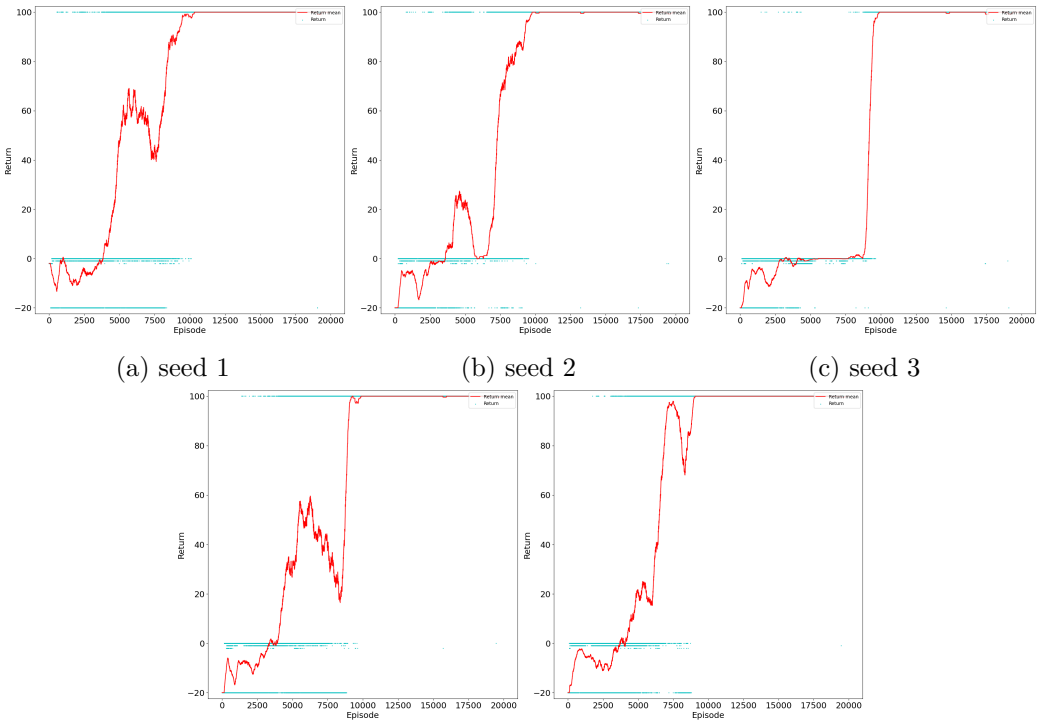
In terms of the definitions in chapter 8, we replace $y_P(s, a, \Theta_1, \Theta_2)$ by $\min(y_P(s, a, \Theta_1, \Theta_2), \rho)$, where $\rho \in \mathbb{R}$ is the highest reward in the binary reward setting, and use the updated variant for the following evaluation.

9.2 DQQL Results

In this section, we report the results of the DQQL method on the Barto-small and Barto-big map.

9.2.1 Barto-small

For DQQL applied on the Barto-small map, we find that the agent was able to solve the environment for every seed when starting from the starting line. When looking at the learning curves shown in Figure 9.7, we partially observe performance collapses during the training process. However, with further learning, the performance of the agent quickly recovers.

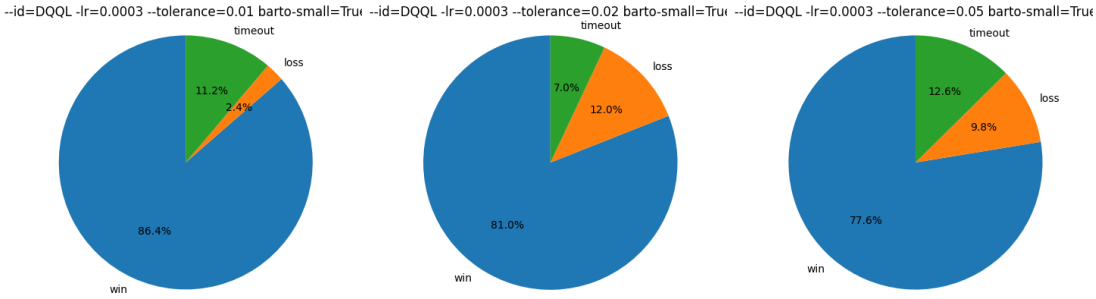


(a) seed 1 (b) seed 2 (c) seed 3

(d) seed 4 (e) seed 5

Figure 9.7: Learning curves of DQQL for the Barto-small map

Figure 9.8 shows the Win-Lose-Timeout percentages when starting in a random initial state. Here, an increased tolerance factor correlates with a lower goal reachability.

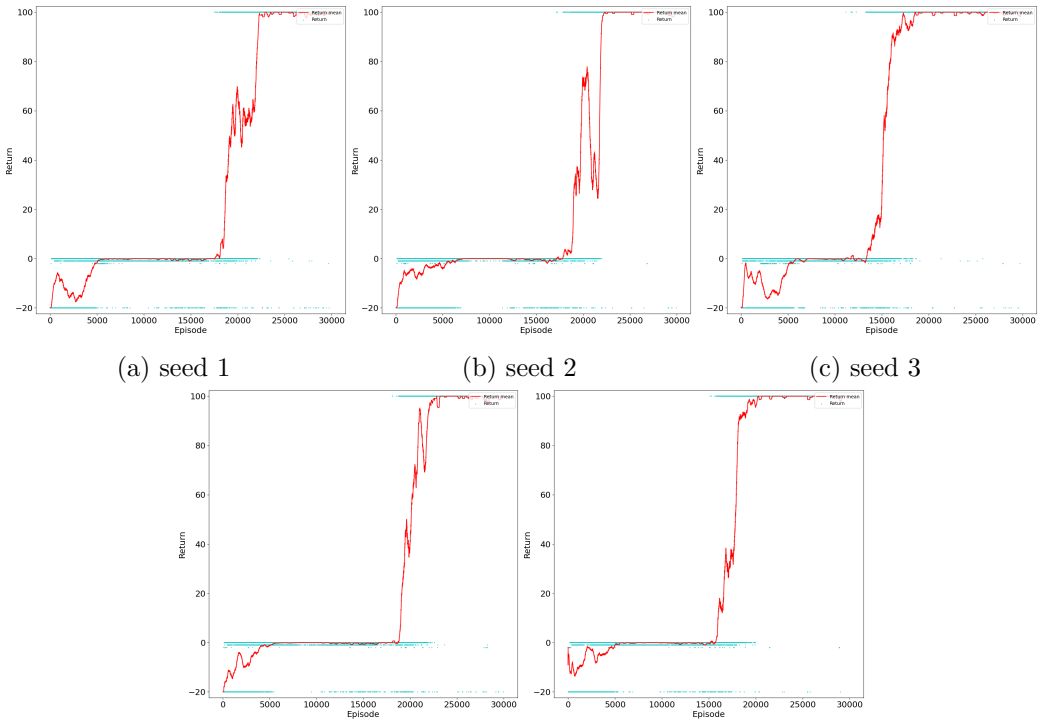


(a) $\alpha = 0.0003$, $\delta = 0.01$ (b) $\alpha = 0.0003$, $\delta = 0.02$ (c) $\alpha = 0.0003$, $\delta = 0.05$

Figure 9.8: Win-Lose-Timeout evaluation of DQQL agents on Barto-small (random start)

9.2.2 Barto-big

When applying DQQL on the Barto-big map, we find the agents being able to solve the environment on all seeds (9.9).

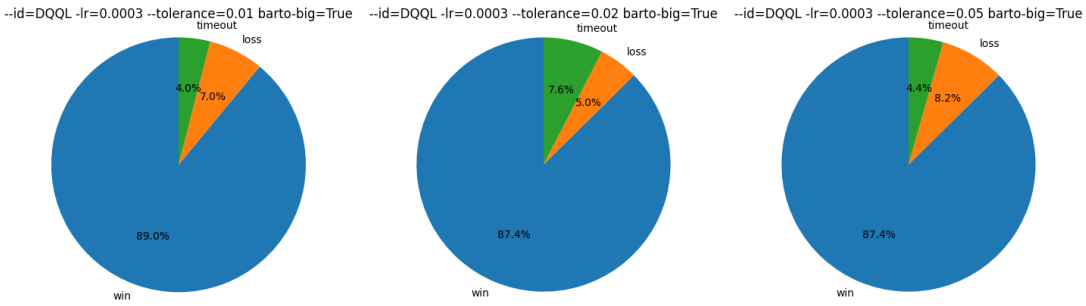


(a) seed 1 (b) seed 2 (c) seed 3

(d) seed 4 (e) seed 5

Figure 9.9: Learning curves of DQQL for the Barto-big map

In Figure 9.10, we observe a higher goal reachability when using a lower tolerance factor.



(a) $\alpha = 0.0003$, $\delta = 0.01$ (b) $\alpha = 0.0003$, $\delta = 0.02$ (c) $\alpha = 0.0003$, $\delta = 0.05$

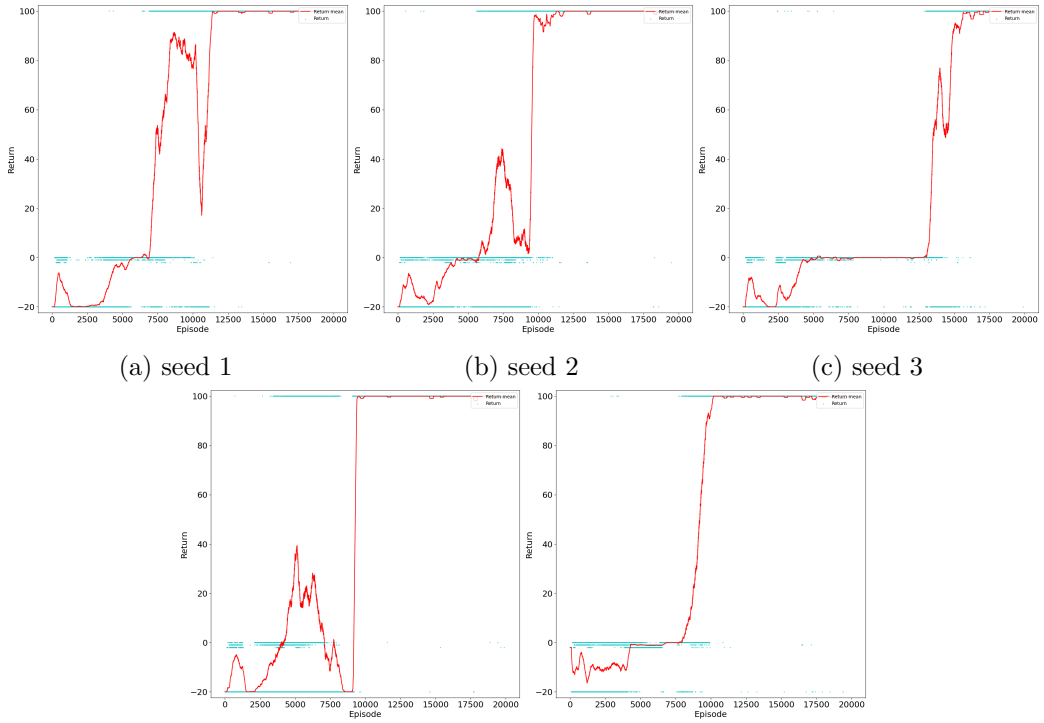
Figure 9.10: Win-Lose-Timeout evaluation of DQQL agents on Barto-big map (random start)

9.3 DQGL Results

Here, we report the results of the DQGL method on the Barto-small and Barto-big map.

9.3.1 Barto-small

For the DQGL variant, we observe similar learning curves as in the DQQL setting (Figure 9.9). We observe partial performance drops during the learning process which do no harm in the long run.

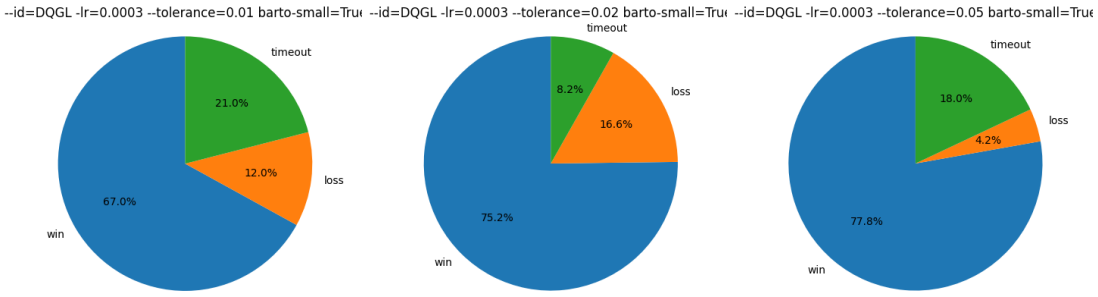


(a) seed 1 (b) seed 2 (c) seed 3

(d) seed 4 (e) seed 5

Figure 9.11: Learning curves of DQGL for the Barto-small map

Considering the evaluation for random starts (9.12), we observe worse results than in the DQQL setting. In contrast to before, we do not see a correlation between the choice of tolerance factor and the goal reachability.

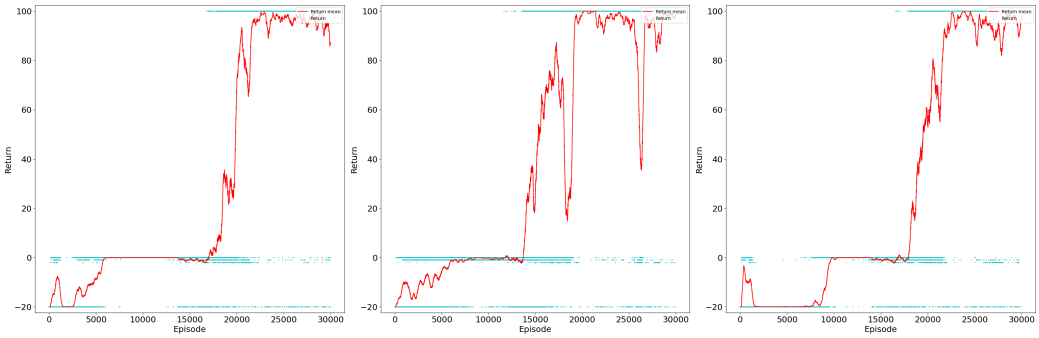


(a) $\alpha = 0.0003$, $\delta = 0.01$ (b) $\alpha = 0.0003$, $\delta = 0.02$ (c) $\alpha = 0.0003$, $\delta = 0.05$

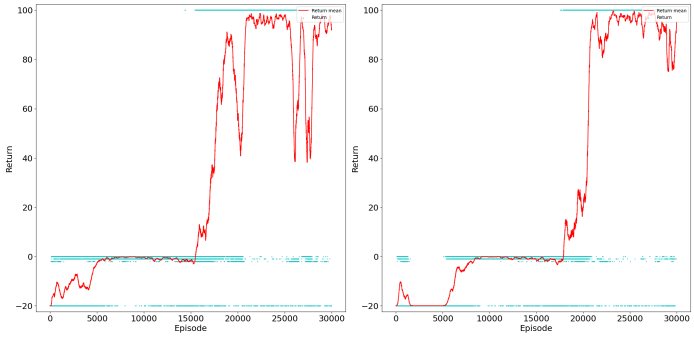
Figure 9.12: Win-Lose-Timeout evaluation DQQL agents on Barto-small map (random start)

9.3.2 Barto-big

For Barto-big we find that agents were able to solve the environment with all 5 seeds (9.9). We partially observe more performance drops during the learning process (9.13d).



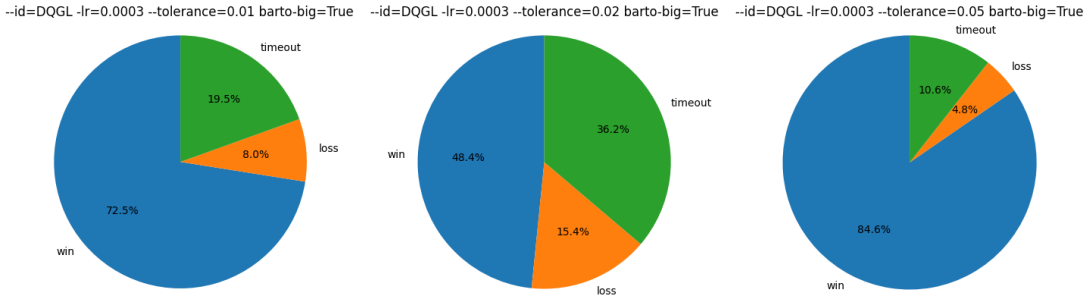
(a) seed 1 (b) seed 2 (c) seed 3



(d) seed 4 (e) seed 5

Figure 9.13: Learning curves of DQGL for the Barto-big map with $\alpha = 0.0003$ and $\delta = 0.01$

In contrast to before, for the goal reachability evaluation, we do not observe a correlation between tolerance factor choice and goal reachability. Instead, we observe worse results for $\delta = 0.02$.



(a) $\alpha = 0.0003$, $\delta = 0.01$ (b) $\alpha = 0.0003$, $\delta = 0.02$ (c) $\alpha = 0.0003$, $\delta = 0.05$

Figure 9.14: Win-Lose-Timeout evaluation DQGL agents on Barto-big map (random start)

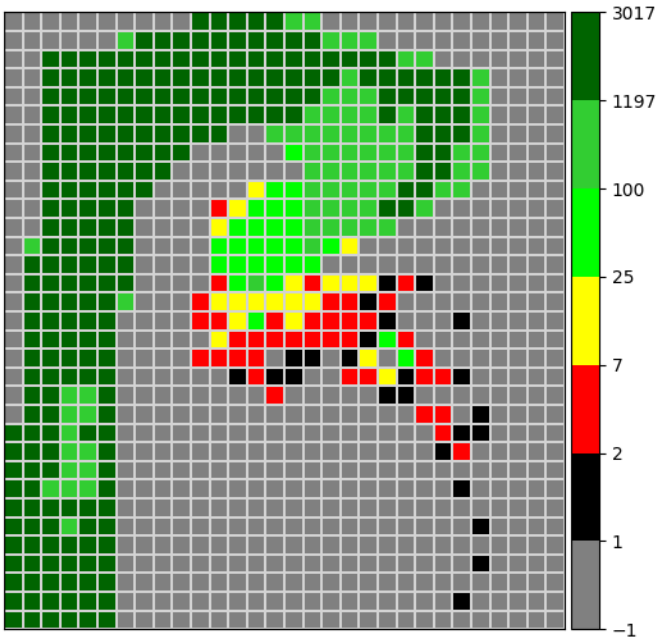


Figure 9.15: Visits per state during training of an DQGL agent with $\alpha = 0.0003$ and $\delta = 0.02$ (all velocities combined)

Upon inspecting the seeds individually, we find that two seeds performed worse than the

other three. A map showing the visit amount per state after training (Figure 9.15) shows that insufficient exploration lead to the bad results. We think that despite capping rewards, overestimation bias favours visiting states near the starting line.

9.4 Magic Button Racetrack

For empirically verifying that TL does not only solve the racetrack but also optimises an undiscounted goal reaching performance measure, we designed a variant of the racetrack called magic button racetrack.

The difference to the racetrack benchmark is that the zero-acceleration action is replaced with the action of pressing a magic button in case the car is not moving. This action has the behaviour that with a 50% chance, the task is immediately considered to be either successful or failed. Formally, if T is the stochastic transition function of the racetrack map, then

$$T_{MB}(s,a,s') = \begin{cases} T(s,a,s'), & y_x \neq 0 \vee v_y \neq 0 \\ \frac{1}{2} \cdot T(s,a,s'), & v_x = v_y = 0 \wedge s' \notin \{\top, \bot\} \\ \frac{1}{2} \cdot T(s,a,\top) + \frac{1}{2}, & s' = \top \wedge v_x = v_y = 0 \wedge s' = \top \\ \frac{1}{2} \cdot T(s,a,\bot) + \frac{1}{2}, & s' = \top \wedge v_x = v_y = 0 \wedge s' = \bot \end{cases} \tag{9.1}$$

is the transition function of the magic button racetrack.

9.4.1 Results

A look at the results reveals that for DQQL as well as for DQGL, the agent learned to take the button press action, thus learning a suboptimal and unreliable policy (9.16).

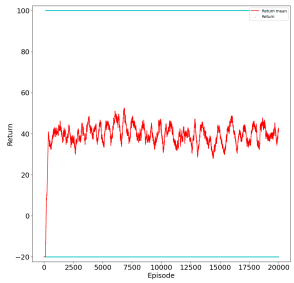


Figure 9.16: Learning curves of a TL agent on the Barto-small map (magic button variant)

In both cases, the agent does not move, since the episode ends immediately.

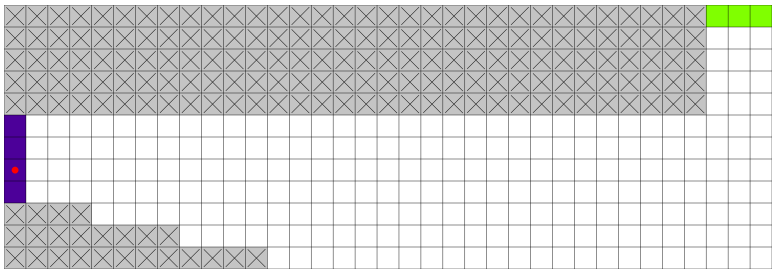


Figure 9.17: Path after final learning step of a TL agent on the Barto-small map (magic button variant)

Looking at the states which were visited during training, we see why TL cannot learn a desired policy. Due to the exploration-exploitation dilemma [21], the probability of finding a better alternative path is low.

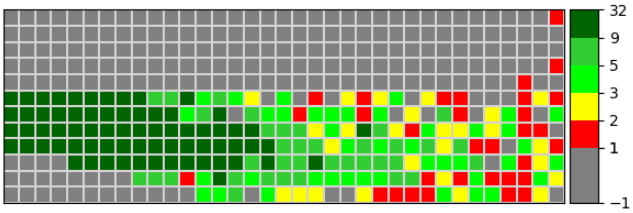


Figure 9.18: Visits per state during training (all velocities combined)

Note that the total number of visits appears lower, since the initial placement on the starting line does not count as a visit and neither does the press of the magic button increase the visits of any state.

However, we can observe the performance of TL if the agent has enough knowledge by disabling the magic button option until episode 15,000. This time, we observe that a change of the environment mid-training did not harm any TL agent (9.19, 9.20).

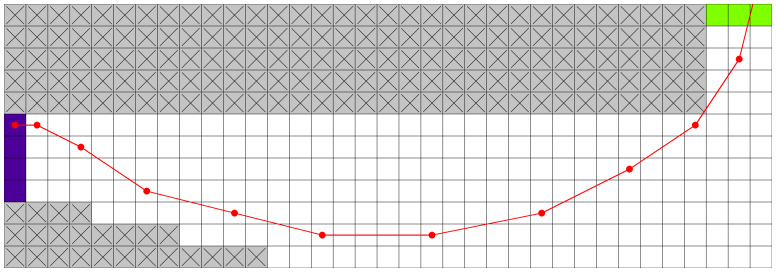


Figure 9.19: Path after final learning step of a DQQL agent on the Barto-small map (magic button variant), delayed button availability

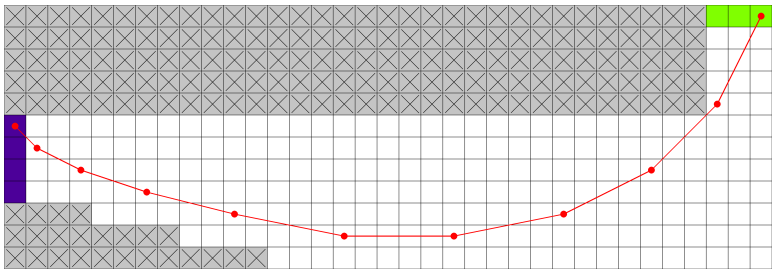


Figure 9.20: Path after final learning step of a DQGL agent on the Barto-small map (magic button variant), delayed button availability

There are learning curves for which we observe a performance drop, shortly after the magic button is enabled, but the agent learns that the action hurts its performance and recovers (9.21).

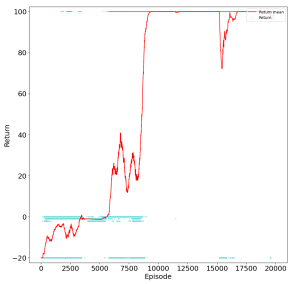


Figure 9.21: Learning curve of a TL agent on the Barto-small map (magic button variant), delayed button availability

Investigating the exact values also show that the primary agent adjusted the expected value of the magic button action, and thus was able to follow a path with higher goal reachability.

9.5 Summarising Deep Learning Methods

In this chapter, we observed that Q-values becoming equal is not the only reason undiscounted DQL is not able to solve the racetrack. Due to an overestimation bias of DQL, we observe that Q-values higher than the actually achievable total reward are learned. Since this issue inevitably carries over to TL, we capped the target values at the maximal achievable reward.

With this change, TL was not only able to solve the standard racetrack but was also able to solve the magic button racetrack, verifying that TL learns to discard actions yielding immediate rewards but are not optimal in regard to optimising the total undiscounted reward. However, we observe that the TL methods proposed are prone to suffering from the exploration-exploitation dilemma.

10 Undiscounted Value Function Reconstruction

To deal with Q-value explosion, we previously capped the maximum value an action-value function may have. For generalising DTL to environments which do not solely yield binary return values, we need to prevent Q-value explosion in the first place instead of dealing with its consequences. We think that Q-value explosion only occurs for sufficiently high discount factors $\gamma \to 1$. Therefore, we propose to learn a discounted state-value function together with additional data, which allows us to reconstruct the undiscounted state-value function without suffering from Q-value explosion.

10.1 Recovering the Data

For this initial approach, we make the assumption that rewards are only received when reaching a goal or a failed state. However, we think that it is possible to transform environments, such that rewards are delayed to final states, which would allow us to generalise this approach.

For a set policy π and state $s \in S$ let

$$x_{\gamma_1} = v_\pi^{\gamma_1}(s) \tag{10.1}$$

$$x_{\gamma_2} = v_\pi^{\gamma_2}(s) \tag{10.2}$$

be the expected return value in the state under the policy with $\gamma_1 \neq \gamma_2 \in (0, 1)$. We denote the unknown expected return value in the state under the same policy with $x_1 v_\pi^1(s)$.

Note that since $x_{\gamma_1}, x_{\gamma_2}$ and x_1 are expected returns when following the same policy, we know that

$$x_1 = \frac{x_{\gamma_1}}{{\gamma_1}^n} = \frac{x_{\gamma_2}}{{\gamma_2}^n} \tag{10.3}$$

where n is the expected number of steps from s to a goal or failed state.

Thus, we can calculate x_1 in case we know the goal distance n. This is the case since

$$\frac{x_{\gamma_1}}{{\gamma_1}^n} = \frac{x_{\gamma_2}}{{\gamma_2}^n} \tag{10.4}$$

$$\Leftrightarrow \left(\frac{\gamma_2}{\gamma_1}\right)^n = \frac{x_{\gamma_1}}{x_{\gamma_2}} \tag{10.5}$$

$$\Leftrightarrow n \cdot \log\left(\frac{\gamma_2}{\gamma_1}\right) = \log\left(\frac{x_{\gamma_1}}{x_{\gamma_2}}\right) \tag{10.6}$$

$$\Leftrightarrow n = \frac{\log(\frac{x_{\gamma_1}}{x_{\gamma_2}})}{\log(\frac{\gamma_2}{\gamma_1})} \tag{10.7}$$

such that n is known. Alternatively, we can learn the goal distance directly.

10.2 Results

We evaluate the idea of the approach by training a DQGL agent with its primary agent learning a discounted state-value function and using undiscounted value function reconstruction for decision-making.

Figure 10.1 shows a learning curve of an uncapped DQGL agent. We observe similar learning curves for every hyperparameter configuration we tried, i.e., none of the uncapped DQGL agents was able to solve the racetrack. Even though when investigating Q-values during training, we partially observe good approximations of the undiscounted values, this is not the case across the board.

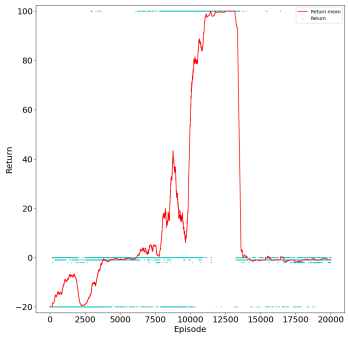


Figure 10.1: Learning curve of an uncapped DQGL agent

We think that the main issue of the approach is that it assumes that the approximations of the undiscounted expected return and the goal distance are perfectly in sync. If that is not the case, the calculated expected undiscounted return might have a large error, which leads to unpredictable decision-making. This also explains why we observe that

the agent becomes uninformed after already having learned a policy which can reach the goal.

11 Conclusion and Further Work

11.1 Conclusion

Throughout this thesis we investigated methods which do not rely on discounting. We differentiated approaches commonly referred to as undiscounted methods from approaches which optimise the total reward while still aiming to reach the goal.

We presented tuple learning methods, alongside its theoretical foundation and derived the QQL and QGL algorithms.

The algorithms were compared with Q-learning and other undiscounted methods. We concluded that in contrast to the other investigated algorithms, the tuple learning methods were able to optimise an undiscounted goal reaching performance measure, thus achieving the goal of this thesis. We observed that due to the exploration-exploitation dilemma, tuple learning methods are prone to being hindered in their learning process.

Alongside the formerly observed problem of agents becoming uninformed when training them using Q-learning with a discount factor of $\gamma = 1$, we observed that overestimation bias caused another problem we called Q-value explosion. Even though this did not pose a problem in the environments we used for evaluation, we expect this issue to hinder the generalisation of tuple learning methods to environments with more complex reward structures.

We proposed the method of undiscounted value function reconstruction to deal with Q-value explosion. Despite having theoretical arguments that this approach might solve the issue, in practice the issue is more complex and requires further investigation.

11.2 Further Work

We think that tuple learning methods can be generalised such that they become feasible learning methods for other environments. For doing so, the two issues of the approach have to be addressed.

Firstly, the method has to be combined with a strategy which ensures better exploration properties to achieve reasonable learning durations.

Secondly, the issue of Q-value explosion needs to be addressed such that tuple learning

methods can be used on environments with more complex reward structures. We think that the undiscounted value function reconstruction approach, despite not being successful as proposed, provides a core idea which can be built and extended on such that the issue might be solved.

It is also of interest if varying the tolerance factor mid-training or during evaluation improves the performance of TL agents.

Bibliography

[1] Nn-svg. http://alexlenail.me/nn-svg/lenet.html.

[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, 08 2019.

[3] Marcin Andrychowicz, Anton Raichuk, Piotr Stańczyk, Manu Orsini, Sertan Girgin, Raphael Marinier, Léonard Hussenot, Matthieu Geist, Olivier Pietquin, Marcin Michalski, Sylvain Gelly, and Olivier Bachem. What matters in on-policy reinforcement learning? a large-scale empirical study, 06 2020.

[4] Alexei Botchkarev. Performance metrics (error measures) in machine learning regression, forecasting and prognostics: Properties and typology, 09 2018.

[5] Heinz-Dieter Ebbinghaus. Einführung in die mengenlehre, 2021.

[6] Timo Gros and Joschka Groß. Rlmate. https://pypi.org/project/rlmate/.

[7] Timo P. Gros. *Tracking the Race: Analyzing Racetrack Agents Trained with Imitation Learning and Deep Reinforcement Learning*. 2021.

[8] Jörg Hoffmann. Lecture notes in artificial intelligence, May 2022.

[9] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.

[10] Timothy Lillicrap, Jonathan Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, 09 2015.

[11] Andreas Meister and Thomas Sonar. Numerik: Eine lebendige und gut verständliche einführung mit vielen beispielen.

[12] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. 2013. cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013.

[13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc Bellemare, Alex Graves, Martin Riedmiller, Andreas 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*, 518:529–33, 02 2015.

[14] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[15] Anton Schwartz. A reinforcement learning method for maximizing undiscounted rewards. pages 298–305, 12 1993.

[16] David Silver, Aja Huang, Chris 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(7587):484–489, January 2016.

[17] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017.

[18] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy P. Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nat.*, 550(7676):354–359, 2017.

[19] Sonali, B. Maind, and Priyanka Wankar. Research paper on basic of artificial neural network. 2014.

[20] Richard Sutton. Learning to predict by the method of temporal differences. *Machine Learning*, 3:9–44, 08 1988.

[21] Richard S. Sutton, Francis Bach, and Andrew G. Barto. *Reinforcement learning: An introduction*. MIT Press Ltd, 2018.

[22] Prasad Tadepalli and Dokyeong Ok. H-learning: A reinforcement learning method to optimize undiscounted average reward. 03 1996.

[23] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015.

[24] Christopher Watkins and Peter Dayan. Technical note: Q-learning. *Machine Learning*, 8:279–292, 05 1992.

[25] Dirk Werner. *Funktionalanalysis*. Springer Spektrum, 2018.

[26] Xiang-Sun Zhang. *Introduction to Artificial Neural Network*, pages 83–93. Springer US, Boston, MA, 2000.