Smart Start

A Directed and Persistent Exploration Framework for Reinforcement Learning

Bart Keulen

Master of Science Thesis

TU Delft
Delft University of Technology

Delft Center for Systems and Control

Smart Start
A Directed and Persistent Exploration Framework for Reinforcement Learning

Master of Science Thesis

For the degree of Master of Science in Systems and Control at Delft University of Technology

Bart Keulen

March 2, 2018

Faculty of Mechanical, Maritime and Materials Engineering (3mE) · Delft University of Technology

The work in this thesis was supported by the Institute for Human and Machine Cognition. Their cooperation is hereby gratefully acknowledged.

Copyright © Delft Center for Systems and Control (DCSC)
All rights reserved.

Voor papa.

Abstract

An important problem in reinforcement learning is the exploration-exploitation dilemma. Especially for environments with sparse or misleading rewards it has proven difficult to construct a good exploration strategy. For discrete domains good exploration strategies have been devised, but are often nontrivial to implement on more complex domains with continuous states and/or actions.

In this work, a novel persistent and directed exploration framework is developed, called Smart Start. Usually, a reinforcement learning agent executes its learned policy with some exploration strategy from the start until the end of an episode, which we call "normal" learning. The idea of Smart Start is to split a reinforcement learning episode in two parts, the Smart Start phase and the "normal" learning phase. The initial Smart Start phase guides the agent to a region in which the agent expects to learn the most. The region is constructed using previous experiences and the guiding is done using a model-based planning or trajectory optimization method. When the agent arrives at the region, it continues its "normal" reinforcement learning. This approach leaves the performance of the used reinforcement learning algorithm unchanged, but augments it with persistent and directed exploration.

The Smart Start framework was evaluated using three reinforcement learning algorithms, a simple model-based reinforcement learning algorithm (MBRL), R-MAX and Q-Learning with ϵ-greedy, Boltzmann and UCB1 exploration. The evaluation was done on four discrete gridworld environments. Three environments with sparse rewards and one with misleading rewards. We showed that the performance of Q-Learning with Smart Start is comparable to R-MAX, which performs near optimal in the used scenarios. The MBRL algorithm with Smart Start is even able to outperform R-MAX in some of the problems. We show that Smart Start is a good framework for exploration that can be incorporated with any reinforcement learning algorithm.

Table of Contents

List of Figures vii

List of Tables ix

List of Symbols xi

1 Introduction 1
- 1-1 Related Work . . . 2
- 1-2 Contributions . . . 4
- 1-3 Thesis Outline . . . 5

2 Reinforcement Learning & Exploration 7
- 2-1 Reinforcement Learning . . . 7
- 2-2 Value Function Methods . . . 9
  - 2-2-1 Dynamic Programming . . . 10
  - 2-2-2 Model-Based Reinforcement Learning . . . 12
  - 2-2-3 Temporal Difference . . . 13
- 2-3 Exploration Strategies . . . 14
  - 2-3-1 Undirected Exploration . . . 15
  - 2-3-2 Directed Exploration . . . 16
  - 2-3-3 Model-Based Exploration . . . 16
- 2-4 Discussion . . . 18

3 Smart Start 19
- 3-1 Method . . . 20
- 3-2 Smart Start State . . . 20
  - 3-2-1 Multi-Armed Bandit Problem . . . 22
  - 3-2-2 Discrete Smart Start State . . . 23

3-3 Trajectory Optimization . . . 24
3-3-1 Transitioning to the Smart Start State using Dynamic Programming . . . 24
3-4 Smart Start Algorithm . . . 26
3-4-1 Q-Learning with Smart Start . . . 26
3-4-2 Continuous Domains . . . 26
3-5 Discussion . . . 29

4 Experimental Setup **31**
4-1 Deterministic Gridworld . . . 31
4-2 Stochastic Gridworld . . . 33
4-3 Misleading Gridworld . . . 34

5 Results **35**
5-1 Exploration Performance . . . 36
5-2 Smart Start Parameters . . . 39
5-3 Deterministic Gridworld with Sparse Rewards . . . 41
5-4 Stochastic Gridworld with Sparse Rewards . . . 45
5-5 Deterministic Gridworld with Misleading Rewards . . . 49
5-6 Discussion . . . 51

6 Conclusion **53**
6-1 Conclusion . . . 54
6-2 Recommendations . . . 55
6-2-1 Non-Sparse Rewards . . . 55
6-2-2 Continuous Environments . . . 56
6-2-3 Smart Start State Selection . . . 56

A Model-Based Reinforcement Learning with Smart Start **57**

B Continuous Smart Start State **59**

Acknowledgments **65**

List of Figures

2-1 Agent environment interaction in reinforcement learning 8

3-1 Flow chart for a reinforcement learning episode with and without Smart Start . . 21

4-1 Gridworld environments 32

5-1 Exploration performance of Smart Start 38
5-2 Influence of Smart Start parameters η and c_{ss} 40
5-3 Rise time of Smart Start on deterministic environments 43
5-4 Normalized average reward of Smart Start for Q-Learning on deterministic environments 44
5-5 Normalized average reward of Smart Start for MBRL and R-MAX on deterministic environments 44
5-6 Normalized average reward of Smart Start for Q-Learning on stochastic environments 47
5-7 Normalized average reward of Smart Start for MBRL and R-MAX on stochastic environments 47
5-8 Policy correctness of Smart Start for Q-Learning on stochastic environments . . . 48
5-9 Policy correctness of Smart Start for MBRL and R-MAX on stochastic environments 48
5-10 Rise time of Smart Start on the Misleading gridworld environment 50
5-11 Normalized average reward of Smart Start for Q-Learning, MBRL and R-MAX on the Misleading gridworld environment 50

List of Tables

4-1 Specifications of the gridworld environments 33

5-1 Reinforcement learning parameters 35

List of Symbols

α	Learning rate
δ_t	Temporal difference error at time-step t
γ	Discount rate
$\mathbb{E}$	Expectation
$\mathbf{a}_t$	Action at time-step t
$\mathbf{s}_t$	State at time-step t
$\mathcal{A}(\mathbf{s}_t)$	Set of actions at state $\mathbf{s}_t$
$\mathcal{P}(\mathbf{s}, \mathbf{a}, \mathbf{s}')$	Transition probability of transferring from state $\mathbf{s}$ to $\mathbf{s}'$ under action $\mathbf{a}$
$\mathcal{R}(\mathbf{s}, \mathbf{a})$	Expected reward associated when action $\mathbf{a}$ is executed in state $\mathbf{s}$
$\mathcal{S}$	Set of states
π	Policy
$\pi(\mathbf{s})$	Deterministic policy
$\pi(\mathbf{s}, \mathbf{a})$	Stochastic policy
$\Pr$	Probability
$C(\mathbf{s})$	State visitation count
$C(\mathbf{s}, \mathbf{a})$	State-action visitation count
$C(\mathbf{s}, \mathbf{a}, \mathbf{s}')$	State-action-state visitation count
J	Expected return
$Q(\mathbf{s}, \mathbf{a})$	Action-value function for state $\mathbf{s}$ and action $\mathbf{a}$
R	Return
$R_{\text{sum}}(\mathbf{s}, \mathbf{a})$	Sum of rewards for state $\mathbf{s}$ and action $\mathbf{a}$
r_t	Reward at time-step t
T	Final time-step of an episode (same as T_{episode})
t	Discrete time-step
T_{episode}	Final time-step of an episode (same as T)

$V(\mathbf{s})$	State-value function for state $\mathbf{s}$
ϵ	Probability of choosing random action with ϵ-greedy exploration
$\mathbf{s}_r$	R-max absorbing state
τ	Boltzmann temperature parameter
C_p	UCB1 constant
m	R-max threshold parameter
$R_{\max}$	R-max replacing reward
$\mathfrak{R}_n$	Expected regret after n plays
$\mu(i)$	Expectation of $X(i)$
K	Number of actions in a multi-armed bandit problem
$T_n(i)$	Number of times action i has been executed during the first n plays
$X(i)$	Random variable associated with action i
η	Smart Start initialization threshold
$\mathbf{s}_0$	Initial state
$\mathbf{s}_{ss}$	Smart Start state
$\mathcal{D}$	Replay buffer
π_{ss}	Policy from $\mathbf{s}_0$ to $\mathbf{s}_{ss}$
θ	Distance threshold for the distance between $\mathbf{s}_t$ and s_t
c_{ss}	Smart Start state selection constant
u	Random value sampled from a uniform distribution
$\Delta\mathbf{s}$	State transition difference
$\mathbf{s}_{\text{subgoal}}$	Sub-goal state gridworld environments
$\mathbf{s}_{\text{goal}}$	Goal state gridworld environments
$\mathbf{s}_{\text{start}}$	Start state gridworld environments
τ_{rise}	Rise time in training steps
ε	Rise time threshold parameter
$l(\pi^*)$	Path length of optimal policy π^*
N_{states}	Total number of accessible states gridworld environments
$T_{\max}$	Maximum number of time-steps of an experiment

Chapter 1

Introduction

Many problems involve optimization over time, ranging from controlling heating systems to playing a game of backgammon. These problems involve sequential decision making and are often modeled as Markov Decision Processes [1]. A lot of problems can be solved by designing a controller, this often requires a model of the system involved in the problem. But for problems where no model is available or the design of a controller is nontrivial a different solution is necessary.

Reinforcement learning [2] is concerned with solving sequential decision making problems. In reinforcement learning an agent learns through interaction with its environment. The environment tells the agent how well it is doing by giving it some reward signal. Based on this reward feedback the agent tries to find the solution that yields the highest accumulated reward signal over time.

We will illustrate the reinforcement learning problem with a simple example. Imagine a completely dark room with a robot in it. The goal of the robot is to learn to find the exit, starting from its charging station randomly placed in the room. Since it is completely dark it is not easy to find the exit. When the robot executes an action and traverses to the next state a supervisor gives the robot a positive reward if it was moving closer to the exit and a negative reward for moving further away. Based on this reward signal the robot learns that moving closer to the exit is good and moving away is bad. At the end of the day the robot returns to its starting position to recharge. The next day it can use the previously learned knowledge to quickly continue with its search. This is called an episodic problem, since every day is a new episode.

In order to learn a “good” solution the agent has to know what “good” is by experiencing high and low rewards. To do so, the agent has to find a proper balance between exploiting its current best solution and exploring new options for a better solution. This is called the exploration-exploitation dilemma [3]. Too much exploration may result in a lower accumulated reward or incredibly long learning time because the agent spends most of its time in low-reward or irrelevant parts of the state space. Too little exploration may result in a solution that is suboptimal as the agent stopped searching before it neared optimality. A good balance of

the exploration-exploitation trade-off requires an efficient exploration strategy. Developing an efficient exploration strategy has proven to be a difficult task.

For environments with sparse or misleading rewards it is even more difficult to design a good exploration strategy. In an environment with sparse rewards the agent often takes a long time before finding any useful rewards, greatly increasing the learning time if no efficient exploration strategy is used. Misleading rewards lead the agent away from the optimal solution, resulting in a suboptimal solution. These problems require a persistent and directed exploration strategy to find the optimal solution.

To illustrate the difficulty with sparse rewards, we will use the example of the robot in the dark room. This time though, the robot does not receive a positive nor negative reward for moving closer or further away from the goal, respectively. Instead, the robot only receives a reward for finding the exit. After spending a day searching around, the robot returns to its charging station. As long as the robot does not find the exit, it does not learn anything, except that nothing it tried resulted in finding the exit. Every day the robot starts with the same knowledge about the problem as the first day and often explores the same areas over and over again, which makes the search highly inefficient. For the misleading reward case, take the same dark room. But this time, the agent is attracted to an area in which a small positive reward is given to the agent. There is a high probability the agent finds this area before the exit, i.e., the agent almost certainly finds this area before it finds the exit. The reward for finding the area is a lot lower than the reward for reaching the exit, but the agent does not know what the rewards are until it finds them. What will happen is that the agent learns that this area is good to go to, since it does not know it can achieve a higher reward when reaching the exit. Therefore, it will go to the low reward area and stay there. To get out of this suboptimal solution, the agent needs a persistent and directed exploration strategy to drive it away and search for a better solution.

The goal of this thesis is to develop a general exploration framework for reinforcement learning problems with sparse or misleading rewards. To create a general strategy, the framework should not be limited to specific reinforcement learning algorithms or exploration strategies. Furthermore, the framework should be compatible for domains with discrete and/or continuous states and actions.

1-1 Related Work

Different exploration strategies have been proposed over the years. We can divide these exploration strategies in two groups: undirected and directed exploration [4].

Undirected exploration uses some form of randomness as exploration strategy. One well known method is ϵ-greedy [5], which has a small probability of executing a random explorative action. Boltzmann exploration chooses actions according to a probability distribution constructed over the available actions, based on the utility of an action. For continuous actions, Gaussian noise is often added to the control signal [6]. For environments in which being greedy is optimal, the performance of undirected exploration is often sufficient. But in environments with sparse of delayed rewards, these methods can struggle to find a good solution.

Directed exploration utilizes knowledge about the learning process to try and achieve more efficient exploration. Count-based exploration [4, 7, 8] keeps track of the frequency state-action

pairs have been executed and derives an exploration bonus from these visitation counts. Other examples of directed exploration strategies are error-based exploration [9, 10] and recency-based exploration [4, 11]. In multi-armed bandit problems [12] a near optimal method exists for choosing an action, called UCB1 [12]. UCB1 constructs an upper confidence bound according to Hoeffding's inequality [13], which ensures the regret rate grows within a constant factor of the optimal regret rate [12]. The UCB1 algorithm has been converted to be compatible with Markov Decision Processes [14]. Often, directed exploration derives an exploration bonus from a learned model. For example, the model-error or model-uncertainty can be used because they indicate which areas require more information to improve the model.

In the directed exploration methods described above, the exploration bonus is often implemented in one of the following two ways. The first implementation is valid for methods using a policy based on the utility of actions, e.g., value function methods [2]. For each action, the utility is updated with the exploration bonus and the policy chooses an action greedily with respect to the updated utilities. An example with count-based exploration can be found in Thrun [4]. These methods only look at the current time-step and do not necessarily drive the agent to areas far away that require more exploration. The second implementation is intrinsic motivation [15, 16]. In most intrinsically motivated reinforcement learning methods a reshaped reward [17] is used, which is the combination of an extrinsic and intrinsic reward. The extrinsic reward is the reward obtained from the environment and the intrinsic reward is the exploration bonus. The agent learns using this modified reward and the policy will be both explorative and exploitive. Examples with count-based exploration can be found in Kolter and Ng [7] and Strehl and Littman [8]. A problem is that reward shaping can change the optimal policy. Ng, Harada and Russel [17] showed that only potential-based reward transformations leave the optimal policy unchanged.

In model-based reinforcement learning [18], the agent learns a model from previous experiences. This model can for example be used for generating simulated experience or planning [19]. The model can also be used for improving the exploration. An example of such an algorithm is R-MAX [20]. In R-MAX a transition model and reward function are learned and subsequently used by a dynamic programming method to plan a policy. The rewards in the reward function are replaced with bonus rewards for state-action pairs that have been executed infrequently. These bonus rewards incentivize exploration for the concerned state-action pairs. Other examples of methods that use a model-based approach are E^3 [21] and MBIE [11].

Model-based exploration works well for discrete environments of relatively small size. These methods have been extended to continuous domains, often by discretization of the state space or using simple feature approximations [22, 23]. For complex environments with continuous state and/or action spaces it can be infeasible to learn a good model. Even planning for a short horizon does often not perform well [24]. Model-free methods do not have this problem and can be extended fairly easily to continuous domains using function approximation [2]. In Nagabandi et al. [25] a model-based approach is used with neural networks as function approximation for the model, but only achieves minor results. A model-free method is required after the model-based approach to achieve excellent performance.

Intrinsic motivation has become popular as an exploration strategy, because it can be used directly with model-free reinforcement learning. Examples are approximation of count-based [26, 27, 28, 29], information gain [30] or surprise-based [31] exploration bonus. Although these

methods perform better than simple undirected exploration strategies, they do have some implications. The first one is reward shaping, which changes the optimal policy [17]. Another issue is the continuously changing reward function the agent is using. The exploration bonus changes continuously as new experiences are observed, which results in a longer convergence time for the agent [11].

1-2 Contributions

The main contribution of this work is the development of a novel exploration framework, called Smart Start. As a framework, Smart Start is not limited to specific reinforcement learning algorithms or exploration strategies, but can be used in combination with the best methods at hand. The idea of Smart Start is to split a reinforcement learning episode into two parts, the Smart Start phase and the "normal" learning phase. In most reinforcement learning problems, each episode starts at the same initial state or at a random initial state. We will focus here on problems that start from the same initial state, $\mathbf{s}_0$, as this is more realistic for physical systems. In "normal" reinforcement learning, the agent starts at $\mathbf{s}_0$ and executes its policy, π, with some exploration strategy from $\mathbf{s}_0$ until the episode ends. With Smart Start, the agent first determines the region in which it expects to learn the most. The region is denoted by a single state, the Smart Start state $\mathbf{s}_{ss}$. In other words, if the agent can choose to start from any known state, $\mathbf{s}_{ss}$ is the state it believes to be most profitable to start from. Subsequently a planning or trajectory optimization method is used to guide the agent efficiently from $\mathbf{s}_0$ to $\mathbf{s}_{ss}$. Once the agent is close to $\mathbf{s}_{ss}$, the exploration policy π is executed until the end of the episode.

We use the example of the robot in the dark room with sparse rewards for illustration. The robot only receives a reward for reaching the exit. The first day, it has wandered around and explored some part of the room. But instead of starting all over again from the beginning the next day, the agent determines the most promising state to start exploring from. In this case, where it expects to be the closest to the exit. This avoids searching potentially useless areas. This state, the Smart Start state $\mathbf{s}_{ss}$, is chosen from all previously seen states. After determining $\mathbf{s}_{ss}$, the agent is guided to $\mathbf{s}_{ss}$ based on the robots prior experience navigating the room. Once the robot reached $\mathbf{s}_{ss}$, it starts exploring again. So instead of starting to explore from the charging station, the robot starts where it left of in previous episodes.

We evaluate the Smart Start framework for sparse and misleading rewards. The evaluation is done on four discrete gridworld environments. Three environments are used for testing the exploration performance and performance in the case of sparse rewards. The fourth environment is used for evaluating the performance when misleading rewards are present. Model-free and model-based reinforcement learning algorithms are used in combination with Smart Start. Q-Learning [2] is used as model-free algorithm with three different exploration strategies: ϵ-greedy, Boltzmann and UCB1. A simple model-based reinforcement learning algorithm was implemented and compared with R-MAX [12]. The guiding was done using Value Iteration [1]. We show that the performance of Q-Learning with Smart Start is similar to R-MAX, even though R-MAX is one of the best known algorithms for these problems. Our simple model-based reinforcement learning algorithm is even able to perform better than R-MAX in some of the problems. Smart Start is applicable to continuous state and/or action spaces as well.

The source code is available on: https://github.com/BartKeulen/smartstart.

1-3 Thesis Outline

The thesis is structured as follows. In Chapter 2 preliminary information on reinforcement learning and several exploration strategies is given. Chapter 3 presents the Smart Start framework proposed in this thesis together with the implementation details using Q-Learning and Value Iteration. The experimental setup, experiments and results are discussed in Chapter 4. The final chapter, Chapter 6, presents the conclusions and recommendations for future work.

Chapter 2

Reinforcement Learning & Exploration

Reinforcement learning solves sequential decision making problems [2]. This is done by trial-and-error to gather experiences to learn from. An important aspect is the exploration-exploitation dilemma [3]. This chapter will provide the background information on reinforcement learning and various exploration strategies.

A few algorithms and exploration strategies will be discussed more thoroughly in this chapter. These algorithms will be used later on in this thesis by the Smart Start framework and/or the evaluation of Smart Start. The concerned algorithms are, Value Iteration, MBRL, Q-Learning and R-MAX. The exploration strategies being discussed are ϵ-greedy, Boltzmann and UCB1.

The chapter is structured as follows. Section 2-1 will define the reinforcement learning problem and notation. The most common methods for solving reinforcement learning problems try to learn a value function, Section 2-2 discusses value function methods. An important part of reinforcement learning is exploration, Section 2-3 presents several exploration strategies for reinforcement learning. A discussion is given at the end of the chapter, in Section 2-4.

2-1 Reinforcement Learning

At each time-step t the agent receives a representation of the environments state $\mathbf{s}_t \in \mathcal{S}$ and a reward $r_t \in \mathbb{R}$, where $\mathcal{S} \in \mathbb{R}^n$ is the set of possible states with arbitrary dimension n. At each state $\mathbf{s}_t$ the agent selects an action $\mathbf{a}_t \in \mathcal{A}(\mathbf{s}_t)$, where $\mathcal{A}(\mathbf{s}_t) \in \mathbb{R}^m$ is the set of possible actions in state $\mathbf{s}_t$ with arbitrary dimension m. As a result of performing action $\mathbf{a}_t$ the agent receives the new modified state $\mathbf{s}_{t+1}$ together with a reward r_{t+1}. Figure 2-1 shows how an agent interacts with its environment in a reinforcement learning problem.

In reinforcement learning the environment is modeled as a Markov decision process. A Markov decision process [1] consists of a set of states $\mathcal{S}$, set of actions $\mathcal{A}$, transition probabilities for each state-action-state triple $\mathcal{P}(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1})$, and the expected reward associated with each state-action pair $\mathcal{R}(\mathbf{s}_t, \mathbf{a}_t)$. The transition probability is defined as the probability to transfer from state, $\mathbf{s}_t$, under action, $\mathbf{a}_t$, to state, $\mathbf{s}_{t+1}$, and is defined as

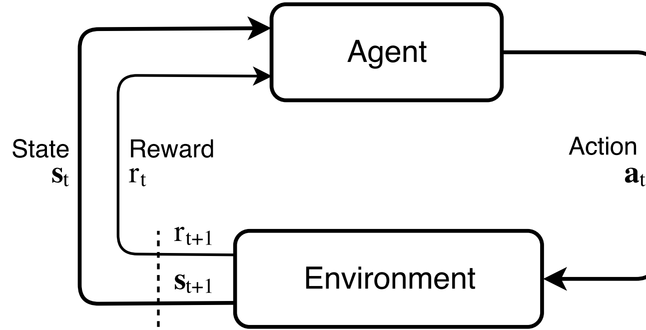


Figure 2-1: Agent environment interaction in reinforcement learning. At time-step t the agent receives a state $\mathbf{s}_t$ and reward r_t from the environment and executes an action $\mathbf{a}_t$ in response. The next time-step $t+1$ the environment emits a new state $\mathbf{s}_{t+1}$ and reward r_{t+1}. Source Barto and Sutton [2].

$$\mathcal{P}(\mathbf{s}, \mathbf{a}, \mathbf{s}') = \Pr\left[\mathbf{s}_{t+1} = \mathbf{s}' \mid \mathbf{s}_t = \mathbf{s}, \mathbf{a}_t = \mathbf{a}\right]. \tag{2-1}$$

The expected reward concerned with this transition is defined as

$$\mathcal{R}(\mathbf{s}, \mathbf{a}) = \mathbb{E}\left[r_{t+1} \mid \mathbf{s}_t = \mathbf{s}, \mathbf{a}_t = \mathbf{a}\right]. \tag{2-2}$$

A Markov decision processes has the Markov property. A system is said to have the Markov property if the current state $\mathbf{s}_{t+1}$ only depends on the previous state $\mathbf{s}_t$ and action $\mathbf{a}_t$. Which means the current state contains the information of all previous state and actions pairs

$$\Pr\left[\mathbf{s}_{t+1}, r_{t+1} \mid \mathbf{s}_t, \mathbf{a}_t\right] = \Pr\left[\mathbf{s}_{t+1}, r_{t+1} \mid \mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t-1}, \mathbf{a}_{t-1}, \ldots, \mathbf{s}_0, \mathbf{a}_0\right]. \tag{2-3}$$

The goal of the agent is to maximize the total accumulated reward, called the return R. Beforehand the return is unknown to the agent, therefore the agent tries to estimate the total reward it is going to receive. This estimate is called the expected return $J = \mathbb{E}[R]$. The simplest form uses the sum of rewards and is defined for problems in episodic setting with a final time-step T. For clarity we will also use T_{episode} instead of T later on in this thesis. For time-step t the return is defined as

$$R_t = r_{t+1} + r_{t+2} + \ldots + r_T = \sum_{k=0}^{T} r_{t+k+1}. \tag{2-4}$$

During each episode the agent tries to complete a task. The goal is to maximize the total reward of each episode. An episode can be terminated before it reaches the final time-step T, this happens for example when the goal is reached before the final time-step T.

Next to an episodic setting we have a continuous setting. In the continuous setting the task spans the whole life-time of the agent. Which means a final time-step of $T = \infty$, therefore

we cannot use the sum of rewards as return. A widely used solution is the discounted return, defined as

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \tag{2-5}$$

where γ is the discount rate with a value $0 \leq \gamma < 1$. Note that when $\gamma = 1$ we would get the sum of rewards with $T = \infty$. The discounted reward is often used in episodic problems as well, since rewards obtained later in the episode have less influence on the action that is currently executed.

The agent uses a policy π to select the actions. The policy π maps states to actions, i.e. it tells you which action to take in a certain state. The policy can either be stochastic or deterministic. A stochastic policy for state $\mathbf{s}$ is defined as the probability distribution

$$\pi(\mathbf{s}, \mathbf{a}) = \Pr\left[\mathbf{a}_t = \mathbf{a} \mid \mathbf{s}_t = \mathbf{s}\right] \quad \forall\, \mathbf{a} \in \mathcal{A}(\mathbf{s}), \tag{2-6}$$

where $\sum_{\mathbf{a} \in \mathcal{A}(\mathbf{s})} \pi(\mathbf{s}, \mathbf{a}) = 1$. A deterministic policy for state $\mathbf{s}$ is defined as

$$\mathbf{a} = \pi(\mathbf{s}). \tag{2-7}$$

2-2 Value Function Methods

Most reinforcement learning methods learn a value function. The value function estimates the expected return the agent is going to receive from a state. Basically the value function tells the agent how "good" a certain state or state-action pair is. Future rewards depend on the policy being executed, hence the value function depends on the policy being used.

For a state $\mathbf{s}$ and policy π the value function using the discounted return is defined as

$$V^\pi(\mathbf{s}) = \mathbb{E}_\pi\left[\sum_{k=0}^{T} \gamma^k r_{t+k+1} \mid \mathbf{s}_t = \mathbf{s}\right], \tag{2-8}$$

where $\mathbb{E}_\pi[\cdot]$ denotes the expected value when following policy π. The value function $V(\mathbf{s})$ only depends on the state $\mathbf{s}$ and is called the state-value function.

We can also estimate the expected return for state-action pairs, i.e. taking action $\mathbf{a}$ in state $\mathbf{s}$ and following policy π

$$Q^\pi(\mathbf{s}, \mathbf{a}) = \mathbb{E}_\pi\left[\sum_{k=0}^{T} \gamma^k r_{t+k+1} \mid \mathbf{s_t} = \mathbf{s}, \mathbf{a}_t = \mathbf{a}\right]. \tag{2-9}$$

$Q(\mathbf{s}, \mathbf{a})$ is called the action-value function.

With value function methods the agent tries to learn the value function from real experiences. The policy is subsequently derived from the value function. To see how the value function is estimated from experiences we first have to introduce an important property of the value function, which is the Bellman equation [1] and is defined as follows

$$\begin{aligned} V^{\pi}(\mathbf{s}) &= \mathbb{E}_{\pi}\left[R_t \mid \mathbf{s}_t = \mathbf{s}\right] \\ &= \sum_{\mathbf{a}\in\mathcal{A}(\mathbf{s})} \pi(\mathbf{s},\mathbf{a}) \sum_{\mathbf{s}'} \mathcal{P}(\mathbf{s},\mathbf{a},\mathbf{s}') \left[\mathcal{R}(\mathbf{s},\mathbf{a}) + \gamma V^{\pi}(\mathbf{s}')\right]. \end{aligned} \tag{2-10}$$

Equation (2-10) is the Bellman equation for V^{π}. For Q^{π} the following Bellman equation is obtained

$$Q^{\pi}(\mathbf{s},\mathbf{a}) = \sum_{\mathbf{s}'} \mathcal{P}(\mathbf{s},\mathbf{a},\mathbf{s}') \left[\mathcal{R}(\mathbf{s},\mathbf{a}) + \gamma V^{\pi}(\mathbf{s}')\right]. \tag{2-11}$$

The recursive nature of the Bellman equation gives rise to some interesting properties. From Equations (2-10) and (2-11) we can see that the value function has to be consistent with the policy. So when we have an optimal policy, the value function is optimal as well. The same holds the other way around. We can now write down an equation for the optimal value function called the Bellman optimality equation [1]. The Bellman optimality equation for the state- and action-value functions are defined as

$$V^{*}(\mathbf{s}) = \max_{\mathbf{a}\in\mathcal{A}(\mathbf{s})} \sum_{\mathbf{s}'} \mathcal{P}(\mathbf{s},\mathbf{a},\mathbf{s}') \left[\mathcal{R}(\mathbf{s},\mathbf{a}) + \gamma V^{*}(\mathbf{s}')\right]. \tag{2-12}$$

$$Q^{*}(\mathbf{s},\mathbf{a}) = \sum_{\mathbf{s}'} \mathcal{P}(\mathbf{s},\mathbf{a},\mathbf{s}') \left[\mathcal{R}(\mathbf{s},\mathbf{a}) + \gamma \max_{\mathbf{a}'\in\mathcal{A}(\mathbf{s}')} Q^{*}(\mathbf{s}',\mathbf{a}')\right]. \tag{2-13}$$

Where the * denotes optimality. Value function methods estimate the value function using the properties described above. Subsequently they derive the policy from the value function. Three main methods exist for solving this problem; dynamic programming, Monte Carlo methods and temporal difference methods. Dynamic programming assumes full knowledge of the environment, i.e., $\mathcal{P}$ and $\mathcal{R}$ are available to the agent. Monte Carlo methods and temporal difference methods do not need a model and estimate the value function from real experiences. The next two sections discuss dynamic programming and temporal difference methods.

2-2-1 Dynamic Programming

Dynamic programming [1] requires full knowledge of the environment. Here, full knowledge means that the transition model $\mathcal{P}$ and reward function $\mathcal{R}$ are known. No real experience is required when solving the problem using dynamic programming. Dynamic programming is often considered a planning method, because the model is known and no real experience is required.

Dynamic programming consists of two parts, policy evaluation and policy improvement. Policy evaluation evaluates the current policy by approximating the value function V^π. This is done by iteratively updating the approximation of V^π using the Bellman equation (2-10). During each iteration k, the agent updates the value function V_k for each state $\mathbf{s} \in \mathcal{S}$ as follows

$$V_{k+1}(\mathbf{s}) = \sum_{\mathbf{a}\in\mathcal{A}(\mathbf{s})} \pi(\mathbf{s},\mathbf{a}) \sum_{\mathbf{s}'} \mathcal{P}(\mathbf{s},\mathbf{a},\mathbf{s}') \left[\mathcal{R}(\mathbf{s},\mathbf{a}) + \gamma V_k(\mathbf{s}')\right]. \tag{2-14}$$

For $k \to \infty$ the sequence V_k will converge to V^π. After finding the value function V^π we can subsequently use it to obtain an improved policy π'. For each state $\mathbf{s}$ we can evaluate which action $\mathbf{a} \in \mathcal{A}(\mathbf{s})$ results in the highest value function value in the next state $\mathbf{s}'$. We can compare the value function of each action using the Bellman equation for the action-value function (2-11). Since the value function is consistent with the policy we can say the following

$$Q^{\pi'}(\mathbf{s}, \pi'(\mathbf{s})) \geq V^\pi(\mathbf{s}). \tag{2-15}$$

This means that the we can always find a new policy, π', that is better or as good as the current policy, π, in state $\mathbf{s}$. From the action-value function Q we can derive the new policy for all states $\mathbf{s} \in \mathcal{S}$

$$\begin{aligned} \pi'(\mathbf{s}) &= \arg \max_{\mathbf{a}\in\mathcal{A}(\mathbf{s})} Q^\pi(\mathbf{s},\mathbf{a}) \\ &= \arg \max_{\mathbf{a}\in\mathcal{A}(\mathbf{s})} \sum_{\mathbf{s}'} \mathcal{P}(\mathbf{s},\mathbf{a},\mathbf{s}') \left[\mathcal{R}(\mathbf{s},\mathbf{a}) + \gamma V^\pi(\mathbf{s}')\right]. \end{aligned} \tag{2-16}$$

The new policy π' can be evaluated using the policy evaluation scheme described in Equation (2-14). Following the evaluation step the policy can be improved again. This cycle of policy evaluation and policy improvement is repeated until the policy has converged, i.e., the policy and value function do not change anymore.

Value Iteration [1] is a dynamic programming algorithm that performs a single iteration during the policy evaluation step instead of iterating until the value function converged. This enables the algorithm to combine the policy evaluation and improvement step resulting in the following update

$$V_{k+1}(\mathbf{s}) = \max_{\mathbf{a}\in\mathcal{A}(\mathbf{s})} \sum_{\mathbf{s}'} \mathcal{P}(\mathbf{s},\mathbf{a},\mathbf{s}') \left[\mathcal{R}(\mathbf{s},\mathbf{a}) + \gamma V_k(\mathbf{s}')\right] \quad \forall\ \mathbf{s} \in \mathcal{S}. \tag{2-17}$$

The full Value Iteration algorithm is given in Algorithm 1. A termination condition has to be given to the algorithm. When the value function does not change more then a threshold θ we say the algorithm has converged, where θ is a small positive number that has to be chosen accordingly. The optimization is terminated when the value function does not converge within a maximum number of iterations. The maximum number of iterations has to be chosen by the user.

Algorithm 1 Value Iteration

Initialize V arbitrarily for all states $\mathbf{s} \in \mathcal{S}$

repeat
  $\Delta = 0$
  for $\mathbf{s} \in \mathcal{S}$ **do**
    $v = V_i(\mathbf{s})$
    $V_{i+1}(\mathbf{s}) = \max\limits_{\mathbf{a} \in \mathcal{A}(\mathbf{s})} \sum_{\mathbf{s}' \in \mathcal{S}} \mathcal{P}(\mathbf{s}, \mathbf{a}, \mathbf{s}') \left[\mathcal{R}(\mathbf{s}, \mathbf{a}) + \gamma V_i(\mathbf{s}')\right]$
    $\Delta = \max\left(\Delta, |v - V_i(\mathbf{s})|\right)$
until $\Delta < \theta$ (small positive number)

// Output deterministic policy π
$\pi(\mathbf{s}) = \arg\max\limits_{\mathbf{a} \in \mathcal{A}(\mathbf{s})} \sum_{\mathbf{s}' \in \mathcal{S}} \mathcal{P}(\mathbf{s}, \mathbf{a}, \mathbf{s}') \left[\mathcal{R}(\mathbf{s}, \mathbf{a}) + \gamma V(\mathbf{s}')\right]$

2-2-2 Model-Based Reinforcement Learning

Reinforcement learning does not require any knowledge of the environment. When the full model of the environment is known, we are not solving a reinforcement learning problem anymore but a planning problem, e.g. dynamic programming. Instead we can use gathered data to learn a model and use the model to improve our learning process, called model-based reinforcement learning [18]. Often the model is used for planning or generating simulated experience, e.g., the Dyna architecture [19]. When the agent has learned an accurate model it does not need to gather new experiences to improve its policy, instead the model can be used directly by a planning method.

For discrete environments a transition model and reward function can be constructed easily. These functions are learned by keeping track of visitation counts of the number of times an action $\mathbf{a}$ has been executed in state $\mathbf{s}$, denoted by $C(\mathbf{s}, \mathbf{a})$, and of the number of times executing action $\mathbf{a}$ in state $\mathbf{s}$ resulted in traversing to state $\mathbf{s}'$, denoted by $C(\mathbf{s}, \mathbf{a}, \mathbf{s}')$. For the reward function a sum of rewards for each state-action pair is kept, $R_{\text{sum}}(\mathbf{s}, \mathbf{a}) = \sum R(\mathbf{s}, \mathbf{a})$. We can now construct the transition model and reward function

$$\mathcal{P}(\mathbf{s}, \mathbf{a}, \mathbf{s}') = \frac{C(\mathbf{s}, \mathbf{a}, \mathbf{s}')}{C(\mathbf{s}, \mathbf{a})}, \tag{2-18}$$

$$\mathcal{R}(\mathbf{s}, \mathbf{a}) = \frac{R_{\text{sum}}(\mathbf{s}, \mathbf{a})}{C(\mathbf{s}, \mathbf{a})}. \tag{2-19}$$

The learned transition model and reward function can for example be used by a dynamic programming method, e.g., Value Iteration, to construct a policy. An implementation of a simple model-based reinforcement learning algorithm is given in Algorithm 2. We will refer to this algorithm as MBRL.

Algorithm 2 MBRL

1: Initialize policy π randomly
2:
3: **for** each episode **do**
4: Initialize s_0 and $t = 0$
5: **repeat**
6: Choose $\mathbf{a}_t$ according to π and some exploration strategy (e.g., ϵ-greedy)
7: Take action $\mathbf{a}_t$ and observe $\mathbf{s}_{t+1}$ and r_{t+1}
8: Increment $C(\mathbf{s}_t)$, $C(\mathbf{s}_t, \mathbf{a}_t)$ and $C(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1})$
9: Update $R_{\text{sum}}(\mathbf{s}_t, \mathbf{a}_t) = R_{\text{sum}}(\mathbf{s}_t, \mathbf{a}_t) + r_{t+1}$
10:
11: *// Update transition model and reward function*
12: $\mathcal{R}(\mathbf{s}_t, \mathbf{a}_t) = \frac{R_{\text{sum}}(\mathbf{s}_t, \mathbf{a}_t)}{C(\mathbf{s}_t, \mathbf{a}_t)}$
13: **for all** $\mathbf{s}' \in C(\mathbf{s}, \mathbf{a}, \cdot)$ **do**
14: $\mathcal{P}(\mathbf{s}, \mathbf{a}, \mathbf{s}') = \frac{C(\mathbf{s}, \mathbf{a}, \mathbf{s}')}{C(\mathbf{s}, \mathbf{a})}$
15:
16: *// Obtain policy using value iteration*
17: $\pi = \text{VALUE ITERATION}(\mathcal{P}, \mathcal{R})$ (See Algorithm 1)
18:
19: $t \leftarrow t + 1$
20: **until** $\mathbf{s_t}$ is terminal or $t = T_{\text{episode}}$

2-2-3 Temporal Difference

Temporal difference methods do not require any knowledge about the environment. Temporal difference methods estimate the value function using experiences obtained in the environment. The value function tries to approximate the expected return. Which can be seen as a minimization problem, where the objective is to minimize the error between the expected return and the value function. This can be written as an incremental update rule

$$V_{t+1}(\mathbf{s}_t) = V_t(\mathbf{s}_t) + \alpha \left[R_t - V_t(\mathbf{s}_t) \right], \tag{2-20}$$

where α is a positive learning rate. The return R_t is not available until the end of an episode. We can estimate the return R_t using Equation (2-10)

$$V^\pi(\mathbf{s}) = \mathbb{E}_\pi \left[R_t \mid \mathbf{s}_t = \mathbf{s} \right] = \mathbb{E}_\pi \left[r_{t+1} + \gamma V^\pi(\mathbf{s}_{t+1}) \mid \mathbf{s}_t = \mathbf{s} \right]. \tag{2-21}$$

If we substitute Equation (2-21) in Equation (2-20) we get the simplest form of temporal difference learning, called TD(0)

$$V_{t+1}(\mathbf{s}_t) = V_t(\mathbf{s}_t) + \alpha \left[r_{t+1} + \gamma V_t(\mathbf{s}_{t+1}) - V_t(\mathbf{s}_t) \right], \tag{2-22}$$

The first two terms between the square brackets, $r_{t+1} + \gamma V(\mathbf{s}_{t+1})$, is called the target. All the terms between the brackets is called the temporal difference error, $\delta_t = r_{t+1} + \gamma V_t(\mathbf{s}_{t+1}) -$

$V_t(\mathbf{s}_t)$. The update is based on an estimate of successor states, this is called bootstrapping. Bootstrapping is seen in many reinforcement learning algorithms, e.g., dynamic programming and temporal difference methods.

Q-Learning [2] is a commonly used method for control in discrete reinforcement learning problems. For control the action-value function is used instead of the state-value function. The update used in Q-Learning is defined as

$$Q_t(\mathbf{s}_t, \mathbf{a}_t) = Q_t(\mathbf{s}_t, \mathbf{a}_t) + \alpha \left[r_{t+1} + \gamma \max_{\mathbf{a}' \in \mathcal{A}(\mathbf{s}_{t+1})} Q_t(\mathbf{s}_{t+1}, \mathbf{a}') - Q_t(\mathbf{s}_t, \mathbf{a}_t) \right]. \tag{2-23}$$

In the update the value of the next state is bootstrapped. For TD(0) this is just the state-value function, but when used for control we have an action-value for each action in the next state. Q-Learning choses the maximum value for bootstrapping, this is called off-policy learning. A different approach can be to use the policy to determine which action-value is used for bootstrapping. This is called on-policy learning, e.g., SARSA [2] is the on-policy version of Q-Learning and uses $Q_t(\mathbf{s}_{t+1}, \pi(\mathbf{s}_{t+1}))$ for bootstrapping. A policy can easily be derived from the action-value function according to

$$\pi(\mathbf{s}) = \arg \max_{\mathbf{a} \in \mathcal{A}(\mathbf{s})} Q(\mathbf{s}, \mathbf{a}). \tag{2-24}$$

The full Q-Learning algorithm is given in Algorithm 3.

Algorithm 3 Q-Learning

Initialize Q arbitrarily for all $\mathbf{s} \in \mathcal{S}$ and $\mathbf{a} \in \mathcal{A}(\mathbf{s})$

for each episode **do**
  Initialize s_0 and $t = 0$
  repeat
    Choose $\mathbf{a}_t$ according to some policy π derived from Q (e.g., ϵ-greedy)
    Take action $\mathbf{a}_t$ and observe $\mathbf{s}_{t+1}$ and r_{t+1}
    $Q(\mathbf{s}_t, \mathbf{a}_t) = Q(\mathbf{s}_t, \mathbf{a}_t) + \alpha \left[r_{t+1} + \gamma \max_{\mathbf{a}' \in \mathcal{A}(\mathbf{s}_{t+1})} Q(\mathbf{s}_{t+1}, \mathbf{a}') - Q(\mathbf{s}_t, \mathbf{a}_t) \right]$
    $t \leftarrow t + 1$
  until $\mathbf{s}_t$ is terminal or $t = T_{\text{episode}}$

2-3 Exploration Strategies

Reinforcement learning agents learn from experiences. In order to evaluate if a certain policy is good the agent has to try out different policies. By just performing the same policy over and over the agent will never learn, therefore exploration is needed. Exploration is an import aspect of reinforcement learning. For exploration, you have to execute actions that may result in poor rewards. Therefore a typical reinforcement learning agent alternates between

exploiting its current best solution and exploring for a better solution. This is known as the exploration-exploitation dilemma [3].

The exploration strategy used has a huge influence on the sample efficiency of the learning process. With a conservative exploration strategy the agent will always see the same states and not learn a lot. On the other hand, a too aggressive exploration strategy will visit new states all the time, but does not give the agent enough time to learn from them. All these factors have to be weighted in the exploration strategy and each problem may require a different exploration strategy, making it a difficult but important problem.

A lot of exploration strategies exist. This section will discuss some different exploration strategies. Exploration strategies can be divided in two groups, undirected and directed exploration [4], which will be discussed in Sections 2-3-1 and 2-3-2 respectively. Section 2-3-3 discusses model-based exploration.

2-3-1 Undirected Exploration

Undirected exploration uses a random process for exploration. Undirected exploration does not use any of the knowledge obtained during the learning process to improve the exploration. The idea is that given enough time the random process will explore throughout the state space. However, this may take a large amount of time, which can grow exponentially with the number of states [32]. The most commonly known methods for undirected exploration are ϵ-greedy and Boltzmann exploration.

ϵ-greedy is one of the most used exploration strategies. In ϵ-greedy there is an ϵ chance the agent will perform a random action and a $1 - \epsilon$ chance of following the current policy. A larger value for ϵ means more random actions are being executed, hence more exploration. Whereas a lower value for ϵ results in more exploitation.

Instead of using a fixed ϵ for the whole training one can start with a high value for ϵ and gradually reduce it during the learning process. This results in more exploration in the beginning of the learning process and more exploitation at the end.

Boltzmann exploration creates a probability distribution over the possible actions. Each action gets assigned a probability based on the associated value function. An action is then sampled according to the probability distribution. The probability for each action is calculated using the soft-max function

$$\pi(\mathbf{s}, \mathbf{a}) = \frac{e^{Q(\mathbf{s},\mathbf{a})/\tau}}{\sum_{\mathbf{a}' \in \mathcal{A}(\mathbf{s})} e^{Q(\mathbf{s},\mathbf{a}')/\tau}} \tag{2-25}$$

The parameter τ, called the temperature, is used to vary the amount of exploration. When $\tau \to \infty$ the distribution becomes more uniform, resulting in a lot of randomness (exploration). For $\tau \to 0$ the difference between probabilities becomes more extreme resulting in a greedy policy (exploitation). The parameter τ can be reduced gradually during the learning process to decrease the amount of exploration.

2-3-2 Directed Exploration

Directed exploration uses knowledge about the learning process to improve exploration. The knowledge is gained through previous experiences. Where undirected exploration only looks at the current time-step, directed exploration utilizes the history of the agent to make better decisions.

UCB1 [12] is a directed exploration method. The UCB1 algorithm, developed for multi-armed bandit problems, uses the upper confidence bound (UCB) to make the most informed decision on what action to choose. A more detailed explanation of the UCB1 algorithm is given in Section 3-2. Kocsis and Szepesvári transformed the algorithm to work with Monte Carlo Planning, called UCT [14]. The UCT algorithm can be used in combination with reinforcement learning problems, the UCT policy is described by the following equation

$$\pi(\mathbf{s}) = \arg\max_{\mathbf{a}\in\mathcal{A}(\mathbf{s})} \left[Q(\mathbf{s},\mathbf{a}) + 2C_p \sqrt{\frac{\log(C(\mathbf{s}))}{C(\mathbf{s},\mathbf{a})}} \right], \tag{2-26}$$

where C_p is an appropriate constant based on the rewards in the environment. $C(\mathbf{s})$ and $C(\mathbf{s},\mathbf{a})$ are the visitation counts, where $C(\mathbf{s})$ is the number of times state $\mathbf{s}$ has been visited and $C(\mathbf{s},\mathbf{a})$ the number of times action $\mathbf{a}$ has been executed in state $\mathbf{s}$. For $C(\mathbf{s}) = 0$, the exploration bonus is defined to be 0. And for $C(\mathbf{s},\mathbf{a}) = 0$ the exploration bonus is ∞. The UCB1 algorithm acts according to an optimistic guess of the value function, this is called "optimism in the face of uncertainty". The UCB1 algorithm gives a bonus for rarely visited state-action pairs. But as $C(\mathbf{s},\mathbf{a})$ increases, the bonus quickly converges to zero. This ensures the agent stops exploring suboptimal actions once it knows enough about those actions.

2-3-3 Model-Based Exploration

Model-based reinforcement learning can improve the exploration by exploiting the learned model for generating exploration policies. Often these policies drive the agent to areas where the model uncertainty is high, in order to learn an accurate model as fast as possible.

R-MAX [20] is such a model-based reinforcement learning algorithm. A model of the environment is learned, which is subsequently used in combination with Value Iteration to plan a policy.

The transition model and reward function are learned according to Equations (2-18) and (2-19) respectively. For stochastic environments a good transition model for a state-pair cannot be constructed using a single sample. Therefore R-MAX requires the count $C(\mathbf{s},\mathbf{a}) \geq m$ for arbitrary $m \in \mathbb{R}$. This threshold ensures that the agent can construct a good model before it uses the model for planning and it makes the agent explore the state space more thoroughly. R-MAX also utilizes uncertainty in the face of optimism. This is done by introducing an absorbing state $\mathbf{s}_r$. Unknown transitions, i.e., $C(\mathbf{s},\mathbf{a}) < m$, lead to the absorbing state and have a transition probability $P(\mathbf{s},\mathbf{a},\mathbf{s}_r) = 1$ and a reward function $R(\mathbf{s},\mathbf{a}) = R_{\max}$. This results in exploring state-action pairs at least m times before exploiting. The full algorithm can be found in Algorithm 4.

Algorithm 4 R-MAX

1: Initialize policy π randomly
2: Initialize $R_{\max}$ and absorbing state $\mathbf{s}_r$
3:
4: **for** $\mathbf{a} \in \mathcal{A}(\mathbf{s}_r)$ **do**
5: $\quad \mathcal{R}(\mathbf{s}_r, \mathbf{a}) = R_{\max}$
6: $\quad \mathcal{P}(\mathbf{s}_r, \mathbf{a}, \mathbf{s}_r) = 1$
7:
8: **for** each episode **do**
9: $\quad$ Initialize $\mathbf{s}_0$ and $t = 0$
10: $\quad$ **repeat**
11: $\quad\quad$ Choose $\mathbf{a}_t = \pi(\mathbf{s}_t)$
12: $\quad\quad$ Take action $\mathbf{a}_t$ and observe $\mathbf{s}_{t+1}$ and r_{t+1}
13: $\quad\quad$ Increment $C(\mathbf{s}_t, \mathbf{a}_t)$ and $C(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1})$
14: $\quad\quad$ Update $R_{\text{sum}}(\mathbf{s}_t, \mathbf{a}_t) = R_{\text{sum}}(\mathbf{s}_t, \mathbf{a}_t) + r_{t+1}$
15:
16: $\quad\quad$ **if** $C(\mathbf{s}_t, \mathbf{a}_t) \geq m$ **then**
17: $\quad\quad\quad$ *// Known state, update model*
18: $\quad\quad\quad$ $\mathcal{R}(\mathbf{s}_t, \mathbf{a}_t) = \frac{R_{\text{sum}}(\mathbf{s}_t, \mathbf{a}_t)}{C(\mathbf{s}_t, \mathbf{a}_t)}$
19: $\quad\quad\quad$ **for all** $\mathbf{s}' \in C(\mathbf{s}_t, \mathbf{a}_t, \cdot)$ **do**
20: $\quad\quad\quad\quad$ $\mathcal{P}(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}') = \frac{C(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}')}{C(\mathbf{s}_t, \mathbf{a}_t)}$
21: $\quad\quad$ **else**
22: $\quad\quad\quad$ *// Unknown state, set optimistic transition*
23: $\quad\quad\quad$ $\mathcal{R}(\mathbf{s}_t, \mathbf{a}_t) = R_{\max}$
24: $\quad\quad\quad$ $\mathcal{P}(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_r) = 1$
25:
26: $\quad\quad$ *Update policy*
27: $\quad\quad$ $\pi = \text{ValueIteration}(\mathcal{P}, \mathcal{R})$ (See Algorithm 1)
28:
29: $\quad\quad$ $t \leftarrow t + 1$
30: $\quad$ **until** $\mathbf{s}_t$ is terminal or $t = T_{\text{episode}}$

2-4 Discussion

Chapter 2 presented the general theory behind reinforcement learning [2] and several methods for solving the reinforcement learning problem. When a model of the environment is available dynamic programming [1] can be used to find an optimal policy, dynamic programming is often considered to be a planning method. Without a model, temporal difference methods can be used. Q-Learning [2] is one of the most well known reinforcement learning method that uses temporal difference learning.

An important characteristic of reinforcement learning is the exploration-exploitation dilemma [3]. Exploration strategies are often divided in two groups, undirected and direct exploration [4]. Undirected exploration uses randomness as exploration strategy, e.g., ϵ-greedy and Boltzmann exploration. Directed exploration uses knowledge about the learning process to improve exploration. UCB1 [12] gives an exploration bonus for actions that have been executed infrequently. UCB1 falls under the "optimism in the face of uncertainty" principle. Another group of exploration strategies is model-based exploration [18], which is part of the directed exploration group. A model of the environment is constructed and used to determine which areas need more exploration, for example based on the model error. R-MAX [20] is such a model-based method.

A problem with most exploration strategies is that they only look at the current state for their exploration policy and do not perform persistent exploration. ϵ-greedy has a small chance of choosing a random exploratory action, UCB1 on the other hand adds an exploration bonus for actions that have been executed infrequently. This mainly results in exploration close to the current policy. UCB1 is smarter compared to ϵ-greedy in making the trade-off between exploration and exploitation, because it uses knowledge about the learning process. But for environments with sparse rewards UCB1 can get stuck in areas that have already been explored thoroughly, this becomes clear if we look at the policy in Equation (2-26). The first term Q will be zero as long as no reward has been received. The action chosen will therefore depend solely on the second term. But $C(\mathbf{s}, \mathbf{a})$ will be equal for several or all actions most of the time. This results in random walk behavior, just like with undirected exploration. This suggests that more intelligent exploration methods are required. R-MAX solves this problem by learning a model and reward function of the environment. The reward function is constructed in such a way that the agent is guided to regions that need more exploration. But R-MAX does need to learn a global model that is accurate enough to be used by a planning method. For small discrete environments this is trivial but for more complex environments with continuous state spaces this is nontrivial [24, 25]. Due to modeling errors, model-based methods often achieve suboptimal performance compared to model-free methods [25]. Furthermore, R-MAX exhaustively explores the full state space and tries each state-action pair a minimum number of times. This exhaustive search can be infeasible or undesirable, especially for environments with large state and/or action spaces.

This suggests a model-free method is wanted for achieving optimal results, whereas the exploration and sample-efficiency of model-based reinforcement learning is desired. The next chapter introduce the main contribution of this work, a directed and persistent exploration framework called Smart Start. The Smart Start framework can be incorporated with any reinforcement learning algorithm to give it directed and persistent exploration.

Chapter 3

Smart Start

Sections 2-3 and 2-4 discussed several different exploration strategies and issues encountered in complex domains. We will now introduce a novel framework for more efficient exploration called Smart Start. Smart Start is developed for episodic problems that start each episode from the same initial state and have a maximum number of time-steps, denoted by T_{episode}. Smart Start tries to solve the problem in a different way and is not a self contained exploration strategy but an addition that can be used in combination with other exploration strategies. Smart Start can be combined with any reinforcement learning algorithm and only guides the agent to a region from which it expects to learn the most new information. This region is denoted by a single state, called the Smart Start state. Therefore Smart Start does not alter the performance of the reinforcement learning algorithm it is used with, but it does give the algorithm more persistent and directed exploration.

There are different ways for choosing the Smart Start state, depending on how you define a good region for exploration or what method you use for choosing the Smart Start state itself. For getting to the Smart Start state there are also several methods, e.g., trajectory replay, trajectory optimization or setting up a separate reinforcement learning problem. In this chapter we will discuss the implementation for discrete systems. We use the UCB1 algorithm for choosing the Smart Start state and dynamic programming for the guiding to the Smart Start state.

First the Smart Start method is introduced in Section 3-1. Smart Start can be divided in two parts. The first part is determining the Smart Start state, i.e., the state we want to start exploring from. The Smart Start state will be discussed in Section 3-2. The second part is guiding the agent to the Smart Start state, discussed in Section 3-3. Section 3-4-1 describes an how a full algorithm with Smart Start looks and gives an example with Q-Learning and Value Iteration. Finally, we end this chapter with a discussion in Section 3-5.

3-1 Method

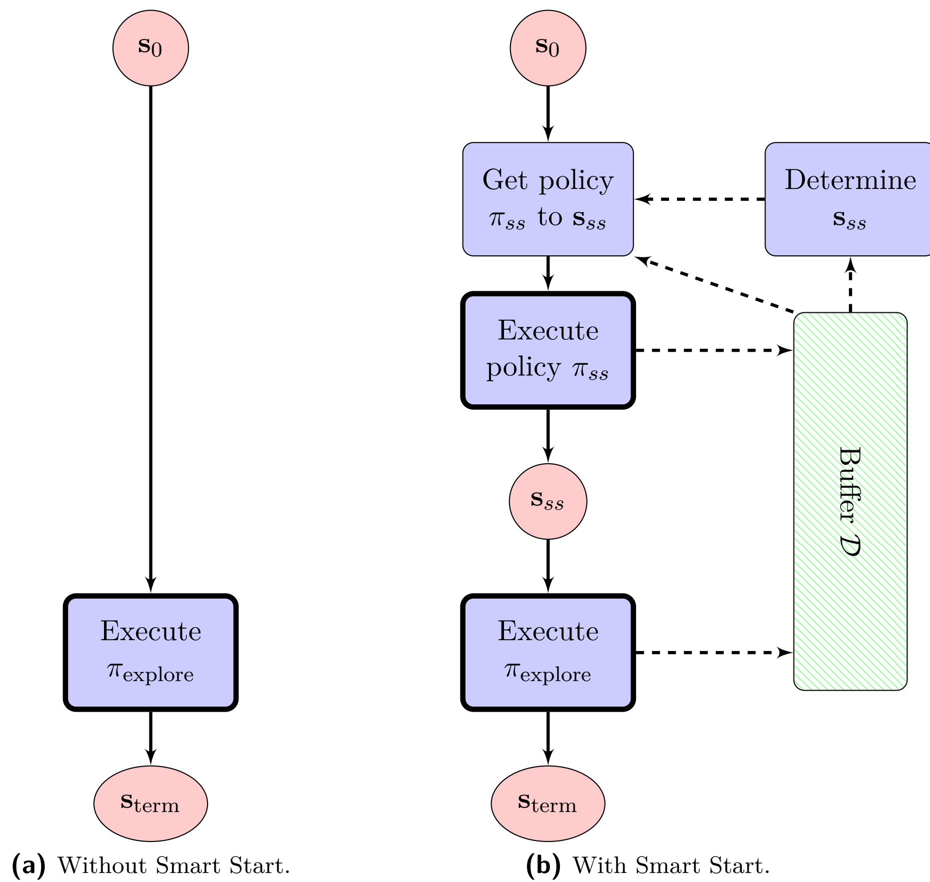
For reinforcement learning problems with an episodic setting, the reinforcement learning agent executes its learned policy together with some exploration strategy from the start until the end of an episode. This often results in staying close to the current policy and exploring locally around it, lacking persistency in its exploration. An episode using Smart Start has a different start of an episode. Instead the agent determines a region where it expects to gain the most information. The region is denoted by a single state, called the Smart Start state, $\mathbf{s}_{ss}$. Subsequently, the agent is guided to the Smart Start state. There are several choices on how to get to the Smart Start state. For example, do you want to get there as fast as possible, without considering regions of low rewards, or do you want to have a high probability of reaching the Smart Start state. A trajectory optimization strategy has to be chosen accordingly, depending on what you want to achieve. Once the agent is in the region of the Smart Start state the learning continues by executing the agent's learned policy and some exploration strategy, e.g., ϵ-greedy or UCB1. In a lot of problems neighboring states are similar, it may also be difficult to get exactly to the Smart Start state, for example in continuous state spaces you never visit exactly the same state twice. Therefore it is sufficient to get near the Smart Start state instead of exactly in it, this does require a metric that describes the relation between states.

Figure 3-1 shows a flow chart for normal reinforcement learning versus reinforcement learning with Smart Start. In normal reinforcement learning, the agent begins in the initial state $\mathbf{s}_0$ and executes its learned policy using some exploration strategy, called π_{explore}, until the episode ends in a terminal state $\mathbf{s}_{\text{term}}$ or when T_{episode} is reached. With Smart Start, the agent first determines the Smart Start state $\mathbf{s}_{ss}$ and tries to find a policy π_{ss} to $\mathbf{s}_{ss}$ using previous experiences. The policy π_{ss} is executed until the agent is close to the $\mathbf{s}_{ss}$ and subsequently the agent executes the learned policy π_{explore} until the episode ends in a terminal state $\mathbf{s}_{\text{term}}$ or when T_{episode} is reached.

For environments with sparse or misleading rewards most normal learning reinforcement learning agents spend a lot of time re-exploring parts of the state space they have already visited or the agent is not persistent enough to explore the state space sufficiently. Smart Start on the other hand uses the information at hand to determine what region is expected to yield the most new information and goes to this region as reliable and quickly as possible. Smart Start is used as extension for reinforcement learning algorithms, therefore any algorithm that is augmented with Smart Start retains its learning performance.

3-2 Smart Start State

The first part of the Smart Start method is determining the Smart Start state. We choose a Smart Start state that is reachable. We define a state reachable when it has been visited at least once by the agent. Therefore, Smart Start can not be used in the first episode, because the agent needs to collect data about the environment first. The number of states grows exponentially in the number of dimensions, which is commonly known as the curse of dimensionality [1]. For many large problems, the number of states the agent can visit during its lifetime is very small compared to all the states in state space. By limiting the set of possible Smart Start states to visited states only, Smart Start does not suffer from the curse



(a) Without Smart Start. **(b)** With Smart Start.

Figure 3-1: Flow chart for a reinforcement learning episode with and without Smart Start. Solid lines depict how the agent travels through the chart. Dashed lines indicate fdata flows, e.g., a state that is being stored in the buffer. States are denote by a red circle. Purple rectangles display steps within each algorithm. Rectangles with a thick border represent the steps in which the agent is learning. The green striped rectangle depicts the buffer.

of dimensionality. We keep track of visited states in a buffer $\mathcal{D} = \{\mathbf{s}_0, \mathbf{s}_1, \ldots\}$, comparable to the replay buffer [33] used in for example DQN [34] and DDPG [6]. All the states in the buffer automatically qualify as possible Smart Start states. The buffer itself is of finite size and uses some strategy for replacing states in the buffer, e.g., first-in first-out (FIFO) or distribution based experience retention [35].

The goal of a reinforcement learning agent is to maximize the return, so naturally we want the Smart Start state to have a high probability of maximizing the return. But more importantly, we want to explore the state space as quickly and efficiently as possible in order to find the optimal solution as fast as possible. There are many ways for choosing the Smart Start state and a balance between these two criteria has to be found, maximizing the return but at the same time exploring sufficiently in order to obtain the best result. Multi-armed bandit problems [12] are concerned with these types of problems. The Smart Start state selection can be modeled as a multi-armed bandit problem, for which lots of algorithms exist that have a near optimal balance between exploration and exploitation.

3-2-1 Multi-Armed Bandit Problem

The name multi-armed bandit, also known as a K-armed bandit, comes from gambling on slot-machines, where the goal is to maximize the total return. Each turn the user has to choose which lever to pull (action) out of K different slot-machines that results in the highest pay-off (reward). The reward is a random variables $X(i) \in [0, 1]$ for $i = 1, \ldots, K$ and distributed according to an unknown distribution with expectation $\mu(i) = \mathbb{E}[X(i)]$. Each turn $t \geq 1$ an action i is executed and a random variable $X_t(i)$ is obtained. Successive plays of machine i yield independent, identically distributed rewards $X_1(i), X_2(i), \ldots, X_t(i)$. A policy π tells us which action to execute at time t based on the sequence of actions and pay-offs. Let $T_n(i)$ be the number of times machine i has been played by policy π during the first n plays. The expected regret of a policy π after n plays is defined as

$$\mathfrak{R}_n = \mu^* n - \sum_{j=1}^{K} \mu(j)\mathbb{E}[T_n(j)], \tag{3-1}$$

where $\mu^* = \max_i[\mu(i)]$ for $i = 1, \ldots, K$. The goal is to find a policy that minimizes the expected regret. Lai and Robbins [36] showed that for a large class of pay-off distributions the regret cannot grow slower than $O(\ln n)$ for all policies.

The UCB1 algorithm [12] is an algorithm whose regret growth rate has been proven to be within a constant factor of the best possible regret rate. The UCB1 algorithm calculates an upper confidence bound for each action i consisting of two terms. The first term is the average reward $\bar{X}_{T_n(i)}(i)$ obtained for action i, we write $\bar{X}_t(i) = \bar{X}_{T_n(i)}(i)$ for shorthand of notation. The second term is a bias term. The bias is chosen according to Hoeffding's inequality [13], to ensure that the true expected reward falls within the upper confidence bound with high probability. An action I is chosen according to the best upper confidence bound

$$I_{t+1} = \arg\max_i \left[\bar{X}_t(i) + c_t(i)\right], \tag{3-2}$$

where I_{t+1} denotes the action chosen at time-step $t+1$ and

$$c_t(i) = \sqrt{\frac{2\log \sum_{j=1}^{K} T_t(j)}{T_t(i)}}. \tag{3-3}$$

Auer et al. [12] showed the probability of $\bar{X}_t(i)$ being outside the upper confidence bound is bounded according to

$$\Pr\left[\bar{X}_t(i) \geq \mu_i + c_t(i)\right] \leq t^{-4}, \tag{3-4}$$

$$\Pr\left[\bar{X}_t(i) \leq \mu_i - c_t(i)\right] \leq t^{-4}. \tag{3-5}$$

3-2-2 Discrete Smart Start State

In the Smart Start state selection we are looking for the best state $\mathbf{s} \in \mathcal{D}$ to start exploring from instead of what machine i to pull the lever from. We are interested in the return obtained when starting from state $\mathbf{s}$ instead of the immediate reward in multi-armed bandit problems. The first term in Equation 3-2 can be replaced by an estimate of the expected return. When a state-value function is used, $V(\mathbf{s})$ can be inserted directly instead of the first term in Equation 3-2. For algorithms that use an action-value function we can use the maximum action-value function $\max_{\mathbf{a}} Q(\mathbf{s}, \mathbf{a})$.

In multi-armed bandit problems, all the actions are independent of each other. For neighboring states in reinforcement learning problems this is often not the case and a metric is required to determine how neighboring states are related. The uncertainty of the UCB1 bound is determined by number of times machine i has been played, denoted by $T_i(n)$. In the Smart Start case, we want this uncertainty to capture the expected new information to gain from a state. We assume that a state or region with a low visitation density has a high probability of being close to unvisited states and therefore has a high probability of resulting in new information. The visitation density of a region requires a metric to determine the relation between states. In this thesis we have employed a simpler approximation and only look at the visitation density of discrete states and not regions. The uncertainty can be determined easily using the visitation counts $C(\mathbf{s})$ for each visited state $\mathbf{s}$.

We can now convert Equation (3-2) for selecting our Smart Start state

$$\mathbf{s}_{ss} = \arg\max_{\mathbf{s}} \left[\max_{\mathbf{a}} Q(\mathbf{s}, \mathbf{a}) + c_{ss}\sqrt{\frac{\log \sum_{\mathbf{s}\in\mathcal{S}} C(\mathbf{s})}{C(\mathbf{s})}}\right] \quad \forall\ \mathbf{s} \in \mathcal{D}. \tag{3-6}$$

A constant $c_{ss} > 0$ is introduced for varying the amount of exploration and exploitation similar to the UCT algorithm [14] and has to be chosen appropriately. Often you want pure exploration in the beginning of the learning process and avoid getting stuck in a suboptimal solution due to misleading rewards for example. A large value for c_{ss} will result in more exploration. The value for c_{ss} can be decreased during the learning process to shift from pure exploration to a good balance between exploitation and exploration.

We have shown how to determine the Smart Start state. The next part elaborates on how to get the agent from the initial state $\mathbf{s}_0$ to the Smart Start state $\mathbf{s}_{ss}$. The guiding can be done in several ways, for example using trajectory replay or a model-based planning approach. The next section discusses how the guiding can be done using dynamic programming.

3-3 Trajectory Optimization

In most reinforcement learning problems the agent cannot be reset to arbitrary states in state space, e.g., a bipedal robot cannot start in the middle of a step. The second part of the Smart Start algorithm is concerned with guiding the agent from the initial state $\mathbf{s}_0$ to the Smart Start state $\mathbf{s}_{ss}$. The Smart Start state is a state that has been visited before, therefore we already have a trajectory to the Smart Start. The easiest way of getting to the Smart Start state would be by replaying the trajectory. But this has certain implications. For deterministic systems this would work, but not with stochastic systems. Another issue might be the length of the trajectory. The trajectory can be a random walk containing a lot of loops. After several iterations the trajectory to the Smart Start state may consist of multiple concatenated random walk trajectories, resulting in an extremely long trajectory to the Smart Start state. Which is undesirable since it decreases the sample efficiency.

The guiding to the Smart Start state can be seen as a trajectory optimization problem. The trajectory optimization can be done while taking into account the environment, e.g., avoiding regions with large penalties. In this thesis we have excluded environment characteristics in the trajectory optimization. Instead, the goal here is to find a policy that results in the most reliable and shortest path to the Smart Start state.

In this work we look at deterministic and stochastic environments with discrete state and action spaces. For these type of environments a model-based approach can be implemented easily and has near optimal performance. The next section discusses how the trajectory optimization can be done using dynamic programming on a learned transition model and reward function.

3-3-1 Transitioning to the Smart Start State using Dynamic Programming

The trajectory optimization problem can be seen as reinforcement learning problem on its own. However it would not make sense to train a reinforcement learning agent to get to the Smart Start state by gathering new real experiences. What we want is to use the knowledge the agent has gathered to construct a policy to the Smart Start state. Here, we learn a model and subsequently use it for planning. The goal here is to find the policy that has the highest probability of reaching the Smart Start state. The planning is done using Value Iteration [1], a discounted reward is used to get the most reliable and shortest path to the Smart Start state.

The agent keeps track of visitation counts and learns a transition model in the same way as the MBRL method, which was presented in Algorithm 2 in Section 2-2-2. We give a reward for transitions to the Smart Start state according to

$$\mathcal{R}(\mathbf{s}, \mathbf{a}) = \begin{cases} 1, & \text{if } \mathcal{P}(\mathbf{s}, \mathbf{a}, \mathbf{s}') > 0 \\ 0, & \text{otherwise} \end{cases}, \tag{3-7}$$

using this reward function, all transitions that have a probability greater than zero for traversing to the Smart Start state receive a reward. The reason for this, is that we are interested in getting in the region of the Smart Start and not exactly in the Smart Start state. An optimal policy to the Smart Start state can now be found using Value Iteration, we refer the reader to Algorithm 1 in Section 2-2-1 for more details on the Value Iteration algorithm.

Algorithm 5 Smart Start Framework

1: Initialize buffer $\mathcal{D}$ and AGENT
2:
3: **for** each episode **do**
4: Initialize $\mathbf{s}_0$ and $t = 0$
5: $u \sim U(0, 1)$
6: **if** $u \leq \eta$ and $|\mathcal{D}| > 0$ **then**
7: *// Choose Smart Start state using upper confidence bound*
8: $\mathbf{s}_{ss} = \arg\max_{\mathbf{s}} \left[\max_{\mathbf{a}} Q(\mathbf{s}, \mathbf{a}) + c_{ss}\sqrt{\frac{\log|\mathcal{D}|}{C(\mathbf{s})}} \right] \forall\ \mathbf{s} \in \mathcal{D}$
9:
10: *// Obtain policy using trajectory optimization*
11: $\pi_{ss} = \text{TRAJOPT}(\mathcal{D},\ \mathbf{s}_0,\ \mathbf{s}_{ss})$
12:
13: *// Execute Smart Start policy*
14: **repeat**
15: Choose $\mathbf{a}_t = \pi_{ss}(\mathbf{s}_t)$
16: Take action $\mathbf{a}_t$ and observe $\mathbf{s}_{t+1}$ and r_{t+1}
17: Add $(\mathbf{s}_t,\ \mathbf{a}_t,\ \mathbf{s}_{t+1},\ r_{t+1})$ to $\mathcal{D}$
18: UPDATEAGENT($\mathcal{D}$)
19: $t \leftarrow t + 1$
20: **until** $d(\mathbf{s}_t,\ \mathbf{s}_{ss}) < \theta$, $\mathbf{s}_t$ is terminal or $t = T_{\text{episode}}$
21:
22: *// Continue using learned policy and exploration strategy*
23: **if** $\mathbf{s}_t$ is not terminal and $t < T_{\text{episode}}$ **then**
24: **repeat**
25: Choose $\mathbf{a}_t$ according to the AGENT's policy π and some exploration strategy
26: Take action $\mathbf{a}_t$ and observe $\mathbf{s}_{t+1}$ and r_{t+1}
27: Add $(\mathbf{s}_t,\ \mathbf{a}_t,\ \mathbf{s}_{t+1},\ r_{t+1})$ to $\mathcal{D}$
28: UPDATEAGENT($\mathcal{D}$)
29: $t \leftarrow t + 1$
30: **until** $\mathbf{s}_t$ is terminal or $t = T_{\text{episode}}$

3-4 Smart Start Algorithm

The pseudo-code of how the full Smart Start framework is given in Algorithm 5. This algorithm can be used as template for implementing the Smart Start framework. We will discuss some important points in the algorithm:

- The AGENT is the reinforcement learning algorithm that is being augmented with Smart Start.
- Instead of using Smart Start every episode it is only used in a certain percentage of the episodes. On line 5 a random value is sampled from a uniform distribution, $u \sim U(0,1)$. If u is below a certain threshold, η, Smart Start is used. Otherwise the episode executes without Smart Start. The size of the buffer, $|\mathcal{D}|$, also has to be larger than *zero*, otherwise there are no states to choose a Smart Start state from.
- On line 8, the buffer size is used instead of the sum of visitation counts as in Equation (3-6). Note that the size of the buffer and the total count are equal, i.e., $|\mathcal{D}| = \sum_{\mathbf{s} \in \mathcal{S}} C(\mathbf{s})$.
- The TRAJOPT function has to be replaced with the trajectory optimization method that you want to use to get to the Smart Start state.
- When the agent is within some distance θ from the Smart Start state, it continues with executing its learned policy and some exploration strategy. The distance function $d(\mathbf{s}_t, \mathbf{s}_{ss})$ has to be chosen according to the metric being used to describe the relation between states.
- The update of the agent is done on lines 18 and 28, denoted by UPDATEAGENT.
- For on-policy algorithms line 18 should be removed. An example of an on-policy algorithm is SARSA [2].

3-4-1 Q-Learning with Smart Start

With the components discussed in this chapter we can now construct an example of the Smart Start algorithm. The pseudo-code of Q-Learning in combination with Smart Start is given in Algorithm 6. In discrete domains, instead of using a buffer to store every experience we can keep track of visitation counts for states and actions. The Smart Start framework can be used in combination with different algorithms. An implementation of Smart Start with the MBRL algorithm is given in Appendix A, the MBRL algorithm itself was presented in Section 2-2-2.

3-4-2 Continuous Domains

So far we have shown how to implement Smart Start in domains with discrete state spaces. For continuous state spaces a few modifications have to be made. This section will provide directions on how the Smart Start framework can be implemented in continuous domains. The framework has not been implemented or evaluated for continuous domains in this work.

Algorithm 6 Q-Learning with Smart Start

1: Initialize $Q(\mathbf{s}, \mathbf{a})$ arbitrarily for all $\mathbf{s} \in \mathcal{S}$ and $\mathbf{a} \in \mathcal{A}(\mathbf{s})$
2:
3: **for** each episode **do**
4: Initialize $\mathbf{s}_0$ and $t = 0$
5: $u \sim U(0, 1)$
6: **if** $u \leq \eta$ and $\sum_{\mathbf{s} \in \mathcal{S}} C(\mathbf{s}) > 0$ **then**
7: *// Choose Smart Start state using upper confidence bound*
8: $\mathbf{s}_{ss} = \arg\max_{\mathbf{s}} \left[\max_{\mathbf{a}} Q(\mathbf{s}, \mathbf{a}) + c_{ss} \sqrt{\frac{\log \sum_{\mathbf{s} \in \mathcal{S}} C(\mathbf{s})}{C(\mathbf{s})}} \right] \ \forall \{\mathbf{s} \in \mathcal{S} : C(\mathbf{s}) > 0\}$
9:
10: *// Fit transition model and reward function*
11: **for all** $\mathbf{s}, \mathbf{a}$ **do**
12: **if** $C(\mathbf{s}, \mathbf{a}) > 0$ **then**
13: **for all** $\mathbf{s}' \in C(\mathbf{s}, \mathbf{a}, \cdot)$ **do**
14: $\mathcal{P}(\mathbf{s}, \mathbf{a}, \mathbf{s}') = \frac{C(\mathbf{s}, \mathbf{a}, \mathbf{s}')}{C(\mathbf{s}, \mathbf{a})}$
15: **if** $\mathbf{s}' = \mathbf{s}_{ss}$ **then**
16: $\mathcal{R}(\mathbf{s}, \mathbf{a}) = 1$
17: **else**
18: $\mathcal{R}(\mathbf{s}, \mathbf{a}) = 0$
19:
20: *// Obtain policy to* $\mathbf{s}_{ss}$ *using value iteration*
21: $\pi_{ss} = \text{Value Iteration}(\mathcal{P}, \mathcal{R})$ (See Algorithm 1)
22:
23: *// Execute Smart Start policy*
24: **repeat**
25: Choose $\mathbf{a}_t = \pi_{ss}(\mathbf{s}_t)$
26: Take action $\mathbf{a}_t$ and observe $\mathbf{s}_{t+1}$ and r_{t+1}
27: Increment $C(\mathbf{s}_t)$, $C(\mathbf{s}_t, \mathbf{a}_t)$ and $C(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1})$
28: *// Q-Learning update*
29: $Q_{t+1}(\mathbf{s}_t, \mathbf{a}_t) = Q_t(\mathbf{s}_t, \mathbf{a}_t) + \alpha \left[r_{t+1} + \gamma \max_{\mathbf{a}'} Q(\mathbf{s}_{t+1}, \mathbf{a}') - Q(\mathbf{s}_t, \mathbf{a}_t) \right]$
30: $t \leftarrow t + 1$
31: **until** $\mathbf{s}_t = \mathbf{s}_{ss}$, $\mathbf{s}_t$ is terminal or $t = T_{\text{episode}}$
32:
33: *// Continue using learned policy and exploration strategy*
34: **if** $\mathbf{s}_t$ is not terminal and $t < T_{\text{episode}}$ **then**
35: **repeat**
36: Choose $\mathbf{a}_t$ according to some policy π derived from Q (e.g. ϵ-greedy)
37: Take action $\mathbf{a}_t$ and observe $\mathbf{s}_{t+1}$ and r_{t+1}
38: Increment $C(\mathbf{s}_t)$, $C(\mathbf{s}_t, \mathbf{a}_t)$ and $C(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1})$
39: *// Q-Learning update*
40: $Q_{t+1}(\mathbf{s}_t, \mathbf{a}_t) = Q_t(\mathbf{s}_t, \mathbf{a}_t) + \alpha \left[r_{t+1} + \gamma \max_{\mathbf{a}'} Q(\mathbf{s}_{t+1}, \mathbf{a}') - Q(\mathbf{s}_t, \mathbf{a}_t) \right]$
41: $t \leftarrow t + 1$
42: **until** $\mathbf{s}_t$ is terminal or $t = T_{\text{episode}}$

The Smart Start state selection needs small modifications for the continuous domain. As mentioned in Section 3-2, a metric is needed that tells us how states are related to each other. In the implementation for discrete domains we did not use a metric and only looked at the visitation counts of discrete states. Lets assume we have an enormous state space that is too large to be fully explored. The current implementation would exhaustively search every single state. A metric would tell us how states are related and allow us to search through regions of states that are related. This gives the agent the ability to search the space more crudely. Once it has a rough idea of how the state space looks and what regions look promising, the metric can be altered to make the search finer and direct the agent to the most promising regions.

The state visitation density can be estimated using kernel density estimation [37], where the used kernel contains the metric. A commonly used kernel is the Gaussian kernel. An implementation of the Smart Start state selection using kernel density estimation is given in Appendix B. The bandwidth of the kernel can be modified to make the relation between states cruder or finer.

The trajectory optimization part is less trivial to implement for continuous environments. As mentioned before, model-based reinforcement learning methods cannot be converted to continuous domains easily. Often, they are not able to learn a good solution because of the poor quality of the learned model. The reason they fail is because the model often needs a high accuracy to be used for planning or simulated experience. With Smart Start on the other hand the model is not directly responsible for the learned policy. Instead, only a local model around the trajectory from the initial state to the Smart Start state is required. Furthermore the model does not have to be perfect, the agent only has to be brought close to the Smart Start state.

One way is to use an approximate dynamic programming method, like fitted Value Iteration [38]. But this still requires a good model to be used for planning to the Smart Start state. A different solution is to use a trajectory optimization method, like iLQG [39]. The trajectory optimization can be done online, also known as model-predictive control [40]. Levine and Koltun [41, 42] used iLQG in their guided policy search algorithm. In subsequent work, Levine and Abbeel [43] incorporated a learned model that is used by the iLQG trajectory optimization. The trajectory optimization and model fitting can be used for the guiding part in our Smart Start framework.

Another interesting implementation is the model-based deep reinforcement learning method proposed by Nagabandi et al. [25]. A neural network dynamics model is fitted and used by a model predictive controller. As more experiences are obtained, the model improves and subsequently the model predictive controller can generate a better control policy. But they suffer from poor performance with just the model-predictive controller. Their solution is to use the learned model-predictive control policy to initialize a policy that can be trained further using model-free reinforcement learning. This is similar to the idea of Smart Start, the neural network dynamics model and model-predictive controller can be used for the trajectory optimization in the Smart Start framework. A big difference is that the method from Nagabandi et al. requires access to the reward function for the model-predictive controller. Smart Start does not and learns a policy based on the unknown reward function.

An example of how the algorithm can look in continuous domains is presented in this paragraph. The algorithm has not been implemented or verified in this work. The pseudo-code given in Algorithm 5 is used as template.

The algorithm presented here requires an off-policy reinforcement learning algorithm that learns a value function, examples of such algorithms are DDPG [6] and NAF [24]. The reinforcement learning algorithm is denoted by AGENT in Algorithm 5. Both algorithms use a replay buffer $\mathcal{D}$ and update their policy and value function by sampling from the buffer. The update of the agent is given by UPDATEAGENT. For the Smart Start state selection on line 8, the count $C(\mathbf{s})$ has to be replaced, for example by the approximate count $\hat{C}(\mathbf{s})$ given in Equation (B-4) in Appendix B. The trajectory optimization can be done using an approach presented above. The samples in the buffer can be used to construct a policy from the initial state to the Smart Start state. In Algorithm 5, the trajectory optimization step is given by TRAJOPT. On line 21 in the algorithm, the $d(\mathbf{s}_0, \mathbf{s}_{ss})$ function represents the distance between $\mathbf{s}_0$ and $\mathbf{s}_{ss}$, which has to be determined according to the metric that is being used to describe the relation between states. When the agent is within a certain distance θ of the Smart Start state, we say that the agent has reached the Smart Start state and continues by executing its learned policy and some exploration strategy.

3-5 Discussion

In this chapter we introduced a novel exploration framework for reinforcement learning, called Smart Start. Smart Start solves the exploration problem by dividing learning episodes in two parts, the Smart Start phase and the “normal” learning phase. During the Smart Start phase the agent is guided to a region in which it expects to learn the most, denoted by the Smart Start state. Once the region is reached the agent will continue with the “normal” learning phase. In the “normal” learning phase the agent executes its learned policy with some exploration strategy.

The Smart Start state selection can be modeled as a multi-armed bandit problem [12], giving access to all the knowledge and algorithms on solving them. One of those methods is UCB1 [12], in which an upper confidence bound is calculated that has an excellent trade-off between exploration and exploitation. We presented the Smart Start state selection for discrete state spaces in which the state visitation counts are used to approximate the state visitation density. But the method can easily be converted to continuous state spaces, for example using kernel density estimation [37] to estimate the state visitation density. In Appendix B an implementation of the Smart Start state selection using kernel density estimation is given. The same approach can be taken for problems with discrete state spaces where states close to each other are related or the goal is a region spanning multiple states. In these cases it can be undesirable to exhaustively search the full state space, instead you want to visit every region containing similar states.

The guiding to the Smart Start state can be seen as a trajectory optimization problem. The agent has visited the Smart Start state before, but this time it wants to get there faster and with high probability. For discrete environments this can be done using a model-based approach [18]. A transition model is fit using visitation counts. The reward function is created by giving a reward for reaching the Smart Start state. The transition model and reward

function can be used in combination with dynamic programming, e.g., Value Iteration [1], to construct an optimal policy to the Smart Start state. Once enough data is available to construct a good model, a robust policy can be planned, making the method compatible with stochastic environments as well. The algorithm fits a global transition model and reward function. For large problems it might be undesirable or intractable to learn a global model. A local trajectory optimization might be sufficient to get the agent to the Smart Start state. An example for continuous states is the iLQG method [44]. Only a local model around the trajectory is necessary for optimization, removing the need to learn a good global model.

An example of the full Smart Start algorithm for discrete environments was presented in combination with Q-Learning [2] and Value Iteration. The Smart Start framework can be used with different reinforcement learning algorithms, an example using a simple model-based reinforcement learning algorithm is given in Appendix A. In this work we focus on discrete environments with sparse or misleading rewards. Directions on using Smart Start in continuous domains were given in Section 3-4-2. The framework can easily be used for non-sparse rewards environments, which will be discussed in the recommendations for future work in Section 6-2-1. Smart Start improves the exploration of an agent by providing persistent and directed exploration and therefore it can improve the overall learning efficiency.

In the next chapter the experimental setup used to evaluate the performance of Smart Start is discussed.

Chapter 4

Experimental Setup

The previous chapter introduced the Smart Start framework. In this chapter the experimental setup that is used to evaluate the Smart Start framework is presented. Four gridworld environments are designed to test the performance in the case of misleading or sparse rewards.

In the sparse rewards case, the agent only receives a reward when the goal is reached. In the misleading reward case, it is difficult to reach the optimal solution and at the same time tempting to go for a suboptimal solution. Therefore we have designed four gridworld environments, depicted in Figure 4-1. The Easy, Medium and Maze gridworld environments are used for evaluating the exploration performance and performance in the case of sparse rewards. The Misleading gridworld environment is used for evaluating the performance when misleading rewards are present. The gridworld environments are surrounded by walls and have interior walls. The agent only receives a reward upon reaching the goal. Each environment has a start state $\mathbf{s}_{\text{start}}$ and a goal state $\mathbf{s}_{\text{goal}}$, denoted by a red and green dot respectively in Figure 4-1. The next three sections discuss the specifications of the gridworld environments.

Sections 4-1 discusses the specifications of the deterministic gridworld environments. In Section 4-2, the details of the stochastic case are given. Section 4-3 will discuss the specifications of the Misleading gridworld environment.

4-1 Deterministic Gridworld

In the gridworld environments the position of the agent is given by a 2-dimensional discrete state $\mathbf{s} = [s_0, s_1] \in \mathbb{Z}^2$, where $\mathbb{Z}$ denotes the set of integers. The set of possible actions for each state is $\mathcal{A}(\mathbf{s}) = [0, 1, 2, 3] \; \forall \; \mathbf{s}$. The deterministic state transition is defined as

$$\mathbf{s}_{t+1} = \mathbf{s}_t + \Delta \mathbf{s}_t, \qquad (4\text{-}1)$$

where

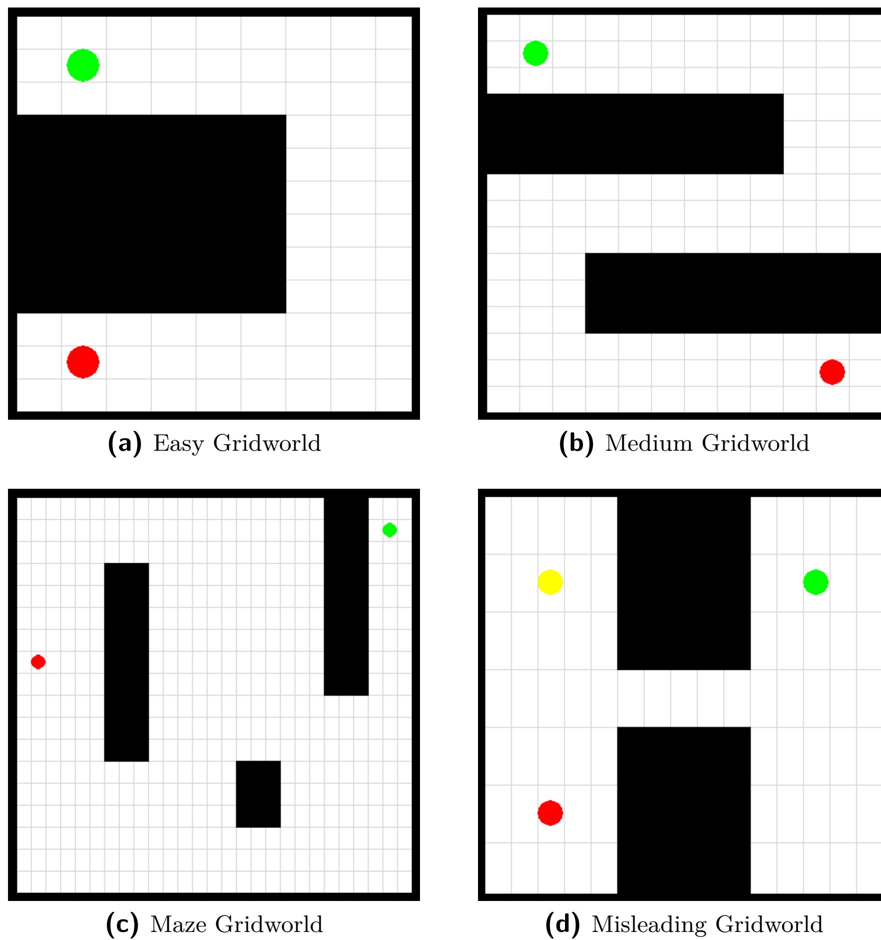


Figure 4-1: Gridworld environments. Intial state s_{start} is denoted by a red dot and the goal state s_{goal} by a green dot. The yellow dot in Figure 4-1d denotes a suboptimal goal. More details can be found in Table 4-1.

Table 4-1: Specifications of the gridworld environments. The position of the agent is denoted by a 2-dimensional discrete state $\mathbf{s} \in [\mathbf{s}_0,\ \mathbf{s}_1]$. The total number of accessible states is given by N_{states}. $\mathbf{s}_{\text{start}}$ is the start state and $\mathbf{s}_{\text{goal}}$ is the goal state. The suboptimal goal in the Misleading gridworld environment is given by $\mathbf{s}_{\text{subgoal}}$. $l(\pi^*)$ denotes the shortest path from $\mathbf{s}_{\text{start}}$ to $\mathbf{s}_{\text{goal}}$.

Gridworld	$s_0 \in$	$s_1 \in$	N_{states}	$\mathbf{s}_{\text{start}}$	$\mathbf{s}_{\text{goal}}$	$\mathbf{s}_{\text{subgoal}}$	$l(\pi^*)$
Easy	$[0,\ \ldots,\ 11]$	$[0,\ \ldots,\ 8]$	72	$\{10,\ 1\}$	$\{1,\ 1\}$	-	19
Medium	$[0,\ \ldots,\ 14]$	$[0,\ \ldots,\ 11]$	126	$\{13,\ 10\}$	$\{1,\ 1\}$	-	35
Maze	$[0,\ \ldots,\ 17]$	$[0,\ \ldots,\ 26]$	423	$\{7,\ 1\}$	$\{1,\ 25\}$	-	40
Misleading	$[0,\ \ldots,\ 6]$	$[0,\ \ldots,\ 14]$	75	$\{5,\ 2\}$	$\{1,\ 12\}$	$\{1, 2\}$	14

$$\Delta \mathbf{s}_t = \begin{cases} [0,1] & \text{if} \quad \mathbf{a}_t = 0 \\ [1,0] & \text{if} \quad \mathbf{a}_t = 1 \\ [0,-1] & \text{if} \quad \mathbf{a}_t = 2 \\ [-1,0] & \text{if} \quad \mathbf{a}_t = 3 \end{cases}. \tag{4-2}$$

The agent cannot walk through walls. When the agent tries to enter a wall the state stays the same, i.e., $\mathbf{s}_{t+1} = \mathbf{s}_t$. The episode is terminated when the agent enters the goal state or the maximum number of time-steps per episode, T_{episode}, is reached. No reward is given except when the agent enters the goal state

$$r_{t+1} = \begin{cases} 1 & \text{if} \quad \mathbf{s}_{t+1} = \mathbf{s}_{\text{goal}} \\ 0 & \text{otherwise} \end{cases}. \tag{4-3}$$

In Table 4-1 the specifications for each gridworld environment are given. The optimal path length is denoted by $l(\pi^*)$, the start and goal states are denoted by $\mathbf{s}_{\text{start}}$ and $\mathbf{s}_{\text{goal}}$ respectively. The total number of accessible states is denoted by $N_{\text{states}} = \max[\mathbf{s}_0] \cdot \max[\mathbf{s}_1] - walls$.

4-2 Stochastic Gridworld

The stochastic gridworld specifications are the same as for the deterministic case. The only difference is the state transition. For the stochastic case there is a 0.75 probability of traversing to the next state according to the deterministic state transition associated with the action being executed, as declared in Equation (4-2). There is a probability of 0.25 that the agent uses one of the three remaining state transitions given in Equation (4-2).

For example, when the agent executes action $\mathbf{a}_t = 0$ in state $\mathbf{s}_t$, it has a 75% chance of using the state transition $\Delta\mathbf{s}_t = [0, 1]$. There is a 25% chance the agent traverses to a different state by randomly choosing one of the other three state transitions. This percentage translates to a 8.33% chance transition $\Delta\mathbf{s}_t = [1, 0]$ is being used, a 8.33% chance transition $\Delta\mathbf{s}_t = [0, -1]$ is being used and a 8.33% chance transition $\Delta\mathbf{s}_t = [-1, 0]$ is being used.

4-3 Misleading Gridworld

The fourth gridworld environment is designed for testing the performance when misleading rewards are present. The Misleading gridworld environment, depicted Figure 4-1d, has the same characteristics as the deterministic gridworld environments. There are three differences:

- All the walls are terminal, this means when the agent walks into a wall the episode is terminated. A reward of $r_{t+1} = -0.1$ is given when this happens.
- A suboptimal goal state, $\mathbf{s}_{\text{subgoal}}$ is added, depicted by a yellow dot in Figure 4-1d. When the agent reaches this suboptimal goal state the episode is terminated and a reward of $r_{t+1} = 1.0$ is given.
- Entering the true goal state is rewarded with a reward of $r_{t+1} = 100.0$.

The moat in the middle of the environment with terminal states, i.e., the walls, on either side makes it hard for the agent to reach the other side. This is hard because the states next to the moat are not only terminal but also give the agent a penalty. It is tempting for the agent to go to the suboptimal goal since it is on the same side of the moat and also gives a positive reward.

The next chapter presents the the results of this thesis.

Chapter 5

Results

We are now going to examine the performance of the Smart Start framework, presented in Chapter 3, in combination with the reinforcement learning methods and exploration strategies discussed in Chapter 2. The four discrete gridworld environments discussed in Chapter 4 are used. Multiple experiments were done for evaluating and comparing the performance of the Smart Start framework. The first experiment measures the exploration performance of different exploration strategies in combination with the Smart Start framework. The second experiment evaluates the influence of the Smart Start parameters. The third and fourth experiments compare the learning performance in deterministic and stochastic environments with sparse rewards. In the final experiment, the learning performance in the case of misleading rewards is evaluated.

For each experiment we use the same parameters for the reinforcement learning agents, given in Table 5-1. The learning rate and discount factor are set to $\alpha = 0.1$ and $\lambda = 0.99$ respectively. The other parameters are chosen by evaluating the performance of different parameter values, the values performing best are used.

Section 5-1 evaluates the exploration performance of various exploration strategies in combination with Smart Start. In Section 5-2 the influence of the various parameters of the Smart Start framework is evaluated. Sections 5-3 and 5-4 evaluate the learning performance

Table 5-1: Reinforcement learning parameters

Parameter	**Symbol**	**Value**
Learning rate	α	0.1
Discount factor	γ	0.99
ϵ-greedy parameter	ϵ	0.1
Temperature Boltzmann exploration	τ	5.0
Exploitation parameter UCB1	C_p	1.0
R-MAX threshold	m	2
R-MAX maximum reward	$R_{\max}$	0.1

of Smart Start in combination with Q-Learning, MBRL and R-MAX on deterministic and stochastic gridworld environments respectively. The results in the case of misleading rewards are presented in Section 5-5. This chapter is finalized with a discussion in Section 5-6.

5-1 Exploration Performance

Reinforcement learning agents learn from the reward signal the environment gives them. In an environment with sparse rewards like ours this has certain implications. For an algorithm like Q-Learning this means the value function is *zero* until the goal has been reached for the first time. The number of steps it takes the agent to reach the goal for the first time is therefore an important characteristic of the exploration strategy.

Experimental Details

In this experiment the average number of training-steps it takes the agent to reach the goal for the first time is measured. The experiment is carried out on the Easy, Medium and Maze gridworld environments and three exploration strategies are evaluated.

The three exploration strategies are evaluated with and without Smart Start. The first exploration method is random exploration, effectively ϵ-greedy and Boltzmann exploration because they have the same policy with a value function of *zero*. The second is UCB1 exploration, which calculates an upper confidence bound that trades-off between exploitation and exploration. The last strategy is model-based exploration, which uses the same approach as R-MAX. The model-based exploration strategy constructs a transition model and reward function from previous experiences and plans to states that have been visited infrequently. Details on the exploration strategies have been discussed in Section 2-3.

The Smart Start framework is designed for episodic problems. Besides the number of training-steps it takes the agent to reach the goal, we are also interested in the effect of different episode lengths on the exploration performance. Thirteen different episode lengths are tested, $T_{\text{episode}} \in$ [25, 50, 75, 100, 150, 200, 250, 500, 750, 1000, 2500, 5000, 10000]. An experiment is terminated if the goal has not been reached after $T_{\max}$ training steps, where $T_{\max} \in$ [50000, 75000, 250000] for the Easy, Medium and Maze gridworld environments respectively.

For the experiments with Smart Start we want the agent to use Smart Start every episode. Therefore, we use $\eta = 1$ in this experiment. Note that the agent cannot use Smart Start in the first episode because no data is collected yet. The value function will be *zero* throughout the experiment. This makes the Smart Start parameter c_{ss} irrelevant in this experiment and can be set to an arbitrary positive value, a value of $c_{ss} = 0.1$ is used in this experiment.

Discussion

Figure 5-1 shows the average number of training steps needed to reach the goal for the first time using different exploration strategies on the Easy, Medium and Maze gridworld environments. The results are averaged over 10 experiments each and depict the mean and standard deviation.

Random exploration shows the worst performance. For smaller episode lengths the agent was not able to reach the goal in the given amount of time-steps. The reason for this is that every time the agent starts a new episode, it performs a new random walk completely disregarding prior information. With shorter episode lengths, the probability of finding the goal decreases exponentially [32], which also explains the high standard deviation. Random exploration with Smart Start achieves consistent results in reaching the goal. For shorter episode lengths random exploration with Smart Start performs better and has a lower standard deviation. At some point random exploration with and without Smart Start results in similar performance. At this point the episode length is long enough for the agent to reach the goal within the first episode with high probability and therefore almost never utilizes the Smart Start method.

UCB1 exploration performs better than random exploration. The same trend is seen as with random exploration. For shorter episode lengths the performance decreases significantly without Smart Start. When a new episode is started, the agent has to cover the same part of state space again. After a while this part has been covered almost uniformly and the policy of the agent starts to look like a random walk for this part of the state space. This greatly reduces the performance of the exploration strategy for shorter episode lengths. This is exactly one of the pitfalls Smart Start tries to solve, namely reducing the time spent wandering through the same areas of the state space over and over and instead providing more persistent and directed exploration. From the results it is clear that UCB1 with Smart Start shows good performance for all episode lengths. At some point the results of UCB1 with and without Smart Start converge to the same values, again this can be explained by the Smart Start method not being utilized because the agent finds the goal in the first episode with high probability.

For model-based exploration the difference does not grow as fast for shorter episode lengths as with random and UCB1 exploration. The model-based policy is guided to regions where the model uncertainty is high. This guiding is done by giving an exploration bonus to state-action pairs that have been executed infrequently. The Smart Start state is constructed in a similar way, giving reward to state-action pairs that result in entering the Smart Start state, therefore similar results are expected. The difference is that the model-based method explores every state-action pair at least m times, here $m = 2$ was used. Smart Start is only guided to the state that has the highest uncertainty according to the upper confidence bound. This results in Smart Start spending less time exhaustively exploring the full state space. This explains the better sample efficiency for shorter episode lengths.

In the next section the influence of the Smart Start parameters is evaluated.

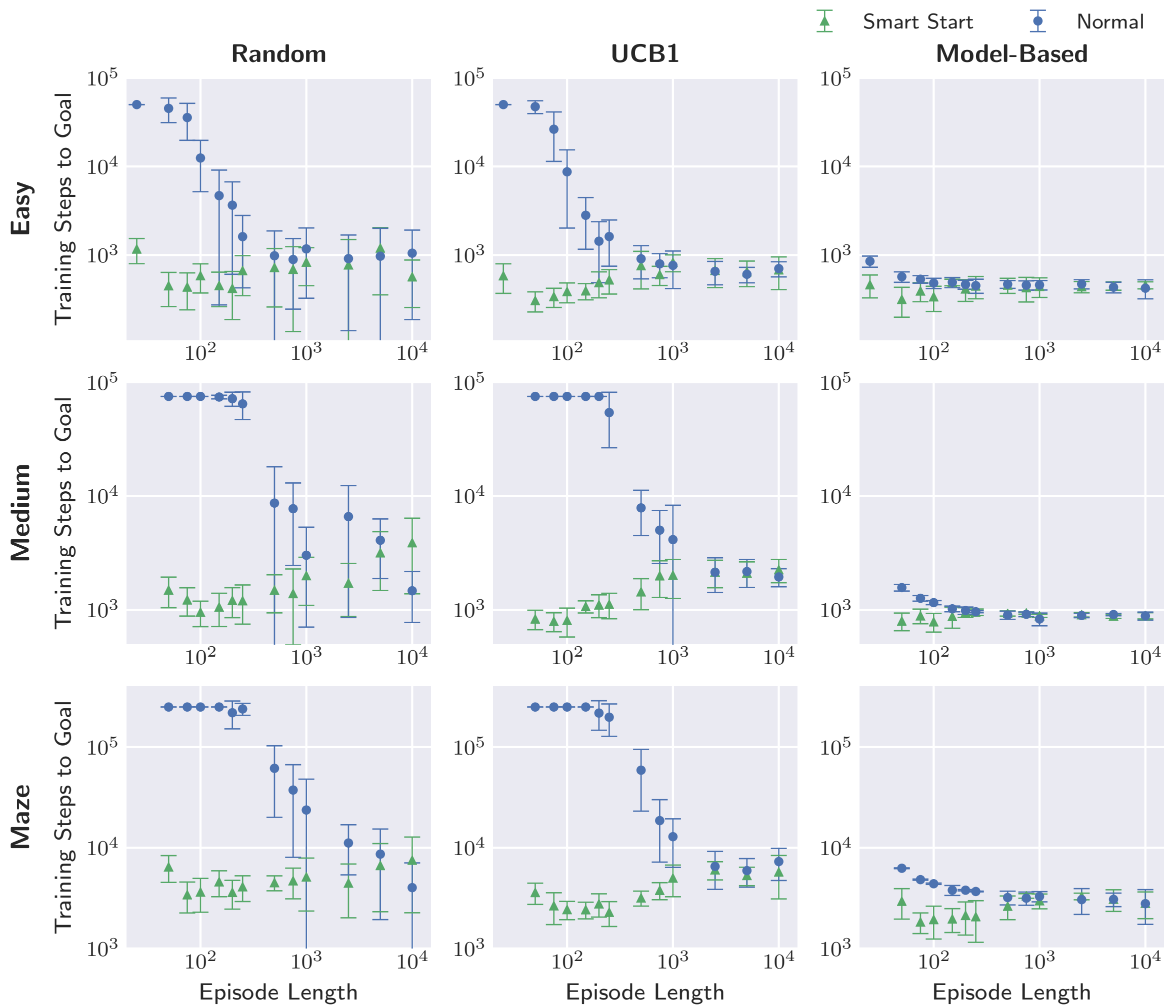


Figure 5-1: Exploration performance on Easy, Medium and Maze gridworld environments. Random, UCB1 and model-based exploration are plotted with and without Smart Start. The performance is measured as the average number of training steps it takes the agent to reach the goal for the first time. The plots depict the episode length versus the average number of training steps and standard deviation. Both axes are in logarithmic scale.

5-2 Smart Start Parameters

The previous experiment showed that Smart Start is able to improve the exploration performance for random, UCB1 and model-based exploration. In this section we are going to assess the influence of the Smart Start parameters, η and c_{ss}. At the start of each learning episode there is an $\eta \in [0,1]$ probability of using Smart Start, i.e., $\eta = 0$ results in Smart Start not being used at all and for $\eta = 1$, the agent will always use Smart Start at the beginning of an episode. Parameter c_{ss} is used in the selection of the Smart Start state and can vary the influence of the bound determined by the state visitation density. A large value for c_{ss} will result in more exploration, whereas a low value results in more exploitation.

Experimental Details

The experiments are done using Q-Learning in combination with ϵ-greedy, Boltzmann and UCB1 exploration. During each experiment a test run is carried out after every 100 training steps. The test run evaluates the learned policy π. The performance is measured as the number of time-steps of the test episode $l(\pi)$, versus the number of training steps. We use the rise time τ_{rise} as numerical measure of performance, this gives us a single value for comparing different experiments. The rise time is defined as the number of training steps it takes to get within ε of the optimal solution

$$\tau_{\text{rise}} = \{t \mid l(\pi) \leq \text{round}\left((1+\varepsilon)\, l(\pi^*)\right)\}, \tag{5-1}$$

where t is the number of training steps that have occurred and $l(\pi^*)$ is the optimal path length and can be found in Table 4-1 for each environment. The value of the threshold is rounded because we work with discrete steps.

The experiment is done on the Easy and Medium gridworld environments. For the rise time a parameter value of $\varepsilon = 0.1$ is used. The following parameter values are tested using a grid search, $\eta \in [0.2,\ 0.4,\ 0.6,\ 0.8,\ 1.0]$ and $c_{ss} \in [0.001,\ 0.005,\ 0.01,\ 0.05,\ 0.1,\ 0.5,\ 1.0,\ 5.0,\ 10.0]$. For the Easy gridworld environment a maximum episode length of $T_{\text{episode}} = 50$ and maximum simulation length of $T_{\text{max}} = 25000$ is used. For the Medium gridworld environment a maximum episode length of $T_{\text{episode}} = 100$ and maximum simulation length of $T_{\text{max}} = 50000$ is used. The episode lengths T_{episode} were derived from the results found in Section 5-1.

Discussion

From the results, depicted in Figure 5-2, it is not immediately clear which parameters values have to be chosen. We will first look at the influence of c_{ss}. Because of the nature of environments it is hard to clearly measure the influence of c_{ss}. This is due to the fact that the upper confidence bound for the Smart Start state is composed of the value function and the uncertainty of a state. However, the value function is *zero* most of the time, because of the sparse reward function that is used. For low c_{ss} the Smart Start method will therefore choose points close to the goal as soon as it has reached the goal for the first time, since the value function has the highest value close to the goal. For higher c_{ss} the Smart Start method will focus more on exploration instead of exploitation.

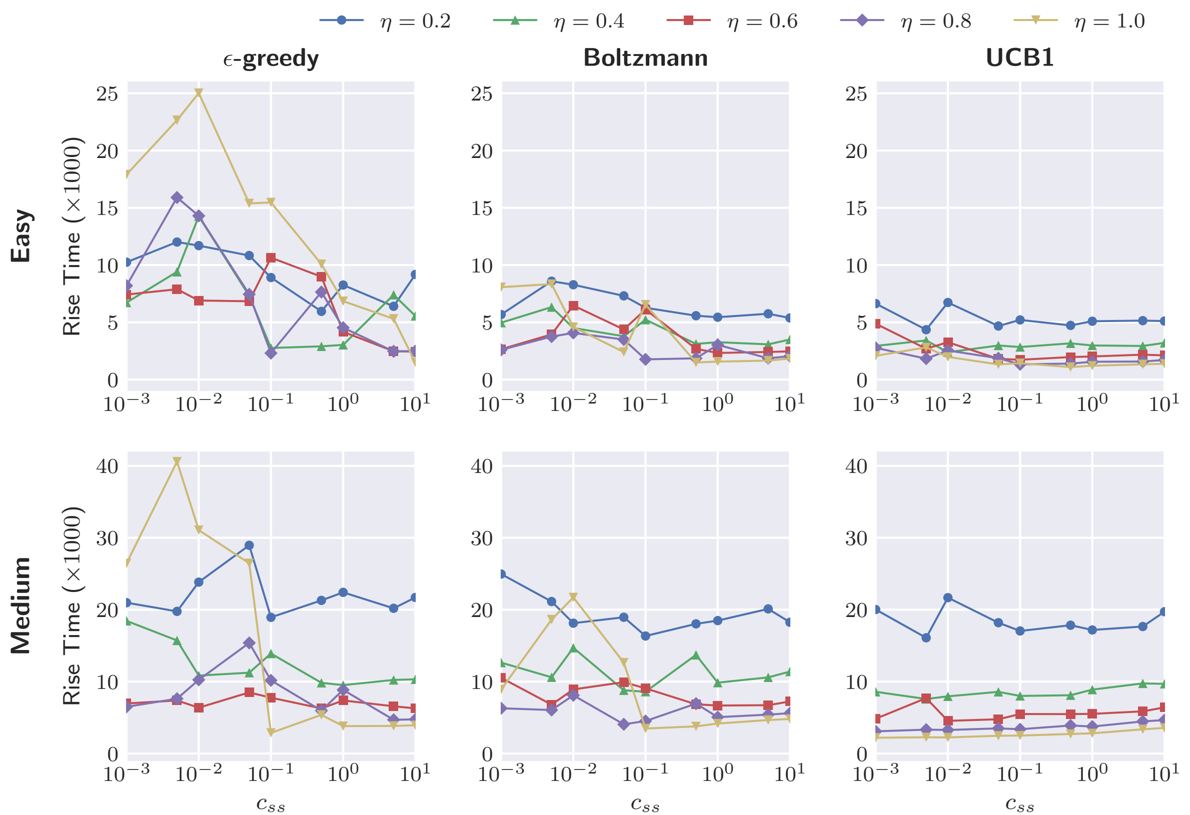


Figure 5-2: Influence of Smart Start parameters η and c_{ss} on the learning performance on the Easy and Medium gridworld environments. The performance is evaluated for Q-Learning with ϵ-greedy, Boltzmann and UCB1 exploration. The rise time is in training steps and was calculated using $\varepsilon = 0.1$.

For parameter η we see a big difference in the results. Increasing the value of η clearly results in better performance. After the agent has explored the full state space and reached the goal it will often choose a Smart Start state close to the goal. This can be explained by the composition of the Smart Start state. First, the value function is largest near the goal. Second, the episode is terminated when the goal is reached and the agent receives a reward when this happens. This means the agent can quickly learn what to do close to the goal. This results in less time wandering around the goal as compared to other parts of the state space. Therefore the bound based on state visitation density will be higher near the goal compared to other parts of the state space. Because of these two factors, the value function and state visitation density, the Smart Start state will often be a state close to the goal. Together with the fact that the agent can construct a perfect model to guide the agent to the Smart Start state, this will result in better performance for higher η.

With the results from the last two section we can now evaluate and compare the full learning performance of the Smart Start framework. The results are presented in the next three sections.

5-3 Deterministic Gridworld with Sparse Rewards

The third experiment compares the learning performance of Smart Start in deterministic environments.

Experimental Details

Smart Start was evaluated for MBRL, R-MAX and Q-Learning with ϵ-greedy, Boltzmann and UCB1 exploration. Details on the algorithms and exploration strategies can be found in Chapter 2. The Smart Start parameters are set to $\eta = 0.8$ and $c_{ss} = 0.1$, because they showed the best combination of consistency and performance in the previous experiment. For experiments with Smart Start, the episode length was set to $T_{\text{episode}} = [100,\ 150,\ 250]$ for the Easy, Medium and Maze gridworld environments respectively. Without Smart Start, $T_{\text{episode}} = 2500$ for Q-Learning and $T_{\text{episode}} = 1000$ for MBRL and R-MAX. In the experiment on exploration performance in Section 5-1 we found that algorithms with Smart Start showed better exploration performance for shorter episode lengths, whereas algorithms without Smart Start for longer episode lengths. Therefore, different episode lengths are used. A maximum simulation time of $T_{\max} = [50000,\ 100000,\ 250000]$ is used for the Easy, Medium and Maze gridworld environments respectively. After every 100 training steps a test run is executed, the test results are used for evaluation. The rise time τ_{rise} is used as performance measure with rise time parameter $\varepsilon = 0.1$. For each experiment the results are averaged over 10 trials. If the agent does not reach a result within ε of the optimal solution, a rise time of $\tau_{\text{rise}} = T_{\max}$ is given. The average reward is normalized around the optimal solution, which is $1.0/l(\pi^*)$ for the Easy, Medium and Maze gridworld environments. The optimal path length $l(\pi^*)$ is given in Table 4-1.

Discussion

The normalized average reward of the test episode versus the training steps for Q-Learning with ϵ-greedy, Boltzmann and UCB1 exploration on the Easy, Medium and Maze gridworld environments are depicted in Figure 5-4. The result for MBRL and R-MAX on all three environments are depicted in Figure 5-5. Figure 5-3 shows the average rise time and standard deviation for each algorithm on the Easy, Medium and Maze gridworld environments from top to bottom respectively. From Figures 5-3 and 5-4 it is immediately clear ϵ-greedy does not perform well on this task. With Smart Start, the performance of ϵ-greedy improves significantly and achieves similar results to Boltzmann and UCB1 in combination with Smart Start. Boltzmann and UCB1 also show a huge improvement when Smart Start is used. The results of Q-Learning with any of the exploration strategies and Smart Start are comparable to the performance of R-MAX. For R-MAX we see a slight decrease in performance. This is especially clear from Figure 5-5. This was expected, because R-MAX performs near optimal in environments like these. The time spent getting to the Smart Start state each episode makes Smart Start less efficient in this situation. MBRL with Smart Start is able to achieve similar or better performance than R-MAX, showing that Smart Start can improve the exploration of simple reinforcement learning algorithms considerably.

The next section discusses the results of the final experiment in stochastic gridworld environments.

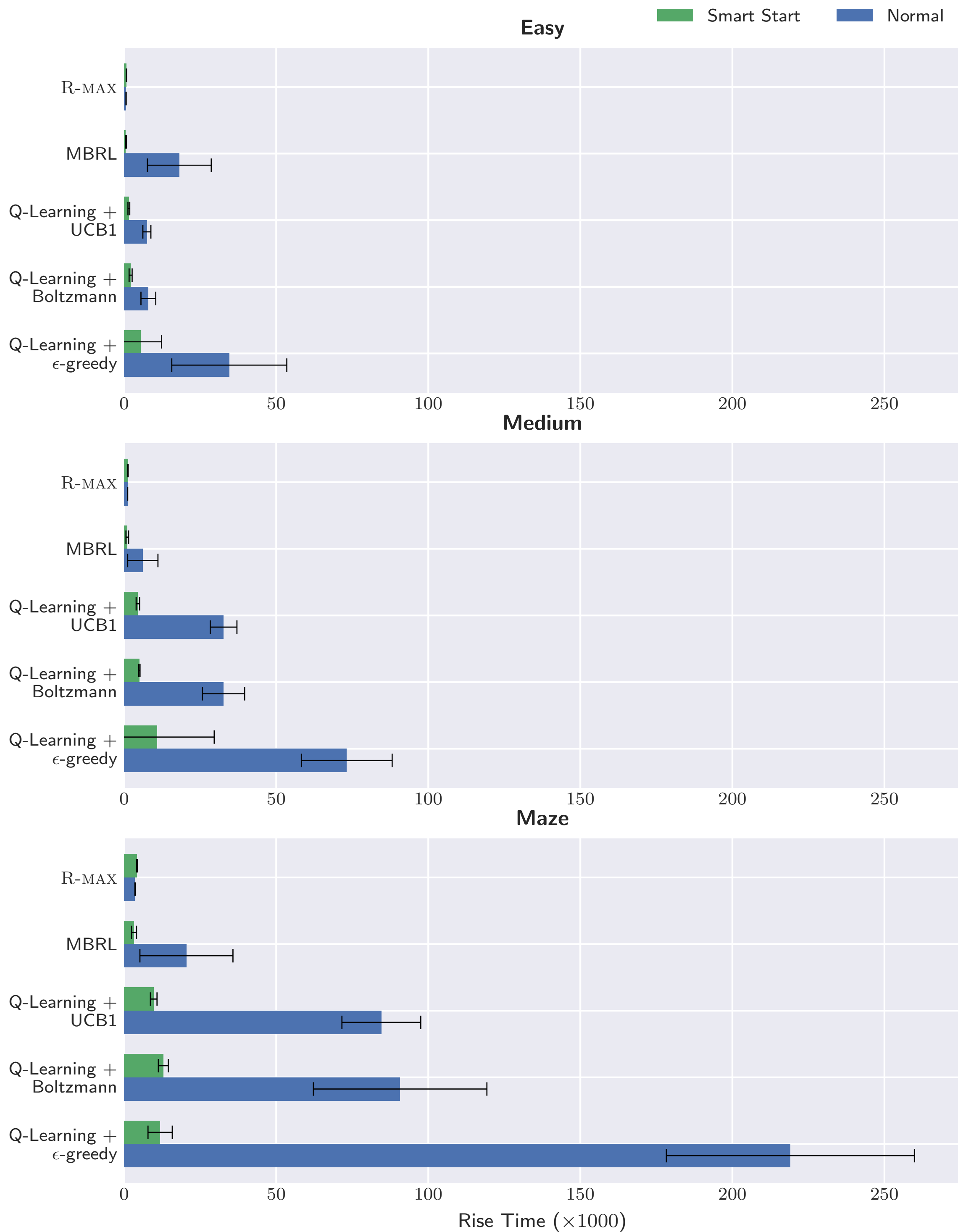


Figure 5-3: Rise time of Smart Start on deterministic environments. MBRL, R-MAX and Q-Learning with ϵ-greedy, Boltzmann and UCB1 exploration are evaluated with and without Smart Start. The results are shown for Easy, Medium and Maze gridworld environments. A rise time parameter of $\varepsilon = 0.1$ was used. The average rise time and standard deviation are depicted.

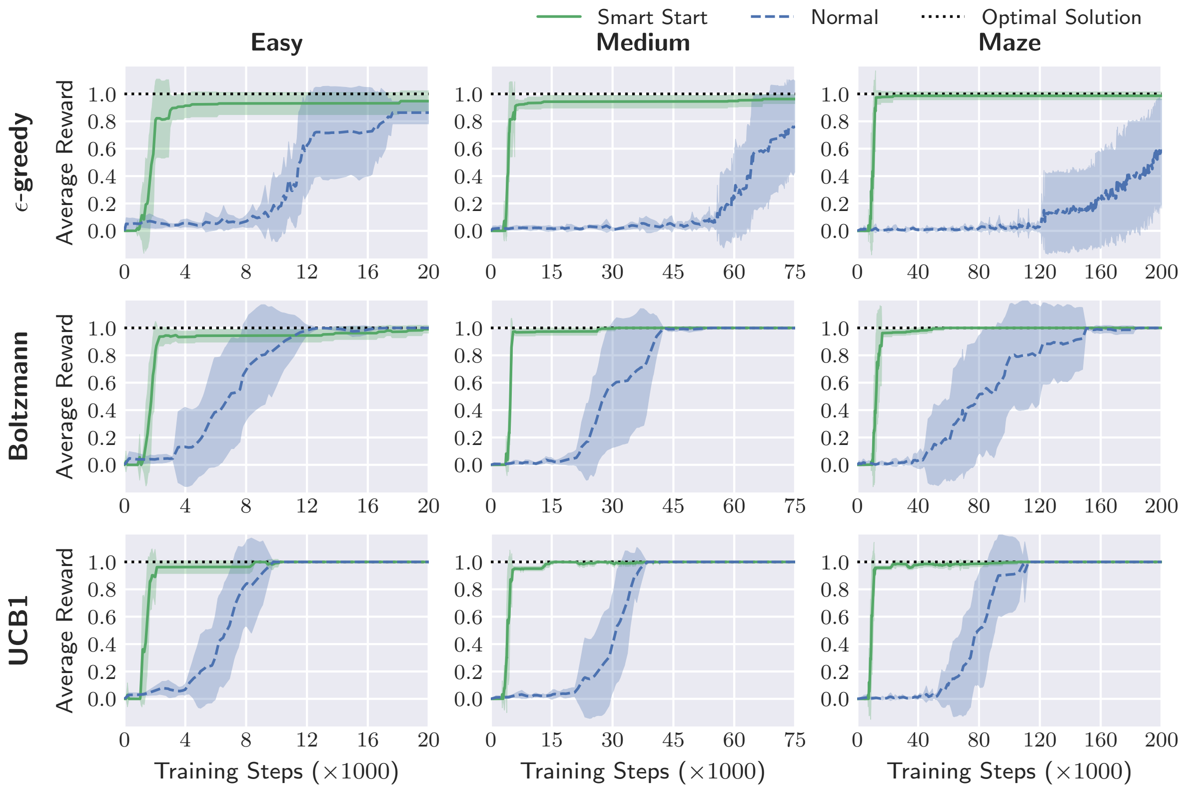


Figure 5-4: Normalized average reward of Smart Start for Q-Learning with ϵ-greedy, Boltzmann and UCB1 exploration on deterministic environments. Evaluated on Easy, Medium and Maze gridworld environments.

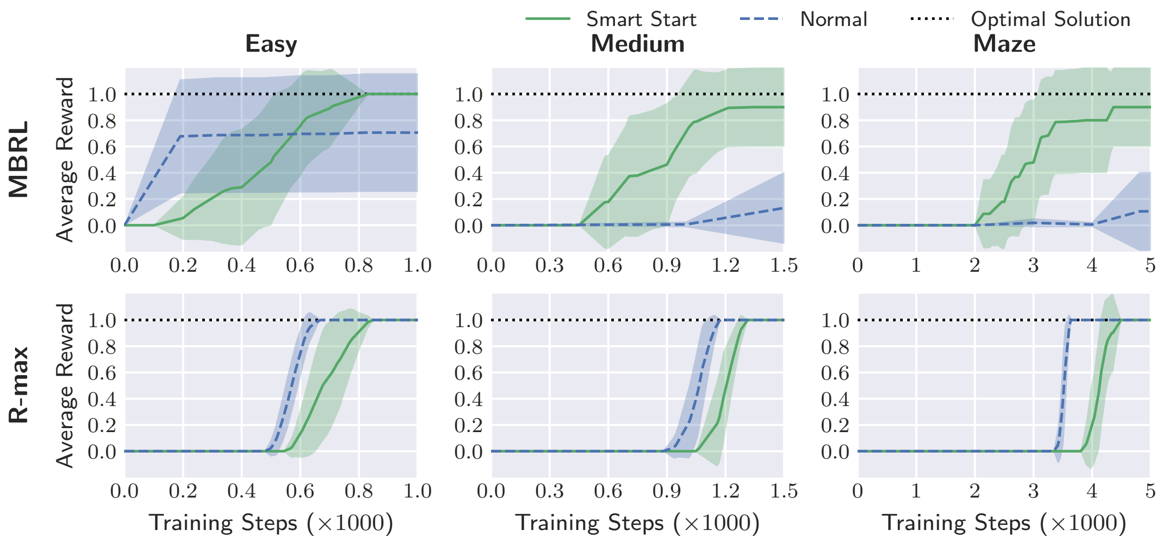


Figure 5-5: Normalized average reward of Smart Start for MBRL and R-MAX on deterministic environments. Evaluated on Easy, Medium and Maze gridworld environments.

5-4 Stochastic Gridworld with Sparse Rewards

The fourth experiment is similar to the previous one, except that we now use stochastic instead of deterministic environments. The details for the stochastic gridworld environment can be found in Section 4-2.

Experimental Details

The Smart Start parameters are set to $\eta = 0.8$ and $c_{ss} = 0.1$. For experiments with Smart Start, the episode length is set to $T_{\text{episode}} = [100,\ 150,\ 250]$ for the Easy, Medium and Maze gridworld environments respectively. Without Smart Start, $T_{\text{episode}} = 2500$ for Q-Learning and $T_{\text{episode}} = 1000$ for MBRL and R-MAX. A maximum simulation time of $T_{\max} = [50000,$ $100000,\ 250000]$ is used for the Easy, Medium and Maze gridworld environments. After every 100 training steps a test run is executed, the test results are used for evaluation. The threshold parameter for R-MAX is set to $m = 5$. For the deterministic environments we were interested in the rise time for the optimal path. In the stochastic case we also need the agent to perform well if it deviates from the optimal path because of stochasticity. Therefore the rise time is not a good evaluation metric in this experiment. Next to the normalized average reward per test episode we also evaluate the correctness of the policy during each test episode. We define the correctness of the learned policy as follows. For each state $\mathbf{s}$ the agent has learned a policy $\pi(\mathbf{s})$, which is compared to the optimal policy $\pi^*(\mathbf{s})$. The optimal policy is determined using dynamic programming with the true transition model and reward function. We use the ratio π/π^* to measure how the policy changes and not as a measure of how good the policy really is, the policy can be correct with a completely wrong value function. Therefore this measure can only give us an indication of what is happening and cannot be used as a definite measure. By averaging over multiple experiments we minimize this error and get a reliable change in ratio that tells us how the policy changes during learning.

Discussion

The normalized average reward per test episode versus the number of training steps for Q-Learning is depicted in Figure 5-6 and for MBRL and R-MAX in Figure 5-7. The Smart Start method shows good performance on all three environments. ϵ-greedy exploration always performs better in combination with Smart Start. Boltzmann and UCB1 exploration do not differ much in performance on the Easy and Medium gridworld environments. On the Maze gridworld environment Boltzmann and UCB1 exploration perform better in combination with Smart Start. For R-MAX we see a slightly worse performance with Smart Start, similar to the results found for the deterministic gridworld environments. R-MAX already has a good directed exploration strategy, going to the Smart Start state just decreases the sample efficiency. MBRL does not show much difference in performance when Smart Start is used. Both R-MAX and MBRL achieve a lower average reward than Q-Learning, suggesting that they did not learn a good policy. The policy is directly derived from the learned model and reward function. This suggest that both R-MAX and MBRL were not able to learn a good model and reward function.

Figures 5-8 and 5-9 show the ratio of policy correctness versus the number of training steps for Q-Learning, MBRL and R-MAX. We see similar results on the Easy and Medium gridworld environments as in Figure 5-6. ϵ-greedy again performs better with Smart Start for each environment. But when we look at the policy correctness for Boltzmann and UCB1 exploration on the Maze gridworld environment it is different. With Smart Start, both learn a decent policy faster but never reach the quality of the policy learned without Smart Start. This is because the Maze environment contains parts that are suboptimal to go through for the agent. These regions can be identified in the top and bottom of Figure 4-1c. With Smart Start, the agent will not go through these regions anymore and not learn a good policy for those regions. In this experiment the agent did learn the optimal policy, but it is possible that the agent learns a suboptimal policy instead. Something interesting can be seen for R-MAX, the policy learned with Smart Start performs better than without Smart Start. But it takes longer to learn the policy. The early peak is due to the characteristics of the environments. The exploration strategy of R-MAX drives the agent to new states. This exploration policy coincides with the correct policy, resulting in a very good policy correctness. But once the agent has explored the state space more thoroughly and starts constructing a policy based on the learned model and reward function, the policy correctness decreases again. This suggest the agent has not gathered enough samples to learn an accurate enough model for the entire state space. Because Smart Start spends more time between getting to the Smart Start state and exploring using the exploration policy of R-MAX, it will gather more samples to construct a more accurate model. The policy correctness of MBRL with Smart Start is slightly better than without Smart Start.

As was shown in Section 5-1, the exploration performance of Smart Start shows a large improvement. This explains why algorithms with Smart Start are able to learn a good solution faster than algorithms without Smart Start. The Smart Start parameters experiment in Section 5-2 showed that the states close to the goal are being chosen as Smart Start states more often. With these findings we can explain why Smart Start learns a less robust policy. After finding the goal, almost every episode that starts with Smart Start will choose a Smart Start state close to the goal. The agent is subsequently guided to the Smart Start state using an optimal policy. The optimal policy will not come close to large parts of the state space in the Maze gridworld environment, because they are far away from the optimal policy. From Figure 4-1c you can see that the bottom and top part are far away from the optimal solution through the middle. Therefore the agent will not learn a good policy for these parts of the state space.

The final experiment on a deterministic gridworld environment with misleading reward is presented in the next section.

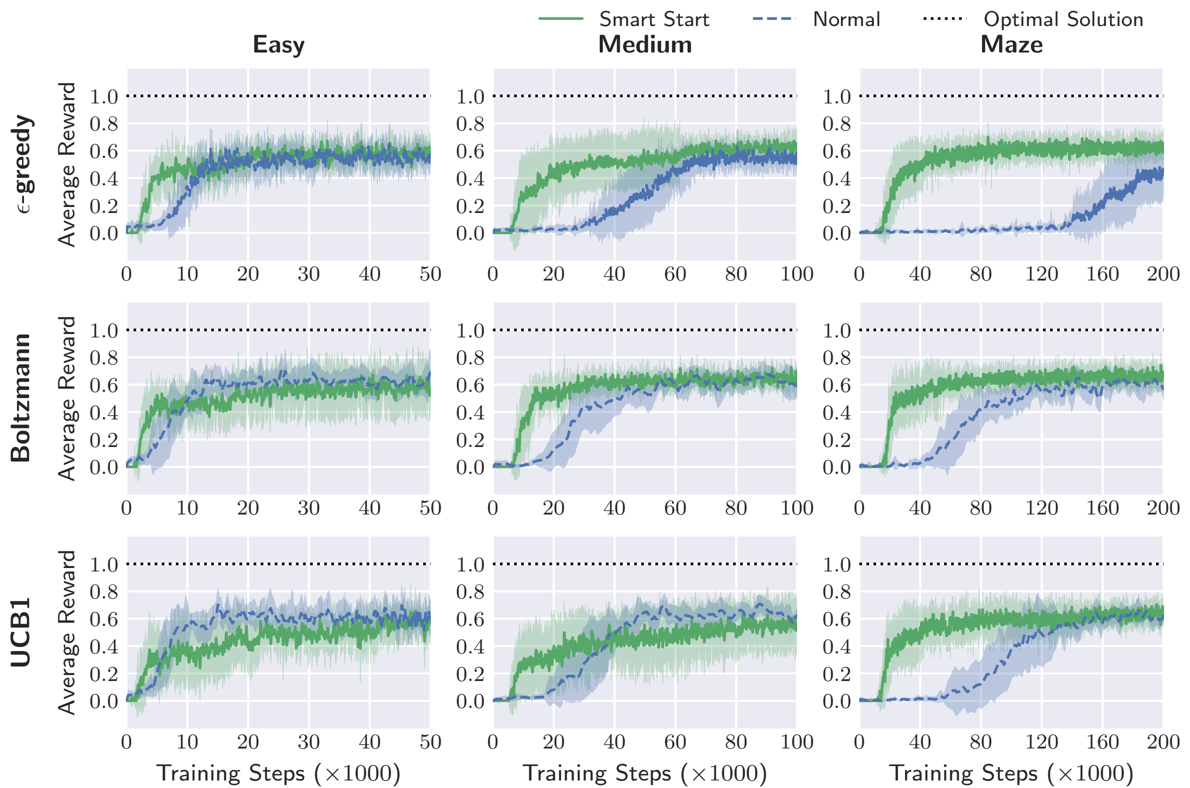


Figure 5-6: Normalized average reward of Smart Start for Q-Learning with ϵ-greedy, Boltzmann and UCB1 exploration on stochastic environments. Evaluated on Easy, Medium and Maze gridworld environments.

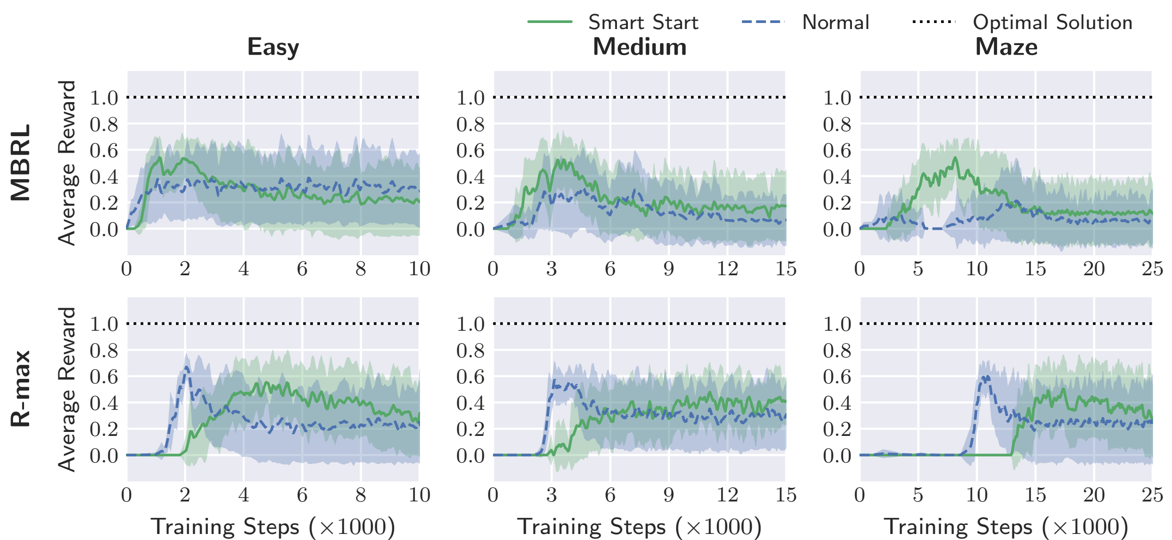


Figure 5-7: Normalized average reward of Smart Start for MBRL and R-MAX on stochastic environments. Evaluated on Easy, Medium and Maze gridworld environments.

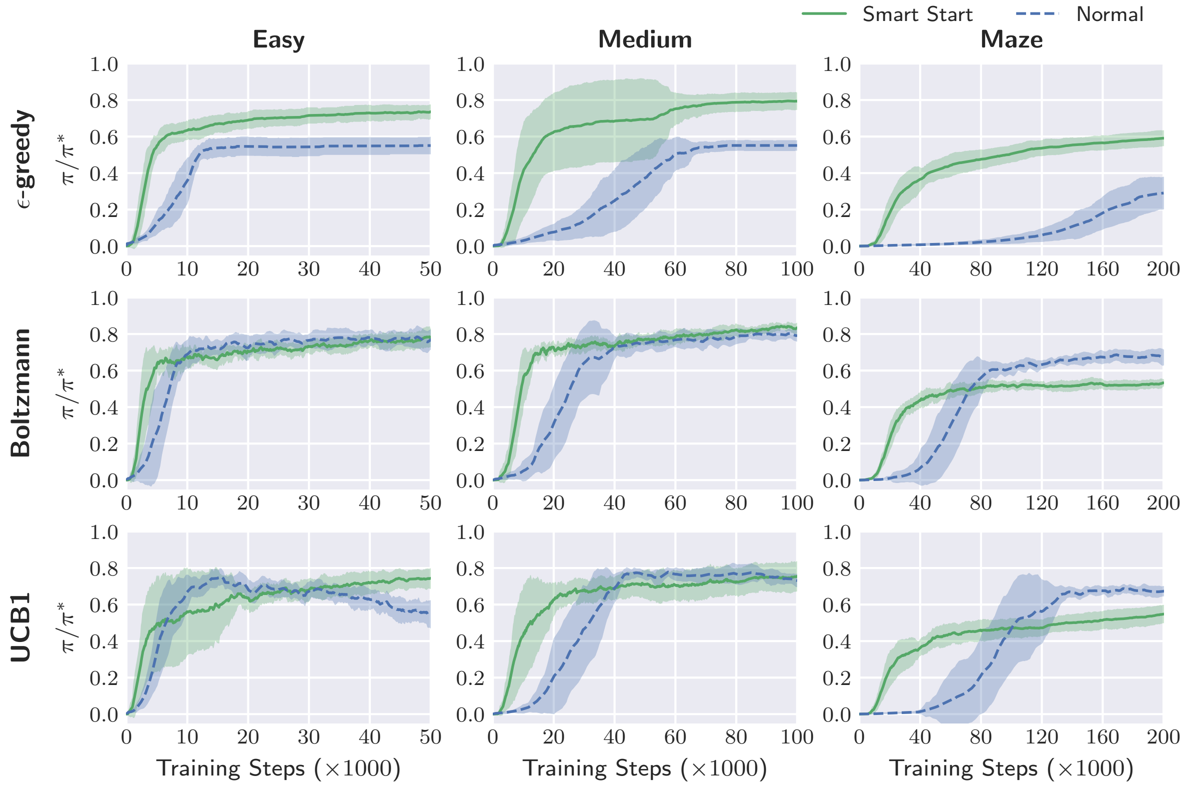


Figure 5-8: Policy correctness of Smart Start for Q-Learning with ϵ-greedy, Boltzmann and UCB1 exploration on stochastic environments. Evaluated on Easy, Medium and Maze gridworld environments.

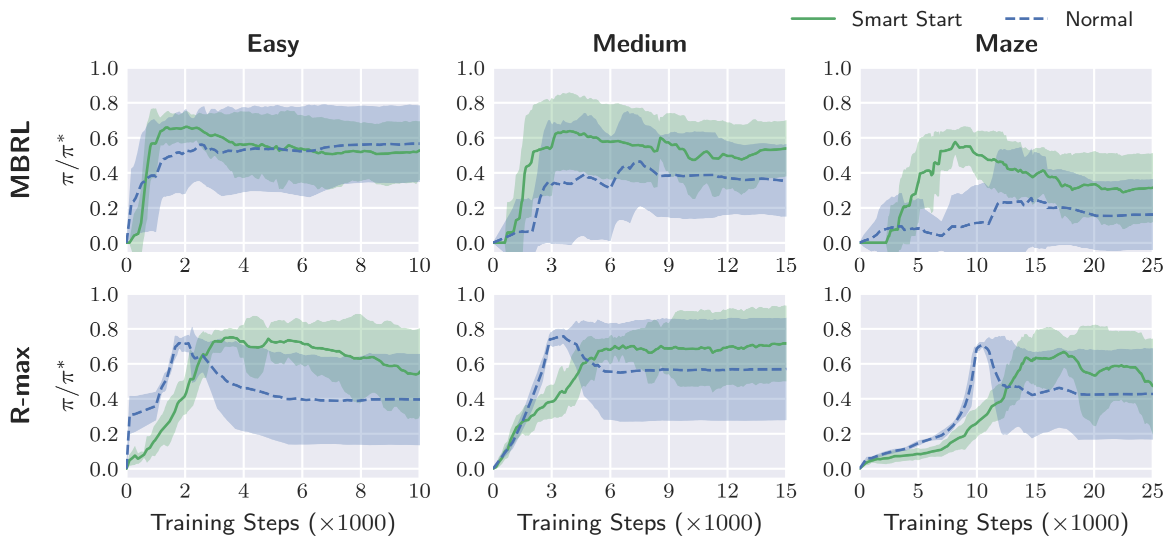


Figure 5-9: Policy correctness of Smart Start for MBRL and R-MAX on stochastic environments. Evaluated on Easy, Medium and Maze gridworld environments.

5-5 Deterministic Gridworld with Misleading Rewards

The final experiment evaluates the performance of Smart Start in a gridworld environment with misleading rewards, see Section 4-3 for details on the Misleading gridworld environment. The environment has been designed to trick the agent into going for the suboptimal goal, which will result in a lower accumulated reward. A persistent and directed exploration strategy is required to make sure the agent explores the state space thoroughly. We only evaluate the performance for an environment with deterministic dynamics. We want to encourage the agent to use Smart Start mainly for exploration, therefore a high value is used for the Smart Start parameter c_{ss}.

Experimental Details

For experiments with Smart Start maximum episode length of $T_{\text{episode}} = 100$ is used, without Smart Start $T_{\text{episode}} = 250$ is used. A maximum simulation time of $T_{\max} = 25000$ is used. The Smart Start parameters were set to $\eta = 0.8$ and $c_{ss} = 100.0$. After every 100 training steps a test episode is executed, the test results are used for evaluation. The rise time is used for evaluating the performance of each algorithm. A maximum reward of $r = 100.0$ can be obtained in the Misleading gridworld environment, which has to be taken into account in calculating the rise time and normalized average reward. The rise time parameter is set to $\varepsilon = 0.1$. When the optimal solution is not reached in the given amount of time-steps a rise time of $\tau_{\text{rise}} = T_{\max}$ is given. For R-MAX the maximum reward value is set to $R_{\max} = 250$ and a threshold parameter $m = 2$.

Discussion

The rise times of different algorithms are depicted in Figure 5-10 and the normalized average reward versus the number of training steps in Figure 5-11. It is immediately clear that random exploration does not perform well for this problem. ϵ-greedy, Boltzmann, UCB1 and MBRL quickly find the suboptimal solution and are not persistent enough to discover the optimal solution. R-MAX does have a persistent exploration strategy and learns the optimal solution. With Smart Start all algorithms are able to find the optimal solution, although Q-Learning does show a lot of variation in the results. MBRL with Smart Start performs best for this task and even outperforms R-MAX. This is due to the fact that R-MAX exhaustively tries every state-action pair a minimum number of times, even state-action pairs that terminate the episode. Varying the value for $R_{\max}$ can improve its behavior, but it is often undesirable to perfectly tune each parameter to the problem at hand.

In the next section a discussion is given on the results.

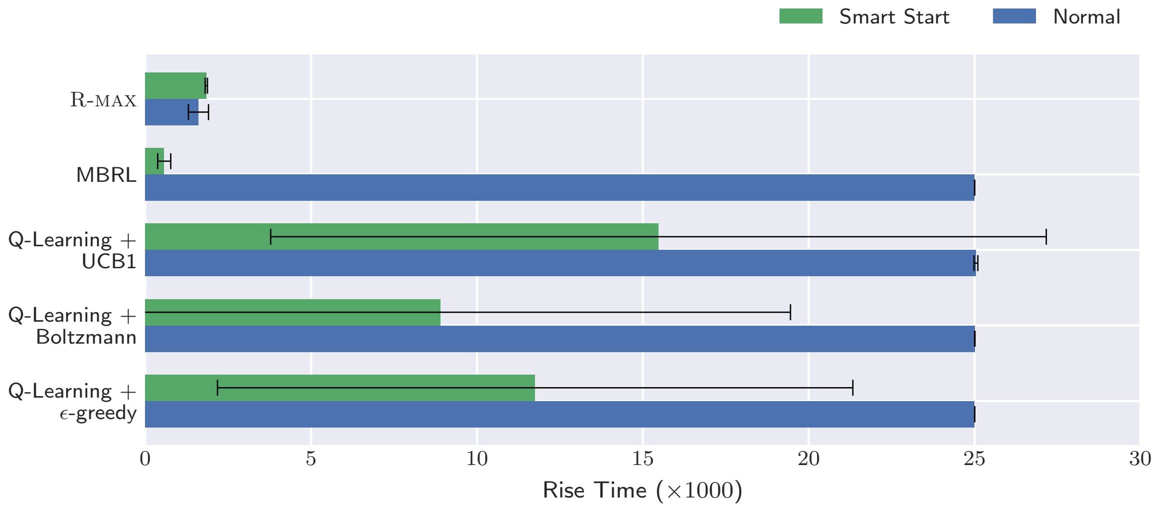


Figure 5-10: Rise time of Smart Start on the Misleading gridworld environment. MBRL, R-MAX and Q-Learning with ϵ-greedy, Boltzmann and UCB1 exploration were evaluated with and without Smart Start. A rise time parameter of $\varepsilon = 0.1$ is used. The average rise time and standard deviation are depicted.

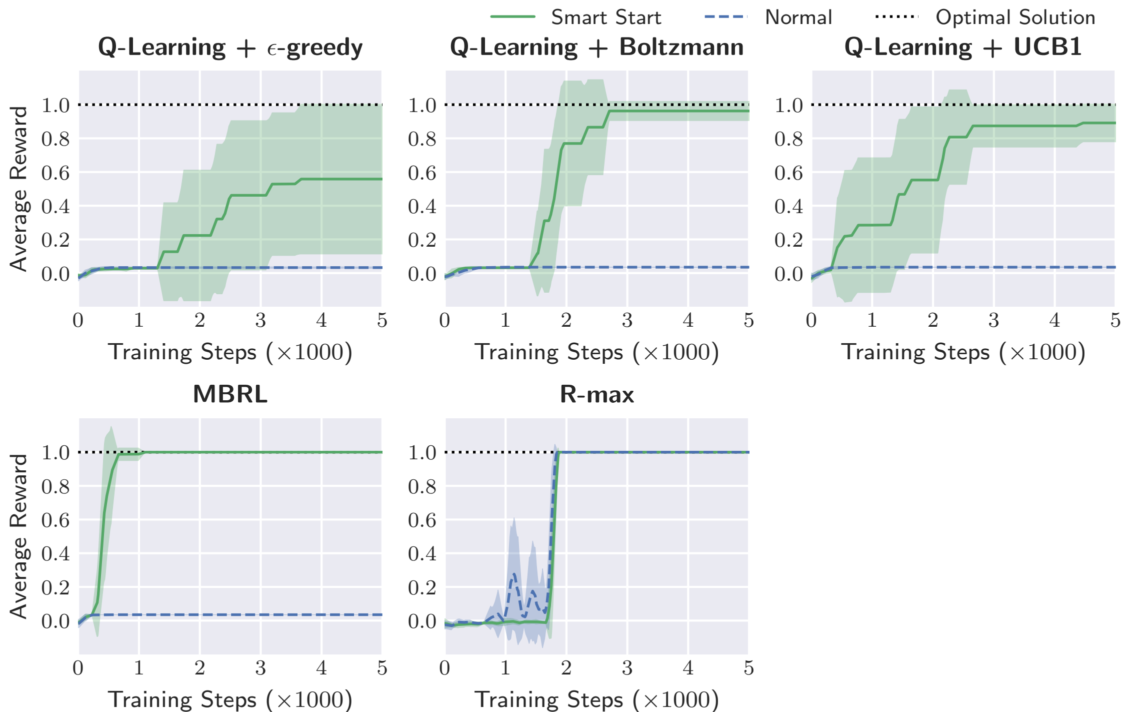


Figure 5-11: Normalized average reward of Smart Start on the Misleading gridworld environment. Evaluated for MBRL, R-MAX and Q-Learning with ϵ-greedy, Boltzmann and UCB1 exploration.

5-6 Discussion

The previous section presented the results of this work. This section gives a discussion on the results.

Exploration Performance

Smart Start showed great performance in exploring the state space for each exploration strategy and outperformed their counterparts without Smart Start. This showed that Smart Start is a good method for efficiently exploration.

Smart Start Parameters

The influence of the Smart Start parameters is hard to asses. The parameters depend on the problem you are trying to solve, so it is hard to deduct a clear answer from the results. We used a high value for η, which resulted in better performance because this brings the agent often close to the goal in a minimal number of time-steps. Varying the parameter c_{ss} did not present clear differences in the results. This is due to the environment characteristics and the way the Smart Start state is selected. The value function will stay *zero* for a long time, meaning the Smart Start state selection only depends on the state visitation density. By the time the goal is reached for the first time, most of the state space is already explored, giving states close to the goal not only the highest value function but also the lowest visitation density. These experiments therefore do not give a clear answer on what values to choose for the Smart Start parameters.

Deterministic Environment

The most interesting part of the results is the learning performance. Smart Start was evaluated for MBRL, R-MAX and Q-Learning with ϵ-greedy, Boltzmann and UCB1 exploration. From Figure 5-3 it is clear that Smart Start gives a huge increase in performance for Q-Learning. With R-MAX it was expected Smart Start would perform less well, because R-MAX already explores uncertain areas of the state space efficiently. In R-MAX an exploration bonus is given to uncertain parts of state space, but the Smart Start state may be on the other side of the state space. This results in going to the Smart Start state first and then going back to explore other uncertain parts. This results in a less sample efficient algorithm. MBRL with Smart Start was able to match the performance of R-MAX and showed that Smart Start can improve the exploration of simple reinforcement learning algorithms considerably.

Stochastic Environment

For the stochastic case, Q-Learning with Smart Start was still able to learn a good policy a lot faster than without Smart Start. But the overall robustness of the policy with Smart Start was less for Boltzmann and UCB1 exploration on the Maze gridworld environment. Smart Start quickly starts choosing Smart Start states close to the goal because the value function is larger in that region. The agent goes to that region using an optimal policy, which does

not go through suboptimal parts of the state space. This results in a less robust policy for parts of the state space that are not close to the optimal policy. This can be varied using the Smart Start parameters η and c_{ss}.

Misleading Rewards

When misleading rewards are present most exploration strategies, like ϵ-greedy, Boltzmann and UCB1, fail to properly explore the state space. Their exploration is not persistent enough to overcome the exploitation of the reinforcement learning agent. A persistent exploration framework like Smart Start does not suffer from this problem and is able to guide the agent efficiently throughout state space. Smart Start showed great performance and was able to outperform R-MAX when combined with MBRL.

We were able to achieve similar performance as R-MAX. R-MAX has to learn a global model for its planning, furthermore the model has to be good enough to be used by a planning algorithm. This is a trivial task in small discrete environments, but for more complex environments with continuous states this is nontrivial. This was already observed in the results on stochastic environments with sparse rewards in Section 5-4. The model-based agents were not able to achieve the same performance as Q-Learning, because they failed to learn a proper model. The guiding to the Smart Start state is done using a global method in this work, but that is not necessary. A local model or trajectory optimization method can also be used to guide the agent to the Smart Start state. The model does not have to be perfect because it is only used for guiding the agent to an area that needs more exploration. This makes Smart Start compatible for a wide range of situations and very promising for future work.

Chapter 6

Conclusion

In this work, a novel exploration framework for reinforcement learning was developed. The framework, called Smart Start, can be used in combination with different reinforcement learning algorithms and exploration strategies. Current state of the art deep reinforcement learning algorithms often take millions of samples to learn a good solution. One of the main issues is the lack of good exploration in large complex, often, continuous environments. Where model-based reinforcement learning performs extremely well for discrete state spaces, they often cannot be converted to problems with continuous state spaces easily. Resulting in sub-optimal performance as to what model-free reinforcement learning can achieve. Smart Start tries to fill this gap by introducing a persistent and directed exploration framework that gives reinforcement learning algorithms the exploration performance of model-based algorithms without reducing the reinforcement learning algorithms its performance.

The Smart Start framework was created for environments with sparse or misleading rewards. The framework is aimed at episodic problems that start each episode at the same initial state. The idea of Smart Start is to split the episode into two parts, the Smart Start phase and "normal" learning phase. In normal reinforcement learning the agent executes its learned policy with some exploration strategy, e.g., Q-Learning with ϵ-greedy, from initial start state until the end of an episode. This is what we call "normal" learning.

With Smart Start the agent first starts with the Smart Start phase before "normal" learning, see Figure 3-1. The Smart Start phase can be split up in two parts. The first part is determining a region where to explore based on previous experiences. The region is denoted by a single state from which the agent expects to learn the most if it was able to start from that state. This state is called the Smart Start state. After the Smart Start state has been selected the agent is guided from the initial state to the Smart Start state. Once the agent is close to the Smart Start state it continues with "normal" learning.

Here, the selection of the Smart Start state is modeled as a multi-armed bandit problem and solved using the UCB1 algorithm. The UCB1 algorithm constructs an upper confidence bound based on the value function of the reinforcement learning algorithm that is being used and uncertainty of the state visitation density. The guiding can, for example, be done using a

trajectory optimization or model-based planning approach. Where model-based reinforcement learning methods often need to learn a good global model for the planner, Smart Start only requires a local model and planner that is good enough to bring the agent close to the Smart Start state. Value Iteration was used as trajectory optimization for the Smart Start framework in this work.

In Section 6-1 the conclusions of this work will be presented. Section 6-2-2 discusses several directions for future work.

6-1 Conclusion

We evaluated the Smart Start framework using three reinforcement learning algorithms, MBRL, R-MAX and Q-Learning. For Q-Learning we used three different exploration strategies, ϵ-greedy, Boltzmann and UCB1. Four gridworld environments of different sizes and complexities were created for the experiments. The first experiment looked at the exploration performance of Smart Start in combination with three exploration strategies: random, UCB1 and model-based exploration. The exploration performance was measured as the average number of training steps it took the agent to reach the goal for the first time. The experiments showed that Smart Start has good exploration performance and does not suffer from shorter episode lengths which is the case when Smart Start is not used.

In the second experiment, the influence of the Smart Start parameters η and c_{ss} was investigated. Because of the nature of the environment with sparse rewards and the selection of the Smart Start state it was difficult to clearly asses the influence of c_{ss}. For η we did find an increase in performance for higher values. This can be explained by the deterministic environment and the way the Smart Start state is chosen. From this experiment we cannot clearly conclude how to choose the parameters and depending on the problem you are trying to solve, the parameters should be chosen accordingly.

Next, we evaluated the learning performance of Smart Start. The third experiment was done on deterministic gridworld environments. Smart Start showed a huge improvement in performance for Q-Learning with various exploration strategies. The agent was able to quickly achieve a near optimal average reward during the test episodes. The performance of R-MAX was slightly worse with Smart Start enabled. This was expected, since R-MAX already performs near optimal in situations like these. Therefore, Smart Start slows down the agent in terms of sample efficiency. Nevertheless, the decrease in performance was minimal. MBRL with Smart Start showed similar and sometimes better performance than R-MAX, showing that Smart Start can improve the exploration of reinforcement learning algorithms significantly.

The fourth experiment also evaluated the learning performance of Smart Start, but on stochastic environments. In terms of average reward per test episode, the agent quickly learned a proper solution. But finding a good solution can take longer in the smaller environments when Smart Start is being used. For the most complex environment, Smart Start clearly outperforms the algorithms without Smart Start. But when we look at the learned policy of the agent, i.e., the learned policy compared to the optimal policy for each state, we see a different result. On the smaller environments Smart Start performs better. But on the most complex environment, UCB1 and Boltzmann exploration outperform Smart Start. This

is because the complex environment contains suboptimal parts to go through for the agent, covering a large portion of the state space. With Smart Start the agent will not be guided through those suboptimal regions anymore. Resulting in Smart Start learning a good policy around the optimal path quicker, but a less robust policy for the state space as a whole. Changing the Smart Start parameters will influence these results. Depending on the problem, the user has to decide what are desired parameter settings. For problems with a large state space it is often infeasible to learn a good robust policy for every state, but just a good policy around the optimal solution is sufficient. The results obtained in this work suggest that Smart Start is able to efficiently solve the problem with large state spaces. MBRL and R-MAX were not able to achieve the same performance as Q-Learning because they failed to learn a proper model. For more complex environments this becomes even more difficult. Smart Start with a model-free reinforcement learning algorithm is a promising alternative to a purely model-based reinforcement learning algorithm.

The final experiment looked at the performance of Smart Start in the case of misleading rewards. Most algorithms without Smart Start were not able to learn the optimal solution, clearly showing the need for a persistent and directed exploration strategy. R-MAX was able to learn the optimal solution, this is expected because R-MAX already has a persistent exploration strategy. Smart Start increased the performance of Q-Learning and MBRL significantly. MBRL with Smart Start even outperformed R-MAX.

We showed that Smart Start can improve the exploration of reinforcement learning agents on discrete gridworld environments. This directly resulted in a more sample efficient performance for the overall learning. The Smart Start framework can easily be incorporated with various reinforcement learning algorithms and exploration strategies. This makes Smart Start a very interesting and promising exploration framework for reinforcement learning problems. The next section proposes directions for future research.

6-2 Recommendations

The Smart Start framework was designed for environments with sparse or misleading rewards. In this work, we evaluated the performance of Smart Start in discrete environments. But, but as a framework it is not limited to discrete environments with sparse or misleading rewards. This immediately gives rise to interesting directions for future research.

The next section, Section 6-2-1, introduces a proposal for environments with non-sparse rewards. Section 3-4-2 provides interesting ideas for using Smart Start in continuous environments. Finally, different ideas for choosing the Smart Start state are discussed in Section 6-2-3.

6-2-1 Non-Sparse Rewards

Smart Start has been designed for environments with misleading or sparse rewards. For environments with non-sparse rewards, Smart Start can be interesting as well. The Smart Start state selection is based on the of the value function and visitation density of states. The value function is a clear measure of how good a state is, especially in non-sparse reward environments. This suggests Smart Start should still be able to perform well in these environments.

In this work, the goal of the trajectory optimization was to get to the Smart Start state as reliable and quickly as possible. For environments with sparse rewards, this results in the shortest path to the Smart Start state. For non-sparse environment this might not be the desirable path to follow, maybe you do not want to go through regions with large penalties for instance. An interesting addition would be to include the value function in the trajectory optimization step. This will result in trajectories to the Smart Start state that are more beneficial to the agent, by passing through regions with high rewards for example.

6-2-2 Continuous Environments

Another interesting class of problems are environments with continuous state and/or action spaces. In this work, we have only looked at discrete problems, but for discrete problems good model-based solutions are already available, e.g., R-MAX. But these algorithms are nontrivial to convert to continuous domains. Smart Start can be a really interesting solution, because it is able to augment reinforcement learning algorithms with a directed and persistent exploration strategy. Guidelines on how Smart Start can be implemented for continuous domains were presented in Section 3-4-2.

6-2-3 Smart Start State Selection

The Smart Start state selection is modeled as a multi-armed bandit problem and in this work we used the UCB1 algorithm for choosing the Smart Start state. Many algorithms exist for solving the multi-armed bandit problem [45]. It would be interesting to look at different algorithms and see which one is most suited for the Smart Start state selection.

We want the Smart Start state to result in the most useful information. Useful information can for example be a high value function or a region close to unvisited states. In this work, the upper confidence bound for the Smart Start state consists of the value function and the bound is based on the state visitation density. We only looked at single discrete states and did not take related states into account. Using a metric that defines the relation between states is an interesting addition. The metric can improve exploration by making the search cruder, avoiding exhaustively visiting every single state. A possible solution is to use kernel density estimation, an implementation is given in Appendix B. Different expressions might be more suited to represent the Smart Start state and are an interesting topic for future research.

In this work we evaluated Smart Start using value function methods. In the given implementation the Smart Start state is based on the value function of the state. For policy search methods this poses a problem, since no value function is learned. The Smart Start framework can still be used to improve exploration, by only using the uncertainty of the state visitation density for the Smart Start state selection and disregarding the value function.

Appendix A

Model-Based Reinforcement Learning with Smart Start

In this appendix the implementation of the MBRL algorithm with Smart Start is given. The full algorithm is given in Algorithm 7. The MBRL algorithm was given in Section 2-2-2. The update of the MBRL algorithm is done in the UPDATE function in Algorithm 7.

No exploration strategy is used by the MBRL algorithm. Therefore, as long as the agent does not find a reward it will use random exploration, since the value function is *zero*. But as soon as a reward is found, the agent switches to exploitation. This gives the MBRL algorithm really bad exploration, but the agent should be able to explore efficiently when Smart Start is being used.

Algorithm 7 MBRL with Smart Start

```
1: Initialize Q(s, a) arbitrarily for all s ∈ S and a ∈ A(s)
2:
3: for each episode do
4:     Initialize s_0 and t = 0
5:     u ~ U(0, 1)
6:     if u ≤ η and Σ_{s∈S} C(s) > 0 then
7:         // Choose Smart Start state using upper confidence bound
8:         s_ss = argmax_s [ max_a Q(s, a) + c_ss sqrt( log Σ_{s∈S} C(s) / C(s) ) ] ∀ {s ∈ S : C(s) > 0}
9:
10:        // Fit reward function
11:        for all s, a do
12:            for all s' ∈ C(s, a, ·) do
13:                if s' = s_ss then
14:                    R_ss(s, a) = 1
15:                else
16:                    R_ss(s, a) = 0
17:
18:        // Obtain policy to s_ss using value iteration
19:        π_ss = VALUE ITERATION(P, R_ss) (See Algorithm 1)
20:
21:        // Execute Smart Start policy
22:        repeat
23:            Choose a_t = π_ss(s_t)
24:            Take action a_t and observe s_{t+1} and r_{t+1}
25:            π = UPDATE(s_t, a_t, s_{t+1}, r_{t+1}) (See Algorithm 2)
26:            t ← t + 1
27:        until s_t = s_ss, s_t is terminal or t = T_episode
28:
29:    // Continue using learned policy and exploration strategy
30:    if s_t is not terminal and t < T_episode then
31:        repeat
32:            Choose a_t = π(s_t)
33:            Take action a_t and observe s_{t+1} and r_{t+1}
34:            π = UPDATE(s_t, a_t, s_{t+1}, r_{t+1}) (See Algorithm 2)
35:            t ← t + 1
36:        until s_t is terminal or t = T_episode
```

Appendix B

Continuous Smart Start State

For continuous state spaces a method is needed for approximating the visitation density of states. The method needs a metric that specifies a relation between states. Instead of keeping track of counts for each state, we store visited states in a buffer $\mathcal{D} = \{\mathbf{s}_0, \mathbf{s}_1, \ldots\}$. For the discrete case, if we randomly choose a state from $\mathcal{D}$, the probability of picking state $\mathbf{s}$ is

$$\Pr[\mathbf{S} = \mathbf{s}] = \frac{C(\mathbf{s})}{|\mathcal{D}|}, \tag{B-1}$$

where $|\mathcal{D}|$ is the number of states in the buffer. We can estimate the count $\hat{C}(\mathbf{s})$ for continuous states using Equation (B-1) when the probability distribution of $\mathcal{D}$ is known. The probability distribution of a random variable can be estimated using kernel density estimation [37]. The kernel density estimation for state $\mathbf{s} \in \mathcal{D}$ is

$$\rho(\mathbf{s}) = \frac{1}{|\mathcal{D}|h} \sum_{j=0}^{|\mathcal{D}|} K\left(\frac{\mathbf{s} - \mathbf{s}_j}{h}\right), \tag{B-2}$$

where K is the kernel and $h > 0$ the bandwidth used for modifying the width of the kernel. The kernel contains the metric that tells us how states are related to each other. Many different kernels exists. The most used kernel is the Gaussian kernel, which uses the euclidean distance as metric

$$K(\mathbf{a}) = e^{-\frac{1}{2}\mathbf{a}^2}. \tag{B-3}$$

The estimation for the count becomes

$$\hat{C}(\mathbf{s}) = \rho(\mathbf{s})|\mathcal{D}| = \frac{1}{h} \sum_{j=0}^{|\mathcal{D}|} K\left(\frac{\mathbf{s} - \mathbf{s}_j}{h}\right). \tag{B-4}$$

For continuous states and discrete actions we get the following equation for determining the smart start state $\mathbf{s}_{ss}$

$$\mathbf{s}_{ss} = \arg\max_{\mathbf{s}} \left[\max_{\mathbf{a}} Q(\mathbf{s}, \mathbf{a}) + \beta \sqrt{\frac{\log |\mathcal{D}|}{\hat{C}(\mathbf{s})}} \right]. \tag{B-5}$$

The inequalities in Equations (3-4) and (3-5) still hold with the assumptions that $\hat{C}(\mathbf{s}) = C(\mathbf{s})$ and the probability decreases with the size of the buffer $|\mathcal{D}|$.

Calculating the kernel density estimation for every state in the buffer $\mathcal{D}$ scales quadratically with the size $|\mathcal{D}|$ and has a computational complexity of $\mathcal{O}(|\mathcal{D}|^2)$. This is undesirable because our set of visited states in buffer $\mathcal{D}$ grows rapidly and usually contains thousands of values. To overcome this problem we do not calculate the density for every state in our set. We uniformly sample a subset $\mathcal{D}_{ss}$ containing n_{ss} states from our buffer $\mathcal{D}$ and assume this gives a good representation of the complete buffer $\mathcal{D}$. The complexity reduces to $\mathcal{O}(n_{ss}|\mathcal{D}|)$, which grows linearly with the size $|\mathcal{D}|$ for constant n_{ss}.

Bibliography

[1] R. E. Bellman, *Dynamic Programming.* Princeton, NJ, USA: Princeton University Press, 2010.

[2] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction.* MIT press Cambridge, 1998, vol. 1, no. 1.

[3] S. B. Thrun, "The role of exploration in learning control," *Handbook of Intelligent Control: Neural, Fuzzy and Adaptive Approaches*, pp. 1–27, 1992.

[4] S. Thrun, "Efficient exploration in reinforcement learning." Pittsburgh, PA, Tech. Rep. CMU-CS-92-102, January 1992.

[5] C. J. C. H. Watkins, "Learning from delayed rewards," Ph.D. dissertation, King's College, Cambridge, 1989.

[6] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[7] J. Z. Kolter and A. Y. Ng, "Near-Bayesian exploration in polynomial time," in *Proceedings of the 26th Annual International Conference on Machine Learning.* ACM, 2009, pp. 513–520.

[8] A. L. Strehl and M. L. Littman, "An analysis of model-based interval estimation for Markov decision processes," *Journal of Computer and System Sciences*, vol. 74, no. 8, pp. 1309–1331, 2008.

[9] J. Schmidhuber, "Adaptive confidence and adaptive curiosity," Technische Universität München, Institut für Informatik, Tech. Rep. FKI-194-91, 1991.

[10] S. B. Thrun and K. Möller, "Active exploration in dynamic environments," in *Advances in Neural Information Processing Systems 4*, 1992, pp. 531–538.

[11] M. Wiering and J. Schmidhuber, "Efficient model-based exploration," in *Proceedings of the Sixth International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, vol. 6, 1998, pp. 223–228.

[12] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multi-armed bandit problem," *Machine Learning*, vol. 47, no. 2, pp. 235–256, 2002.

[13] W. Hoeffding, "Probability inequalities for sums of bounded random variables," *Journal of the American Statistical Association*, vol. 58, no. 301, pp. 13–30, 1963.

[14] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," in *Machine Learning: ECML 2006*. Springer Berlin Heidelberg, 2006, pp. 282–293.

[15] P.-Y. Oudeyer and F. Kaplan, "How can we define intrinsic motivation?" in *in Proceedings of the 8th International Conference on Epigenetic Robotics: Modeling Cognitive Development in Robotic Systems*. Lund University Cognitive Studies, 139, 2008.

[16] A. G. Barto, "Intrinsic motivation and reinforcement learning," in *Intrinsically Motivated Learning in Natural and Artificial Systems*. Springer Berlin Heidelberg, 2013, pp. 17–47.

[17] A. Y. Ng, D. Harada, and S. J. Russell, "Policy invariance under reward transformations: Theory and application to reward shaping," in *Proceedings of the Sixteenth International Conference on Machine Learning*, 1999, pp. 278–287.

[18] T. Hester and P. Stone, "Learning and using models," in *Reinforcement Learning: State-of-the-Art*. Springer Berlin Heidelberg, 2012, pp. 111–141.

[19] R. S. Sutton, "Integrated architectures for learning, planning, and reacting based on approximating dynamic programming," in *Proceedings of the Seventh International Conference on Machine Learning*, 1990, pp. 216–224.

[20] R. I. Brafman and M. Tennenholtz, "R-MAX - A general polynomial time algorithm for near-optimal reinforcement learning," *Journal of Machine Learning Research*, vol. 3, pp. 213–231, 2002.

[21] M. J. Kearns and S. P. Singh, "Near-optimal reinforcement learning in polynomial time," *Machine Learning*, vol. 49, no. 2-3, pp. 209–232, 2002.

[22] N. K. Jong and P. Stone, "Model-based exploration in continuous state spaces," in *Abstraction, Reformulation, and Approximation*. Springer Berlin Heidelberg, 2007, pp. 258–272.

[23] T. Jung and P. Stone, "Gaussian processes for sample efficient reinforcement learning with RMAX-like exploration," in *Machine Learning and Knowledge Discovery in Databases*. Springer Berlin Heidelberg, 2010, pp. 601–616.

[24] S. Gu, T. P. Lillicrap, I. Sutskever, and S. Levine, "Continuous deep Q-Learning with model-based acceleration," in *Proceedings of the 33nd International Conference on Machine Learning*. PMLR, 2016, pp. 2829–2838.

[25] A. Nagabandi, G. Kahn, R. S. Fearing, and S. Levine, "Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning," *arXiv preprint arXiv:1708.02596*, 2017.

[26] M. G. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos, "Unifying count-based exploration and intrinsic motivation," in *Advances in Neural Information Processing Systems 29*. Curran Associates, Inc., 2016, pp. 1471–1479.

[27] H. Tang, R. Houthooft, D. Foote, A. Stooke, X. Chen, Y. Duan, J. Schulman, F. De Turck, and P. Abbeel, "#Exploration: A study of count-based exploration for deep reinforcement learning," in *Advances in Neural Information Processing Systems 30*. Curran Associates, Inc., 2017, pp. 2753–2762.

[28] G. Ostrovski, M. G. Bellemare, A. van den Oord, and R. Munos, "Count-based exploration with neural density models," in *Proceedings of the 34th International Conference on Machine Learning*, vol. 70. PMLR, 2017, pp. 2721–2730.

[29] J. Fu, J. Co-Reyes, and S. Levine, "EX2: Exploration with exemplar models for deep reinforcement learning," in *Advances in Neural Information Processing Systems 30*. Curran Associates, Inc., 2017, pp. 2577–2587.

[30] R. Houthooft, X. Chen, X. Chen, Y. Duan, J. Schulman, F. De Turck, and P. Abbeel, "Vime: Variational information maximizing exploration," in *Advances in Neural Information Processing Systems 29*. Curran Associates, Inc., 2016, pp. 1109–1117.

[31] J. Achiam and S. Sastry, "Surprise-based intrinsic motivation for deep reinforcement learning," *arXiv preprint arXiv:1703.01732*, 2017.

[32] L. Avena, M. Heydenreich, F. den Hollander, E. Verbitskiy, and W. van Zuijlen, "Lecture notes on random walks," Mathematical Institute, Leiden University, 2016. [Online]. Available: http://websites.math.leidenuniv.nl/probability/lecturenotes/RandomWalks.pdf

[33] L.-J. Lin, "Reinforcement learning for robots using neural networks," Ph.D. dissertation, School of Computer Science, Carnegie Mellon University, 1993, uMI Order No. GAX93-22750.

[34] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[35] T. de Bruin, J. Kober, K. Tuyls, and R. Babuška, "Improved deep reinforcement learning for robotics through distribution-based experience retention," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2016, pp. 3947–3952.

[36] T. L. Lai and H. Robbins, "Asymptotically efficient adaptive allocation rules," *Advances in Applied Mathematics*, vol. 6, no. 1, pp. 4–22, 1985.

[37] E. Parzen, "On estimation of a probability density function and mode," *The Annals of Mathematical Statistics*, vol. 33, no. 3, pp. 1065–1076, 1962.

[38] J. A. Boyan and A. W. Moore, "Generalization in reinforcement learning: Safely approximating the value function," in *Advances in Neural Information Processing Systems 7*. MIT Press, 1995, pp. 369–376.

[39] E. Todorov and W. Li, "A generalized iterative LQG method for locally-optimal feedback control of constrained nonlinear stochastic systems," in *Proceedings of the 2005 American Control Conference*, vol. 1, 2005, pp. 300–306.

[40] Y. Tassa, T. Erez, and E. Todorov, "Synthesis and stabilization of complex behaviors through online trajectory optimization," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2012, pp. 4906–4913.

[41] S. Levine and V. Koltun, "Guided policy search," in *Proceedings of the 30th International Conference on Machine Learning*, vol. 28, no. 3. PMLR, 2013, pp. 1–9.

[42] ——, "Learning complex neural network policies with trajectory optimization," in *Proceedings of the 31st International Conference on Machine Learning*, vol. 32, no. 2. PMLR, 2014, pp. 829–837.

[43] S. Levine and P. Abbeel, "Learning neural network policies with guided policy search under unknown dynamics," in *Advances in Neural Information Processing Systems 27*. Curran Associates, Inc., 2014, pp. 1071–1079.

[44] W. Li and E. Todorov, "Iterative linear quadratic regulator design for nonlinear biological movement systems." in *Proceedings of the First International Conference on Informatics in Control, Automation and Robotics*. IEEE, 2004, pp. 222–229.

[45] J. Vermorel and M. Mohri, "Multi-armed bandit algorithms and empirical evaluation," in *Machine Learning: ECML 2005*. Springer Berlin Heidelberg, 2005, pp. 437–448.

Acknowledgments

I would like to thank the Institute for Human and Machine Cognition (IHMC) in Florida, United States, for giving me the opportunity to conduct my research at their robotics lab. I am particularly grateful for the supervision that was given to me by Dr. Jerry Pratt. I would like to thank everybody else at IHMC for the advice and good times they gave me.

From Delft University of Technology I would like to express my great appreciation for my supervisor Dr.-Ing. Jens Kober. Also from Delft, I would like to thank the Committee members of my thesis defence, Prof. Dr. Ir. Martijn Wisse and Ir. Wouter Wolfslag.

I would like to offer my special thanks to my mother, brother and sister for all the support and advice they gave me. Lastly, I would like to thank my father, whose knowledge, inspiration and guidance has been indispensable to me.

Delft, University of Technology — Bart Keulen
February, 2018