

SEMESTER PROJECT REPORT
LASA



ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Learning from noisy demonstrations: the
role of compliance in exploration-exploitation
trade-off

Laboratory	Learning Algorithms and Systems Laboratory
Supervisors	Mahdi Khoramshahi, Sutcliffe Andrew
Semester	Spring

Contents

1	Introduction	2
1.1	Motivations	2
1.2	Background	2
1.2.1	Learning from demonstration	2
1.2.2	Transfer learning	2
1.3	Proposed approach	2
2	Reinforcement Learning	3
2.1	Formulation	3
2.1.1	Notations and first definitions	3
2.1.2	Markov decision process	3
2.1.3	State and action value function	4
2.1.4	Optimal policies	4
2.2	Dynamic Programing	5
2.2.1	Generalized policy iteration	5
2.2.2	The value iteration algorithm	5
2.3	Temporal differences methods	6
2.3.1	On-policy method : SARSA	6
2.3.2	Off-policy method : Q-learning	6
2.4	Grid-world examples	7
2.4.1	Dynamic Programing solving	7
2.4.2	SARSA solving	8
2.4.3	Q-learning solving	9
3	Compliant-based reinforcement learning	10
3.1	Problem selection	10
4	Results	11
	Conclusion	12

Chapter 1

Introduction

1.1 Motivations

1.2 Background

1.2.1 Learning from demonstration

1.2.2 Transfer learning

1.3 Proposed approach

Chapter 2

Reinforcement Learning

2.1 Formulation

2.1.1 Notations and first definitions

Reinforcement learning is a framework in which an *agent* (or a *learner*) learns its actions from interaction with its environment. The environment generates scalar values called *rewards*, that the agent is seeking to maximize over time.

Let \mathcal{S} denote the state space in which our agent evolves (the localization of a robot on a grid for instance), and $\forall s \in \mathcal{S}$ we will define the action state $\mathcal{A}(s)$, describing all possible action that can be taken by the agent at state s . When taking an action from a state s_t , the agent finds itself in a new state s_{t+1} where it receives a reward $r_{t+1} \in \mathbb{R}$. The action taken is sampled over a probability distribution from the joint space of state and action :

$$\begin{aligned} \pi : \mathcal{S} \times \mathcal{A}(s) &\rightarrow [0, 1] \\ (s, a) &\rightarrow \pi(s, a) \end{aligned} \quad (2.1)$$

where $\pi(s, a)$ is the probability of picking action a in state s . Such a distribution is called the agent's *policy*. The key goal of reinforcement learning is teaching an agent on how to change its policy to maximize its reward on the long run.

The agent indeed seeks to maximize the *expected return* R_t mapping the reward sequence into \mathbb{R} . A commonly used expression for this value employs a *discount factor* $\gamma \in [0, 1]$, allowing to make the agent's more sensible to rewards it will get in a close future :

$$R_t = \sum_{i=0}^T \gamma^i r_{t+1+i} \quad (2.2)$$

This also allows to adapt this formulation to continuous tasks, where there are no terminal states and the task goes on indefinitely (there are no *episodes* in the learning).

2.1.2 Markov decision process

To make the problem tractable, we ask for the state signal to comply with Markov's property, hence to be *memory-less*. For instance, we want to be able to write that, in a stochastic environment, $\forall s' \in \mathcal{S}$:

$$\mathbb{P}(s_{t+1} = s' | a_t, s_t, \dots, a_1, s_1) = \mathbb{P}(s_{t+1} = s' | s_t, a_t) \quad (2.3)$$

Hence, every reinforcement learning problem can be represented by a *Markov Decision Process*, that consists in a 5-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}(\cdot, \cdot), \mathcal{R}(\cdot), \gamma)$ where :

- ▷ \mathcal{S} is the agent's state space
- ▷ \mathcal{A} is the agent's action space
- ▷ $\forall s, s' \in \mathcal{S}, \forall a \in \mathcal{A}(s), \mathcal{P}_a(s, s') = \mathbb{P}(s_{t+1} = s' | s_t = s, a_t = a)$ is the probability that action a in state s will lead the agent to transitioning to state s' .

- ▷ $\forall s, s' \in \mathcal{S}, \forall a \in \mathcal{A}(s), \mathcal{R}_a(s, s')$ is the immediate reward perceived by the agent when transitioning from state s to s' when taking action a .
- ▷ γ is the discount factor.

A *finite Markov decision process* designates a MDP for which both the action and state space are finite.

2.1.3 State and action value function

Most of the reinforcement learning algorithms are based on evaluation value function. A value function is a function mapping the state space in \mathbb{R} , estimating how good (in terms of expected future reward) it is for the agent to be in a given space. More precisely, a value function $V^\pi(\cdot)$ evaluates the expected return of a state when following the policy π . $V^\pi(\cdot)$ is called the **state-value function**.

$$\forall s \in \mathcal{S}, \quad V^\pi(s) = \mathbb{E}_\pi [R_t | s_t = s] \quad (2.4)$$

The **action-value function** evaluates the value of taking a given action, and then following the policy π :

$$\forall s, a \in \mathcal{S} \times \mathcal{A}(s), \quad Q^\pi(s, a) = \mathbb{E}_\pi [R_t | s_t = s, a_t = a] \quad (2.5)$$

Both those functions satisfy particular recursive relationships known as the *Bellman equations*. It is shown that (see TODO quote Sutton) we have the following results :

Bellman equations for Markov Decision Process

- ▷ Bellman equation for the state-value function : $\forall s \in \mathcal{S}$

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(s, a) \sum_{s'} \mathcal{P}_a(s, s') [\mathcal{R}_a(s, s') + \gamma V^\pi(s')] \quad (2.6)$$

- ▷ Bellman equation for the action value function : $\forall s, a \in \mathcal{S} \times \mathcal{A}(s)$:

$$Q^\pi(s, a) = \sum_{s'} \mathcal{P}_a(s, s') [\mathcal{R}_a(s, s') + \gamma V^\pi(s')] \quad (2.7)$$

2.1.4 Optimal policies

The value functions define a partial ordering in the policy space. A policy π is therefore said to be better than π' (or $\pi \geq \pi'$) if $\forall s \in \mathcal{S}, V^\pi(s) \geq V^{\pi'}(s)$. We are looking for π^* so that :

$$\forall \pi, \quad \pi^* \geq \pi \quad (2.8)$$

TODO quote showed that for finite MDPs, there is always at least one policy that is better our equal to all others, and therefore is called the *optimal policy* π^* . As shown in TODO quote, the state-value and action-value function verify the *Bellman optimality equations*.

Bellman optimality equations

- ▷ Bellman optimality equation for the state-value function : $\forall s \in \mathcal{S}$

$$V^\pi(s) = \max_{a \in \mathcal{A}(s)} \pi(s, a) \sum_{s'} \mathcal{P}_a(s, s') [\mathcal{R}_a(s, s') + \gamma V^\pi(s')] \quad (2.9)$$

- ▷ Bellman optimality equation for the action value function : $\forall s, a \in \mathcal{S} \times \mathcal{A}(s)$:

$$Q^\pi(s, a) = \sum_{s'} \mathcal{P}_a(s, s') \left[\mathcal{R}_a(s, s') + \max_{a' \in \mathcal{A}(s')} Q(s', a') \right] \quad (2.10)$$

Those relations are essential in understanding the solving algorithms that will be presented later.

There exists several ways of solving (i.e computing the optimal policy) of a Markov Decision Process, that can generically be separated in two categories : *model-based* and *model-free* methods.

2.2 Dynamic Programing

Dynamic programing is a mathematically well-developed theory. It requires the complete and accurate model of the environment, making it a model-based method.

Dynamic programing methods aims at computing the optimal value function at every state of state space. This could, of course, be done by solving the $|\mathcal{S}|$ equations of $|\mathcal{S}|$ unknowns that are the Bellman equations for a given policy, and then evolve that policy toward a better one, based on the current value function. Of course, this approach is computationally intractable for large state space and therefore needs to be adapted, but gives a first approach of the idea behind dynamic programing.

2.2.1 Generalized policy iteration

The generalized policy iteration methods rely on alternating two processes, known as **policy evaluation** and **policy improvement**.

- ▷ Policy evaluation deals with estimating the value function of a given policy π , without directly solving the full system given by Bellman equations. The idea is actually pretty simple : use Bellman's equation as an update rule, using that the value function is a fixed point for this update rule. The algorithm, after setting the tabled value function to an initial value, iterate by performing what is called *full Bellman backups* :

$$\forall s \in \mathcal{S}, \quad V_{k+1}^\pi(s) = \sum_{a \in \mathcal{A}} \pi(s, a) \sum_{s'} \mathcal{P}_a(s, s') [\mathcal{R}_a(s, s') + V_k^\pi(s')] \quad (2.11)$$

This algorithm converges under the same assumptions that guarantee the existence of the value function, and has the generic name of *iterative policy evaluation*. They are many refining for speeding up that process (reduced backups, prioritized sweeping) that we won't address here.

- ▷ Policy improvement is a process that from a given policy value function, returns a better or equal policy compared to the latter. The simplest way to do that is for every state $s \in \mathcal{S}$, consider every action-value functions :

$$Q(s, a) = \sum_{s'} \mathcal{P}_a(s, s') [\mathcal{R}_a(s, s') + \gamma V^\pi(s')], \quad a \in \mathcal{A}(s) \quad (2.12)$$

and to build π' to be *greedy* with respect to those actions-values :

$$\pi'(s) = \operatorname{argmax}_{a \in \mathcal{A}(s)} \{Q(s, a)\} \quad (2.13)$$

The policy improvement theorems then ensures that $\pi' \geq \pi$.

Hence, generalized policy improvement are a set of methods that iteratively combine those two sub-methods to compute the optimal policy for a given MDP. Of course, one does not have to perform all sweeps of value evaluation before improving the policy to converge toward an optima (indeed, many times our sweeps won't have any affect on the greedy policy). They are many ways to combine the two (prioritized sweeping, asynchronous dynamic programing), but the most used and one of the most quickest way to converge is to use the value iteration algorithm.

2.2.2 The value iteration algorithm

The value iteration algorithm takes the limit of the behavior we just described, and stops the value evaluation procedure after only *one state space sweep*. It therefore performs a simple backup procedure :

$$\forall s \in \mathcal{S}, \quad V_{k+1} = \max_{a \in \mathcal{A}} \sum_{s'} \mathcal{P}_a(s, s') [\mathcal{R}_a(s, s') + \gamma V_k^\pi(s')] \quad (2.14)$$

For any arbitrary V_0 , it is shown that $V_k \rightarrow V^*$ as $k \rightarrow \infty$, under the same hypothesis that ensure the existence of the optimal value function V^* . As one can notice, it actually implements the *Bellman optimality conditions* as an update rule !

2.3 Temporal differences methods

Temporal difference methods can be seen as a combination of dynamic programming and another kind of learning called Monte Carlo methods, where the expected return are approximated via sampling. Like dynamic programming, TD methods are said to bootstrap (meaning that they build their estimators through already estimated values), but are *model-free* methods and learn from experience.

The justification, proof of convergences and literature and those models is pretty wide, hence we will not cover them in this report. However, full description of those methods can be found in TODO quote.

2.3.1 On-policy method : SARSA

The SARSA algorithm is an *on-policy* control method, meaning that the algorithm updates the value function and improves the current policy it is following. At state s_t , it chooses an action a_t from its policy and follows it. After observing the reward r_{t+1} and the next state s_{t+1} , it again chooses an action a_{t+1} using a soft policy and performs a one-step backup :

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \quad (2.15)$$

It therefore relies on a 5-tuple $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$ to perform the update, giving it the State Action Reward State Action (SARSA) name.

The General Sarsa Algorithm

1. *Initialize* $Q(s, a)$ arbitrarily $\forall (s, a) \in \mathcal{S} \times \mathcal{A}(s)$
 2. *Repeat* for each episode :
 - Initialize s
 - Choose $a \in \mathcal{A}(s)$ using a soft policy derived from Q (typically ε -greedy)
 - Repeat for each step of the current episode :
 - Take a , observe r, s'
 - Choose a' from s' using policy derived from Q
 - $Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma Q(s', a') - Q(s, a)]$
 - $a \leftarrow a', s \leftarrow s'$
- until $s \in \mathcal{S}^+$.

The convergence properties of SARSA depend on the nature of the policy's dependency on Q . Indeed, SARSA converges with probability 1 to the optimal policy as long as all the state and actions pairs are visited an infinite number of time, and the policy converges in the limit to the greedy policy. This is done, for instance, by turning the temperate of a softmax based policy to 0, or by having $\varepsilon \rightarrow 0$ for a ε -greedy policy. For SARSA to converge, we also as the learning rate to comply with the stochastic approximation conditions :

$$\sum_k \alpha_k(a) = +\infty \quad \text{and} \quad \sum_k \alpha_k(a)^2 < +\infty \quad (2.16)$$

where $\alpha_k(a)$ is the learning rate for the k^{th} visit of the pair (s, a) .

2.3.2 Off-policy method : Q-learning

The Q-learning algorithm is an off-policy method who learns to directly approximate Q^* , independently of the policy being followed. Its update rule is given by :

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_{a \in \mathcal{A}(s_{t+1})} Q(s_{t+1}, a) - Q(s_t, a_t) \right] \quad (2.17)$$

The actual policy being followed still has an effect in that it determines which state-actions pairs are visited and updated. However all that is required for convergence it that all pairs continue to be updated.

Q-Learning Algorithm

1. Initialize $Q(s, a)$ arbitrarily $\forall (s, a) \in \mathcal{S} \times \mathcal{A}(s)$
 2. Repeat for each episode :
 - Initialize s
 - Repeat for each step of the current episode :
 - Choose $a \in \mathcal{A}(s)$ using arbitrary policy
 - Take a , observe r, s'
 - $Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a' \in \mathcal{A}(s')} Q(s', a') - Q(s, a)]$
 - $s \leftarrow s'$
- until $s \in \mathcal{S}^+$.

Along with this hypothesis and a slight variation in the usual stochastic approximation conditions, the learned action value function by Q-learning has been shown to converge to Q^* with probability 1.

2.4 Grid-world examples

We hereinafter describe two grid-world space, on which we will apply the learning rule derived in the latest section. Such example are trivial and are displayed here just to show convergence and behavior of the different algorithms.

We'll consider the two following state space :

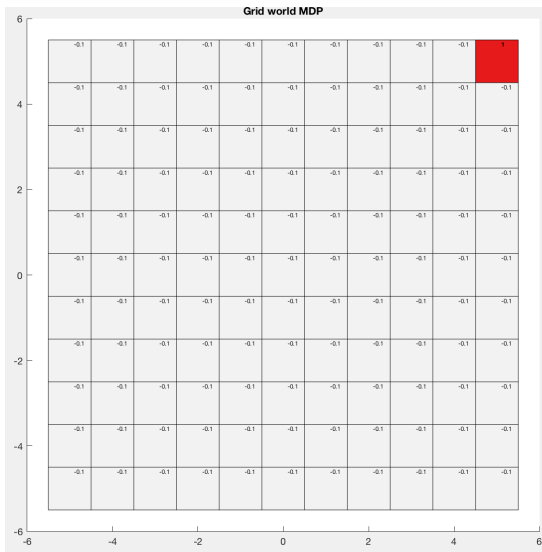


Figure 2.1: The *free_grid* state space

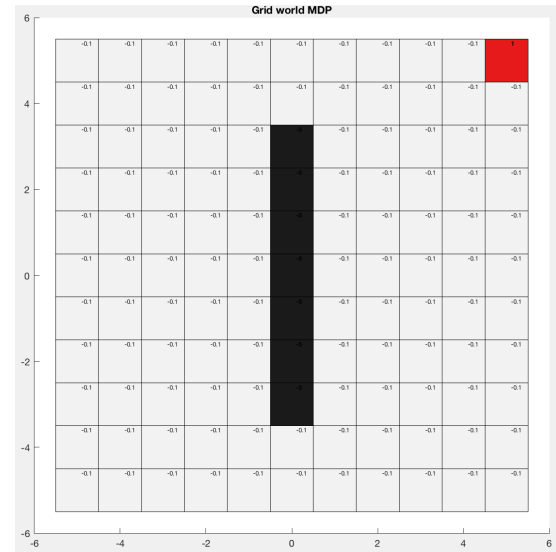


Figure 2.2: The *bar_grid* state space

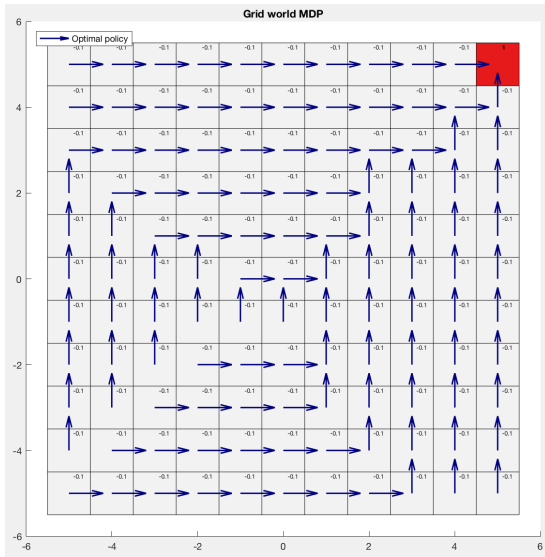
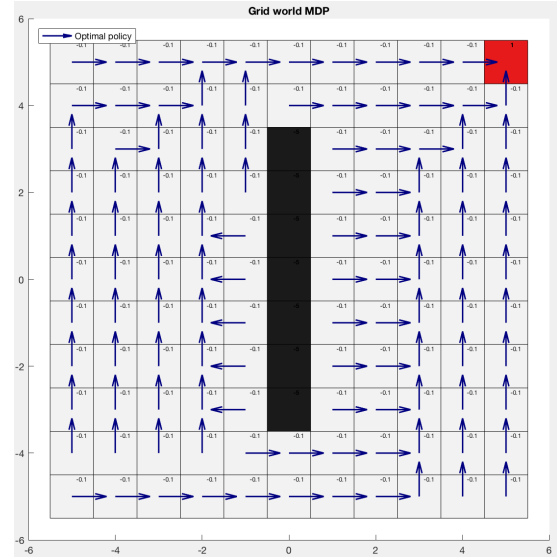
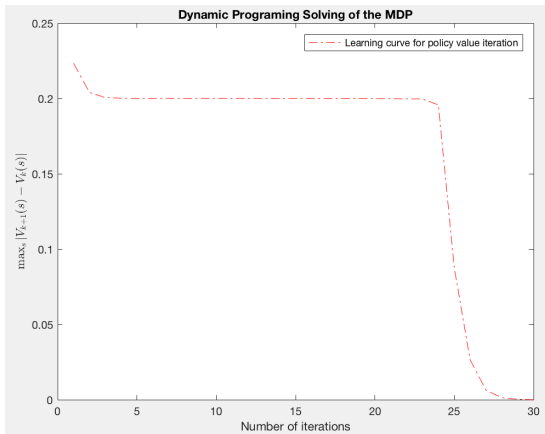
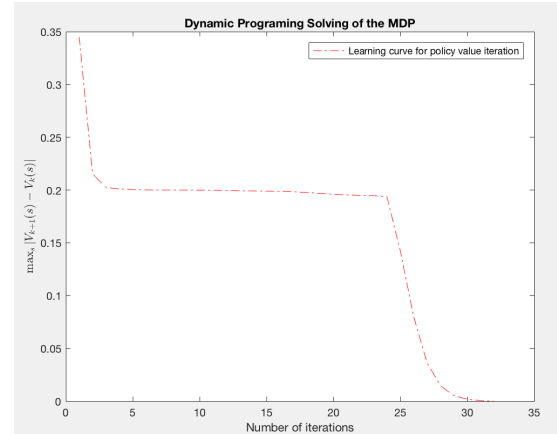
2.4.1 Dynamic Programming solving

Let us run the DP algorithm on such grid worlds. We'll consider a stochastic environment, with the transition probability :

$$\mathcal{P}_{s,s'}^a = \begin{cases} 0.9 & \text{if } s' = s(a) \\ 0.1 & \text{otherwise} \end{cases} \quad (2.18)$$

Running the value iteration algorithm (assuming we now the environment model), we obtain the following policies and learning curves. The stopping criterion addresses the maximum absolute change brought to the value function as the sweeping goes through the state space :

$$\text{If } \max_{s \in \mathcal{S}} |V_{k+1}(s) - V_k| < \delta \text{ then stop} \quad (2.19)$$

Figure 2.3: The *free_grid* learned optimal policyFigure 2.4: The *bar_grid* learned optimal policyFigure 2.5: The *free_grid* value iteration learning curveFigure 2.6: The *bar_grid* value iteration learning curve

One can notice that for the *bar_grid* environment, the agent undergoes a longer trajectory than necessary, at the left of the obstacle. This is because of the stochastic nature of the environment, causing the agent to learn to take its distance with the obstacle in order not to accidentally hit it (and then receive a very negative reward).

The learned policy are indeed optimal. The next algorithms (SARSA and Q-learning) will try to reproduce them without a model for the environment.

2.4.2 SARSA solving

We display here the learning curves for the *free_grid* state space using SARSA. The algorithm manages to learn the optimal policy and the right action-value functions. We use *optimistic initialization* to encourage exploration, and Gibbs sampling for following a soft-policy : $\forall(s, a) \in \mathcal{S} \times \mathcal{A}(s)$

$$\pi(s, a) = \frac{e^{Q(s, a)/\tau}}{\sum_{a' \in \mathcal{A}(s)} e^{Q(s, a')/\tau}} \quad (2.20)$$

We'll tune the distribution's temperature τ to zero, in order to converge toward the greedy policy w.r.t the learned action-value function.

Following this strategy and tuning our learning rate to comply with the stochastic approximation conditions, we obtain the following learning curves. Again, our stopping criterion addresses the *maximum change in the action-value function over all the trajectories of a mini-batch*.

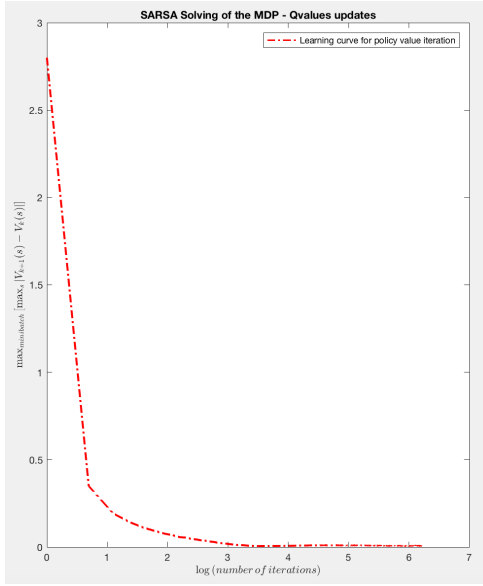


Figure 2.7: Learning curve for SARSA on *free_grid*

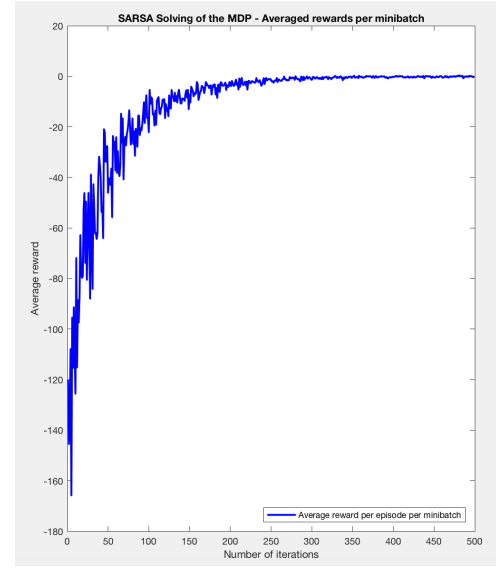


Figure 2.8: Averaged rewards over mini-batch for SARSA on *free_grid*

2.4.3 Q-learning solving

We display here the learning curves for the *bar_grid* state space using Q-learning. Again, the algorithm manages to learn the optimal policy and the right action-value functions. We use Gibbs sampling for the behavior policy, without any tuning for the temperature (the behavior policy doesn't need to be greedy in limit). The learning curves obtained are given hereinafter.

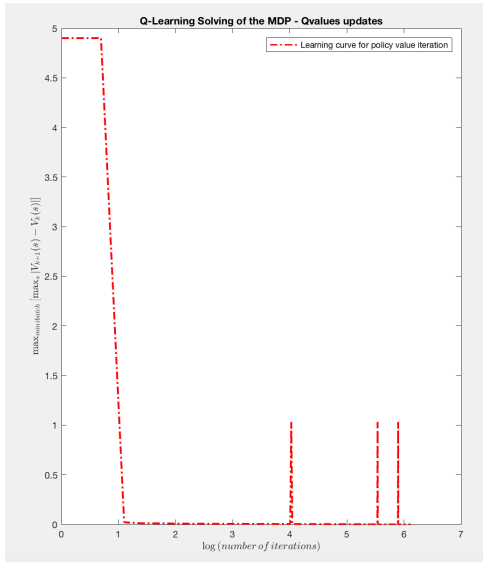


Figure 2.9: Learning curve for Q-learning on *bar_grid*

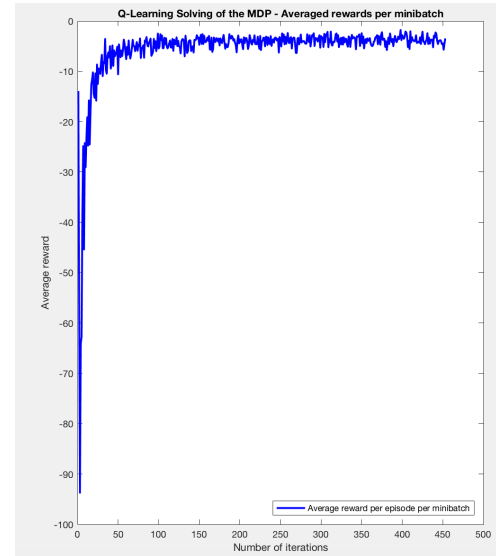


Figure 2.10: Averaged rewards over mini-batch for Q-learning on *bar_grid*

Chapter 3

Compliant-based reinforcement learning

3.1 Problem selection

Chapter 4

Results

Conclusion

Bibliography

- [SB98] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning : An Introduction*. MIT Press, 1998.