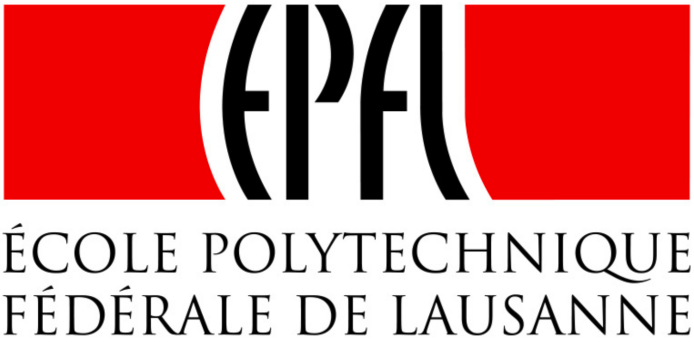
ÉCOLE POLYTECHNIQUE FEDERALE DE LAUSANNE
MT SECTION
FAURY Louis

SEMESTER PROJECT REPORT

LASA



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

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

Contents

Chapter 1

Introduction

1.1 Motivations

For many long and complex tasks, many existing machine learning algorithms; when initialized without any prior knowledge of their environment or of the task that they are suppose to complete are unable to lead (at least in acceptable computational times) to feasible solutions. Hence, much like we would do with a child in the process of learning a skill (dancing, kicking a ball,..), we provide the learning algorithm with demonstration, most often coming from a human teacher. However, because the teacher is older, or not an expert in the skill he is trying to teach, he might give the child a suboptimal way of performing the task. The child would use the prior information given by the teacher, until it becomes pretty confident about performing the given skill. It will also learn by itself, and maybe find that indeed, by slightly changing how he performs the tasks, he is able to perform better than the teacher. Therefore, the learning child will first be *compliant* with the teacher, before trying things out by itself once it has become pretty good at performing the learned skill.

This semester project aims at introducing a theoretical formulation of such a compliance-based behavior, in the Learning from demonstration (LfD) framework.

Learning from demonstration is indeed a useful tool when it comes to robots learning complex tasks. A good way of helping them in their learning process is to provide them with demonstrations of the task. Such a demonstration coming from a human teacher (and therefore without the same abilities of the robot), some adaptation has to be made, as well as exploration by the robot to make sure the solution given by the teacher is not a suboptimal one. Another possibility would be the case of receiving a demonstration from a trained robot, or even a set of different agent with inhomogeneous level of training. Such variants will also be studied in this project.

After a short reminder on Learning from demonstration (LfD) problematics and Reinforcement Learning (RL) techniques, we will introduce a compliant based exploration/exploitation tradeoff for policy search techniques, and assess its abilities before trying to generalize our approach for several mentors.

1.2 Background

1.2.1 Learning from demonstration

As discussed before, Learning from demonstration (LfD) is a framework where a robot can learns from interacting with a human. It particularly focus on the case where a mentor (human) provides demonstration on how to perform a task. Such an approach is also known as Programming from demonstration (PbD) or imitation learning.

LfD mostly lies on the principle that the learning robot can be taught new tasks by end-users, without having to be programmed again. We therefore are dealing with robot that are able to generalize from demonstration, be it only to understand what is the task that they must learn. They are many underlying problematics and opened questions on the subject, detailed in [BCD16].

On approach of LfD combines with reinforcement learning techniques. A could argument for this is that LfD could easily speed-up the learning process by providing a finite or infinite set of good solutions. However, this assume some kind of reproducibility by the robot, as well as a similar enough context between demonstration and reproduction.
Under such assumptions, RL techniques can use LfD to speed up their learning process and discover new control policies through a directed exploration of its action-state space. Indeed, demonstrations are used to guide the RL algorithm exploration, reducing the time and iterations needed for an agent to discover a new control policy, and eventually overcoming the teacher's apparent policy sub-optimalities.

They are many ways of performing such a demonstration-based learning. We remind here a few of them, taken from [ACVB09], all related to Imitation Reinforcement Learning (IRL) :

- policy derivations techniques : directly approximate the robot's mapping from state to action by reproducing and generalizing the teacher's one.
- use demonstration data to learn a model from the world's dynamic (or transition probability model) to compute the optimal policy (see [PB03]).
- use demonstration data and additional user intention to learn the rule that associate a set of pre and post-condtions which each action as well as a sparse transition model to plan for actions.

The framework we described hereinbefore can easily be extended to more than one teacher. Moreover, such teachers can actually be other learning algorithms with different levels of training. Quickly, the problem drifts toward the underlying issue of performing transfer skill between agents. As this will also be tackled during this project, the next section reminds a few useful facts about the theory of transfer learning.

1.2.2 Transfer learning

Transfer learning deals with speeding a learning process thanks to another learning experience - for instance training an agent based on the previous training on another agent, performing a similar task. The literature of transfer learning in reinforcement learning is pretty large, and for a full description of the different problematics (homogeneous / inhomogeneous sets of actions, multi task learning, ..) we'll refer to [TS09].

As we said before, one of our goal is to assess our method as a transfer learning one, and up to a point, consider our teacher to be a trained (fully or not) agent, before considering many different agents as potential mentors. We would consider a really simple case of transfer learning, where all agents display the same state-action state, with the same abilities and constraints.

Hence, we remind here some metrics to evaluate transfer learning methods that we would have to use to compare our different approaches :

- *Jump start* : the initial performance of an agent can be improved by transfer.
- *Asymptotic performance* : how close is the asymptotic learned policy from the optimal policy ?
- *Total or averaged reward* : averaged reward may be improved by transfer
- *Transfer ratio* : Ratio of total reward accumulated by the transfer learner and the total of reward accumulated by a non-transfer learner.
- *Time to threshold* : the time required by a transfer learning to reach a pre-specified level of skills in achieving its task.

All those metrics actually compared different aspects of how well a RL algorithm learns. Comparing two RL methods is actually still an open question, and hence we'll need to address several of those metrics to decide on the wellness of our derived TL methods.

1.3 Proposed approach

As we started to explained before, we are going to start with a fairly simple model in order to address the problem of compliance in learning from a teacher. We hope that a simple enough state space and framework will allow us to better understand how compliance in learning by demonstration should be tackled, in a reinforcement learning framework. We also hope we would be able to generalize to more complex situations once the understanding on a simple but generic model is mastered.

After defining a simple two dimensional grid-world state space with a simple action set, we will quickly study how well the different classical RL policy search algorithms performs on such a space.

Once this done, we will introduce a new exploration policy, based on the learned confidence the learner has in its teacher. The teacher does not provide the learner with trajectories, but with recommandation of actions point-wise in the state-action space they share (*homogeneous settings*). We want the learner, based on a prior knowledge of the confidence it should give to its teacher (the confidence is understood locally : we can be very confident that the teacher is right for one point of the state space and really doubting the teacher's recommandation elsewhere) to gradually update (based on a *compliance term*) this confidence to be able to decide to explore new trajectories once it has quickly learned a possibly suboptimal one from the teacher.

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 :

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

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 :

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

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, $\forall s' \in \mathcal{S}$:

$$\mathbb{P}\left(s_{t+1} = s' \,|\, a_t, s_t, \ldots, a_1, s_1\right) = \mathbb{P}\left(s_{t+1} = s' \,|\, s_t, a_t\right) \tag{2.3}$$

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

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

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

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

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 : $\forall s \in \mathcal{S}$

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

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

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

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 :

$$\forall \pi, \quad \pi^* \geq \pi \tag{2.8}$$

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 : $\forall s \in \mathcal{S}$

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

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

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

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* :

$$\forall s \in \mathcal{S}, \quad V^{\pi}_{k+1}(s) = \sum_{a \in \mathcal{S}} \pi(s,a) \sum_{s'} \mathcal{P}_a(s,s') \left[\mathcal{R}_a(s,s') + V^{\pi}_k(s')\right] \tag{2.11}$$

  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 evaluation 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 :

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

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

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

  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 :

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

For any arbitrary V_0, it is shown that $V_k \to V^*$ as $k \to \infty$, 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 programing 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 :

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

It therefore relies on a 5-tuple $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$ to perform the udpate, 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
   $Q(s, a) \longleftarrow Q(s, a) + \alpha\big[r + \gamma Q(s', a') - Q(s, a)\big]$
   $a \leftarrow a'$, $s \leftarrow s'$

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 sate 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 $\varepsilon \to 0$ for a ε-greedy policy. For SARSA to converge, we also as the learning rate to comply with the stochastic approximation conditions :

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

where $\alpha_k(a)$ is the learning rate for the k$^{\text{th}}$ 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 :

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

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'
   $Q(s,a) \longleftarrow Q(s,a) + \alpha\big[r + \gamma \max_{a' \in \mathcal{A}(s')} Q(s',a') - Q(s,a)\big]$
   $s \leftarrow s'$
   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.

2.3.3 Eligibility traces

In TD(0) approach (described in the latest section), we update the value function in the direction of the *one-step return* :

$$\Delta V_t(s_t)^{(1)} = \alpha\left[r_t + \gamma V_t(s_{t+1}) - V_(s-t)\right] \tag{2.18}$$

The idea behind eligibility traces is to expand that update rule in order to steer the value fonction towards the *n-step return* (or at least until a terminal state is reached) :

$$\begin{aligned} \Delta V_t(s_t)^{(n)} &= \alpha\left[r_t + \gamma r_{t+1} + \ldots + \gamma^n V_t(s_{t+n}) - V_t(s_t)\right] \\ &= \alpha\left[R_t^{(n)} - V_t(s_t)\right] \end{aligned} \tag{2.19}$$

The backups can not only be done toward any n-step return, but toward any average of such returns, as long as the corresponding weights sum-up to one. In this way, the TD(λ) algorithm can be understood as a particular way of averaging n-steps returns. With $\lambda < 1$, the resulting backup is known as the *λ-return* :

$$R_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(n)} \tag{2.20}$$

where the weights are fading with n. When the runs are episodic, we can write this return as :

$$R_t^\lambda = (1-\lambda)\sum_{n=1}^{T-t-1} \lambda^{n-} R_t(n) + \lambda_{T-t-1} R_t^{(T)} \tag{2.21}$$

Sarsa(λ) algorithm

1. *Initialize* $Q(s,a)$ arbitrarily $\forall(s,a) \in \mathcal{S} \times \mathcal{A}(s)$

2. *Repeat* (for each episode) :
   Initialize s, a
   Repeat for each step of the current episode :
   Take action a, observe r, s'.
   Choose $a' \in \mathcal{A}(s')$ using soft policy derived from Q
   $\delta \longleftarrow r + \gamma Q(s',a') - Q(s,a)$
   $e(s,a) \longleftarrow e(s,a) + 1$
   For all s, a :
   $Q(s,a) \longleftarrow Q(s,a) + \alpha\delta e(s,a)$
   $e(s,a) \longleftarrow \gamma\lambda e(s,a)$
   $s \leftarrow s'$, $a \leftarrow a'$
   until $s \in \mathcal{S}^+$.

Such a formulation of eligibility traces is known as the *forward view* of TD(λ), and shows how eligibility traces build the bridge between TD(0) methods and Monte-Carlo one. It is not implementable as is since it is non-causal. There exist a more mechanistic view, equivalent to the forward view (see [SB98]), known as the *backward-view*. It gives birth to causal version of the TD(λ) method. We give as an example the pseudo-code for the SARSA(λ) algorithm above. Its Q-learning equivalent can be found in [SB98].

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 :

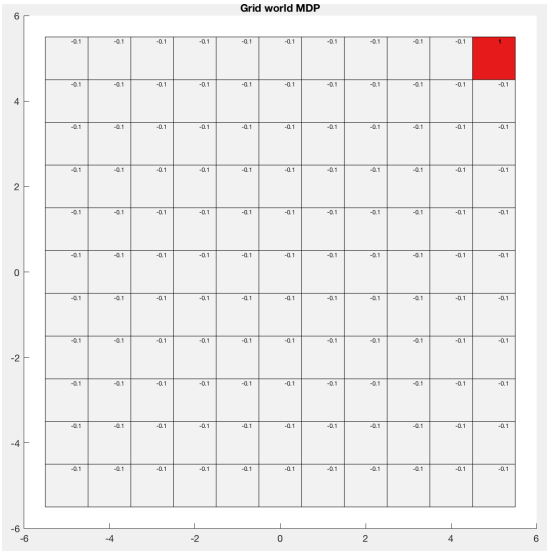


Figure 2.1: The *free_grid* state space

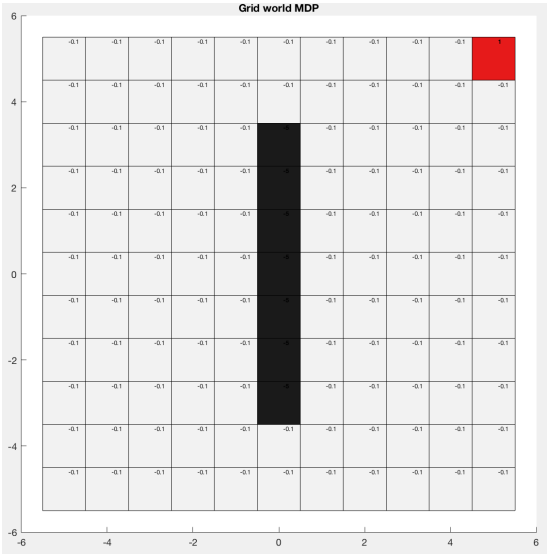


Figure 2.2: The *bar_grid* state space

2.4.1 Dynamic Programing solving

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

$$\mathcal{P}^a_{s,s'} = \begin{cases} 0.9 & \text{if } s' = s(a) \\ 0.1 & \text{otherwise} \end{cases} \tag{2.22}$$

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

$$\text{If } \max_{s\in\mathcal{S}} |V_{k+1}(s) - V_k| < \delta \text{ then stop} \tag{2.23}$$

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 takes 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.

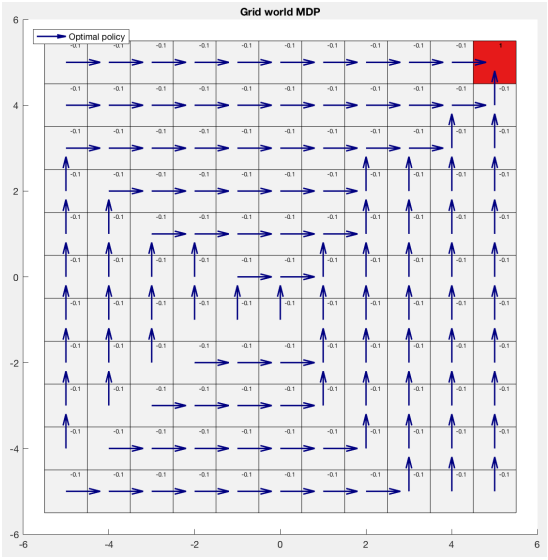


Figure 2.3: The *free_grid* learned optimal policy

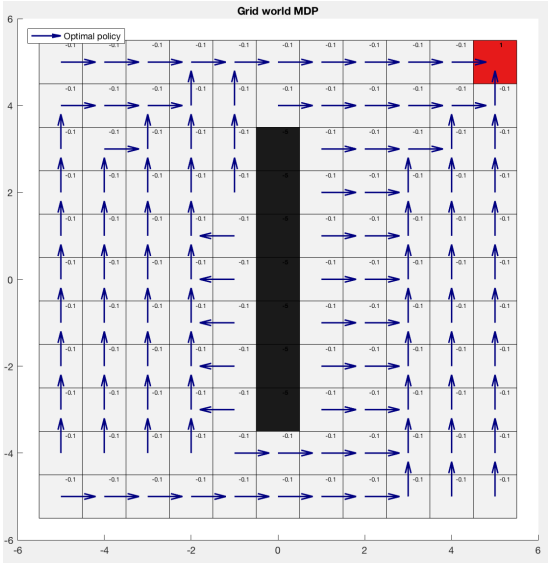


Figure 2.4: The *bar_grid* learned optimal policy

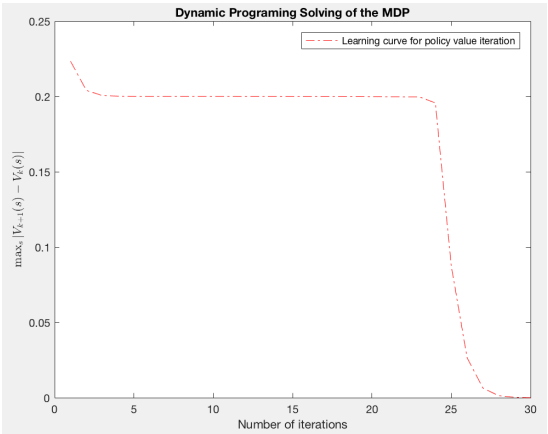


Figure 2.5: The *free_grid* value iteration learning curve

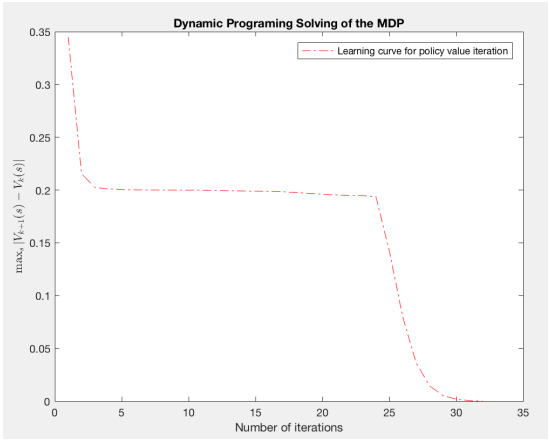


Figure 2.6: The *bar_grid* value iteration learning curve

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 : $\forall (s,a) \in \mathcal{S} \times \mathcal{A}(s)$

$$\pi(s,a) = \frac{e^{Q(s,a)/\tau}}{\sum_{a' \in \mathcal{A}(s)} e^{Q(s,a')/\tau}} \tag{2.24}$$

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 acton-value function over all the trajectories of a mini-batch.*

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

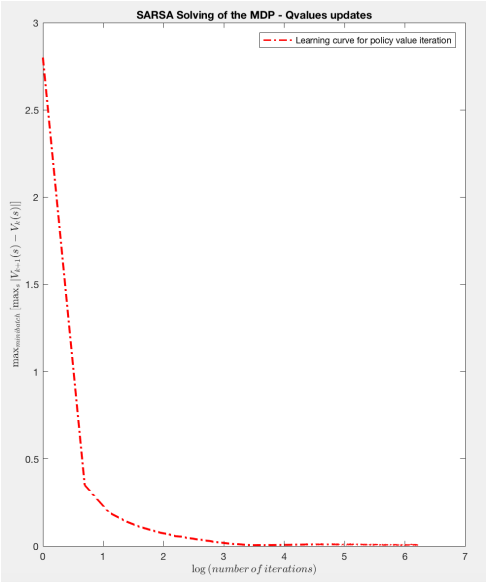


Figure 2.7: Learning curve for SARSA on *free_grid*

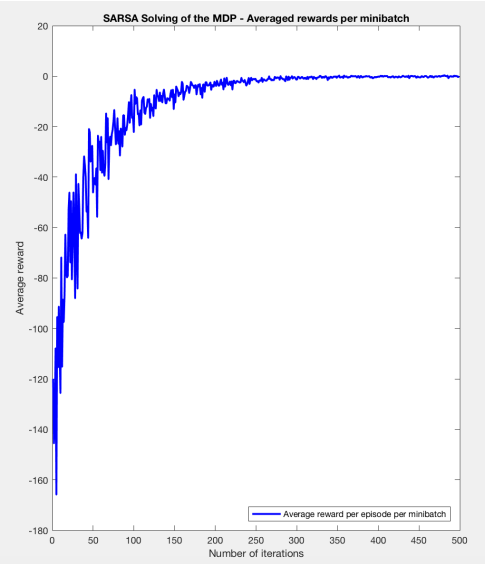


Figure 2.8: Averaged rewards over minibatch for SARSA on *free_grid*

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.

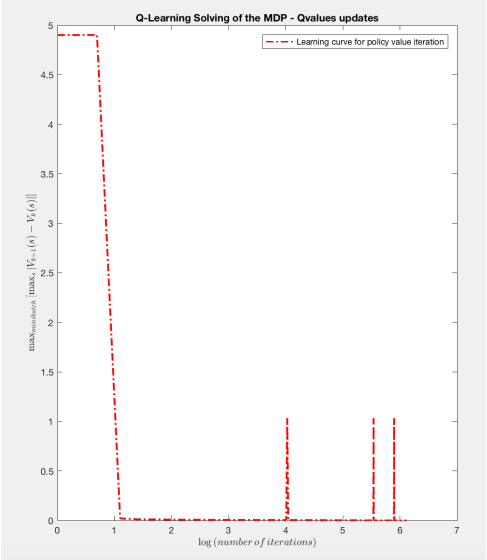


Figure 2.9: Learning curve for Q-learning on *bar_grid*

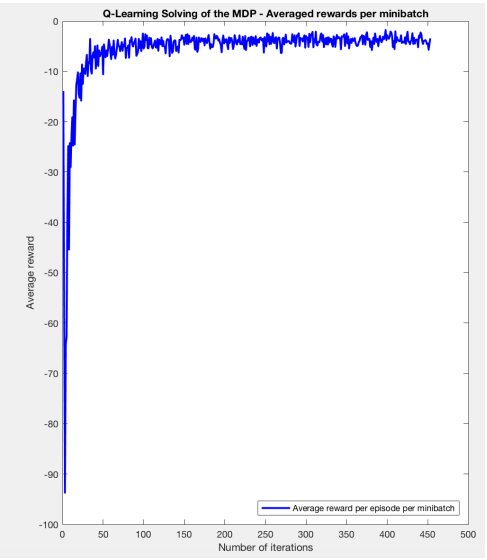


Figure 2.10: Averaged rewards over minibatch for Q-learning on *bar_grid*

Chapter 3

Compliant-based reinforcement learning

3.1 Problem selection

With our approach, we wish to tackle two topics : on one hand, we wish to display a imitation based learning framework that *accelerate* the learning process. Also, we wish to have a shifting compliance based behavior so that an agent can *overtake a suboptimal teacher*. To test the effectiveness of the solutions we would propose, we decided to provide ourself with a model that would stay fixed all along the experiments, in order to compare the different algorithms we could come up with.

We therefore needed to come up with an environment that, with the algorithms described before, cannot solve its MDP quickly, and can sometimes get stuck in local minima. We chose the state space displayed in figure (3.1). In this environment, all black cells are obstacles. They give out highly negative rewards ($r = -10$). Whenever an agent take the action to enter such a cell, it immediately perceives the negative reward but stays in its current cell. The only positive reward in this environment is at the middle of the grid ($r = 10$), also a terminal state. Any trajectory of the learner starts at one of the corner of the grid (green cells), and every step spent on a non-terminal cell gives out a small negative reward ($r = -0.1$). The environment is stochastic, with the transition probability model :

$$s' = \begin{cases} a(s) \text{ with proba } 0.95 \\ s'' \neq a(s) \text{ otherwise, uniformly sampled} \end{cases} \tag{3.1}$$

It is big enough for the usual algorithms to learn rather slowly, even if they are greatly enhanced by the use of eligibility traces. Also, all tested algorithms (SARSA, Q-learning, SARSA(λ) and Watkins Q(λ)), because they do not perform infinite exploitation / exploration moves, renders slightly suboptimal policies.

Figure (3.3) displays the convergence (expressed as average reward on minibatch) for Qlearning, Sarsa(λ), Sarsa(0) and Watkins Q(λ). By average reward on minibatch, we mean that at every iteration, the reward is average around a given number of trajectories, following the same *exploratory* policy. This allows to reduce the stochasticity of trajectories while learning and give a smoother estimation of how well the algorithm is learning. As a way of comparing them to the optimal and random policies, we also plot the average rewards perceived by the latest along many trajectories.
For all the learning algorithms tested, we used optimistic action-value initialisation to promote exploration. This explains why many negative rewards are perceived in the beginning.

As explained before, this environment is considered as a base line in all that follows, and will serve as a sandbox model where we will test our different methods for an imitation learning based approach.

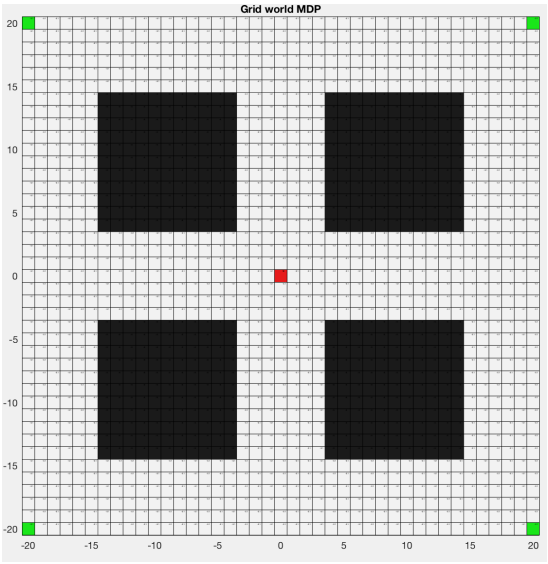


Figure 3.1: The *maze_grid* state space

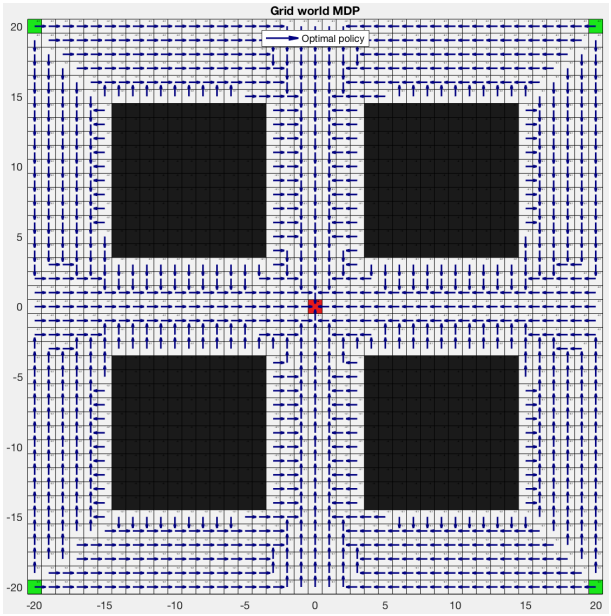


Figure 3.2: Optimal policy, computed with DP

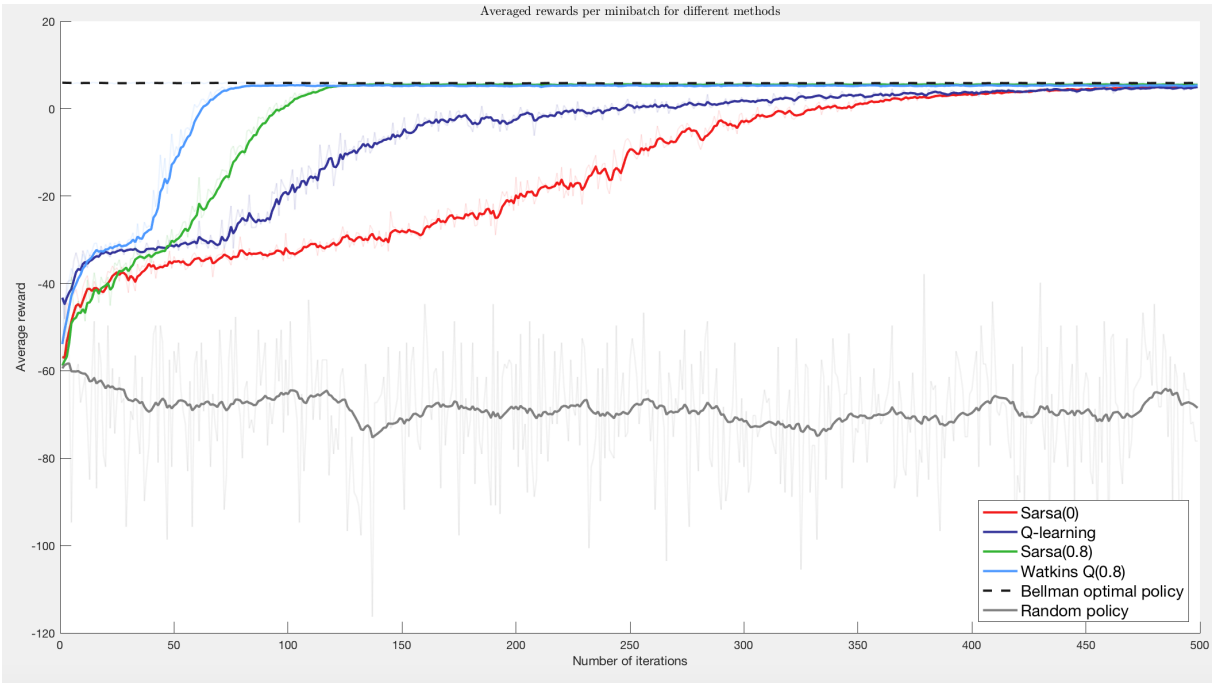


Figure 3.3: Average rewards on minibatch for learned policies, optimal policy and random policy.

Chapter 4

Results

Conclusion

Bibliography

[ACVB09] Brenna D Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robotics and autonomous systems*, 57(5):469–483, 2009.

[BCD16] Aude G. Billard, Sylvain Calinon, and Rüdiger Dillmann. *Learning from Humans*, pages 1995–2014. Springer International Publishing, Cham, 2016.

[PB03] Bob Price and Craig Boutilier. Accelerating reinforcement learning through implicit imitation. *Journal of Artificial Intelligence Research*, 19:569–629, 2003.

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

[TS09] Matthew E Taylor and Peter Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(Jul):1633–1685, 2009.