Introducing interactive help for reinforcement learners

Fabien Montagne[1] and **Philippe Preux**[2] and **Samuel Delepoulle**[3]

Abstract. The reinforcement learning problem is a very difficult problem when considering real-size applications. To solve it, we think that many issues should be studied altogether. To achieve such an endeavor, we also think that it is quite common that human begins can provide help on-the-fly to the reinforcement learner, that is when he/she sees how the learner is (mis)behaving, or could perform better than what it is currently performing. This work precisely aims at designing a principled way to help a reinforcement learner in such a context. We put the emphasis on the fact that the help being provided cannot be expected to be perfect (whatever "perfect" might mean here...).

1 Introduction

In this paper, we deal with sequential decision problems and particularly with reinforcement learning to solve them.

We consider the problem of providing help to a reinforcement learner (RLearner in the sequel). The underlying idea is that we (as humans) can generally provide useful help/information to a RLearner. This is implicitly done during the design phase: the specification of the sequential decision problem (as far as we are given some choice in it) already helps the learning algorithm by restricting the size of state space, or of action space, having more informative states, designing the problem to be markovian, ... We will not deal with these design issues here. Once the sequential decision problem has been designed and the RLearner is actually learning, whenever we are able to "see" what the learner is doing, we generally have the feeling (and even the urge!) to help it. The question we address here is how we can actually provide on-line help to a learner. More precisely, the learner has a certain representation of what it has already learnt (e.g. its current estimate of the value function in the case of the resolution of a Markov Decision Problem – MDP); then, how can we transform this informally defined help into something that can usefully be combined with what the learner has already learnt? There are several issues here:

- the combination between the two sources of information,
- the fact that this combination should be done on-line,
- more primarily, the very representation of the help,
- the fact that the help cannot be expected to be "perfect": the human helper does its best but still, assuming perfection of the help is much too much.

We want here to stress the fact that we want to keep the RLearner "untouched", that is, un-modified to provide it with help. The RLearner should be designed so as to be able to incorporate new information that helps it solve its assigned task. In this regard, what we present here can be combined with other approaches that modify the task to solve (see shaping [9, 5], hierarchical MDPs [4], dynamic abstraction [2]...).

In the sequel of this paper, we will concentrate on Markov Decision Problems, and on reinforcement learning: thus, we consider that a reinforcement learner has to solve a given MDP. Then, how can we design such a RLearner so that it is able to take advantage of some information obtained on-the-fly, while solving a given MDP. We think that some of the presented ideas may be drawn outside of this context to more general sequential decision problems, and other learning problems (not necessarily sequential decision problems, but also supervised and unsupervised learning problems). The MDP has been formalized by Puterman [8] as a 4-uplet $M = (\mathscr{S}, \mathcal{A}, \mathcal{R}, \mathcal{T})$ where $\mathscr{S}$ is the state space, $\mathcal{A}$ is the action space, $\mathcal{R} : (\mathscr{S} \times \mathcal{A} \times \mathscr{S}) \to \mathbb{R}$ is the reward function, and $\mathcal{T} : (\mathscr{S} \times \mathcal{A} \times \mathscr{S}) \to [0, 1]$ is the transition function, to reach the state s' when performing a certain action a in a certain state s.

This paper is organized as follows: in section 2, we discuss what kind of help can be provided; in section 3, we discuss the design of a reinforcement learner that takes advantage of this sort of help; in section 4, we present some experimental results; section 5 concludes and discusses this work.

2 What is help?

Various sorts of help can be provided. This help may be provided using a certain "high-level" language [6]. Help can also be provided as actions to perform in certain states [10]. In a multi-agent context, interactions between two agents with the same goal could share partial knowledge [7]. Like in some previous works, we want to give help incrementally, our help must not modify the MDP, the trainer gives information when it thinks it's worthwhile, and the given help is not necessary optimal (this implies that the learner must be able to correct, modify the trainer advice). At this constraint we add two new ones: the learner does not need to know the dynamics of state (he does not need to be able to give explicitly a policy or partial policy, or to know the transition function), the trainer state space can be different of the learner one.

Which help is interesting to a certain RLearner also depends on how the RLearner actually solves the MDP, either by estimating its value function, or by direct policy search to name the two most used approaches. In this paper, we favor the estimation of the value function. This being said, we might consider the problem as that of a regression (supervised learning) problem: we seek at learning a real-valued function and we provide some examples of this function. However, reinforcement learning is surely not a regression problem, even though, in the considered framework here, the RLearner learns a real-valued function. So, there is a mismatch between what

[1] LIL, ULCO, Calais, France, email: montagne@lil.univ-littoral.fr
[2] LIFL, UMR CNRS, Université de Lille 3, Villeneuve d'Ascq, France, email: philippe.preux@univ-lille3.fr
[3] LIL, ULCO, Calais, France, email: delepoulle@lil.univ-littoral.fr

we may provide the RLearner – couples (transition, average reward on this transition) – and what the RLearner has to learn, that is $V(s), \forall s \in \mathscr{S}$, where $\mathscr{S}$ is the set of states of the MDP.

In this paper, we consider that the help is made of trajectories, each being a set of transitions between states. However, to make things easier for the helper, we do not assume that these trajectories go from an initial state to a final state, nor that they may be represented exactly as continuous trajectories in the RLearner states space. Indeed, we do not expect that the helper represents its help in the same state space as the state space of the RLearner. Furthermore, we do not assume that the helper knows anything about the reward function of the MDP, merely that he knows (or, that he feels) that a certain set of transitions should be quite correct for the learner to reach its goal.

To give the flavor of what we will do so that the RLearner can use this help, these trajectories will be represented in the RLearner state space and a numerical gradient is created alongside, from its first extremity towards its other extremity (the help trajectories are orientated from their beginning to their end). This gradient will be used by the RLearner as an indication stating that it is a good thing for it to go along this path in the state space. Furthermore, this information should be somehow generalized so that neighboring transitions can be considered as close to those belonging to the help trajectories.

To be more precise, let us define Σ the state space in which the help is provided to the RLearner. Typically, Σ is a subspace of the MDP state space $\mathscr{S}$. (We consider here continuous state spaces). We suppose that then exists a function Φ: $\Sigma \rightarrow \mathscr{S}$, and we note $\Phi(s_i) = \sigma_i$.

Then, a trajectory Tr is defined as a finite, ordered set of elements of Σ: $Tr = \{\sigma_1, \sigma_2, ...\sigma_m\}, \sigma_i \in \Sigma$. A transition $t \in Tr$ is then a couple of subsequent elements: (σ_i, σ_{i+1}).

We then create a potential A along a trajectory. This potential assigns a real value to each transition of the trajectory: $A(\sigma_i, \sigma_{i+1})$.

The help provided by the helper is then a set of trajectories. With the previous definitions, and in particular, the way we create a gradient on transitions (we do not assign a value to states), we do not have any problem with intersecting trajectories, or with two (or more) trajectories going into reverse directions.

The way we actually create the gradient (the very definition of A) is not settled. Since we define $A(\sigma_i, \sigma_{i+1}) > 0$, this defines a positive gradient, and indicate that from the state σ_i, it seems to be a "good idea" to try to reach the state σ_{i+1}. The main benefit to define a gradient for each transition beside define a gradient for each states, is that our gradient is independent of the quality of our trajectory and independent of other examples. To define a gradient we have many possibilities, such as assigning random values to each transition of a trajectory, or assigning a constant gradient to each transition,...

As long as the RLearner is not working in the same state space as the one in which the help is represented, we must solve this issue, as well as provide some generalization abilities to avoid performing mere rote-learning: neighboring transitions from those present on the help trajectories should be rated, according to their closeness. Regarding generalization, we are left here with all the methods of supervised, regression problems: to each transition, according to the rated transitions that appear in the help trajectories, we have to induce a rate for other transitions, which do not appear in the help. We should favor a regression algorithm that may be used incrementally. This led us to locally weighted regression (LWR) algorithms, which have proven their efficiency and good performance in regression and in reinforcement learning [1].

To use LWR techniques, we basically have to choose a kernel function $k(t_1, t_2)$ which measures the similarity between two transitions t_1 and t_2. Transitions may be seen as segments in some space, that we suppose here euclidian (if it is not euclidian, we need that it is a metric space at least, which is a rather mild requirement). So, the kernel function has to measure the similarity between two segments. In the present work, we have used a measure that combines two things: the angle between the segments, and their relative distance. So, k is decomposed into $k_1 k_2$. k_1 is defined by:

$$k_1(t_1, t_2) = \begin{cases} \exp^{-c \sin^2(t_1, t_2)} & \text{when the angle between } t_1 \text{ and } t_2 \\ & \in [-\frac{\pi}{2}, \frac{\pi}{2}] \\ 0 & \text{otherwise} \end{cases}$$

where t_1 and t_2 are two transitions. k_2 is defined by:

$$k_2(t_1, t_2) = \exp^{-d \times \text{max\_dist}}$$

where max_dist is the maximal distance among (s_1, s_3), (s_1, s_4), (s_2, s_3), (s_2, s_4), where (s_1, s_2) are the extremities of t_1, and (s_3, s_4) are the extremities of t_2. c and d are constants.

Then, to estimate the potential $A(t)$ of a certain transition t, we use the following equation:

$$A(t) = \sum_{t' \in T} A(t') k(t, t') \quad (1)$$

where T is the set of all the transitions in all the trajectories that have been provided to the RLearner up to now.

3 A RLearner that takes advantage of this help

At this point, we have described how we represent the help provided by a helper. Now, we describe how an RLearner can make use of it. As above, we consider here that the learner learns the value function of the MDP under consideration.

So, as seen in the previous section, the help is somehow transformed into some real-valued function denoted A. It should be very clear that A is not meant to be an approximation of the MDP value function, that we will denote with the usual V. The thing is that the RLearner, while interacting with its environment, has to learn V. V will let it select its action at each iteration. We will note C the value function which is estimated in this manner; we consider here that C is learnt by a usual Temporal-Difference method, say the TD(λ) algorithm. The problem is to combine C and A. Both functions are learnt incrementally: C is learnt by interaction with the environment, while A is learnt from the help trajectories, which may be provided at any moment to the RLearner. According to the general idea of providing help, the helper provides help whenever he/she sees that the RLearner is in trouble; so, the help trajectories should be taken into account immediately; for example, they should take the RLearner out of a bad position, or help it find a way out of a place where it is stuck. Instead of trying to combine A and C in a single representation of the estimate of V, we use two function approximators, one for A and an other for C to represent each function. And it is only when the estimate of the value of a state s is required that both are combined to provide $\hat{V}(s)$. This has several advantages: it makes the on-line incorporation of new help trajectories simpler since A and C are not defined on the same space; the learner can always access to former helps; help does not fade away as learning by interaction goes on.

So, to sum-up, the estimate $\hat{V}(s), s \in \mathscr{S}$ is obtained according to the following equation:

$$\forall s \in \mathscr{S},\ \hat{V}(s) = \quad (1-\zeta(s)) \times \hat{C}(s) + \\ \zeta(s) \sum_a \sum_{s'} A(t(\sigma_i, \sigma)) \times \mathcal{T}^a_{s,s'} \tag{2}$$

where:

- $s \in \mathscr{S}$ is the state for which the value is required,
- $\hat{C}(s)$ is the current estimate of the value of s,
- $A(t(\sigma_i, \sigma))$ is the information obtained from the help (as explained in sec. 2, and according to eq. 1)
- $\sigma \in \Sigma,\ \sigma = \Phi(s)$ and $\sigma_i \in \Sigma,\ sigma_i = \Phi(s')$
- $\mathcal{T}^a_{s,s'}$ is the probability (in the MDP) to reach s' when action a is emitted in state s
- $\zeta(s) : \mathscr{S} \rightarrow [0, 1]$ is a parameter which balances the information available in the help, and what has already been learnt. Typically, it decreases along time; the estimation of the value of a state initially takes into account the help and, progressively, as learning is being performed by the trainee, the information learnt by the trainee gets more and more accurate, so that the help gets less and less importance. ζ depends on the state to keep on with the incrementally of help: newly obtained help is assigned a value of ζ close to 1.

The previous equation may be easily stated for the estimate of the quality function as follows:

$$\forall s \in \mathscr{S}, \forall a \in \mathcal{A},\ \hat{Q}(s,a) = \quad (1-\zeta) \times \hat{C}(s,a) + \\ \zeta \sum_{s'} A(Tr(\sigma_i, \sigma)) \times \mathcal{T}^a_{s,s'}. \tag{3}$$

So, the RLearner simply:

- learns C as any TD(λ) algorithm does, by interacting with its environment
- updates A whenever new help trajectories are provided, according to eq. (1)
- selects its action according to state value estimates obtained by way of eq. (2)

3.1 The Actor-Critic-Critic algorithm

The actor-critic-critic architecture has been designed to integrate the information available from the usual interaction loop in reinforcement learning TD methods, and from the help receives incrementally from the trainer. The actor-critic-critic has been inspired by the Actor-Critic architecture [11]. The actor uses the value estimation $\hat{V}$ to determine its policy. There are two critics: one critic represents the current estimate of the value function in a usual way in reinforcement learning (hence, provides the $C(S)$ estimate of eq. 2), while the second critic contains an internal representation of the help by way of a gradient and provides the term $A(Tr(\sigma_i, \sigma))$ in eq. 2.

The actor-critic-critic algorithm is sketched in fig. 1.

In the actor-critic-critic, the help does not modify the relation between the trainee and its environment. Indeed, the trainee still uses usual reinforcement learning algorithms and update equations of state value estimations. The help interacts only during the action selection. This allows us to suppose that our agent is in the usual convergence conditions [3]. Indeed, as long as we assume that the sequence $1 - \zeta$ converges to zero faster than the learning rate, the help can be seen formally as a noise decreasing along time.

The actor-critic-critic is able to integrate at any time a new trajectory without stopping the trainee because we use two critics. Algorithm 1 sketches the reinforcement learning critic, while algorithm 2 sketches the critic that manages the help.

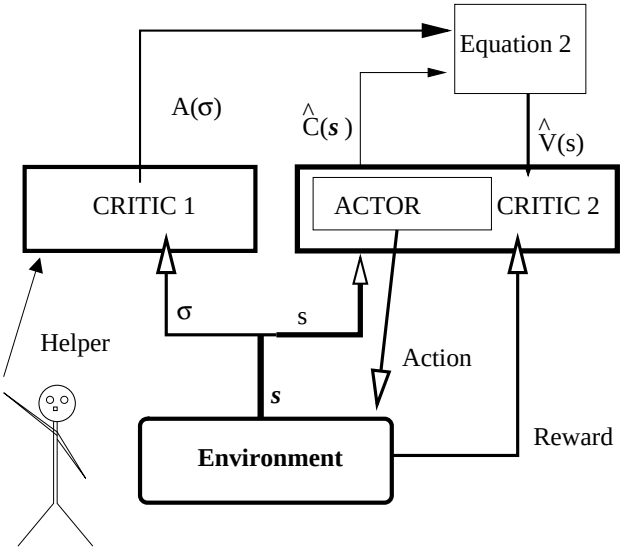

Figure 1. The actor-critic-critic architecture.

Algorithm 1 *Q-learning adapted to the ACC:* α, γ are the usual learning rate and discount factor, $Q(S, a)$ is the Q-value estimation using equation 3, $C(S, a)$ the internal Q-value estimate of the reinforcement learner.

for all episode e **do**
  perceive current state s_t
  while s_t not terminal **do**
    for all action a **do**
      estimate $Q(s_t, a)$ with equation 3
    end for
    select action a
    execute action a
    perceive next state s' and reward r
    $C(s, a) \leftarrow C(s, a) + \alpha(r + \gamma \max_{a'} C(s', a') - C(s_t, a))$
    $s_t \leftarrow s'$
  end while
end for

Algorithm 2 *Critic managing the help:* it is merely an infinite loop that awaits for two kinds of events: one kind of events is the reception of a new trajectory; the second kind of events is to answer to a state value estimate request. To simplify the notation, we suppose that there exists an application $\Phi : \mathscr{S} \rightarrow \Sigma$.

for ever **do**
  new event E
  if E = New trajectory Tr inserted **then**
    potential construction
    for all $s \in \mathscr{S}$ such as $\Phi(s) = \sigma$ and $\sigma \in Tr$ **do**
      $\zeta(s) \leftarrow 1.0$
    end for
  end if
  if E = Calculate Q(s,a) **then**
    update ζ
    return $(1 - \zeta(s))C(s, a) + \zeta(s) \sum_{s'} A(\Phi(s), \Phi(s'))\mathcal{T}^a_{s,s'}$
  end if
end for

A note on the convergence of this algorithm: it is sufficient that $\zeta(s)$ decreases faster than the learning rate $\alpha(s)$ to keep convergence properties, in the case where the TD(λ) itself converges (see e.g. [3]).

4 Experiments

The ideas presented above are very difficult to study theoretically. Indeed, there is a major issue which is related to the pragmatism of the approach: help is provided when one sees it necessary. Furthermore, as with most learning algorithms, the actual performance relies on a set of parameters, some of them being functional:

- the space in which help is provided and represented in A, thus the mapping from Σ to $\mathscr{S}$,
- the way the gradient is created along help trajectories, thus the way A is created from the help trajectories,
- the kernel k used to compute the potential A of any transition,
- the way A and C are combined to provide its estimate $\hat{V}(s)$, and in the case used here, the way $\zeta(s)$ varies.

Although conscious that a more fundamental demonstration would be desirable, we are only able to provide experimental evidences for the moment. So, we use the mountain-car problem as described in [11] as a testbed. This problem may seem overly simple, but its simplicity lets us study the effect of the help more easily than on a more complex problem. Furthermore, to provide help, it is nice to have some way to visualize the behavior of the agent; in the case of the mountain-car, this is fairly straightforward.

The problem is defined as follows: a car has to be driven up on a steep road in mountains, starting from the bottom of the hill. The state is specified by the position ($x \in [-1.2; 0.5]$) and the velocity ($\dot{x} \in [-0.07; 0.07]$) of the car; the initial state is $(-0.57, 0)$. There are three actions available: full throttle forward ($a = +1$), full throttle backward ($a = -1$), and no action ($a = 0$). However, from its starting position, even at full throttle, the car cannot reach its goal; it has to first move away from its goal, going up the opposite hill and then, full throttle and thanks to gravity and inertia, reach the goal on the other hill. There is a -1 return at each step, and a $+1.0$ return is giving since the car is reaching the abscissa $+1.2$ (the abscissa of the top of the mountain).

The dynamic of the environment is represented as follows :

$$\begin{aligned} x(t+1) &= x(t) + \dot{x}(t) \\ \dot{x}(t+1) &= \dot{x}(t) + 0.001 * a - 0.0025 * \cos(3.0 * x(t)) \end{aligned}$$

The speed is bounded by 0.07 and -0.07, if $x(t+1) < -1.2$ the $x(t+1)$ is fixed at -1.2 and $x(t+1)$ is set to 0.

We use the same representation as [11] for the value function ($\hat{C}$ with our notation), that is, ten regular 9×9 tilings, each offset by a random fraction of a tile width. The algorithm is a SARSA(λ) which acts as an actor-critic. λ is set to 0.9, the discount factor γ is set to 0.9, the learning rate is set constant to $\alpha = 0.05$, and the action selection is greedy. We use replacing eligibility traces, so that, basically, we use the algorithm given in fig. 8.8 in [11]. An episode lasts until the goal is reached.

As the number of actions is finite, the initial state is always the same, and the problem is deterministic, we made an exhaustive search and found that the best policy reaches the goal in a minimum of 98 steps.

In the implementation of eq. 1, we set the kernel constants to $d = c = -5$. To create the potential on transitions, the gradient is linear with 100 divided by the length of the trajectory.

The RLearner's state space $\mathscr{S} = [-1.2; 0.5] \times [-0.07; 0.07]$. The helper provides its help by specifying coordinates in the plane (x, y) as illustrated in fig. 2. So, $\Sigma = [-1.2; 0.5] \times [-1; 1]$. In our experiment, the helper has no means to specify a velocity, and naturally could not give any policy in sense of action to . Of course, this is a choice to conduct our experiment; this is not a limitation of our approach. Clearly, the velocity is very important in this problem; so the lack of this information in the provided help makes it interesting, but far from fully informative for the RLearner.

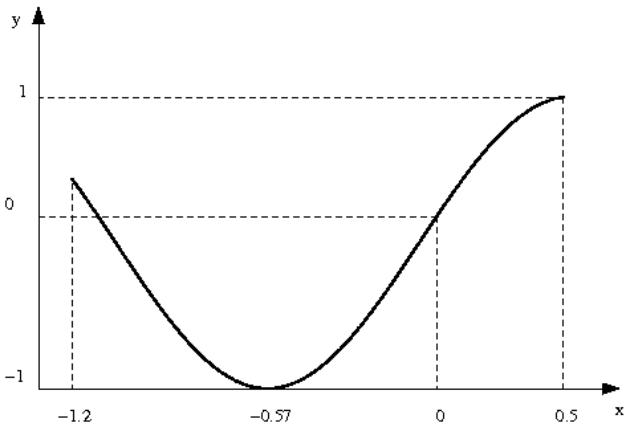

Figure 2. The mountain car task. The initial state is at the bottom ($x = -0.57$); the goal state is at $x = 0.5$. The state of the RLearner is $(x, \dot{x})$. The helper only perceives (x, y).

The help is based on four trajectories which are sketched in fig. 3. Each of these trajectories means "go straight in a direction, do not oscillate". These trajectories are not positioned perfectly on the road; furthermore, they are straight while the road is curved.

These two points make the provided help far from perfect, and rather crude.

We have then performed several experiments providing different helps made of those 4 trajectories. In all cases, there is a clear advantage when help is provided, even in cases which might seem counter-intuitive, such as merely using the trajectories 3 and 4 that together specify to go straight up and down along the hillside opposite to the goal. We have also experimented regarding the lengths of the segments that make the trajectories. With regards to the segments indicated in fig. 3 which extremities are on end-point positions and at the bottom of the hills, we noticed that shortening them to 60 % of their length still yielded very good performance.

As said before, there are various parameters to set. For instance, we have taken some time to try to figure out a correct ζ functional parameter. We have used a function that decreases along an episode, likewise the learning rate α. However, the intuition of providing help is that help should be used immediately, during the current episode, by the RLearner; then, in the next episodes, it should be used less and less, since the information should become more and more captured in the C function. So, it also seems natural that ζ remains constant during an episode, and decrease along episodes. We actually compared both approaches and obtained much better learning curves with the latter approach (see Fig. 4). With this approach, and by a suitable choice in the variation of ζ, the performance of the RLearner is greatly improved. Furthermore, there is a subtle balance to keep with regards to the use of help: when the help is provided, it is immediately taken into account and if the help is really useful , there is a clear

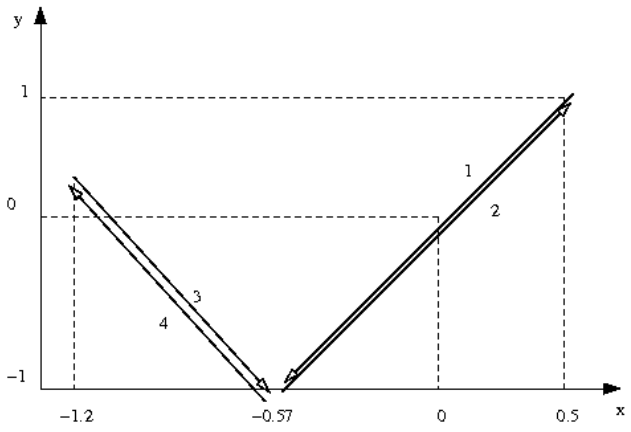

Figure 3. The transitions with which the different trajectories are built. The extremities of the transitions do not lie on points that can be reached by the car (they are not located on the road, but beneath or above the road).

immediate improvement in performance; however, at each episode, the help is less and less used, while the estimated value function (C) is conversely more and more used; the shift between these two information has to be tuned finely to avoid degradation of performance.

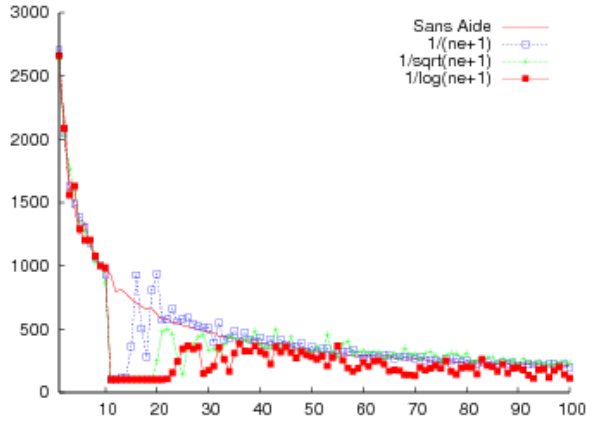

Figure 4. The learning curve for various ζ settings. "ne" denotes the number of the current episode. The help is provided at the 10^{th} episode. We see on this plot that the $\frac{1}{\log 1+ne}$ is the best one: during the first episodes after the help has been provided, the performance remains the same; along episodes, there is a shift in the importance given to the help, with regards to C; this implies a degradation of performance which is rather mild with this function. After some more episodes, the performance is good again: the help has been captured in the C function.

5 Conclusion and discussion

In this paper, we have presented our on-going work aiming at designing a principled way to help a reinforcement learner while it is solving a certain Markov decision problem. The idea is that this help cannot be expected to be thorough, nor perfect, and that the help is provided as trajectories in a certain space, which is not necessary the states space in which the RLearner is acting. To this end, we have defined an extension of temporal-difference algorithms that is able to store this help, and take advantage of it, while interacting with its environment. The algorithm sketched here is fairly general. We have tested experimentally its efficiency. The performance are those expected.

There are many directions to go on with this work. Obviously, experiments on other problems should be done. The way the value of a state is estimated also deserves more work. The representation of the help (the function A) is based on a function approximator to perform a regression. If the locally weighted regression seems a good algorithm for this purpose, the kernel function that is used may be discussed; at least, the kernel function has to be adapted to the problem and the help that is provided. However, the way we create the gradient should be further investigated. The ζ function that drives the combination between the help A and the value estimate C should also be investigated. We are currently thinking on a definition of it for each state, that is a $\zeta(s, ne)$. This would allow variations of ζ based on the number of visits to the states: if help has been given in a certain area of the state space and the learner has not visited this area, this help should be used when the learner will visit this area.

In the same spirit as what we did here, an other kind of help would be to provide "don't go in this area" information, that is, areas that would better be fled from.

Even if experimental evidences that our method "works" are not enough, it is already rather difficult to design a fully convincing way to do it. The criterion according to which we may judge the advantage gained by the algorithm is not clear. We try to study the "quality" of help, that is, how much a certain set of trajectories helps the RLearner; it is very difficult to define a quantitative measure that accurately handles this.

Finally, as we said in the introduction, providing help to a learner clearly goes well beyond from the reinforcement field,know as active learning. Partially interactive learning is a way to help solving large learning applications where human beings may provide help on-the-run, whenever he/she sees the algorithm in trouble.

REFERENCES

[1] Chris Atkeson, Andrew Moore, and Stefan Schaal, 'Locally weighted learning for control', *AI Review*, **11**, 75–113, (April 1997).
[2] A. Barto and S. Mahadevan. Recent advances in hierarchical reinforcement learning, 2003. Discrete event systems (2003, to appear).
[3] D.P. Bertsekas and J.N. Tsitsiklis, *Neuro-Dynamic Programming*, Athena Scientific, 1996.
[4] M. Dorigo and M. Colombetti, 'Robot shaping: developing autonomous agents through learning', *Artificial Intelligence*, **71**, 321–370, (1994).
[5] A. Laud and G. DeJong, 'The influence of reward on the speed of reinforcement learning: an analysis of shaping', in *Proc. 20th Int'l Conf. on Machine Learning*, (2003).
[6] Richard Maclin and Jude W. Shavlik, 'Creating advice-taking reinforcement learners', *Machine Learning*, **22**(1-3), 251–281, (1996).
[7] B. Price and C. Boutilier, 'Accelerating reinforcement learning through implicit imitation', *Journal of Artificial Intelligence Research*, **19**, 569–629, (2003).
[8] Martin L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*, John Wiley & Sons, Inc., 1994.
[9] J. Randløv, 'Shaping in reinforcement learning by changing the physics of the problem', in *Proc. of the Int'l Conf. on Machine Learning*, pp. 767–774, (2000).
[10] M.T. Rosenstein and A.G. Barto, *Learning and Approximate Dynamic Programming: Scaling Up to the Real World*, chapter Supervised actor-critic reinforcement learning, 359–380, John Wiley and Sons, Inc, 2004.
[11] R. Sutton and A. Barto, *Reinforcement learning*, MIT Press, 1998.