Propagation of Q-values in Tabular TD(λ)

Philippe Preux

Laboratoire d'Informatique du Littoral
UPRES-EA 2335
Université du Littoral Côte d'Opale
BP 719, 62228 Calais Cedex, France
philippe.preux@lil.univ-littoral.fr

Abstract In this paper, we propose a new idea for tabular TD(λ) algorithm. In TD learning, rewards are propagated along the sequence of state/action pairs that have been visited recently. In complement to this, we propose to propagate rewards towards neighboring state/action pairs along this sequence, though unvisited. This leads to a great decrease in the number of iterations required for TD(λ) to be able to generalize since it is no longer necessary that a state/action pair is visited for its Q-value to be updated. The use of this propagation process makes tabular TD(λ) coming closer to neural net based TD(λ) with regards to its ability to generalize, while keeping unchanged other properties of tabular TD(λ).

1 Introduction

Time derivative (TD) algorithms [9] are important reinforcement learning methods. Assuming discrete time, at each time step $t \in \mathbb{N}$, being in a certain state $s_t \in \mathcal{S}$, a reinforcement learning algorithm learns which action $a_t \in \mathcal{A}(s_t)$ to emit in order to optimize the total amount of rewards $R_T = \sum_{t=0}^{T} \gamma^t r_t$ it will receive, where r_t is the reward received at time step t, γ is the discount factor ($\gamma \in [0,1]$), and T can be ∞. In the sequel, we assume that conditions to apply dynamic programming techniques are not met.

A key point in the design of a TD(λ) algorithm lies in the choice of a structure to store estimates of qualities (or values). One possibility is to use a look-up table in which each state/action pair is associated to one element. The access as well as the update of a quality costs a single array element access. An update only concerns one state/action pair and to obtain an estimate of all state/action pairs, all pairs should be visited once at the very least. However, the size of the table is $\mathcal{O}(|\mathcal{S}|)$ which may be considerable. The other possibility is to use some sort of approximation architecture which represents the information in a much more compact form. Different architectures have been used [2]. Among them, neural networks are rather popular and well-known for their ability to generalize from their training. They have been used to tackle problems of large size, such as TD-Gammon which learnt to play Backgammon at a grand master level [13]. In this case, states are encoded in some way to be fed into the network. The output of the network provides the current estimate of the state value. This estimate is a

function of the network weights. The number of weights to be learnt is very small with regards to the number of possible states. In TD-Gammon, $|\mathcal{S}|$ is estimated to be 10^{20}, while the number of neurons is varying around 300-400 depending on the version of the program, resulting in $\mathcal{O}(10^4)$ weights to learn. These weights are typically learnt using a backpropagation process. So, the access to a state value costs the computation of the output of the network, while its update costs a backprop; clearly, these computaitonal costs are much larger than in the case of tabular TD(λ). However, when weights are updated for a given input state, the estimated value of all states are updated in the same time. There is thus some sort of implicit parallel update of the estimation of all values. This confers the neural based TD(λ) much greater ability for generalization: it is no longer required that a state is visited to have an estimate of its value.

Tabular TD(λ) is appealing when the number of states is small, whereas the ability for generalization of neural networks is very attractive. So, in this paper, we propose a variation of tabular TD to enhance its ability to generalize. This variation can be embedded in Q-learning, Q(λ) [14], Sarsa, Sarsa(λ) [8, 7] and their derivatives, and each of these algorithms can benefit of it. Basically, the idea is to add a propagation process of Q-values (and we call the resulting algorithm, the Propagation-TD, or PTD, as well as PQ(λ) and PSarsa(λ)). This idea relies on the observation that two neighboring states s and s' are such that, let a an action that can lead from s to s', the quality of (s, a) (denoted $Q(s, a)$) is likely to be closely related to the value of s' (denoted $V(s')$): if $V(s')$ is high, then the quality of state/action pairs that lead to s' is likely to be high too, and conversely, unless the return when transiting from s to s' via action a is very large. Actually, neural network based TD(λ) faces exactly the same problem. This propagation process transforms tabular TD(λ) into something coming close to neural TD with regards to its generalization ability. Indeed, it is no longer required that a state/action pair is visited for its quality to be estimated. Except with regards to the compactness of the neural net representation of Q-values, we end-up with an algorithm that combines the advantages of both approaches to TD learning.

In the sequel of the paper, we detail the propagation process in Sec. 2. Then, we evaluate its interest using an experimental approach in Sec. 3. In this evaluation, we are mainly interested in PTD as an agent behavior control learning algorithm, rather than as a value learning algorithm, although both issues are closely related. We are also interested in combining supervised learning with reinforcement learning in order to show the agent how to achieve its goal and makes learning faster. After that, we discuss and conclude in Sec. 4.

2 The Propagation Algorithm

In this section, we detail the keypoint of PTD, that is the propagation process. Before going into details, we wish to make it clear that this is a generic idea from which different strategies can be drawn for general cases, and some specific strategies can be drawn for specific cases, such as episodic tasks. So, we will not detail all these specificities. Some of them will be discussed in the experimental

discussion, in Sec. 3. We use the notation $s' \xrightarrow{b} s$ to mean that the action b emitted in state s' has a non null probability to lead to state s in a single transition; s' is a predecessor of s.

The propagation process acts as follows. Let us consider a TD(λ) process, and let us suppose that it has visited the state/action pairs $(s_t, a_t)_{t\in[0,n]}$ since it began. Then, the generic idea of PTD is as follows:

- build $\mathcal{S}_0 = \{s_{t\in[0,n]}\}$.

  Then, $Q(s', b)$ should be updated for any (s', b) such that $s' \xrightarrow{b} s \in \mathcal{S}_0$, then
- build $\mathcal{S}_1 = \mathcal{S}_0 \cup \{s', \text{such that } Q(s', b) \text{ has just been updated }\}$.

  Then, $Q(s', b)$ should be updated for any (s', b) such that $s' \xrightarrow{b} s \in \mathcal{S}_1$, and consequently,
- build $\mathcal{S}_2 = \mathcal{S}_1 \cup \{s', \text{such that } Q(s', b) \text{ has just been updated }\}$,
- iterate by building $\mathcal{S}_d$ using $\mathcal{S}_{d-1}$ until some criterion is fulfilled.

Fig. 1 sketches the propagation algorithm embedded in a naive[1] Q(λ) using accumulating eligibility traces. The propagation process is handled in the "Propagate" procedure. We could equally have given the propagation algorithm embedded into Sarsa(λ) by simply adding a call to "Propagate" at the end of it.

The propagate procedure relies on the assumption that we know whether a transition between one state to an other state via a certain action is possible or not; however, we do not need to know the probability of this transition.

Let us now discuss some details.

The propagation can be iterated as long as new state/action pairs are reached by way of a saturation process. However, the discount factor involves a fast decrease of the amount of update of qualities. So, after some iterations, this amount can be neglected and the propagation can be stopped: this may avoid a huge amount of updates, thus, and may save a huge amount of computation time. Then, propagation can be stopped after a certain amount of iterations either by limiting d, or by cutting off the propagation as the update becomes smaller than a given threshold.

The update of Q-values is discounted by the factor γ at each iteration of the propagation (repeat/loop of the "Procedure Propagate" in Fig. 1). This tends to create a gradient of qualities in the state/action space. Visually, in this space, the standard Q(λ) digs some sort of a very narrow furrow while PTD creates a whole gradient field. The update of Q-values is performed as follows using the update equation of Q-learning:

$$Q(s', b) \leftarrow Q(s', b) + \alpha[\gamma^d \max_{c\in\mathcal{A}(s)} Q(s, c) - Q(s', b)]$$

There are two differences with Q-learning update rule. First, γ appears with exponent d which is the distance from s' to the closest state in the eligibility trace. Second, the current reward term is absent since the emission of b in state

[1] the word "naive" is used according to [12]. In the case of Watkins Q(λ), a naive update of eligibility traces involves that these are not reset after exploratory steps.

Procedure PQ(λ)
Initialize $Q(s,a)$ arbitrarily, for all s,a
Repeat (for each episode):
    Initialize s,a
    Initialize $e(s,a)=0$ for all s,a
    Repeat (for each step in the episode):
        Take action a, observe r,s'
        Choose a' from s' (e.g., ϵ-greedy strategy)
        $a* \leftarrow \arg\max_b Q(s',b)$
        $\delta \leftarrow r+\gamma Q(s',a*)-Q(s,a)$
        $e(s,a) \leftarrow e(s,a)+1$
        For all s,a
            $Q(s,a) \leftarrow Q(s,a)+\alpha\delta e(s,a)$
            $e(s,a) \leftarrow \gamma\lambda e(s,a)$
        Propagate
        $s \leftarrow s'; a \leftarrow a'$
    Until s is terminal

Procedure Propagate
    $d \leftarrow 0$
    $\mathcal{S}_d \leftarrow \{(s,a), \text{such that } e(s,a) \neq 0\}$
    Repeat
        $d \leftarrow d+1$
        $\mathcal{S}_d = \mathcal{S}_{d-1}$
        For all $s \in \mathcal{S}_{d-1}$
            For all $(s',b) \notin \mathcal{S}_{d-1}$
                If $s' \xrightarrow{b} s \in \mathcal{S}_{d-1}$ Then
                    update $Q(s',b)$
                    add (s',b) to $\mathcal{S}_d$
    Until stopping criterion is fulfilled

Figure 1. Generic outline of Propagation-Q(λ). The basis is the tabular Q(λ) as expressed in [12, p. 184], using a naive strategy, and accumulating eligibility traces. We assume that the "For all" construction has a SIMD semantic, that is, all iterations of a "For all" loop are executed at the same time.

s' has not been performed actually. This choice might be discussed. If this can seem troublesome, it should be pointed out that this is precisely what happens in neural TD(λ). Indeed, in neural TD, the quality of a state/action pair being stored in the weights of the network, changing the weights after a state/action pair has been visited involves an alteration of the quality of many state/action pairs. However, the consequences associated with the emission of the action are not observed for these pairs. In some cases of application of PTD, a certain term could be used to approximate a systematic consequence of an action, such as an action cost. More generally, one might use a model of expected rewards.

In episodic tasks, a possible variation on this basic algorithm is to propagate only at the end of the episode and not at each step. For episodic tasks where a reward is always null except when the end of the episode is reached, propagating the qualities only at that time does not alter the way the algorithm works and saves a lot of computation time.

3 Experimental Assessment

In this section, we provide an experimental assessment of propagation Q(λ) by comparing its performance with regards to Q(λ) on a test problem. The test problem consists in finding an outlet or a goal state in a 2D labyrinth. This problem can be made more complex by making it more or less random, stationary or not, ... So, this is actually a whole family of problems rather than a single problem that is used.

3.1 Experimental Setup

The labyrinth is defined in a gridworld where each cell can be either empty or filled with a piece of wall. The state space is the set of cells of the labyrinth. Each cell is numbered and the algorithm knows the cell in which it lies. Only states corresponding to empty cells can be occupied by the agent. In each state, the agent has the choice to stay in its current state, move upward, downward, leftward, or rightward if it does not hit a piece of wall; only non wall-hitting moves are allowed in any state. The agent receives a positive reward when it reaches a goal state of the labyrinth, otherwise it does not receive any reward. Clearly, this problem is markovian, and it is also fully deterministic. This is an episodic task in which a reward is given only at the end of the episode. It is not necessary to propagate at each iteration of the inner repeat/until loop of procedure PQ(λ). It is only necessary to propagate it at the end of the episode.

We are mainly interested in algorithm to control an agent so that we are mainly interested in algorithms that learn to behave correctly, rather than in algorithms that learn to predict accurately state values or state/action pair qualities. So, to evaluate our approach, we are interested in the number of iterations required to learn to achieve the task, as well as the number of correct decisions that the algorithm is making regarding actions that have been emitted to reach a

goal state, as well as the number of correct decisions the algorithm has learnt regarding actions that would have been emitted if the algorithm had followed other trajectories. This latter somehow measures the generalization the algorithm has made from its experience along trajectories it followed to reach goal states. With the word "iteration", we mean here one iteration of the inner repeat/until loop in procedure PQ(λ) of Fig. 1. We compare the performance of Q(λ) and PQ(λ). Both are run on the two leftmost mazes shown at Fig. 2. The first one (called the "Pacman maze" later on) is composed of 206 states (wall cells are not counted), while the second one is composed of 2310 states. This latter maze is drawn from the partigame paper [4] and thus called the "Partigame maze". Apart from the difference in their size, the two mazes differ greatly in that in the pacman maze, there are lots of walls and in most states, the number of possible actions is reduced to 2 (or 3 if immobility is possible). In the partigame maze, there are 4 (resp. 5) possible actions in the large majority of states. Finally, in the pacman maze, goal states are the two outlets while in the partigame maze, the goal state is the one used as such in the partigame paper.

For each run of the algorithm, we set it into an initial cell and let it find a goal state. Then, we reset its position and run it again performing 100 reaches of the goal (without resetting Q-values along these 100 runs). To obtain correct statistics, we average the performance over 10 runs. In the pacman maze, initial states are drawn at random at each new run while in the partigame maze, initial and goal states are set to those used in the partigame paper [4]. To avoid certain biases due to the pseudo-random generator and be able to discuss experimental results more thoroughly, the algorithms can be run so that the initial states are the same for the different algorithms. Propagation is stopped whenever the amount being propagated becomes smaller than a certain threshold (10^{-10}). γ is set to 0.9, α to 0.5, and λ to 0.9. Q-values are initialized to 0. The selection of action is ϵ-greedy, with ϵ set to 0.1, that is 10% of exploratory moves (exploratory moves are random moves).

3.2 Results

As expected, the average number of iterations is significantly smaller for PQ(λ). More preciselly, for both Q(λ) and PQ(λ), the naive strategy provides much better results than the non naive version. Consequently, we now use naive versions of the algorithms unless explciitely mentioned. At the least, Q(λ) needs 20% more iterations than PQ(λ) in the pacman maze, 4 times more for the partigame maze: this is clearly an effect of the size of the state/action space. Both algorithms perform approximately the same amount of backups (2 10^6 for the pacman maze along 100 episodes); PQ(λ) performs much more backups during the first episode than during the next ones. However, it should be said that the way we performed the comparison is unfair for PQ(λ). Indeed, for this kind of problems (deterministic), once Q-values have been propagated to all state/action pairs, exploration is no longer necessary. The amount of exploratory moves being fixed by the value of ϵ (0.1), PQ(λ) always performs 10% exploratory moves that are almost always useless: once the gradient field has been created, a mere

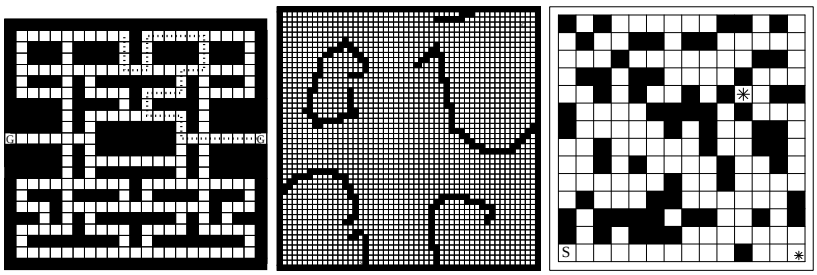

Figure 2. Three mazes used in this paper. The leftmost maze has two goal states (the outlets) as indicated by G's leftmost and rightmost cells at mid-heigth). In the rightmost maze, the algorithm has to find its way from an initial cell (S located in the bottom line) to a goal state (G located in the upper right corner). The rightmost maze is used at section 3.4. It has one initial state (S) and two goal states (stars), one goal being better than the other.

greedy selection of action is optimal. During a single execution of PQ(λ), the first episodes require much more iterations that the others. If we do not take the 10 first episodes into account in the measure of performance, the relative performances of the compared algorithms remain unchanged. This shows that the difference of performance does not rely on a transient effect.

We have performed an analysis of the role of the parameter (α, γ, and λ). The results are contrasted since Q(λ) is at its best with high values for α and λ ($\alpha = 1.0, \lambda = 0.75$), while PQ($\lambda$) performs at its best with small values ($\alpha = \lambda = 0.1$); both algorithms perform better with $\gamma = 0.5$.

Table 1. This table summarizes some results obtained on the labyrinth problem for the pacman maze and the partigame maze. Figures are averaged over 10 runs, each made of 100 episodes. The second and third lines gives the size of the problems, either as the number of states, or as the number of possible state/action pairs (this number is given considering that immobility is forbidden; when immobility is a valid action, the number of state/action pairs is the sum of the second and third lines). The "Length" column gives the average number of iterations performed to reach the goal (this number takes into account the distance between the initial state and the closest goal state), the "Backups" column gives the average number of backups per run, while the "Greedy actions" column gives the average percentage of states for which the learnt greedy action is correct.

	Pacman maze			Partigame maze		
states	206			2310		
state/action pairs	488			8750		
Algorithm	Length	Backups	Greedy actions	Length	Backups	Greedy actions
Q(λ)	46.7	$2.3\ 10^6$	48%	14.8	$6\ 10^6$	5%
PQ(λ)	38.3	$2.7\ 10^6$	77%	3.5	$3\ 10^6$	60%

3.3 Capacity of Generalization

It is interesting to try to measure the capacity of generalization of the algorithms. The capacity of generalization can be assessed as follows: having learnt a good trajectory from an initial state to a final state, for what fraction of the state space have correct actions also been learnt? Clearly, after one run of Q-Learning, a one step trajectory has been learnt; for Q(λ), a several step trajectory has been learnt, according to the length of the eligibility trace when the goal state is reached. In PQ(λ), much more correct actions have been learnt. For the two mazes that are used here, we obtain the results of table 2 after 1 and 100 episodes. As expected, PQ(λ) obtains the highest measures. Of course, the first episode of PQ(λ) requires larger run times than for the other algorithms (approximately 10 times with our non optimized version). However, the next runs of PQ(λ) are very efficient and very fast: as far as a gradient is already available, the algorithm has just to follow it greedily to reach the goal.

Table 2. Proportion of states for which the correct action has been learnt.

Algorithm	after 1 episode		after 1000 episodes	
	Pacman maze	Partigame maze	Pacman maze	Partigame maze
Q-Learning	0.4%	0.04%	16.9%	6.0%
Watkins' Q(λ)	0.4%	0.04%	18.7%	7.3%
naive Q(λ)	1.8%	0.2%	16.9%	8.0%
PQ(λ)	11.1%	7.6%	72.4%	59.0%

It is also interesting to discuss the proportion of correct behaviors that are learnt after a certain amount of episodes, or after having used a certain amount of CPU time. After 100 episodes on the partigame maze, PQ(λ) has learnt 59% correct greedy actions (that is, in 59% of the state space, PQ(λ) greedy selection selects the correct action to perform – indeed running 100 or 1000 episodes does not increase significantly this figure); after 1 500 episodes, naive Q(λ) has only learnt 9%. Using the same CPU duration (1 minute on a Pentium III, 500 MHz running Linux), PQ(λ) performs 100 episodes, while Q(λ) performs 10^4 episodes. In this case, Q(λ) has only learnt 15% correct actions in the whole state space, that is one quarter of what PQ(λ) does in the same amount of time. Regarding the number of backups, PQ(λ) perfoms approximately 3 10^6 backups, while Q(λ) performs 5 10^7 within this amount of time. When plotted against the number of episodes, this proportion of correctly learnt greedy actions levels; to get closer to 100%, one has to perform more episodes so that certain yet unexplored regions of the state space get explored.

From what has been reported, it is clear that, as expected, PQ(λ) is able to generalize much better than classical Q(λ).

3.4 Dealing with Local Optima

For the moment, reaching either one of the two outlets of the pacman maze provides the same positive return. We now consider a problem closely related in which one outlet is sub-optimal: reaching one of the two outlets (say, the left-most) provides a return equal to +5.0, while reaching the other outlet provides a return of +10.0. We expect that PQ(λ) (as well as regular Q(λ)) will be able to learn to reach both outlets, though favoring the outlet associated with the largest return. Results are displayed in table 3 for two mazes, the pacman maze where one outlet is made suboptimal, and a misleading maze drawn from [3]. In the pacman maze, the two goals are equally easy to find, while in the misleading maze, the sub-optimal goal is much easier to find than the real optimum. PQ(λ) and Q(λ) reach the best outlet much more often than the sub-optimal one.

Table 3. This table displays the proportion of executions that reach either the sub-optimal or the optimal goal in the pacman maze and in the misleading maze where the two goals are distinguished with regards to the return they provide when reached. Results are obtained over 10^3 runs of each algorithm.

	PQ(λ)	Q-learning
pacman maze	91%	84%
misleading maze	24.5%	3%

3.5 Combining Reinforcement Learning with Training

One can hope to greatly improve the performance of Q(λ) by showing it a trajectory (also called "training" technique), for instance, by way of a graphical interface. However, this "obvious" improvement is not so successful or, for the least, it is less successful that one would expect. Indeed, after having been shown a trajectory and subsequently having been reset to its initial position, Q(λ) begins by following the demonstrated trajectory but, after some steps and depending on the value of ϵ, it escapes from this trajectory because it has performed an exploratory move. Once Q(λ) has left the demonstrated trajectory, it is completely lost and is generally unable to return to it. Then, Q(λ) has to reach the target by its own means, the demonstrated trajectory being then totally unused.

PTD solves this problem. When being shown the trajectory, PTD creates a gradient towards the taught trajectory. Then, when behaving autonomously, this trajectory is followed and exploratory moves simply lead out of the trajectory to which it is attracted back by the gradient field when performing a greedy move.

Furthermore, exploratory moves naturally lead to the optimization of the taught trajectory. Indeed, when shown via the graphical interface, the trajectory is generally not perfect, that is, it is seldomly the best possible trajectory. Starting from an already good approximation of the best trajectory, PTD optimizes it little by little.

To illustrate this point, on the pacman maze, during the first episode, we train the algorithm: instead of selecting itself the action to emit, the algorithm follows a training trajectory. This trajectory is voluntarily sub-optimal, being 1.94 longer than the shortest trajectory from the initial state to the closest goal (the training trajectory is the dashed line on the Pacman maze in Fig. 2). During the next episodes, Q(λ) trajectories are generally getting a little bit longer, while those of PQ(λ) are getting shorter. Averaged over 100 runs, after 1000 episodes, the length of the trajectories followed by PQ(λ) has shrunk down to an average of 1.27 times longer than the shortest one, while the average length of those followed by Q(λ) is 1.83 times longer. The shortest trajectory followed by PQ(λ) is 1.11 times longer than the shortest, the 10% extra-length being explained by $\epsilon = 0.1$ leading to 10% exploratory moves; the longest trajectory is 1.74. In the case of Q(λ), the shortest trajectory is 1.42 and the longest is 4.0.

An other worthy point regards whether if the training trajectory leads towards a sub-optimal goal, the algorithm is still able to reach the optimal goal. To check this, in the pacman maze, we make the rightmost outlet sub-optimal, while the leftmost outlet is the best rewarding goal and we train the algorithm with the same trajectory as before, leading to the sub-optimal goal. Performing 10^5 episodes after having been trained during the first one, PQ(λ) finds the best optimum after 10^3 epsodes for the first time. On average, the best optimum is found 66% along these 10^5 episodes.

More generally, we think that the use of PTD (instead of tabular TD), in combination with training, can be applied in many cases and can bring important speed-ups despite its initial overcost which is largely compensated by its ability to generalize from its own experience and from training.

4 Discussion and Conclusion

In this paper, we have proposed the propagation TD algorithm. Based on tabular TD methods, PTD tends to bridge the gap between tabular TD and neural TD with regards to its generalization capabilities. While tabular TD updates only Q-values of state/action pairs that are visited, PTD propagates the updates to state/action pairs that lead to states that have been visited. Propagation is grounded on the idea that if a state s has a high value, then state/action pairs that lead to s are likely to have a high quality. This idea is general and can be applied to all tabular TD algorithms. There are a number of nice features of PTD. First, PTD does not involve any extra parameter. Second, as a natural extension of tabular TD(λ) algorithms, propagation can be used instead of these algorithms in many places and applications. For example, it can take advantage of techniques to speed-up Q(λ), or be used in hierarchical Q(λ) to solve POMDPs [15], ... Third, though not studied here, existant convergence proofs should be able to be adapted from other algorithms to PTD[2]. Fourth, PTD is worthy when combining reinforcement learning with training (or supervised learning)

[2] as a matter of fact, the idea of PTD has been proposed very recently and independently by other authors, accompagnied with such a convergence proof [16]

by avoiding tabular TD to be unable to come back to the taught trajectory after an exploratory move. In some sense, the rather blind tabular TD method becomes far sighted. Fifth, the experimental assessment has shown that, even though it has been implemented very crudely and we have not spent any time to optimize neither the propagation of Q-values, nor the number of backups, the run time of PTD is very reasonable and the trade-off between the run time and the number of learning iterations is not bad for PTD. These last three points make us think that PTD is worthy for applications based on Q-Learning that require generalization abilities. The fact that PTD only needs to know which transitions are possible is also a nice point with regards to real time dynamic programming [1] for which a complete model is necessary.

With regards to existing work, PTD shares some similarities with Dyna, Prioritized sweeping, and queue-Dyna. All three algorithms build a model $(\hat{T}, \hat{R})$ where $\hat{T}(s, a, s')$ is the estimated probability that taking action a in state s leads to state s', and $\hat{R}(s, a)$ is the estimated return of taking action a in state s. Dyna was introduced by [10, 11]. In complement to regular Q-learning, Dyna maintains and updates the model $(\hat{T}, \hat{R})$ at each iteration and uses it to update its estimates of the quality of k other state/action pairs drawn at random among those that have been visited. Prioritized sweeping and queue-Dyna are two similar techniques that have been proposed independently, respectively by [3] and [5]. They are both derived from Dyna from which they differ in that state values are estimated instead of state/action pair qualities, and updated estimates are not drawn at random. Each state is characterized by its predecessors, as well as a priority. Value estimates are updated according to the priority of states: the value of the states having the highest priority is updated. For each updated value $V(s)$, the priority of s is reset to 0, while the priority of the predecessors s' of s is updated proportionaly to the change in $V(s)$ and $\hat{T}(s, a, s')$. Thus, PTD is yet an other strategy. First, PTD does not make use of such a thorough model: it solely relies on whether a transition between two states is possible or not. Second, the updated state/action pairs are all those that have been visited since the beginning of the episode as PTD uses eligibility traces, as well as neighboring state/action pairs of updated state/action pairs.

In the near future, we wish to optimize the propagation process. More fundamentally, we wish to compare more precisely the ability to generalize of PTD with regards to neural TD. We also wish to evaluate the generalization abilities of PTD with regards to the size of the state/action space, and the performance of PTD in a non deterministic environment. We are also currently evaluating the usefulness of using PTD instead of Q-learning to control the animat MAABAC [6]: this is a multi-segmented artefact in which multiple reinforcement agents learn to collectively solve a task. In this application, the environment is no longer markovian, nor stationary.

Acknowledgements

The author would like to thank anonymous reviewers for their constructive remarks, as well as pointing towards the recently published and very closely related paper [16].

References

[1] A.G. Barto, S.J. Bradtke, and S.P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72:81–138, 1995.

[2] D.P. Bertsekas and J.N. Tsitsiklis. *Neuro-dynamic programming.* Athena Scientific, 1996.

[3] A.W. Moore and C.G. Atkeson. Prioritized sweeping: reinforcement learning with less data and less real time. *Machine Learning*, 13:103–130, 1993.

[4] A.W. Moore and C.G. Atkeson. The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. *Machine Learning*, 21, 1995.

[5] J. Peng and R.J. Williams. Efficient learning and planning within the dyna framework. *Adaptive Behavior*, 1(4):437–454, 1993.

[6] Ph. Preux, Ch. Cassagnabère, S. Delepoulle, and J-Cl. Darcheville. A non supervised multi-reinforcement agents architecture to model the development of behavior of living organisms. In *Proc. European Workshop on Reinforcement Learning*, October 2001.

[7] G.A. Rummery. *Problem Solving with Reinforcement Learning.* PhD thesis, Cambridge University, 1995.

[8] G.A. Rummery and M. Niranjan. On-line Q-learning using connectionist systems. Technical Report TR 166, Cambridge University, Enginerring Department, September 1994.

[9] R.S. Sutton. Learning to predict by the method of temporal difference. *Machine Learning*, 3:9–44, 1988.

[10] R.S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proc. Seventh Int'l Conf. on Machine Learning*, pages 216–224. Morgan Kaufmann, 1990.

[11] R.S. Sutton. Planning by incremental dynamic programming. In *Proc. Eighth Int'l Conf. on Machine Learning*, pages 353–357. Morgan Kaufmann, 1991.

[12] R.S. Sutton and A.G. Barto. *Reinforcement learning: an introduction.* MIT Press, 1998.

[13] G. Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38:58–68, 1995.

[14] C.J.C.H. Watkins. *Learning from delayed rewards.* PhD thesis, King's college, Cambridge, UK, 1989.

[15] M. Wiering and J. Schmidhuber. HQ-Learning. *Adaptive Behavior*, 6(2):219–246, 1997.

[16] W. Zhu and S. Levinson. PQ-learning: an efficient robot learning method for intelligent behavior acquisition. In *Proc.* 7^{th} *Int'l Conf. on Intelligent Autonomous Systems*, March 2002.