Q-Batch: initial results with a novel update rule for Batch Reinforcement Learning

João Cunha, Nuno Lau, and António J. R. Neves

DETI/IEETA - University of Aveiro
Campo de Santiago
3810-193 Aveiro, Portugal
{joao.cunha,nunolau,an}@ua.pt,

Abstract. Batch Reinforcement Learning has established itself as a valuable alternative to develop learning and adaptive agents. Batch Reinforcement Learning algorithms are characterized by obtaining a policy from a set of collected data. Common methods apply adapted versions of RL update rules, such as Q-Learning, on the transitions of the batch, building a pattern set. The target values of the pattern represent a value function, which is latter "fitted" with a function approximator using batch supervised learning methods. This paper presents the first results with a novel update rule, Q-Batch. The proposed method is benchmarked against the batch version of Q-Learning and Watkins Q(λ) in the Neural Fitted Q Iteration framework. The proposed work is tested in the Predator-Prey simulated environment. Empirical results show that the proposed method is able to achieve comparable or better asymptotical performance while requiring fewer interactions with the environment.

1 Introduction

Reinforcement Learning [15] is a sub-area of Machine Learning drawing inspiration on biology and animal behaviour learning. Although not a new field of study, it has received a lot of attention in recent years from researchers worldwide having sprawled in a multitude of different methods [19, 16].

Reinforcement Learning is usually formalized using a Markov Decision Process (MDP). MDPs are defined by the tuple $\langle S, A, P, R\rangle$: a state set S, an action set A, a probability distribution model of the system dynamics $P(s'|s, a)$ and an immediate reward function $R(s, a, s')$. In every time step t the agent is in a state $s_t \in S$ and takes an action $a_t \in A$. In the following time step $t+1$ the agent observes a transition to state s_{t+1} and collects a reward r_{t+1}. The key goal of Reinforcement Learning is to find an optimal policy π, that maximizes the cumulative sum of rewards, or the return at time t, $R_t = \sum_{k=0}^{+\infty} \gamma^k r_{t+k+1}$, with a discount factor $\gamma \in [0, 1]$.

In this work we will focus on value function based methods. As the name implies these methods find policies by first estimating value functions. We will further focus on methods estimating Q-functions since these methods are able to cope with situations where a model of the environment is not available or is too complex to sample from. The most common update rule to estimate a Q-function, is Q-Learning [18], which is defined by:

$$Q(s_t, a_t) = \alpha(r_{t+1} + \gamma \max_b Q(s_{t+1}, b)) + (1 - \alpha) Q(s_t, a_t) \quad (1)$$

Having estimated a Q-function, it is straightforward to obtain a policy, $\pi(s) = \arg\max_a Q(s, a)$.

Common employed methods able to estimate value functions are guaranteed to converge to the optimal policy if all the states (or state-action pairs) are visited infinitely often. This is a major drawback in physical or real-world systems where the interaction with the environment may be rather costly. On the other hand, convergence is only guaranteed if the value function is represented by a table which limits the applicability of these methods to discrete state spaces.

A class of methods that aim to address the shortcomings highlighted before is Batch Reinforcement Learning. A famous method among this class of methods is the Fitted Q Iteration (FQI) [3] on which other methods are inspired. FQI can be regarded as Q-Learning for Batch Reinforcement Learning. However it seems counter-intuitive to apply Q-Learning, an online and transition based update rule, when Batch Reinforcement Learning methods have a (sometimes rather large) set of experiences available. However, the application of episodic update-rules such as Monte-Carlo policy evaluation limits the reusability of the existing batch. In this paper we aim to improve the learning time of FQI methods, by introducing a novel update-rule, Q-Batch, that is able to estimate a Q-Function in an episodic manner while maintaining data reusability.

The remainder of this paper is structured as follows: Section 2 briefly presents the basic concepts of Batch Reinforcement Learning. Section 3 introduces our proposed approach. Section 4 presents the testing environment and the conducted experiments. Section 5 discusses the obtained results and section 6 draws the conclusions of our contribution.

2 Batch Reinforcement Learning

Batch Reinforcement Learning [8] differs from other RL methods in that it estimates a policy π from a set $\mathcal{F}$ of N transitions sampled from the environment. Batch Reinforcement Learning is sub-divided in two problems. In the first, N, the size of the batch is fixed, giving rise to the the Fixed Batch Problem. In this setting no assumptions can be made from the policy used to build the set $\mathcal{F}$. The transitions can be sampled from arbitrary policies and are not guaranteed to be sampled from connected trajectories.

On the other hand, Batch RL methods can estimate policies *while* interacting with the environment, solving the Growing Batch Problem. As the name implies the size of $\mathcal{F}$ increases as the learning agent alternates between an interaction phase, sampling experience from the environment, and a learning phase, determining the best policy from the collected data. This allows control over the sampling process generating $\mathcal{F}$.

A notable feature that characterizes Batch RL methods is synchronous update. While purely online methods, such as (1), update a Q-function as soon as a transition is observed, Batch RL methods update a Q-function once for all state-action pairs in $\mathcal{F}$. To

achieve a synchronous update, a so-called Batch RL operator is applied, in a kernel fashion, to all collected transitions, generating a pattern $\mathcal{P}$ mapping state-action pairs to the target Q-function value, $\mathcal{P} = \{(s_i, a_i), \bar{Q}(s_i, a_i) | i = 1, \ldots, N\}$. This presents an opportunity to apply batch supervised learning methods to generate an approximate Q-function $\hat{Q}$, combining function approximators and $\mathcal{P}$ as a training set, through regression.

Among other methods, we highlight the Neural Fitted Q Iteration (NFQ) [12], an instance of the class of FQI methods, which relies on multilayer perceptrons, a powerful function approximator, to represent the approximate Q-function $\hat{Q}$. Additionally, NFQ applies batch supervised learning approaches, such as RPROP [13], which are more sophisticated than simple gradient descent techniques.

NFQ builds the pattern $\mathcal{P}$ by applying a Dynamic Programming adapted version of the Temporal Difference Q-Learning update rule (1), over all the collected transitions:

$$\bar{Q}(s_i, a_i) = r_i + \gamma \max_b \hat{Q}(s'_i, b), \forall i \in 1..N \tag{2}$$

where s_i, a_i and s'_i are the current state, action chosen and following state of the i^{th} transition, respectively.

A careful analysis of (2) reveals that it is equal to (1) with maximum learning rate, $\alpha = 1$. While the learning rate is omitted, the stochastic approximation is solved by minimizing the squared error during the approximation phase, effectively yielding an expected return.

Figure 1 represents the execution flow of FQI methods: the Interaction, Pattern Generation and Supervised Learning modules can be implemented differently to realize different Fitted Iterations. The work presented here focuses on the Pattern Generation module. A complete and detailed discussion of Batch Reinforcement Learning is out of the scope of this paper. For more details, we refer interested readers to [8] and [14].

3 Q-Batch

While in the Fixed Batch Problem no assumptions can be made about the connectivity of the transitions in $\mathcal{F}$, in the Growing Batch Problem, the most current Q-function is used to derive a policy, which is in turn used to interact with the environment. It is then safe to assume that we can sample from the environment along connected trajectories. With this assumption, we can quickly realise that an update rule such as (2) is not the most efficient to apply in such situation. For one, the method only uses immediate information (the reward and the Q-value in the following state) although it has information available about the effect of its actions along a trajectory.

To have a sense of the value of such information, consider the following situation. We have a randomly initialized Q-Function $\hat{Q}$ and luckily we are able to sample from an informed policy, a near-optimal policy that is able to generate near optimal trajectories. Although this is a very uncommon situation, where we would have such a policy at the initial stages of learning, it serves to show that after having sampled trajectories, and applying (2), we would need to repeat the fitting process possibly many times, in a

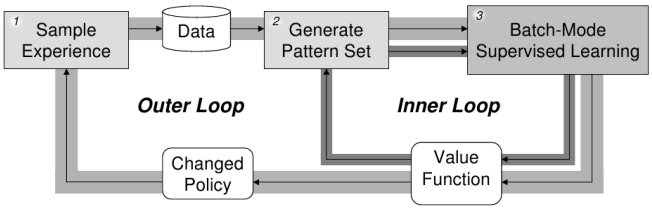

Fig. 1. The Batch Reinforcement Learning framework, adapted from [4]. The framework is divided in three distinct modules: sampling experience from the environment, generating a pattern set from the collected experience and batch supervised learning to train a function approximator that represents the value function.

similar manner to Experience Replay [9], in order to propagate back the values all the way from the end of the trajectory to the initial states of the trajectory.

It might be tempting to apply a Monte-Carlo update rule based on trajectory rollouts [17] such as (3), since we could easily propagate the value on the final states to the initial states within an episode.

$$\hat{Q}(s_i, a_i) = R_i, \forall i \in [1, N] \tag{3}$$

However we need to keep in mind that these kind of update rules don't bootstrap. The consequence is that since the Q-value does not depend on an estimate of the Q-function, data collected can only be used to determine the current Q-function and must be discarded afterwards. Considering that applications of Batch RL involve real world physical systems, discarding collected data is highly undesirable.

In order to develop a more efficient Batch RL operator, we first start by rearranging the structure of data set $\mathcal{F}$, with the assumption that transitions are sampled along connected trajectories. The basis of the proposed structure are the episodes. The set $\mathcal{F}$ is now composed of N episodes. Each episode i is a time consistent sequence of T_i states, actions and rewards. This representation allows for a more compact data set $\mathcal{F}$ since the following state of a given transition and the current state of the following transition are now redundant information, thus we can avoid storing following states s' explicitly. Considering the timestep j of episode i, the corresponding state, action and reward are now represented by $s_{i,j}$, $a_{i,j}$ and $r_{i,j}$, respectively.

Rewards can also be removed from the data set $\mathcal{F}$ altogether. Since the evaluation of a value function is performed offline, with regards to the interaction with the environment, and it is reasonable to assume that for many applications, the reward function, $R(s, a, s')$, is known, the immediate rewards can be calculated during the application of the update-rule. This features two main advantages: not only it allows for a more compact data set $\mathcal{F}$, an advantage considering that the number of collected trajectories can grow to a large number on long-term autonomy applications, but also by recalculating the rewards, the data set $\mathcal{F}$ can be reused for different, but similar, learning tasks. Here we support a point of view that in a Reinforcement Learning context, the reward function is the most succinct description of a learning task [10]. Therefore using a pre-

viously collected $\mathcal{F}$ and changing the reward function to learn a different task is not only perfectly reasonable, but a valuable asset in real world applications.

A well known approach to unify Temporal Difference and Monte-Carlo methods is the application of eligibility traces [15]. An eligibility trace can be regarded as a temporary record of the occurrence of a transition. When performing a backup, eligible transitions propagate a fraction of the observed rewards towards the state being evaluated. While this concept has been extended to the *off-policy* case, such as Watkins' Q(λ) and Peng's Q(λ) [11], both variants rely on the existence of a policy, considered the current optimal policy, to propagate the eligible rewards. However before a close-to-optimal policy is obtained it may happen that some "good" rewards will not be propagated, which would accelerate convergence to the optimal policy faster.

Instead, we recover Watkins idea of n-step return [18]. It builds on the basis that the return can be calculated not only by shallow backups, such as TD methods, or full backups, such as Monte Carlo methods, but from an intermediate number of n steps of real rewards and the estimated value of the n-th state. Therefore a one-step return is based on the first reward and the value of the state one step later, a two-step return is based on the two first rewards and the value of the state two steps later, and so on as shown in (4).

$$\begin{aligned}
R_t^1 =& r_{t+1} + \gamma V(s_{t+1}) \\
=& r_{t+1} + \gamma \max_{a \in A} Q(s_{t+1}, a) \\
R_t^2 =& r_{t+1} + \gamma r_{t+2} + \gamma^2 \max_{a \in A} Q(s_{t+2}, a) \\
R_t^3 =& r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 \max_{a \in A} Q(s_{t+3}, a) \\
& \vdots \\
R_t^n =& \sum_{i=0}^{n-1} \gamma^i r_{t+1+i} + \gamma^n \max_{a \in A} Q(s_{t+n}, a)
\end{aligned} \tag{4}$$

Historically, n-step returns were deprecated over other update rules, since a backup could only occur after n time steps had passed. In a Batch Reinforcement Learning context however, it provides a valuable alternative, since all backups are performed synchronously and all data required to evaluate n-step returns is available in $\mathcal{F}$.

We then propose a novel update rule based on n-step returns applicable to Batch RL applications, named Q-Batch. Instead of trying to find an optimal value for n, under Q-Batch, $\bar{Q}$ is the maximum n-step return found, as shown in (5).

$$\begin{aligned}
&\bar{Q}(s_{i,j}, a_{i,j}) = \max_k R_{i,j}^k \\
&= \max_k \left(\sum_{l=0}^{k-1} (\gamma^l r_{i,j+1+l}) + \gamma^k \max_{b \in A} \hat{Q}(s_{i,j+k}, b) \right), \\
&\forall i \in 1..N, \forall j \in 1..T_i - 1, \forall k \in 1..T - j
\end{aligned} \tag{5}$$

where $R^k_{i,j}$ is the k^{th} observed return in time step j of the episode i.

This allows for an *off-policy* update-rule that is able to reuse every transition in $\mathcal{F}$ regardless of the policy used in the interaction with the environment. One can easily observe that from the analysis of (4) that $R^1_{i,j}$ is equal to (2). Therefore if the trajectory described in episode i is not optimal, in the worst case the maximum n-step return, found by (5) will be the immediate reward plus the discounted value in the following state, which is itself an off-policy update rule. Figure 2 presents the Q-Batch backup diagram.

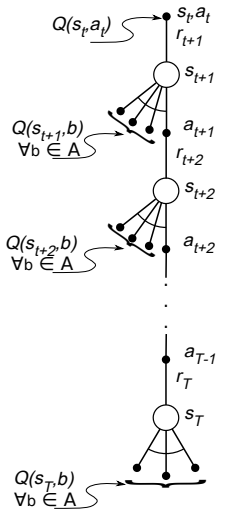

Fig. 2. Q-Batch backup diagram.

While Q-Batch appears to be an improvement over the application of Q-Learning like update rules in Batch RL, it also carries an increase in computational costs. One can clearly conclude that the complexity of (2) is constant, $O(1)$, with respect to the size of the trajectory. On the other hand, by implementing the sum of the rewards using Dynamic Programming, the complexity of (5) is linear, O(T). One could argue that, since the application of Q-Batch is offline during the Pattern Generation module of a Fitted Iteration, such complexity is not particularly negative. However, as stated before, the number of collected transitions can grow exponentially, which will harshly degrade the system with any increase of complexity.

A solution to the increasing size of the data set $\mathcal{F}$, is the application of sampling methods [2, 6]. These methods are able to sample transitions from $\mathcal{F}$ into a representative distribution of the state space, consequently reducing the number of transitions to be processed, resulting in shorter training times. However such methods are based on heuristics, and it is currently unclear which heuristic is best suited to yield a sub-set of

$\mathcal{F}$ able to generate policies with a similar performance to the policies trained with the complete set $\mathcal{F}$.

Luckily, with some modifications, the complexity of (5) can be improved. Initially, the maximum n-step return can be rewritten in the following recursive form:

$$\max_k R_t^k = \max(R_t^1, r_{t+1} + \gamma \max_{k'} R_{t+1}^{k'}) \tag{6}$$

Combining (5) with (6), yields:

$$\begin{aligned} \bar{Q}(s_{i,j}, a_{i,j}) &= \max_k R_t^k \\ &= \max(R_t^1, r_{t+1} + \gamma \max_{k'} R_{t+1}^{k'}) \\ &= \max(R_t^1, r_{t+1} + \gamma \bar{Q}(s_{i,j+1}, a_{i,j+1})) \\ &= r_{t+1} + \gamma \max(\max_b \hat{Q}(s_{i,j+1}, b), \bar{Q}(s_{i,j+1}, a_{i,j+1})) \end{aligned} \tag{7}$$

Analysing (7), we observe that the target Q-value depends on the next immediate reward, the approximated maximum Q-value in the following state, and the target Q-value of the following transition. Considering that all backups are performed synchronously, it is possible to apply Q-Batch in the *inverse* order for every episode in $\mathcal{F}$, thus achieving a constant complexity, comparable to (2).

4 Experiments and Results

To evaluate the performance of Q-Batch, we tested the proposed approach on the Predator-Prey environment [7]. In this simulated environment, predator agents chase prey agents in a discrete toroidal world. While the simulator supports the existence of multiple predators and preys, we will focus the testbed in single agent systems, therefore in all our tests one predator will learn how to chase and capture a single prey. Both the predator and prey can move north(up), south(down), east(right), west(left) as well as stay in the same cell. In every control cycle the predator acts first, and if after the predator action, both agents occupy the same cell, the prey is captured and an episode is finished. Otherwise the prey moves randomly, with a given probability of staying in the same cell, p(stay), and the remaining actions are chosen with probability $\frac{1-p(stay)}{4}$. Please note the following exception, if after the predator moves, the agents are in adjacent cells, the prey will not move onto the cell occupied by the predator. In our tests we used an environment size of 15×15. Figure 3 presents the simulated environment used.

As mentioned before, we focused our implementation on the Neural Fitted Q Iteration framework, which approximates the Q-function with a multilayer perceptron. However the proposed approach is applicable to other function approximators.

The state vector of the predator agent is composed of the relative position to the prey, in cartesian coordinates. To maximize the generalization of the neural network, the actions are coded in the following manner:

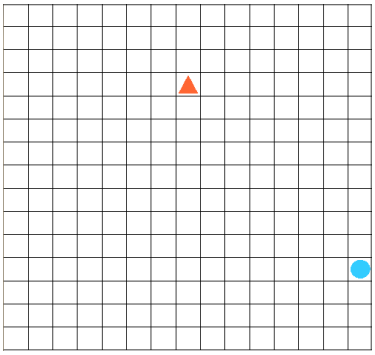

Fig. 3. The Predator-Prey environment. The prey is represented by the triangle and the predator is represented by the circle.

north	south	east	west	stay
1,0,0,0	0,1,0,0	0,0,1,0	0,0,0,1	0,0,0,0

Given this representation of the learning task, the neural network structure used to approximate the Q-function is 6-5-1. All different test conditions ran for 1000 NFQ iterations of interaction with the environment and the neural network was trained with 600 RPROP epochs. A common approach to speed up learning in the NFQ framework is the addition of artificial patterns (also called *hint-to-goal*). For this application, artificial patterns were added with the state (0,0) and target Q-values of 0, thus hinting the predator to this state. The reward function used in all tests is given by (8). In our implementation we formulate the reward signal in terms of costs, therefore a positive signal corresponds to a penalization. Correspondingly, a greedy policy selects the action that minimizes the value of the Q-function.

$$R(s, a, s') = \begin{cases} 0, & s' = (0, 0) \\ 0.01, & \text{else} \end{cases} \tag{8}$$

We strived to test our proposed approach under a number of different conditions. Firstly we separately tested the approach under a fixed and growing batch setting. Additionally we tested the performance with a static prey, p(stay) = 1, and a moving prey with p(stay) = 0.2. In the growing batch problem, we also tested the effect of exploration in the performance of the approach, comparing greedy with ϵ-greedy (ϵ=0.2) policies. In all experiments carried out, no discount factor was used, $\gamma = 1$.

In the fixed batch problem, the resulting set $\mathcal{F}$ of ϵ-greedy learning tasks is not partitioned, but instead used as a whole in the Pattern Generation module in all iterations. Notice that while a policy is tested no more data is collected. This resembles a Learning from Demonstration application. A final test was devised where a clearly sub-optimal policy sampled trajectories of the environment. This policy repeats the same action along the entire trajectory until the predator captures or the agent revisits the starting state. The set $\mathcal{F}$ is built by sampling trajectories starting in all states of the environment with all possible actions, visiting the entire state-action space.

To evaluate the performance of an update-rule we measured the asymptotic performance, which is the sum of the costs obtained in an episode after the agent has achieved a near-optimal policy, the number of NFQ iterations as well as the interaction time until the asymptotic performance was achieved, similarly to the evaluation conducted in [4].

For comparison purposes, in addition to Q-Batch, we tested the batch version of Q-Learning, (2), and a batch version of Watkins' Q(λ), with $\lambda = 0.1$. Each test configuration was repeated 10 times and averages formed. Table 1 presents the obtained results for the tests described above.

Table 1. Obtained results in the different test configurations.

Batch	Prey Policy	Agent Policy	Learning Method	Asymptotic Performance	NFQ iterations to AP	Interaction Time to AP (s)
Growing Batch	P(Stay) = 1	Greedy	Q-Learning	0.076 ± 0.0052	177.9 ± 31.12	39.55 ± 5.67
			Q(λ)	0.086 ± 0.0029	129.7 ± 26.77	31.70 ± 6.50
			Q-Batch	$\mathbf{0.076 \pm 0.0052}$	$\mathbf{130.0 \pm 31.83}$	$\mathbf{26.65 \pm 7.27}$
		ϵ-Greedy	Q-Learning	0.080 ± 0.0034	160.6 ± 43.78	37.56 ± 9.09
			Q(λ)	0.080 ± 0.0051	134.0 ± 29.32	32.00 ± 6.71
			Q-Batch	$\mathbf{0.076 \pm 0.0034}$	$\mathbf{141.6 \pm 34.26}$	$\mathbf{31.90 \pm 6.75}$
	P(Stay) = 0.2	Greedy	Q-Learning	0.096 ± 0.0052	95.1 ± 16.04	21.11 ± 4.20
			Q(λ)	0.095 ± 0.0054	104.6 ± 19.48	26.33 ± 5.07
			Q-Batch	$\mathbf{0.089 \pm 0.0020}$	$\mathbf{105.4 \pm 19.29}$	$\mathbf{25.65 \pm 4.31}$
		ϵ-Greedy	Q-Learning	0.098 ± 0.0068	104.0 ± 19.94	26.37 ± 4.75
			Q(λ)	0.104 ± 0.0068	96.7 ± 27.08	25.81 ± 7.83
			Q-Batch	$\mathbf{0.096 \pm 0.0034}$	$\mathbf{98.4 \pm 22.26}$	$\mathbf{24.70 \pm 5.53}$
Fixed ϵ-greedy sampled Batch	P(stay) = 1	Greedy	Q-Learning	0.083 ± 0.0077	72.6 ± 31.28	14.87 ± 6.80
			Q(λ)	0.080 ± 0.0040	71.3 ± 21.82	14.51 ± 4.93
			Q-Batch	$\mathbf{0.078 \pm 0.0063}$	$\mathbf{29.7 \pm 17.98}$	$\mathbf{5.21 \pm 3.16}$
	P(stay) = 0.2	Greedy	Q-Learning	0.123 ± 0.0080	22.0 ± 5.62	5.67 ± 1.46
			Q(λ)	0.123 ± 0.0100	35.67 ± 16.45	9.39 ± 4.58
			Q-Batch	$\mathbf{0.091 \pm 0.0041}$	$\mathbf{9.7 \pm 4.35}$	$\mathbf{1.94 \pm 0.93}$
Fixed sub-optimal Batch	P(stay) = 1	Greedy	Q-Learning	0.073 ± 0.0043	63.5 ± 10.82	12.11 ± 2.59
			Q(λ)	0.068 ± 0.0025	61.7 ± 3.77	11.89 ± 1.06
			Q-Batch	$\mathbf{0.075 \pm 0.0040}$	$\mathbf{55.1 \pm 16.91}$	$\mathbf{10.12 \pm 3.30}$

5 Discussion

The analysis of the Table 1 shows that Q-Batch achieves a comparable or better performance compared to the other tested update-rules, especially when compared to Q-Learning. While in some tests the asymptotic performance achieved by both methods is similar, Q-Batch requires less NFQ iterations and interaction time to achieve the same level of performance. The exception to this is the last test devised, where the data set $\mathcal{F}$ is fixed and formed by trajectories sampled with a sub-optimal policy. This indicates, as was expected, that Q-Batch is best applied in the presence of informed policies. These policies generate trajectories leading to desirable states, where Q-Batch can quickly propagate rewards to the initial states of the trajectories. This makes Q-Batch specially suited for Learning from Demonstrarion applications.

We also observe that, in a growing batch, simple exploration strategies do not yield better results. However, given the argument presented previously, one can expect Q-Batch to perform better when combined with directed exploration strategies [5] or heuristics to speed up learning [1]. Moreover, since the developed update-rule is off-policy, it should not be negatively affected by the employed exploration strategy. This idea is greatly reinforced by the results obtained in the tests with a fixed batch with trajectories sampled from an ϵ-greedy policy. Since $\mathcal{F}$ contains optimal trajectories from the early stages of learning, we can clearly observe that rewards are propagated backwards along the trajectory much faster than using Q-Learning. Figure 4 presents the averaged results over ten repeated tests ilustrating the difference in the performance of the tested methods with a fixed batch with trajectories sampled from an ϵ-greedy policy, against a static prey.

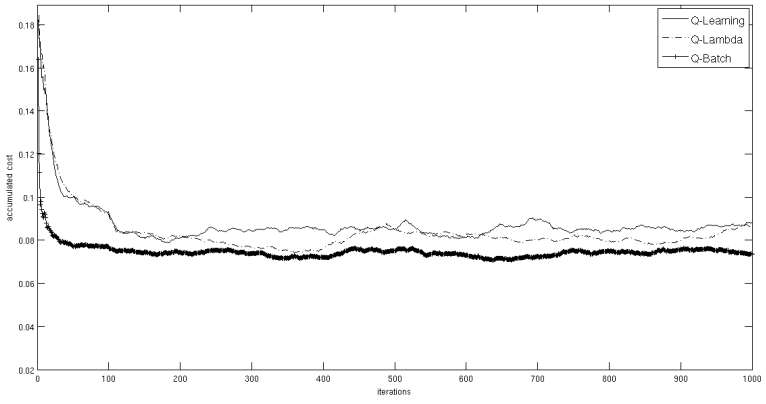

Fig. 4. The performance of the tested methods over time, with a fixed batch composed of ϵ-greedy trajectories, against a moving prey.

While in the fixed batch tests, the entire data set $\mathcal{F}$ was used, which can result in many repeated patterns, the proposed approach can be combined with sampling methodologies to select a more concise and representative set of $\mathcal{F}$. To achieve this goal, sampling methods should sample trajectories, or even partial trajectories, which contain interesting features of the learning task, to take full advantage of Q-Batch.

6 Conclusions

This paper presented a novel update rule based on n-step returns, particularly well suited for Batch RL. It presents an improvement over update-rules similar to Q-learning, which use immediate information, ignoring the fact that in a Batch RL scenario there is valuable non-immediate information available. This paper also presents a strategy to reduce the computational complexity of the update-rule minimizing the execution times. The proposed approach focuses on the Pattern Generation module of the Fitted Q Iteration framework, allowing the application of different function approximators as well as different batch supervised learning algorithms. Additionally since the developed update-rule is off-policy, different exploration techniques can be applied. The sole requirement to apply the proposed approach is to able to sample trajectories from the environment, instead of transitions (s, a, s'), which offer no guarantee to form connected trajectories. Additionally we propose a reorganization of the data set that stores the collected experiences, based on episodes and their corresponding trajectories, which is more compact than current used structures.

While tested on a simulated environment, the initial results are promising. Future work will involve the application of Q-Batch in more complex and real-world learning tasks. The authors envision such applications in the context of physical robots, in learning tasks ranging from robotic soccer to service robotics environments.

Acknowledgments

The authors would like to thank Dr. Martin Riedmiller for his availability to clarify essential concepts of Batch Reinforcement Learning and Neural Fitted Q Iteration implementation.

This research is funded by FEDER through the Operational Program Competitiveness Factors - COMPETE, by National Funds through FCT - Foundation for Science and Technology in the context of the project FCOMP-01-0124-FEDER-022682 (FCT reference PEst-C/EEI/UI0127/2011) and project Cloud Thinking (funded by the QREN Mais Centro program, ref. CENTRO-07-ST24-FEDER-002031).

References

1. Bianchi, R.A.C., Ribeiro, C.H.C., Costa, A.H.R.: Accelerating autonomous learning by using heuristic selection of actions. Journal of Heuristics 14(2), 135–168 (2008)
2. Ernst, D.: Selecting concise sets of samples for a reinforcement learning agent. In: Proceedings of the 3rd International Conference on Computational Intelligence, Robotics and Autonomous Systems. Singapore (10–14 December 2005)

3. Ernst, D., Geurts, P., Wehenkel, L.: Tree-based batch mode reinforcement learning. Journal of Machine Learning 6, 503–556 (2005), http://orbi.ulg.ac.be/handle/2268/9360
4. Gabel, T., Lutz, C., Riedmiller, M.: Improved neural fitted Q iteration applied to a novel computer gaming and learning benchmark. In: Proceedings of the IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning. pp. 279–286. IEEE Press, Paris (Apr 2011)
5. Hester, T., Stone, P.: Real Time Targeted Exploration in Large Domains. In: Proceedings of the 9th International Conference on Development and Learning (ICDL). Ann Arbor, Michigan (Aug 2010)
6. Kietzmann, T.C., Riedmiller, M.: The Neuro Slot Car Racer: Reinforcement Learning in a Real World Setting. In: Proceedings of the International Conference on Machine Learning Applications. Springer, Miami, Florida (2009)
7. Kok, J.R., Vlassis, N.: The pursuit domain package. Tech. rep., Informatics Institute, University of Amsterdam, Amsterdam (2003)
8. Lange, S., Gabel, T., Riedmiller, M.: Batch Reinforcement Learning. In: Wiering, M., van Otterlo, M. (eds.) Reinforcement Learning: State of the Art, chap. 2, pp. 45–74. Springer (2012)
9. Lin, L.J.: Self-improving reactive agents based on reinforcement learning, planning and teaching. Machine Learning 8(3-4), 293–321 (May 1992)
10. Ng, A.Y., Russell, S.J.: Algorithms for Inverse Reinforcement Learning. In: Langley, P. (ed.) Proceedings of the Seventeenth International Conference on Machine Learning (ICML). pp. 663–670. Morgan Kaufmann, Stanford, California (2000)
11. Precup, D., Sutton, R.S., Singh, S.P.: Eligibility traces for off-policy policy evaluation. In: Langley, P. (ed.) Proceedings of the Seventeenth International Conference on Machine Learning. pp. 759–766. Morgan Kaufmann (2000)
12. Riedmiller, M.: Neural fitted Q iteration–first experiences with a data efficient neural reinforcement learning method. In: Gama, J.a., Camacho, R., Brazdil, P., Jorge, A., Torgo, L. (eds.) Proceedings of the European Conference on Machine Learning (ECML). Lecture Notes in Computer Science, vol. 3720, pp. 317–328. Springer, Porto (2005)
13. Riedmiller, M., Braun, H.: A direct adaptive method for faster backropagation learning: the RPROP algorithm. In: Ruspini, H. (ed.) Proceedings of the IEEE International Conference on Neural Networks. pp. 586–591. San Francisco, CA (1993)
14. Riedmiller, M., Gabel, T., Hafner, R., Lange, S.: Reinforcement learning for robot soccer. Autonomous Robots 27(1), 55–73 (May 2009)
15. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. MIT Press, Cambridge (MA) (1998)
16. Szepesvári, C.: Algorithms for Reinforcement Learning. Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool (2010)
17. Tesauro, G., Galperin, G.R.: On-line policy improvement using Monte Carlo search. In: Neural information processing systems (NIPS). pp. 206–221. Denver (1996)
18. Watkins, C.J.C.H.: Learning from Delayed Rewards. Ph.D. thesis, University of Cambridge (1989)
19. Wiering, M.A., van Otterlo, M. (eds.): Reinforcement Learning: State of the Art, Adaptation, Learning, and Optimization, vol. 12. Springer (2012)