Ex$\langle a \rangle$: An Effective Algorithm for Continuous Actions Reinforcement Learning Problems

José Antonio Martín H.
Sistemas Informáticos y Computación
Universidad Complutense de Madrid
Madrid, España 28040
Email: jamartinh@fdi.ucm.es

Javier de Lope
Sistemas Inteligentes Aplicados
Universidad Politécnica de Madrid
Madrid, España 28031
Email: javier.delope@upm.es

***Abstract*—In this paper the Ex$\langle a \rangle$ Reinforcement Learning algorithm is presented. This algorithm is designed to deal with problems where the use of continuous actions have clear advantages over the use of fine grained discrete actions. This new algorithm is derived from a baseline discrete actions algorithm implemented within a kind of k-nearest neighbors approach in order to produce a probabilistic representation of the input signal to construct robust state descriptions based on a collection (knn) of receptive field units and a probability distribution vector $p(knn)$ over the knn collection. The baseline continuous-space-discrete-actions kNN-TD(λ) algorithm introduces probability traces as the natural adaptation of eligibility traces in the probabilistic context. Later the Ex$\langle a \rangle(\lambda)$ algorithm is described as an extension of the baseline algorithms. Finally experimental results are presented for two (not easy) problems such as the Cart-Pole and Helicopter Hovering.**

I. Introduction

Reinforcement Learning (RL) [1], [2] is a paradigm of Machine Learning (ML) in which rewards and punishments guide the learning process. One of the central ideas of RL is learning by a "direct-online" interaction with the environment. In RL there is an Agent (learner) which acts autonomously and receives a scalar reward signal which is used to evaluate the consequences of its actions. The framework of RL is designed to guide the learner in maximizing the expected reward that it gathers from its environment in the long run.

Let us start with a classical temporal difference (TD) learning rule [3, Q-Learning]:

$$Q(s,a)_{t+1} = Q(s,a)_t + \alpha[x - Q(s,a)_t], \quad (1)$$

where $x = r_{t+1} + \gamma \max_a Q(s_{t+1}, a)_t$.

This rule combines two strong ideas which are central to RL: (i) the notion that knowledge is, in essence, made of expectations and (ii) the temporal-difference notion [4] which is a general method of prediction very well suited to direct online (incremental[1]) learning from immediate experience. The basic rule (1) can be derived directly from the formula of the expected value of a discrete random variable. We know that the expected value μ of a discrete random variable (x) with possible values $x_1, x_2 \ldots x_n$ and probabilities represented by the function $p(x_i)$ can be calculated as:

$$\mu = \sum_{i=1}^{n} x_i \, p(x_i), \quad (2)$$

thus for a discrete variable with only two possible values the formula becomes:

$$\mu = (1-\alpha)a \; + \; b\alpha \;\; , \quad (3)$$

where α is the probability of the second value (b). Then taking a as a previously stored expected value μ and b as a new observation x, we have:

$$\mu = (1-\alpha)\mu \; + \; x\alpha \;\; , \quad (4)$$

and doing some operations we get:

$$\begin{aligned} \mu &= (1-\alpha)\mu \; + \; x\alpha \\ &= \mu - \alpha\mu + \alpha x \\ &= \mu + \alpha[-\mu + x] \\ &= \mu + \alpha[x - \mu] \end{aligned} \quad (5)$$

Then we can see that the parameter α, usually described as the learning rate, express the probability that the random variable μ get the value of the observation x and that replacing μ by $Q(s,a)_t$ and then replacing x by the expected cumulative future reward (usually the learning target):

$$r_{t+1} + \gamma \max_a Q(s_{t+1}, a)_t \quad (6)$$

we finally get again the Q-Learning update rule (1).

In this paper we propose the Ex$\langle a \rangle$ Reinforcement Learning algorithm. This algorithm is designed to deal with problems where the use of continuous actions have clear advantages over the use of fine grained discrete actions; some of the reasons of such advantage may include tractability-feasibility of the action space, learning speed and the accuracy of the approximated suboptimal actions with respect to the optimal actions. This algorithm is derived from a baseline discrete actions algorithm implemented within a kind of k-nearest neighbors approach in order to produce a probabilistic representation of the input signal to construct robust state descriptions based on

[1]—the steps in a sequence should be evaluated and adjusted according to their immediate or near immediate successors, rather than according to the final outcome [4].

a collection (knn) of receptive field units and a probability distribution vector $p(knn)$ over the knn collection.

The k-nearest neighbors (k-NN) technique [5]–[7] is a method for classifying objects based on closest training examples in the feature space. The determinant parameters in k-NN techniques are the number k which determines how many units are to be considered as the neighbors and the distance function, generally the euclidian distance is used.

There has been other uses of the k-NN rule, e.g. in Dynamic Programming (DP) [8], as a function approximator with successful results. Also, we must mention that the there is a close relation between the kind of probabilistic state-space representation, and the derived learning rules that we propose, and the so called "Locally Weighted Learning" (LWL) methods surveyed by Atkeson and Moore [9].

The baseline algorithms are based on some previous works [10], [11] proposing the use of k-NN as a perception scheme in RL. This baseline continuous-space-discrete-actions kNN-TD(λ) algorithm introduces probability traces as the natural adaptation of eligibility traces in the probabilistic context.

Later the Ex$\langle a \rangle(\lambda)$ algorithm is described as an extension of the baseline algorithms. Finally experimental results are presented for two hard control problems: the Cart-Pole with continuous actions and the problem of learning Helicopter Hovering.

II. Baseline kNN-TD RL Algorithm

The algorithm description will follow the classical interaction cycle in RL.

A. Perception

The first task is to determine the k-nearest neighbors set (knn) of the current observation s, where for each defined neighbor of the observation will be an associated action-value predictor $Q(i, a)$ and also a weight w_i which is calculated as follows:

$$w_i = \frac{1}{1 + {d_i}^2} \;\; \forall i \in [1, ..., k] \tag{7}$$

where ${d_i}^2$ is a distance function and then for each time step will be k active classifiers whose weights are inverse to the distance from the current observation (s).

The second one is to obtain the probability distribution $p(knn)$ over knn. We must note that each element in the knn set will provide information on the current observed state s. This information could be in form of an associated value, i.e. an associated action-value predictor $Q(i, a)$, or indeed implicitly defined just by the meaning of being x_i a particular neighbor of s. In any respect, the probabilities are implied by the current weights vector (w). This vector must be normalized in order to express a probability distribution $p(knn)$, thus the probabilities for each element in the knn set are calculated by the equation (8).

$$p(i) = \frac{w_i}{\sum w_i} \;\; \forall i \in knn, \tag{8}$$

Then we can state that a perceptual representation, or simply a perception, is uniquely and completely defined by two components (knn, p) obtained from a currently observed state (s):

1) A set knn which contains the k-nearest neighbors of the observation s.
2) A probability distribution vector p of each element in the knn set.

B. Action Selection

Once obtained a perceptual representation (knn, p) of an observed state (s) the Agent should emit an action (a) to the Environment. For doing so, the Agent has to apply some "rational" action selection mechanism based on its expectations about the consequences of such action (a). These consequences are represented by expectations of future rewards. The discussion about exploration vs. exploitation and the corresponding methods like greedy, ϵ-greedy or soft-max methods can be found in the literature [2]. We will center the attention on calculating the expectations for each action by means of a collective prediction process which can then be used directly by any classical action selection mechanism.

The objective of the prediction process consist of calculating the expected value of the predictions for each available action a_i in the observed state s. Since this process involves many different predictions, one for each element in the knn set, the procedure gets reduced to estimate an expected value, that is:

$$\mu = \sum_{i=1}^{n} x_i \, p(x_i) \; , \tag{9}$$

In this way, the expected value of the learning target for a given action a is estimated by (10).

$$\langle \mathcal{Q}(knn, a) \rangle = \sum_{i=1}^{knn} Q(i, a) p(i) \; , \tag{10}$$

hence we can see clearly that $p(i)$ acquire the meaning of the probability $P(\mathcal{Q}(knn, a) = Q(i, a)|s)$ that the $\mathcal{Q}(knn, a)$ takes the value $Q(i, a)$ given the observation (s) of the environment. Then the action selection mechanism can directly derive a greedy policy from the expectations for each perceptual representation as shown in (11).

$$a_{greedy} = argmax_a \; \mathcal{Q}(knn, a) \tag{11}$$

This can be understood, in k-NN terminology, as that the Agent will select the most "frequent" recommended action in the neighborhood of the observed state s or, in bayesian terms, the action for which there is more evidence of a high future reward.

C. Learning

The learning process involves the use of the accumulated experience and thus it will rely on past and current experienced observations, actions performed and received reward (s, a, s' and r). Given that the Agent have experienced the state s and performed an action a and then observed a new state s' and received a reward r there are two perceptual representations, once for each state respectively. These perceptual representations are the tuples (knn, w) for observation s and (knn', w') for observation s'.

Following the TD-Learning scheme we need an estimation of the future cumulative reward. This estimation is generally obtained by the action-value of the current state s', i.e. $\max_a \mathcal{Q}(knn', a)$ in off-policy learning methods and by $\mathcal{Q}(knn', a')$ in on-policy learning.

For whichever (off/on policy) method be followed, an expected (predicted) value is calculated:

$$\langle \mathcal{Q}(knn', a)\rangle = \sum_{i=1}^{knn'} Q(i, a)p'(i) \ , \tag{12}$$

where knn' is the set of the k-nearest neighbors of s', $p'(i)$ are the probabilities of each neighbor in the set knn'. In this way, the TD-error (δ) can be calculated as shown in (13) or (14) (on/off policy respectively).

$$\delta = r + \gamma \mathcal{Q}(knn', a') - \mathcal{Q}(knn, a) \tag{13}$$

$$\delta = r + \gamma \max_a \mathcal{Q}(knn', a) - \mathcal{Q}(knn, a) \tag{14}$$

Thus we can simply update the prediction of each point in the knn set by applying the update rule (15).

$$Q(i, a)_{t+1} = Q(i, a)_t + \alpha\delta p(i) \ \ \forall i \in knn, \tag{15}$$

D. Probability Traces $e(i) \leftarrow p(i)$: kNN-TD(λ)

The use of eligibility traces [2, p.161] is a useful technique for improving the performance of RL algorithms. Eligibility traces are controlled by a parameter λ that takes values in the range $[0, 1]$. This parameter makes that the methods based on eligibility traces behaves between two extremes: one-step TD-Learning ($\lambda = 0$) and Monte Carlo methods ($\lambda = 1$) [2, p.163]. A pseudo-code of the kNN-TD(λ) RL algorithm is presented in figure 1.

In order to implement eligibility traces it is necessary to define a matrix e in which the traces for each classifier will be stored. A consequence of the use of eligibility traces is that the update rule should be applied to all the classifiers and not only to the knn set. Thus the update rule for the case of eligibility traces becomes:

$$Q_{t+1} = Q_t + \alpha\delta e \tag{16}$$

In general, replacing traces are more stable and perform better than cumulating traces [2], [12]. Replacing traces in the kNN-TD(λ) algorithm are defined in a simple way by establishing the value of the trace for each classifier as its probability $p(i)$ for the performed action (a) while for actions different from a the traces reset to 0. This form of trace resetting is a specialization of replacing traces that exhibits better performance than classical replacing traces schemes [2, p.188]. The expression (17) shows the way in which this kind of traces are defined for the kNN-TD(λ) algorithm.

$$e(knn, j) = \begin{cases} p(knn) & j = a \\ 0 & j \neq a \end{cases} \tag{17}$$

Fig. 1. The kNN-TD(λ) reinforcement learning control algorithm.

```
1: Initialize the space of classifiers $cl$
2: Initialize $Q(cl, a)$ arbitrarily and $e_i(cl, a) \leftarrow 0$
3: **repeat** {for each episode}
4:    Initialize $s$
5:    $knn \leftarrow k$-nearest neighbors of $s$
6:    $p(knn) \leftarrow$ probabilities of each $cl \in knn$
7:    $\mathcal{Q}(knn, a) = Q(knn, a) \cdot p(knn)$ for all $a$
8:    Chose $a$ from $s$ according to $V(a)$
9:    **repeat** {for each step of episode}
10:      Take action $a$, observe $r$, $s'$
11:      $knn' \leftarrow k$-nearest neighbors of $s'$
12:      $p(knn') \leftarrow$ probabilities of each $cl \in knn'$
13:      $\mathcal{Q}(knn', a') = Q(knn', a) \cdot p(knn')$ for all $a$
14:      Chose $a'$ from $s'$ according to $V'(a)$
15:      Update Q and e:
16:         $e(knn, \cdots) \leftarrow 0$ {optional}
17:         $e(knn, a) \leftarrow p(knn)$
18:         $\delta \leftarrow r + \gamma\mathcal{Q}(knn', a') - \mathcal{Q}(knn, a)$
19:         $Q \leftarrow Q + \alpha\delta e$
20:         $e \leftarrow \gamma\lambda e$
21:      $a \leftarrow a'$, $s \leftarrow s'$, $knn \leftarrow knn'$
22:    **until** $s$ is terminal
23: **until** learning ends
```

where knn is the set of k-nearest neighbors of the observation s, $p(knn)$ are its corresponding probabilities and a is the last performed action. The traces always decay following the expression (18).

$$e_{t+1} = \gamma\lambda e_t \tag{18}$$

As we can see, the eligibility traces scheme is replaced by a probability traces scheme given the fact that the stored values are just probabilities and not the value 1 as is used in classical eligibility traces theory.

III. Continuous Actions: the Ex$\langle a\rangle(\lambda)$ Algorithm

A pseudo-code of the Ex$\langle a\rangle(\lambda)$ RL algorithm is presented in algorithm 2.

As for the discrete actions method, kNN-TD(λ), an action list needs to be defined. Nevertheless the form and use of this action list will differ notoriously. When the actions are discrete these can have any type of nonnumeric interpretation, i.e. they can be symbols that indicate a label of an arbitrary action and these labels stored in the action list can be independent between them, whereas for continuous actions the action list is a granulation of a continuous space represented by points in a given interval, for instance:

$$\mathcal{A} = \{-1.0, -0.5, 0.25, 0.0, 0.25, 0.5, 1.0\}$$

In this case the actions are related to each other and are interdependent. As in the version for discrete actions, each classifier will maintain an expectation $Q(a_i)$ for each action $a_i \in \mathcal{A}$.

Fig. 2. The Ex$\langle a \rangle(\lambda)$ reinforcement learning control algorithm for continuous state and action spaces.

```
1:  Initialize $\mathcal{A} \leftarrow linspace(lower, upper, n)$
2:  Initialize $\mathcal{L} \leftarrow 1 \dots n$
3:  Initialize the space of classifiers $cl$
4:  Initialize $Q(cl, \mathcal{L})$ arbitrarily and $e(cl, \mathcal{L}) \leftarrow 0$
5:  **repeat** {for each episode}
6:    Initialize $s$
7:    $knn \leftarrow k$-nearest neighbors of $s$
8:    $p(knn) \leftarrow$ probabilities of each $cl \in knn$
9:    $I \leftarrow \underset{\mathcal{L}}{argmax}\ Q(knn, \mathcal{L})$
10:   $a \leftarrow \mathcal{A}[I] \cdot p(knn)$
11:   $\mathcal{Q}(knn, I) = \max_{\mathcal{L}} Q(knn, \mathcal{L}) \cdot p(knn)$
12:   **repeat** {for each step of episode}
13:     Take action $a$, observe $r$ and $s'$
14:     $knn' \leftarrow k$-nearest neighbors of $s'$
15:     $p(knn') \leftarrow$ probabilities of each $cl \in knn'$
16:     $I' \leftarrow \underset{\mathcal{L}}{argmax}\ Q(knn', \mathcal{L})$
17:     $a' \leftarrow \mathcal{A}[I'] \cdot p(knn')$
18:     $\mathcal{Q}(knn', I') = \max_{\mathcal{L}} Q(knn', \mathcal{L}) \cdot p(knn')$
19:     Update Q and e:
20:       $e(knn, \cdots) \leftarrow 0$ {optional}
21:       $e(knn, I) \leftarrow p(knn)$
22:       $\delta \leftarrow r + \gamma \mathcal{Q}(knn', I') - \mathcal{Q}(knn, I)$
23:       $Q \leftarrow Q + \alpha \delta e$
24:       $e \leftarrow \gamma \lambda e$
25:     $a \leftarrow a'$, $s \leftarrow s'$, $knn \leftarrow knn'$, $I \leftarrow I'$
26:   **until** $s$ is terminal
27: **until** learning ends
```

The main modification to the method is made in the action selection mechanism and in the calculation of the value of the best action. That is, it is necessary to modify the equations (10) and (11). The action selection mechanism is now independent of the mechanism to calculate the value $\mathcal{Q}(knn', a')$ (the best prediction of future reward).

First, the action selection mechanism selects the recommended (best expected future reward) actions of each classifier in the knn set and stores them in the optimal actions list A^*. Thus given an action list $\mathcal{A}$ the best actions are selected according to each classifier:

$$I = \underset{a}{argmax}\ Q(i, a) \ \ \forall i \in [knn] \ \text{ and } \forall a \in \mathcal{A}\ , \tag{19}$$

where I is an index list over the action list $\mathcal{A}$ that will contain the indices of the values of $\mathcal{A}$ where the values of $Q(i, a)$ are maximal.

This index list I is then translated to its values within the list $\mathcal{A}$ to obtain the list A^* of the optimal values of action a recommended by each classifier:

$$A^* = \mathcal{A}[I] \tag{20}$$

Of this form the list A^* could contain e.g. the values $A^* = \{0.25, 0.0, 0.25, 0.25\}$ for an index list $I = \{4, 3, 4, 4\}$ and $k = 4$.

Having the actions A^*, recommended by the classifiers in the knn set, and its respective probabilities given by $p(i)$ the *expected optimal action* is calculated according to the classical formulation of the expected value of a variable with discrete values. Thus the (collectively recommended) action to be performed by the agent can be obtained calculated in the following way:

$$\langle a \rangle = \sum A^*(i) p(i) \ \ \forall i \in knn, \tag{21}$$

where $\langle a \rangle$ is the expected value of the continuous optimal action to be executed, $A^*(i)$ is the best recommended action by the classifier i and $p(i)$ is the probability $P(a = A^*(i)|s)$ that a takes the value $A^*(i)$ given the state of the environment s.

Finally, as in the kNN-TD algorithm, the update of the predictions of each classifier is based on the use of *probability traces* and the TD-error (δ) to estimate the expected value of the future reward. Probability traces are defined in this case by the following equation:

$$e(knn, I) = \begin{cases} p(knn) & \text{for all } a \in I \\ 0 & \text{for actions not in } I \end{cases}, \tag{22}$$

where $p(knn)$ are the probabilities of each classifier in the knn and I is the index list over the actions in $\mathcal{A}$ with maximal action-value. Then the update rule (23) can be applied to each classifier:

$$Q_{t+1} = Q_t + \alpha \delta e, \tag{23}$$

the TD-error (δ) can be calculated in the following form:

$$\delta = r + \gamma \mathcal{Q}(knn', I') - \mathcal{Q}(knn, I), \tag{24}$$

and the action-value $\mathcal{Q}(knn, I)$ is calculated according to the expression:

$$\langle \mathcal{Q}(knn, I) \rangle = \sum_{i=1}^{knn} \max_{\mathcal{L}} Q(i, \mathcal{L}) p(i), \tag{25}$$

where $\mathcal{L}$ is the list $1 \dots n = |\mathcal{A}|$.

IV. Experimental Results

The family of algorithms kNN-TD has been intensively tested on various kinds of problems with very robust results. These problems range from the classical Mountain-Car, Acrobot, Cart-Pole to some harder control problems such as the control of a simulated Helicopter.

A. Solving the Cart-Pole (Pole-Balancing) Problem with Continuous Actions

The pole-balancing is a control benchmark historically used in engineering as an example of unstable, multiple-output and dynamic systems that appear in many balancing situations and it has been used to show modern and classical control-engineering techniques [13]. It involves a pole affixed to a cart via a joint which allows movement along a single axis. The cart is able to move along a track of fixed length.

Figure 3 shows the learning convergence graph for the discrete actions method kNN-TD(λ) in the Cart-Pole problem

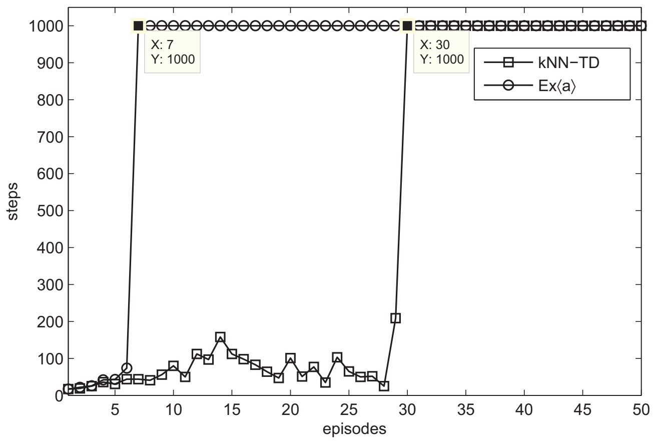

Fig. 3. Learning convergence curve for the discrete actions method kNN-TD (k=4; λ=0.95 ; ϵ=0; α=0.3) and learning convergence curve for the continuous actions method Ex$\langle a\rangle$ (k=4; λ=0.9 ; ϵ=0; α=0.3) in the Cart-Pole problem.

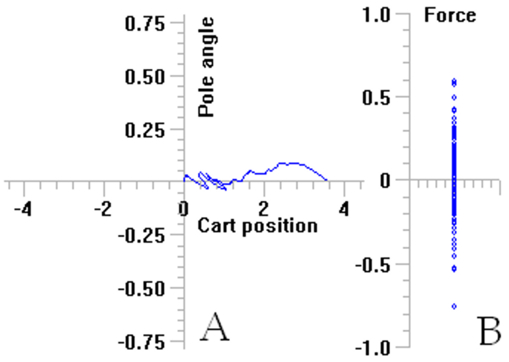

Fig. 4. Example episode of the Cart-Pole problem where the route in the state space in shown in (A) and the continuous performed actions are shown in (B). The value of the parameter k in this example was $k = 8$ and the base action list was $\mathcal{A} = linspace(-1, 1, 11)$

and the learning convergence graph for the continuous actions method Ex$\langle a\rangle(\lambda)$. As one can see, the continuous actions algorithm outperforms the discrete version by learning a near optimal control policy in only 7 episodes while the discrete actions method takes at least 30 episodes in learning a near optimal policy. Also, when taking the softness of the control actions as a key factor, the quality of the control policies of the Ex$\langle a\rangle(\lambda)$ algorithm can be considered better.

Finally in figure 4 a graph of an example episode where the route in the state space (A) and the continuous performed actions (B) can be seen. As it is observed in this example the actions that the algorithm performs are continuous and they almost completely cover the continuous action space.

B. Learning to Hover an Helicopter

Here we present a method to obtain a near optimal RL controller for autonomous helicopter hovering. The Helicopter simulator was created at Stanford University [14] and simulates an XCell Tempest helicopter (figure 5) in flight regime close to hover.

Fig. 5. A picture of the real XCell Tempest RC Helicopter from [15].

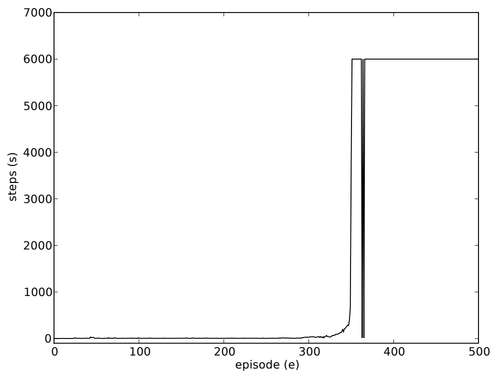

Fig. 6. Learning curve in steps for the Helicopter problem.

The agent's objective is to hover the helicopter during 6000 time steps by manipulating four continuous control actions based on a non-uniform twelve dimensional state space. If the helicopter crashes before 6000 steps then a very large negative regard is given and the episode ends. The learning parameters used for the experiment where $\lambda = 0.5$ and $\mathcal{A} = linspace(-1, 1, 21)$.

The action space is defined as a 4-dimensional continuous valued vector:

a_1:	aileron, longitudinal (front-back) cyclic pitch $[-1, 1]$.
a_2:	elevator, latitudinal (left-right) cyclic pitch $[-1, 1]$.
a_3:	rudder, main rotor collective pitch $[-1, 1]$.
a_4:	coll, tail rotor collective pitch $[-1, 1]$.

The observation space is 12 dimensional continuous valued defined as follows:

u:	forward velocity.
v:	sideways velocity (to the right).
w:	downward velocity.
x_{err}:	helicopter x-coord position $-$ desired x-coord position.
y_{err}:	helicopter y-coord position $-$ desired y-coord position.
z_{err}:	helicopter z-coord position $-$ desired z-coord position.
ϕ:	angular rate around helicopter's x axis.
θ:	angular rate around helicopter's y axis.
ω:	angular rate around helicopter's z axis.
p:	angular velocity helicopter's x axis.
q:	angular velocity around helicopter's y axis.
r:	angular velocity around helicopter's z axis.

Figure 6, 7 and 8 show the results of the experiments on applying the Ex$\langle a\rangle(\lambda)$ to the helicopter hovering problem. As can be observer, about 375 episodes are needed to learn a safe policy that does not crash the helicopter.

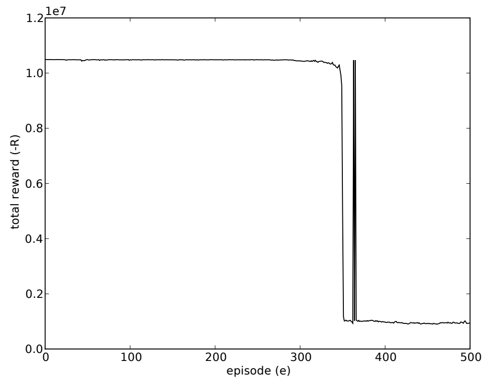

Fig. 7. Learning curve in total reward per episode (less is better) for the Helicopter problem.

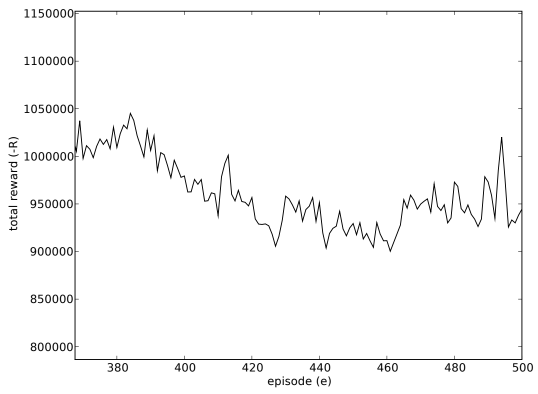

Fig. 8. A zoom in of the final steps of the learning curve in total reward per episode for the Helicopter problem.

The obtained results are only presented for the Ex$\langle a\rangle(\lambda)$ since there was no way of producing any convergence with the kNN-TD(λ) indeed using discrete action vectors significative larger than the used for the Ex$\langle a\rangle(\lambda)$ algorithm. For more information about this problem see: *http://2009.rl-competition.org/helicopter.php*.

V. Conclusions and Further Work

An Effective Reinforcement Learning algorithm designed to deal with problems where the use of continuous actions have clear advantages over the use of fine grained discrete actions have been presented. This continuous actions algorithm Ex$\langle a\rangle(\lambda)$ has been derived from an extension of the baseline kNN-TD(λ) algorithm which has been also presented and described in detail. Experiments have been carried out in order to study the validity and performance of the methods. The experimental results shows that the proposed Ex$\langle a\rangle(\lambda)$ is by far a better choice for continuous action domains than the kNN-TD(λ) which exhibits a state of the art performance in discrete action domains.

Acknowledgments

This work has been partially funded by the Spanish Ministry of Science and Technology, project: DPI2006-15346-C03-02.

References

[1] L. P. Kaelbling, M. L. Littman, and A. W. Moore, “Reinforcement learning: A survey,” *Journal of Artificial Intelligence Research*, vol. 4, pp. 237–285, 1996.

[2] R. Sutton and A. Barto, *Reinforcement Learning, An Introduction*. The MIT press, 1998.

[3] C. J. Watkins and P. Dayan, “Technical note Q-learning,” *Machine Learning*, vol. 8, p. 279, 1992.

[4] R. S. Sutton, “Learning to predict by the method of temporal differences,” *Machine Learning*, vol. 3, pp. 9–44, 1988.

[5] T. M. Cover and P. E. Hart, “Nearest neighbor pattern classification,” *IEEE Transactions on Information Theory*, vol. IT-13, no. 1, pp. 21–7, Jan. 1967.

[6] R. O. Duda and P. E. Hart, *Pattern Classification and Scene Analysis*. Wiley, 1973.

[7] S. A. Dudani, “The distance-weighted k-nearest-neighbor rule,” *IEEE Transactions on Systems, Man and Cybernetics*, vol. SMC-6, 4, pp. 325–327, 1976.

[8] G. J. Gordon, “Stable function approximation in dynamic programming,” in *ICML*, 1995, pp. 261–268.

[9] C. Atkeson, A. Moore, and S. Schaal, “Locally weighted learning,” *AI Review*, vol. 11, pp. 11–73, April 1997.

[10] J. A. Martin H. and J. de Lope, “A k-nn based perception scheme for reinforcement learning,” in *Computer Aided Systems Theory EUROCAST 2007*, ser. Lecture Notes in Computer Science, vol. 4739. Springer Verlag, 2007, pp. 138–145.

[11] J. A. Martin H., J. de Lope Asiaín, and D. Maravall, “The knn-td reinforcement learning algorithm,” in *IWINAC Part I, Methods and Models in Artificial and Natural Computation*, ser. Lecture Notes in Computer Science, vol. 5901. Springer, June 2009, pp. 305–314.

[12] S. P. Singh and R. S. Sutton, “Reinforcement learning with replacing eligibility traces,” *Machine Learning*, vol. 22, no. 1-3, pp. 123–158, 1996.

[13] C. W. Anderson, “Strategy learning in multilayer connectionist representations,” in *Proceedings of the Fourth International Workshop on Machine Learning, Irvine, CA, June 1987*. Morgan Kaufmann, 1987, pp. 103–114.

[14] P. Abbeel, V. Ganapathi, and A. Y. Ng, “Learning vehicular dynamics, with application to modeling helicopters,” in *NIPS*, 2005. [Online]. Available: http://books.nips.cc/papers/files/nips18/NIPS2005_0824.pdf

[15] A. Y. Ng, H. J. Kim, M. I. Jordan, and S. Sastry, “Autonomous helicopter flight via reinforcement learning,” in *NIPS*, S. Thrun, L. K. Saul, and B. Schölkopf, Eds. MIT Press, 2003. [Online]. Available: http://books.nips.cc/papers/files/nips16/NIPS2003_CN07.pdf