Proceedings of the 2007 IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning (ADPRL 2007)

Reinforcement Learning in Continuous Action Spaces

Hado van Hasselt and Marco A. Wiering
Intelligent Systems Group, Department of Information and Computing Sciences, Utrecht University
Padualaan 14, 3508 TB Utrecht, The Netherlands
Telephone: +31 - 30 - 253 2814, Fax: +31 - 30 - 251 3791
Email: {hado,marco}@cs.uu.nl

***Abstract*— Quite some research has been done on Reinforcement Learning in continuous environments, but the research on problems where the actions can also be chosen from a continuous space is much more limited. We present a new class of algorithms named Continuous Actor Critic Learning Automaton (CACLA) that can handle continuous states and actions. The resulting algorithm is straightforward to implement. An experimental comparison is made between this algorithm and other algorithms that can handle continuous action spaces. These experiments show that CACLA performs much better than the other algorithms, especially when it is combined with a Gaussian exploration method.**

I. Introduction

Reinforcement Learning (RL) can be used to make an agent learn to interact with an environment. The goal is to optimize the behavior of the agent in respect to a reward signal that is provided by the environment. The actions of the agent can also affect the environment, complicating the search for the optimal behavior. RL can be used to find solutions for Markov Decision Processes (MDPs). For a detailed introduction in the field of RL, see the book by Sutton and Barto [1].

In conventional RL, only MDPs with finite sets of actions are considered. However, in many real-world applications an a priori discretization of the action space is not very useful. Some parts of the action space may be much more important than others, requiring a very fine grained discretization to reach good solutions. Furthermore it may not be evident at first exactly what these important regions of the action space are. Also, when the action space is discretized, it is hard to generalize from past experiences and learning may be slow when the resulting discrete action space has many elements. To avoid these problems, we will look at algorithms capable of dealing with real continuous action spaces.

We will present the Continuous Actor Critic Learning Automaton (CACLA) algorithm, which has all the characteristics that we think are important for a continuous state and action space RL algorithm. These characteristics are: (1) The ability to find real continuous solutions; (2) Good generalization properties and (3) Fast action selection. Though we will not do so in this paper, CACLA is easily transformed into a batch algorithm with similar properties.

The CACLA algorithm is model-free, which we think is a good property to have since we do not want to assume the agent has an a priori model of the environment and we do not want to wait until the agent has built a model before it starts learning. Also, in some cases finding a good model for a real-world process can be hard, while the optimal behavior is in fact quite simple. Consider a situation where the agent can only choose between two actions, one of which is clearly inferior. A model-based approach would possibly require a lot of exploration just to construct a rough model before choosing the correct action, whereas a model-free approach will find the correct behavior after just a few observations. Furthermore, constructing a good model in continuous state and action spaces can be even harder, because of the added complexity of the need for good generalization and exploration while retaining the information gathered from earlier observations.

This paper will only discuss online algorithms and will therefore not cover batch algorithms for similar problems. This automatically excludes batch algorithms such as Episodic Natural Actor Critic [2] and Neural Fitted Q Iteration [3]. Since CACLA is easily extended to a batch algorithm, in the future it may be interesting to compare a set of batch algorithms including the aforementioned ones to the batch version of CACLA. We also make this distinction since online algorithms have different properties than batch algorithms, such as better performance in dynamic environments and a faster learning slope. Batch algorithms on the other hand can potentially perform better in problems with few observations, for instance when simulations are (computationally) expensive or when dealing with actual physical objects instead of simulations.

In section II we give a short summary of RL. In section III we extend RL to continuous situation and action spaces. CACLA and the algorithms used to compare CACLA to are presented in section IV. The experiments we conducted are described in section V after which the results are given in section VI. Section VII concludes this paper.

II. Reinforcement Learning

RL can be used to solve problems that can be modeled as an MDP. In this section we will show how an agent can learn to solve such problems. An MDP can be viewed as a tuple (S, A, R, T) where:

- S is the set of all states and $s_t \in S$ is the state the agent is in at time t.
- A is the set of all possible actions and $a_t \in A$ is the action the agent performs at time t.

- $R : S \times A \times S \rightarrow \mathbb{R}$ is the reward function that maps a state s_t, an action a_t and the resulting state s_{t+1} into a reward $R(s_t, a_t, s_{t+1})$. This reward is known to the agent when reaching the state s_{t+1}. We use r_t to denote the possibly stochastic reward drawn from a distribution with mean $R(s_t, a_t, s_{t+1})$.
- $T : S \times A \times S \rightarrow [0, 1]$ is the transition function, where $T(s, a, s')$ gives the probability of arriving in state s' when performing action a in state s.

A. Values and Q-Functions

An agent can learn by storing values for each state or for each state-action pair. State values, denoted by $V(s)$, represent the cumulative discounted reward that the agent expects to receive in the future after reaching state s. State-action values, denoted by $Q(s, a)$, represent the cumulative discounted reward it expects to receive after performing action a in state s. The goal for the agent is to learn an action selection policy $\pi : S \times A \rightarrow [0, 1]$ that optimizes the cumulative reward. Here $\pi_t(s, a)$ gives the probability of selecting action a in state s at time t. Formally, starting at time t, we want the agent to optimize the total discounted reward:

$$r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

where $0 \leq \gamma \leq 1$ is a discount factor. We regard the discount factor as part of the algorithm and not of the MDP, since some algorithms perform much better with specific discount factors, while other algorithms require different discount factors for optimal performance. Of course, a different discount factor can mean a different optimal policy, which is why we measure performance in average reward and not in discounted reward.

Let Q^π and V^π denote the Q-function and state value function corresponding to some policy π. We denote the optimal policy by π^* and its corresponding state and state-action values by V^* and Q^*. There is always at least one optimal policy. We then have:

$$\max_a Q^*(s, a) \;=\; V^*(s) \;=\; \max_\pi V^\pi(s)$$
$$\forall s \in S : \pi^* \;=\; \arg\max_\pi V^\pi(s)$$

B. Learning the Values

We know that the value function corresponding to the optimal policy will have the following property:

$$V^*(s) = \max_a \sum_{s'} T(s, a, s') \Big(R(s, a, s') + \gamma V^*(s') \Big) \quad (1)$$

which is called the Bellman optimality equation for V^* [4], [1]. The transition function T is usually not known. The values can then be updated using Temporal Difference (TD) learning [5]:

$$V_{t+1}(s_t) = V_t(s_t) + \alpha_t \delta_t \;, \quad (2)$$

where δ_t is the TD-error, defined as $r_t + \gamma V_t(s_{t+1}) - V_t(s_t)$ and $0 \leq \alpha_t \leq 1$ is a learning rate. It has been proven that when these values are stored in a table, using update (2) will result in convergence of the values to the actual expected returns for a fixed policy [5], [6]. Convergence to the optimal policy has been proven under certain conditions for variants of this update using Q-values instead of state values, such as Q-Learning [7] and SARSA [8], [9]. These Q-values are dependent on states and actions instead of just states.

III. Continuous Spaces

First a short introduction in handling continuous state spaces will be given. This is a relatively well known field, where a lot of work has already been done. Then we will continue with the harder problem of continuous action spaces.

A. Continuous State Spaces

When the state space is continuous, parametrized function approximators (FAs) can be used to store the value of observed states and generalize to unseen states. The update is then performed on the parameters of the FA. We used neural networks, whose weights are then the parameters. Let θ^V denote the parameter vector of an FA. The update rule corresponding to Temporal Difference (TD) learning is derived from update (2):

$$\theta^V_{i,t+1} = \theta^V_{i,t} + \alpha \delta_t \frac{\partial V_t(s_t)}{\partial \theta^V_{i,t}} \quad (3)$$

Here $\theta^V_{i,t}$ is the i^{th} component of vector θ^V at time t and $V_t(s)$ is the output of the FA at time t with state s as input. These methods have been extensively studied. See for instance the book by Bertsekas and Tsitsiklis [10].

B. Continuous Action Spaces

A harder problem is to extend RL to continuous action spaces. Even if we have a good approximation of the value function, we still have the problem that we cannot trivially find the action that corresponds to the highest value, in a given state. Therefore, we would like an algorithm to quickly output an approximation of the optimal action, given a certain state. For this again an FA can be used. The question then becomes how to improve this approximation.

In section IV we show some possibilities to apply RL to real continuous action spaces. First some considerations on exploration in continuous action spaces are given.

C. Exploration

Exploration is crucial in RL, since it is the only way to discover new and better policies. We discuss two methods for exploration in continuous spaces. The first method of exploration is ϵ-greedy exploration. An exploratory random action is selected with probability ϵ and the greedy, current approximation for the optimal action is selected with probability $(1 - \epsilon)$. It is then possible to decrease exploration by simply decreasing the factor ϵ.

The second method of exploration is Gaussian exploration around the current approximation of the optimal action. The action that is performed is sampled from a Gaussian distribution with the mean at the action output of the algorithm we

are using. When $\text{Ac}_t(s_t)$ denotes the action that the algorithm outputs at time t, the probability of selecting action a is:

$$\pi_t(s_t, a) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(a-\text{Ac}_t(s_t))^2/(2\sigma^2)}$$

Here $\pi_t(s, a)$ denotes the policy, while π denotes the mathematical constant.

IV. Continuous Algorithms

In this section we will describe RL algorithms that can handle problems with both continuous state and action spaces. First we will present a new algorithm, then we will describe two older algorithms.

A. Continuous Actor Critic Learning Automaton

The Actor Critic Learning Automaton (ACLA) algorithm is a new algorithm that is easily extendable to continuous spaces. First we consider the tabular case with discrete states and actions. One table stores the values of the states. These can be updated with the TD-learning update (2). Another table stores the probabilities of performing each action for all states.

The values of the states converge to the actual discounted future rewards, given the current policy. If performing some action results in a positive change for the value of a state, then that action could potentially lead to a higher discounted future reward and thus to a better policy. Therefore, we reinforce that action. In pseudo-code we get:

$$\text{IF} \quad \delta_t > 0 \quad : \quad \text{increase}_t(\pi_t(s_t, a_t)) \tag{4}$$

Here $\text{increase}_t(\pi(s_t, a_t))$ increases the probability of selecting action a_t in state s_t. The probabilities for other actions in state s_t are scaled down accordingly. Note that we do not decrease the probability of a_t when $\delta_t < 0$. A version that also used negative feedback to adjust the action selection probabilities is possible in the tabular case. However, when we extend the algorithm to continuous action spaces, the resulting algorithm makes more sense when just using the positive feedback. A comparison between these algorithms will be given in the experimental section.

The main difference that distinguishes the ACLA algorithm from conventional actor critic systems is that ACLA only uses the sign of the TD-error to determine the update to the actor, as opposed to using the exact value of the TD-error. This approach has several advantages. The algorithm is easy to implement and to extend to continuous actions. The learning parameter of the actor in Continuous ACLA (CACLA) is invariant to different scalings of the reward function. The algorithm is robust with regard to learning interference. Further, our experimental results will show that the continuous version outperforms other algorithms in continuous action spaces.

To extend the algorithm to continuous spaces, we can just replace the tables by FAs that output the value and the action that needs to be performed, given a certain state. The update to the critic FA is simply equation (3). Denoting the output of the actor FA at time t as $\text{Ac}_t(s_t)$ and its parameter vector as θ^{Ac}, the update to the parameters of the actor is:

$$\text{IF} \quad \delta_t > 0 \quad : \quad \theta^{\text{Ac}}_{i,t+1} = \theta^{\text{Ac}}_{i,t} + \alpha(a_t - \text{Ac}_t(s_t)) \frac{\partial \text{Ac}_t(s_t)}{\partial \theta^{\text{Ac}}_{i,t}}$$

The actor learns to output something more similar to action a_t in state s_t if the value of the state is increased. The actor is not updated when the value is not increased. The reason for this is that when also negative updates are allowed, the algorithm will update away from the last action, which was perceived as bad. However, this is equivalent to updating towards some unknown action, which is not necessarily better than our present approximation of the optimal action. We might have discovered that a_t is not perfect, but we do not yet know of any better alternatives.

To stress the effects of actions that improve the value more than usual we can extend the update. First, a running average of the variance of the TD-error is stored:

$$\text{var}_{t+1} = (1 - \beta)\text{var}_t + \beta\delta_t^2 \ ,$$

Using this variance, we can determine if an action was exceptionally good. We link the number of updates towards an action to the number of standard deviations that the target value lies above the old value. The number of updates then is $\lceil \delta_t / \sqrt{\text{var}_t} \rceil$. When $\delta_t \leq 0$ no updates will be performed. Note that var_0 should not be set too low compared to usual values of δ, since this could lead to too many updates early in learning. We used $\text{var}_0 = 1$ and $\beta = 0.001$.

We shall refer to the algorithm that uses the variance as CACLA+Var. The algorithm that always updates at most once will be referred to as CACLA. We will also use CACLA to refer to the algorithm in general.

In the broadest sense, CACLA is an Actor Critic method, since it uses the value of δ_t to update its actor. A similar class of algorithms named Continuous Actor Critic (CAC) systems use a less step-wise update, linearly relating the size of the update to the size of δ_t. In Fig. 1 we compare these algorithm graphically. Note that the unit for the x-axis is δ_t/stddev_t, where $\text{stddev}_t = \sqrt{\text{var}_t}$ is the average standard deviation of δ_t. This is clearly a function of δ_t, though we have no knowledge of earlier Actor Critic algorithms using a similar function.

In Fig. 1 the line labeled CAC is an example of how a CAC system might update on a specific time step. Since the x-axis is dependent on stddev_t, on another time step the same CAC system may have a corresponding line with a steeper, or less steep slope. A special case where this does not happen is shown as CAC+Var. This CAC system is in between CACLA+Var and CAC in that it does use δ_t/stddev_t to determine its update, but it does not update in steps. Rather CAC+Var updates the actor with a learning rate linearly dependent on the size of δ_t/stddev_t.

The figure makes clear that the precise value of δ_t is not important for CACLA, only its sign matters. The update of CACLA+Var increases in steps with each standard deviation. In practice this means that CACLA+Var will update once

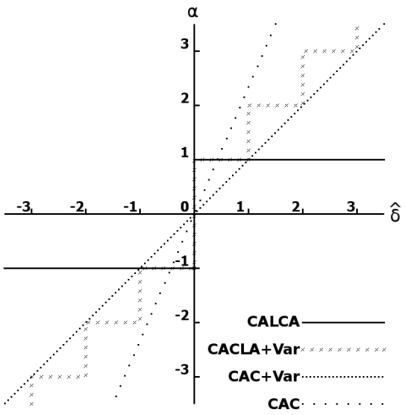

Fig. 1. Comparing CALCA with Continuous Actor Critic (CAC) methods. The unit $\hat{\delta}_t$ of the x-axis is defined as the TD-error divided by the standard deviation of that error: $\hat{\delta}_t = \delta_t / \text{stddev}_t$. Since CAC is dependent only on δ_t and not on stddev_t, in fact the line of CAC in the figure is only an example and it can be steeper or less steep on different time steps.

approximately 65% of the time, twice 30% of the time and more than twice 5% of the time.

In discrete action spaces, conventional Actor Critic (AC) methods attempt to optimize the performance of the policy in terms of the value function. The algorithm described as ACLA is slightly different in its behavior, because it uses less of the available information from δ_t. Consider a stochastic setting with two possible actions a_1 and a_2 in a state s. The following table lists the transition probabilities and corresponding rewards and discounted values of the next states:

a	$T(s_t, a, s_{t+1})$	r_t	$\gamma V_t(s_{t+1})$	$V_t(s_t)$	δ_t
a_1	0.1	-100	-100	0	-200
a_1	0.9	10	10	0	20
a_2	0.1	100	100	0	200
a_2	0.9	-10	-10	0	-20

In such a setting, the expected update of ACLA increases the probability of selecting a_1 and therefore decreases the probability for a_2, because the probability of a positive δ_t is higher for a_1. However, the expected value of δ_t is in fact lower for a_1 than for a_2. Instead of learning a policy that optimizes the size of δ_t, ACLA learns the policy that optimizes the probability of receiving a positive δ_t. This implies that convergence to sub-optimal policies is possible for ACLA in environments with stochastic transitions. It can be shown that this problem does not occur in deterministic environments.

The behavior of CACLA when compared to ACLA differs slightly, because CACLA does not change independent probabilities of actions, but instead updates towards an action that has been found to be better than the present approximation for the optimal action. As we will show in our experiments, faster convergence to good policies can be obtained using CACLA than using CAC even though the experiments feature stochastic transitions due to noisy interactions with the environments.

B. Wire Fitting

Baird and Klopf propose the Wire Fitting (WF) algorithm, which efficiently stores an approximation of the complete Q-function [11]. They propose using a function approximator to output a fixed number of actions and corresponding values, given a certain state. Each action and corresponding value are output independently and concurrently. The outputs can be interpolated if the value of an interlying action is required. The interpolation function is given as follows:

$$\begin{aligned} f(s,a) &= \lim_{\varepsilon \downarrow 0} \frac{\sum_{i=0}^{n} \frac{q_i(s)}{||a-a_i(s)||^2 + c_i(q_{max}(s) - q_i(s)) + \varepsilon}}{\sum_{i=0}^{n} \frac{1}{||a-a_i(s)||^2 + c_i(q_{max}(s) - q_i(s)) + \varepsilon}} \\ &= \lim_{\varepsilon \downarrow 0} \frac{\sum_{i=0}^{n} q_i(s)/d_i(s,a)}{\sum_{i=0}^{n} 1/d_i(s,a)} \\ &= \lim_{\varepsilon \downarrow 0} wsum(s,a)/norm(s,a) \ , \end{aligned}$$

where (s, a) is the state-action pair of which the value is wanted. Time indicating subscripts are left out for increased legibility. n is the number of action-value pairs output and $a_i(s)$ and $q_i(s)$ are the outputs corresponding to the i^{th} action and action-value for this state, respectively. $q_{max}(s) \stackrel{def}{=} \max_j q_j(s)$ is the maximum of all q_i. c_i is a small smoothing factor and ε prevents division by zero. Given a state and an action, the interpolation gives a weighted average of the different values, depending on the distance of the given action to the output actions on this particular state. The results for our experiments were obtained when using $c = 10^{-3}$ and 9 wires. This makes the WF algorithm computationally by far the most expensive algorithm. Using more wires did not increase performance levels, while using less wires resulted in slightly worse performance.

Consider that an agent using WF has an experience: (s_t, a_t, r_t, s_{t+1}). We will now explain how this experience is used to update the system. As mentioned, in this algorithm the output of the interpolation is to be interpreted as the value of a given state-action pair, which we shall denote as $Q(s, a)$. Let $\theta_i^{a_j}$ denote the parameters of the function approximator that outputs action a_j. The following update will change these parameters according to gradient descent on the error of the value:

$$\theta_i^{a_j} \leftarrow \theta_i^{a_j} + \alpha(r_t + \gamma \max_b Q(s_{t+1}, b) - Q(s_t, a_t)) \frac{\partial Q(s,a)}{\partial a_j} \frac{\partial a_j}{\partial \theta_i^A}$$

Where:

$$\frac{\partial Q(s,a)}{\partial a_j} = \lim_{\varepsilon \downarrow 0} \frac{2(wsum(s,a) - norm(s,a) q_j)(a_j - a)}{(norm(s,a) d_i(s,a))^2}$$

A similar update can be done for the parameters of the function approximators that output the values. Then we use:

$$\frac{\partial Q(s,a)}{\partial q_j} = \lim_{\varepsilon \downarrow 0} \frac{norm(s,a)(d_i(s,a) + q_j c) - wsum(s,a) c}{(norm(s,a) d_i(s,a))^2}$$

Because of the nature of the interpolator, the action with the highest value is always amongst the ones output by the function approximator. This means that to find this action we do not have to use the interpolation function. We just select

the action with the highest corresponding value. This results in fast action selection. In a fully trained system with a low error between experienced action-value pairs and the interpolation output, we can assume that this is the optimal action, though convergence guarantees can not be given.

The approach is called Wire Fitting, because essentially the interpolation function describes a surface in $S \times A \times \mathbb{R}$ space, which is draped, so to say, over wires defined by the outputs of the function approximator. Because the whole value function is approximated, using enough outputs, the system can reach any real valued optimal policy, generalize well, while allowing fast action selection.

C. Gradient Ascent on the Value

Prokhorov and Wunsch described Adaptive Critic Designs of which we implemented a version of their Action Dependent Heuristic Dynamic Programming (ADHDP) algorithm [12]. Since more similar algorithms exist, we will use the more general name of Gradient Ascent on the Value (GAV). The algorithm uses a single actor function approximator to output the optimal action, given a state. We shall denote the output of the actor at time t as $\text{Ac}_t(s)$. When this action is selected, the Q-value can be determined using a critic function approximator that outputs the Q-value given a state and action.

The following continuous version of Q-Learning handles the updates to the parameters of the critic function approximator. An update can be made after each experience of a state, action, reward and new state:

$$\theta_i^Q = \theta_i^Q + \alpha(r_t + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t)) \frac{\partial Q_t(s_t, a_t)}{\partial \theta_i^Q} \tag{5}$$

Training the actor function approximator is only slightly more complex. We need to find a target towards which we can train the output of the actor, given a certain state. For this we can use gradient information to determine how the value function would change if the action is changed locally. The gradient information of the value can be propagated back immediately to the parameters of the actor. Calling the parameters of the actor θ^{Ac}, this results in the following update:

$$\theta_i^{\text{Ac}} \leftarrow \theta_i^{\text{Ac}} + \alpha \frac{\partial Q_t(s_t, a_t)}{\partial \text{Ac}_t(s_t)} \frac{\partial \text{Ac}_t(s_t)}{\partial \theta_i^{\text{Ac}}} \tag{6}$$

So in summary, an update consists of a gradient descent update on the TD-error of the output value for the critic function approximator and a gradient ascent update on the Q-value for the actor function approximator. A potential problem lies in the backpropagation of the value through the critic. When the critic is not yet fully trained, the gradient information on the value will not always be accurate. This can lead to divergence.

In the original ADHDP algorithm, an update was made to the actor every time step. Our adaptation consists of slowly increasing the probability for an update to the actor's parameters during learning. This prevents early divergence of the actor function approximator due to the incorrect early gradient information provided by the critic. In our experiments, the probability of an update being performed was $(1-p)$, where p decreases exponentially from 1 to 0.01 by multiplication with a constant. Some experiments with both the original algorithm and the version we just described showed the adaptation performed significantly better.

Prokhorov and Wunsch note that they did not get ADHDP to work on one of their problems [12]. They propose using similar algorithms that do solve that problem and should have better performance on other problems. However, these algorithms require that the agent knows the reward function or a model of the environment. Because we do not want to assume such knowledge, we do not discuss these algorithms.

V. Experimental Setup

Now we will describe the experimental setup we used to test the algorithms. The experiments were a Tracking experiment and a Cart Pole experiment. A comparison between CACLA and CAC was performed, but only in the cart pole setting. Because these results showed that CACLA outperforms CAC, the latter was not included in the comparison with WF, GAV and a random policy. All experiments were run with discount factors of 0.0, 0.8, 0.9, 0.95 and 0.99. Gaussian noise was added to the actions and reward since noisy interactions are inevitable in most real-world applications.

When using ϵ-greedy exploration, the exploration rate (ϵ) was exponentially dropped by multiplication with a constant from 1 - a random action every time step - to 0.01 - a probability of 0.01 to perform a random action. Setting the decay of exploration higher or lower did not result in better performance for any of the algorithms. When using Gaussian exploration, the standard deviation for the distribution was 0.1. Different settings were not tried. In contrast with the ϵ-greedy exploration described above, this exploration did not decay.

For all FAs used by the algorithms, we used feedforward neural networks with 12 hidden neurons. Preliminary tests showed the number of hidden neurons did not severely affect performance. In both experiments, the components of state and action vectors and the rewards were linearly scaled to fall within the $[-1, 1]$ interval, essentially scaling the inputs and expected outputs of the neural networks to this interval. The activation function of the hidden layer of the networks was a sigmoidal function $\mathbb{R} \rightarrow [-1, 1]$. The output layer used a linear activation function. The learning rate for the weights of the networks was 0.01 in all cases. Different settings for the learning rates did not increase performance for any of the algorithms in preliminary testings.

A. Tracking

In this experiment the objective is to get an agent to follow a target. The positions of the agent and the target are real valued two dimensional vectors $\in [0, 10]^2$. The state description is a four dimensional vector $\in [0, 10]^4$ containing the positions of the agent and the target. The actions are two dimensional vectors $\in \mathbb{R}^2$, which represent the goal position for the agents. The components of the action need not be in the $[0, 10]$ interval, though the agent can never actually step outside the 10x10 area. The reward that should be maximized is the

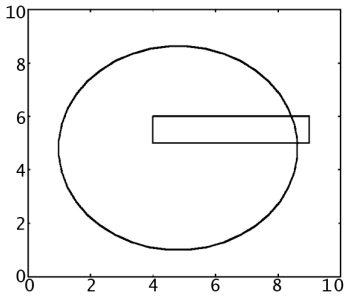

Fig. 2. Tracking agent. The circle represents the movement of the target. The rectangular shape represents the obstacle. The target is allowed to pass through it, but the agent is not.

negative squared distance between the learning agent and the target. The movement of the target is shown in Fig. 2. The target takes 40 time steps to return to its initial position. Also an obstacle is shown that the target can pass through. However, the agent cannot and therefore needs to find a way around it. When an agent tries to step outside the area or through the obstacle, the agent's new coordinates will be the place where it would hit the boundary, assuming a continuous straight trajectory. Gaussian noise was added to the actions and reward with a standard deviation of 0.1. The initial position of the agent and the target were $(5.0, 5.0)$ and $(1.0, 4.5)$, respectively.

B. Cart Pole

This problem involves a cart that stands on a one dimensional track. The goal is to balance a pole on top of the cart by pushing the cart left or right. Also the cart must not stray too far from its initial position. The state description to the RL agent consists of a four dimensional vector containing the angle ϕ and (radial) speed of the pole $\phi' = \partial\phi/\partial t$ and the position x and speed $x' = \partial x/\partial t$ of the cart. The action of the actor consist of a real valued force that is used to push the cart. A positive force results in pushing the cart right, while a negative value results in pushing the cart left.

The weights of cart and pole are 1.0 kg and 0.1 kg respectively. The length of the pole is 1 m and the duration of one time step 0.1 s. The reward is calculated as follows:

$$r_t = \begin{cases} 1 & \text{if } |\phi| < \frac{1}{15}\pi \text{ and } |x| < 1 \\ -1 & \text{otherwise} \end{cases} \quad (7)$$

This reward function gives a reward of -1 on failure and a reward of 1 on all other time steps. Failure is defined as a state where the angle of the pole is more than 12 degrees in either direction, or the cart is further than 1 meter from its initial position. Using this reward function forces the algorithms to look ahead, since the reward function gives little information on how good the current situation in fact is.

When one of the conditions for failure was reached, the system was reset to $x = 0$, $x' = 0$, $\phi' = 0$ and $\phi = x$, where x is a random number in the interval $[-0.05, 0.05]$. This was also the initial state of the system.

TABLE I

TRACKING RESULTS. THIS TABLE GIVES THE RESULTS AFTER 102400 TIME STEPS. RANDOM GAUSSIAN NOISE WAS ADDED TO ACTIONS AND REWARDS WITH A STANDARD DEVIATION OF 0.1. AVERAGED OVER 20 SIMULATIONS.

	ϵ-greedy		
	γ	**mean**	**std dev**
WF	0.80	0.861	0.056
GAV	0.00	0.783	0.163
CACLA	0.95	0.829	0.012
CACLA+Var	0.90	0.843	0.014
RND	0.00	0.467	0.020
	Gaussian		
WF	0.00	0.816	0.261
GAV	0.95	0.088	0.265
CACLA	0.95	0.922	0.011
CACLA+Var	0.80	0.938	0.011
RND	0.00	0.467	0.020

VI. RESULTS

Next we will present the final results, which were obtained after 102400 time steps of learning. Also the performance at the intermediate steps is discussed. We give the discount factor that resulted in the best average results along with the results. For the settings of other parameters a coarse search was performed. The results presented below are from experiments with added random noise. Conducting the same experiments without this noise resulted in similar results.

First we present the results comparing CACLA to WF, GAV and the random policy. Then, we show the results of the comparison of CACLA and CAC on the cart pole task.

A. Tracking

In Table I the results for the Tracking experiment are shown. Fig. 3 shows intermediate results during training. The y-axis represents a running average of the rewards. The CACLA algorithms in combination with Gaussian exploration perform significantly better than the other algorithms, as can be seen in the right half of the plot in Fig. 3. Furthermore, the CACLA algorithms find good solutions very quickly.

B. Cart Pole

As can been seen in Fig. 4 and Table II, in this experiment the CACLA algorithms outperform the other algorithms for both types of exploration. Again, the results for CACLA with Gaussian exploration are better than with ϵ-greedy exploration. The differences with the other algorithms are significant. As can be concluded from Table II, the CACLA algorithms managed to learn to balance the pole perfectly in every simulation when using Gaussian exploration, also for higher discount factors than the ones given in the table. The duration of 100 s corresponds to 1000 actions. In Fig. 4 we see that good solutions are found very fast, especially by CACLA+Var.

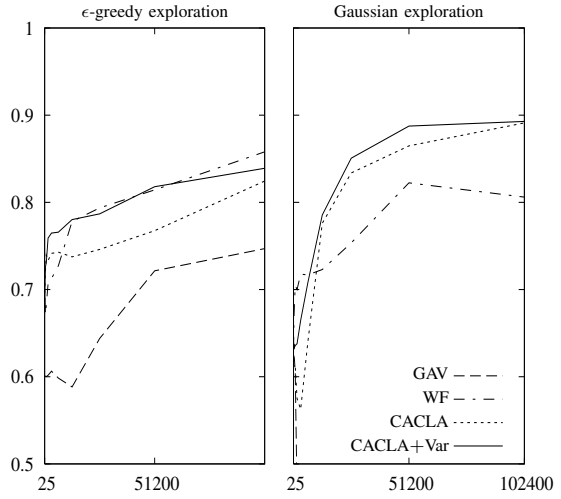

Fig. 3. Tracking results. Average intermediate rewards are plotted against the number of training iterations. Results with ϵ-greedy exploration are shown on the left. The results on the right were obtained with Gaussian exploration. Random Gaussian noise was added to actions and rewards with a standard deviation of 0.1. Averaged over 20 simulations. GAV is not visible in the right plot, because its performance is too low.

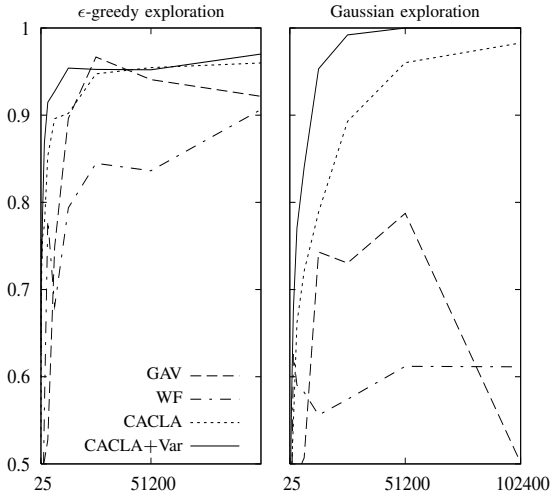

Fig. 4. Cart Pole results. Average intermediate rewards are plotted against the number of training iterations. Results with ϵ-greedy exploration are shown on the left. The results on the right were obtained with Gaussian exploration. Random Gaussian noise was added to actions and rewards with a standard deviation of 0.3. 20 simulations were performed.

TABLE II

CART POLE RESULTS. THE TABLE GIVES THE PERCENTAGE OF TRIALS THAT ENDED WITH A SOLUTION THAT CAN BALANCE THE POLE FOR AT LEAST 100 S. RANDOM GAUSSIAN NOISE WAS ADDED TO ACTIONS AND REWARDS WITH A STD DEV OF 0.3. 20 SIMULATIONS WERE PERFORMED.

	ϵ-greedy		**Gaussian**	
	γ	**success**	γ	**success**
WF	0.80	10 %	0.80	0 %
GAV	0.00	15 %	0.80	20 %
CACLA	0.95	40 %	0.80	100 %
CACLA+Var	0.99	80 %	0.95	100 %
RND	0.00	0 %	0.00	0 %

Finally, we present some results comparing the different versions of CACLA and CAC. In Table III the results are given. Fig. 5 shows the corresponding intermediate results. The results for CACLA differ from Fig. 4 because these were obtained with $\gamma = 0.9$ instead of $\gamma = 0.8$, showing that though the final behavior is in both cases perfect, intermediate results are somewhat better with $\gamma = 0.9$.

It is clear that CACLA and CACLA+Var with only positive updates perform the best of all algorithms. Once again, CACLA+Var learns the fastest. When also negative updates are performed, all algorithms except CAC+Var perform worse. The most striking difference is in the case of CACLA, which performs so bad that it is not even visible on the right in Fig. 5. The algorithm never succeeds in balancing the pole for 100 seconds and it even diverges to solutions where it topples the pole with every action in some cases. CAC, CAC+Var and CACLA+Var suffer less from the updates when $\delta_t < 0$. However, they do not reach as good solutions as CACLA and CACLA+Var when only positive updates are performed. It can be noted that the performance of CAC+Var is significantly better than that of CAC when updates are done every time-step.

When comparing CAC and CAC+Var to WF and GAV by comparing Fig. 5 with Fig. 4, we see that when CAC and CAC+Var only use positive updates, they reach better solutions than both WF and GAV do. However, because the CACLA algorithms significantly outperform the CAC algorithms, we did not extensively study this comparison.

The reason that CAC+Var performs better than CAC most probably lies in the fact that CAC+Var is less sensitive to variations in the reward feedback. By using an approximation of the variance to scale the learning rate, the algorithm becomes less sensitive to noise. Also CACLA, CACLA+Var and CAC+Var will be able to maintain the same level of performance when the reward function is linearly scaled. CAC will require a new search for the optimal learning rate in such a case. In our experiments, preliminary experiments were performed with different learning rates, but no algorithm performed better with other settings than 0.01. When the reward function is scaled $\times 100$, the learning rates for the actor of CACLA, CACLA+Var and CAC+Var can still be set to 0.01. However, in the case of CAC performance will deteriorate unless we scale the learning rate down.

The invariance to the precise size of δ_t of the new algorithms in this paper has another advantage. In almost all cases, the performance of CACLA and CACLA+Var barely decreases when higher discount factors are chosen. In most cases performance at $\gamma = 0.95$ or $\gamma = 0.99$ is only slightly worse than at $\gamma = 0.9$. In some cases, it is even better at those higher discount factors. For the other algorithms, this was not the case. This makes sense when considering the

TABLE III

CART POLE RESULTS. THE TABLE GIVES THE PERCENTAGE OF TRIALS THAT ENDED WITH A SOLUTION THAT CAN BALANCE THE POLE FOR AT LEAST 100 S. RANDOM GAUSSIAN NOISE WAS ADDED TO ACTIONS AND REWARDS WITH A STD DEV OF 0.3. 20 SIMULATIONS WERE PERFORMED.

	Update only when $\delta_t > 0$		**Update always**	
	γ	**success**	γ	**success**
CACLA	0.90	100 %	0.90	0 %
CACLA+Var	0.95	100 %	0.90	80 %
CAC	0.90	50 %	0.90	45 %
CAC+Var	0.99	60 %	0.90	60 %

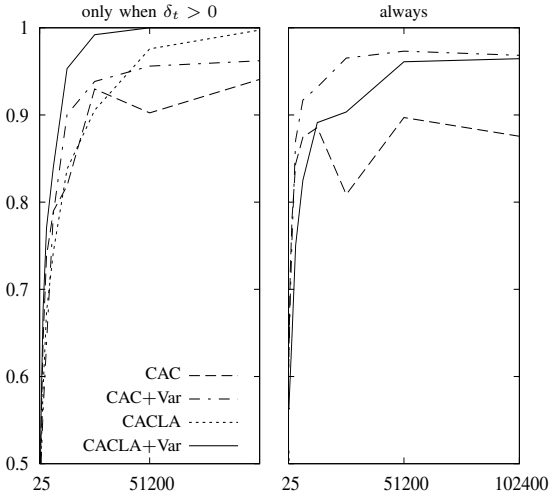

Fig. 5. Cart Pole results for CAC and CACLA. Average intermediate rewards are plotted against the number of training iterations. The results on the right are obtained when updating the algorithms only when $\delta_t > 0$. For the results on the left, the actor was updated every time step. All results were obtained with Gaussian exploration. Random Gaussian noise was added to actions and rewards with a standard deviation of 0.3. 20 simulations were performed.

impact of a higher discount factor on the value function. When the discount factor is higher, the values will also be. The algorithms using the variance of δ_t to scale their learning rates are invariant to this scaling of the value function. The other algorithms show decreasing performance for larger discount factors, usually peaking at discount factors of 0.8.

VII. CONCLUSION

We have presented a new class of algorithms named Continuous Actor Critic Learning Automaton (CACLA) for the RL framework that we extended to handle problems that involve continuous state and action spaces. In continuous Tracking and Cart Pole experiments the performance of this algorithm was very good when compared to the performance of two other algorithms that can handle continuous states and actions. This is interesting, because the algorithm is relatively simple to implement and its computational requirements are low. A similar class of algorithms named Continuous Actor Critic (CAC) systems was defined and found to be inferior in performance to the CACLA algorithms.

More specifically, the performance of CACLA with Gaussian exploration was unequaled when considering the rate of convergence as well as the final performance. Added noise did not severely hinder the performance of the algorithm, showing that it is resistant to noisy interactions with an environment.

CACLA and CAC only update when the TD-error is positive. Also considering negative updates may allow more updates to the actors of the algorithms, but was shown to result in worse behavior. The main difference between CAC and CACLA lies in their intended optimization. CAC tries to optimize the policy in terms of δ_t, while CACLA optimizes the probability of positive δ_t. This approach increases learning rates, especially in areas where δ_t is relatively small.

An extension to the CACLA and CAC algorithms was presented that uses the variance δ_t to determine if an update has been significantly large. This version performs slightly better than the versions of the algorithms that treat all updates equally. We updated the variance as one running average over all visited states. However, the true variance of states may differ greatly. Probably better results can be obtained when an approximation of the current variance is stored per state. A function approximator could be used that maps states to approximations of the variance of δ_t.

REFERENCES

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. The MIT press, Cambridge MA, A Bradford Book, 1998.

[2] J. Peters, S. Vijayakumar, and S. Schaal, "Natural actor-critic," in *16th European Conference on Machine Learning, Porto, Portugal, October 3-7, 2005, Proceedings*, ser. Lecture Notes in Computer Science, J. Gama, R. Camacho, P. Brazdil, A. Jorge, and L. Torgo, Eds., vol. 3720. Springer, 2005, pp. 280–291.

[3] M. Riedmiller, "Neural fitted Q iteration - first experiences with a data efficient neural reinforcement learning method," in *16th European Conference on Machine Learning, Porto, Portugal, October 3-7, 2005, Proceedings*, ser. Lecture Notes in Computer Science, J. Gama, R. Camacho, P. Brazdil, A. Jorge, and L. Torgo, Eds., vol. 3720. Springer, 2005, pp. 317–328.

[4] R. E. Bellman, *Dynamic Programming.* Princeton University Press., 1957.

[5] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine Learning*, vol. 3, pp. 9–44, 1988.

[6] P. Dayan, "The convergence of TD(λ) for general lambda," *Machine Learning*, vol. 8, pp. 341–362, 1992.

[7] C. J. C. H. Watkins, "Learning from delayed rewards," Ph.D. dissertation, King's College, Cambridge, England, 1989.

[8] G. Rummery and M. Niranjan, "On-line Q-learning using connectionist systems," Cambridge University, UK, Tech. Rep. CUED/F-INFENG-TR 166, 1994.

[9] R. S. Sutton, "Generalization in reinforcement learning: Successful examples using sparse coarse coding," in *Advances in Neural Information Processing Systems 8*, D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, Eds. MIT Press, Cambridge MA, 1996, pp. 1038–1045.

[10] D. P. Bertsekas and J. N. Tsitsiklis, *Neuro-dynamic Programming*. Athena Scientific, Belmont, MA, 1996.

[11] L. C. Baird and A. H. Klopf, "Reinforcement learning with high-dimensional, continuous actions," Wright-Patterson Air Force Base Ohio: Wright Laboratory, Tech. Rep. WL-TR-93-1147, 1993. [Online]. Available: http://leemon.com/papers/index.html#b93b

[12] D. V. Prokhorov and D. C. Wunsch II, "Adaptive critic designs," *IEEE Transactions on Neural Networks*, vol. 8, no. 5, pp. 997–1007, September 1997. [Online]. Available: citeseer.csail.mit.edu/prokhorov97adaptive.html