Using Continuous Action Spaces to Solve Discrete Problems

Hado van Hasselt Marco A. Wiering

Abstract— **Real-world control problems are often modeled as Markov Decision Processes (MDPs) with discrete action spaces to facilitate the use of the many reinforcement learning algorithms that exist to find solutions for such MDPs. For many of these problems an underlying continuous action space can be assumed. We investigate the performance of the Cacla algorithm, which uses a continuous actor, on two such MDPs: the mountain car and the cart pole. We show that Cacla has clear advantages over discrete algorithms such as Q-learning and Sarsa, even though its continuous actions get rounded to actions in the same finite action space that may contain only a small number of actions. In particular, we show that Cacla retains much better performance when the action space is changed by removing some actions after some time of learning.**

I. Introduction

A large body of literature exists about learning to map an input set to a finite set of outputs. A large subset of these learning problems involve outputs that can be thought of as lying on a continuous underlying real valued line. The actual set of outputs is then a finite set of points in this continuous space. It can be beneficial to approximate a continuous mapping of the input space to this continuous output space and only to discretize the output of the algorithm at the last moment by rounding to one of the actually available options. In particular, such an algorithm can generalize better over the available options. This better generalization has two main advantages. First, we expect an algorithm that makes use of this to be able to learn faster, especially when the number of discrete options is large. Second, it can be helpful to be able to generalize when the output space is not fully fixed.

In this paper we consider reinforcement learning problems that attempt to learn the mapping of a state space to an action space. As an example, consider an agent trained to drive a manual transmission car. First, due to generalization, a continuous algorithm may more quickly learn that the third gear will have an effect somewhere between the effects of the second and fourth gears, even if it has not been tried often yet. Second, assume that after a good policy has been learned the third gear stops working altogether. It is an interesting topic to see how an algorithm can adapt to such a situation and our intuition is that algorithms that assume continuous underlying spaces can more easily change their behavior to compensate for missing actions. This kind of a changing action space may also be an issue when training an algorithm on an imperfect simulation before trying to exploit its learned behavior in an actual system.

Hado van Hasselt is with the Intelligent Systems Group, Utrecht University, Utrecht (email: hado@cs.uu.nl) and Marco Wiering is with the Department of Artificial Intelligence, University of Groningen, Groningen (email: mwiering@ai.rug.nl)

We will use the Continuous Actor Critic Learning Automaton (Cacla) reinforcement learning algorithm [2] to approximates a continuous mapping from state space to action space. Cacla outputs real valued actions that will only be discretized at the last moment and we will show that it performs well compared to discrete reinforcement learning algorithms that learn a separate value for each action on two classical reinforcement learning problems: the cart pole and the mountain car. Next to showing that the Cacla algorithm can perform well on these problems, we also present a new discrete version of the Cacla algorithm, named Actor Critic Learning Automaton (Acla) [14] and show that it also outperforms most of the conventional algorithms on the selected problems. Finally, we will adapt the action space of the cart pole after the algorithms are trained by removing some of the possible actions and show that Cacla is far more robust to such changes than the other tested algorithms.

The paper is organized as follows. In the next section we will explain the reinforcement learning algorithms that are used. Section III describes the experimental setting for the mountain car and the cart pole problem. We will present and analyze the results of the experiments in section IV. The last section concludes the paper and gives directions for future research.

II. Methods

Problems that can be solved with reinforcement learning algorithms can be modeled as Markov Decision Processes (MDPs). A finite MDP can be viewed as a tuple (S, A, R, T, γ), where $s_t \in S$ denotes the state the agent is in at time t; $a_t \in A$ denotes the action it performs in that state; r_t denotes the possibly stochastic reward received at time t and $R^a_{ss'}$ denotes the expected value of such a reward when moving from state s to state s' after performing action a; T is the transition function, where $T^a_{ss'}$ gives the probability of ending up in state s' after performing a in s; and $0 \leq \gamma \leq 1$ is the discount factor that discounts future rewards.

In the rest of this paper, we only look at problems with continuous state spaces and discrete action spaces. We will distinguish between conventional, discrete algorithms that learn how to choose between the finite possible actions in each state and the continuous algorithm Cacla that assumes an underlying continuous action space. Each of the algorithms has slightly different properties and as will we see in section IV, in many cases there are significant differences in performance.

All algorithms keep track of a state value or a state-action value that gives the expected value of state s or of a state-action pair (s, a) respectively. How these values are updated and used to get a policy for behavior is outlined below for each of the algorithms. We use the following notation:

$$X_{t+1} \xleftarrow{\alpha} T_t \ , \tag{1}$$

to denote an update of X towards target T_t with a step size of α. This notation is shorthand for the following update:

$$X_{t+1} = X_t + \alpha(T_t - X_t) \ . \tag{2}$$

In this paper we use neural networks as function approximators. If X is approximated with a neural network, we use the shorthand in (1) for the gradient descent update on the squared difference between target and former value:

$$w^X_{t+1} = w^X_t + \alpha(T_t - X_t)\nabla X_t \ , \tag{3}$$

where w^X_t is the weight vector at time t of the network approximating X and ∇X_t is the gradient of the output of the network to this weight vector. Now we will describe all the different algorithms, starting with the discrete algorithms.

A. Discrete Algorithms

1) Q-learning: One of the best known and most used reinforcement learning algorithms is Q-learning [11]. This algorithm uses a one step temporal difference update to update its approximation of the value of the state-action pair (s_t, a_t). The update of Q-learning looks as follows:

$$Q_{t+1}(s_t, a_t) \xleftarrow{\alpha_t} r_t + \gamma \max_a Q_t(s_{t+1}, a) \ , \tag{4}$$

where $0 \leq \alpha_t \leq 1$ is a step size parameter.

Q-learning is an off-policy algorithm, which means the optimal policy is approximated even if a non-optimal (i.e. exploring) policy is followed.

2) Sarsa: The update for Sarsa [5], [9] is similar to that of Q-learning:

$$Q_{t+1}(s_t, a_t) \xleftarrow{\alpha_t} r_t + \gamma Q_t(s_{t+1}, a_{t+1}) \ . \tag{5}$$

The main difference lies in the lack of a $\max$ operator in the case of Sarsa. This implies that Sarsa is on-policy, which means that Sarsa learns an approximation of the values of the state-action pairs including exploration steps.

3) Actor Critic: Actor Critic (AC) is the first of three algorithms that we discuss that use a state value function updated with Temporal Difference (TD) learning [8]:

$$V_{t+1}(s_t) \xleftarrow{\beta_t} r_t + \gamma V_t(s_{t+1}) \ . \tag{6}$$

An advantage of using update (6) is that because the V values get updated every time a state gets visited, they can become reliable approximations more quickly than Q values, that are only updated whenever the corresponding action is selected in that state.

The AC algorithm uses the state values to update a preference function P, which gives the preference for each action:

$$P_{t+1}(s_t, a_t) = P_t(s_t, a_t) + \alpha_t \delta_t \ , \tag{7}$$

where δ_t is called the TD-error and is defined as follows:

$$\delta_t = r_t + \gamma V_t(s_{t+1}) - V_t(s_t) \ . \tag{8}$$

Using the general update (1), which also includes the possibility for neural networks, this update can be written as follows:

$$P_{t+1}(s_t, a_t) \xleftarrow{\alpha_t} P_t(s_t, a_t) + \delta_t \ . \tag{9}$$

4) QV-learning: A newer algorithm that has been shown to perform well on some problems is the QV-learning algorithm [13], [14], which also uses the value function as updated with update (6) and approximates the state-action values as follows:

$$Q_{t+1}(s_t, a_t) \xleftarrow{\alpha_t} r_t + \gamma V_t(s_{t+1}) \ . \tag{10}$$

This algorithm can be viewed as a mix of Sarsa and the AC algorithm. It uses state values that may learn quickly, but it approximates actual Q values, and not preference values such as AC does.

5) R-learning: We also consider the R-learning algorithm [6], [3]. This algorithm uses a slightly different paradigm in that it does not try to approximate the value of the discounted cumulative future rewards, but rather approximates the gain when comparing the expected value of the present action to the average rewards over all states. This means that in principle the average rewards are optimized instead of the discounted cumulative rewards. The algorithm updates its R values with the following update:

$$R_{t+1}(s_t, a_t) \xleftarrow{\alpha_t} r_t + \max_a R_t(s_{t+1}, a) - \rho_t \ , \tag{11}$$

where ρ_t gives an approximation of the average reward, updated as follows:

$$\rho_{t+1} \xleftarrow{\beta_t} r_t + \max_a R_t(s_{t+1}, a) - \max_a R_t(s_t, a) \ , \tag{12}$$

where ρ_t is only updated when the greedy action was selected in state s_t.

R-learning is an important addition to our set of algorithms since amongst other performance measures we will be looking at average rewards. R-learning is the only algorithm that explicitly optimizes this performance measure, but as we will see this does not necessarily mean that it reaches better performance levels in the experiments in this paper when compared to algorithms optimizing the related cumulative discounted rewards.

6) Acla: The last discrete algorithm we consider is a new and simpler version of the Actor Critic Learning Automaton (Acla) algorithm [14]. It also uses update (6) and then updates preference values for the selected action as follows:

$$\begin{array}{ll} P_{t+1}(s_t, a_t) \xleftarrow{\alpha_t} 1 & \text{if } \delta_t > 0 \\ P_{t+1}(s_t, a_t) \xleftarrow{\alpha_t} 0 & \text{if } \delta_t \leq 0 \end{array} \tag{13}$$

For Acla, only the sign of the TD-error matters and not its size.

This version of Acla differs from the original version that also updated the preferences of all actions that were not selected in order to let the preferences of all actions sum to one in every state. We relaxed this constraint by only updating the last performed action and have observed better results with this new, simpler version.

B. Cacla

The forementioned algorithms will be compared to an algorithm that was originally designed for continuous action spaces. The algorithm we will use is called the Continuous Actor Critic Learning Automaton (Cacla) [2].

Cacla uses a critic that stores the expected sum of discounted rewards for states, using update (6). Then, the TD-error δ_t is used to determine if a_t was a good action or not. A function approximator Ac is used as an actor that approximates a function $Ac^* : S \rightarrow A$, where $Ac^*(s)$ denotes the optimal action for state s. This actor is updated by noting that if the TD-error is positive, the action that was just performed is better than expected and should therefore be enforced. The actor is then updated towards this action, with the intent that its output becomes more similar to the last performed action for the present state. The update then becomes:

$$Ac_{t+1}(s_t) \xleftarrow{\alpha_t} a_t \quad \text{if } \delta_t > 0 \ . \tag{14}$$

Note that this update only changes the actor when the action a_t that is actually performed differs from $Ac_t(s_t)$, which is why the Cacla algorithm can only learn new actions on exploratory steps. Of course, non-exploratory steps can still improve the accuracy of the value function.

Like Acla, Cacla only uses the sign of the TD-error and not its size. If the TD-Error is negative, the policy is not updated, because it can not be guaranteed that updating away from an action that resulted in a negative TD-error will result in an update towards an action that would have resulted in a positive TD-error [2]. This is different from Acla, where the action is updated towards a preference value of zero when the TD-error is negative. However, in Acla there is no interference between the separate actions, as there is in Cacla since we only keep track of a single approximation of the optimal action instead of a preference value for each action. In a sense, this algorithm performs a form of hill-climbing in the policy function space, using the TD-error as guidance.

For our comparison to the discrete algorithms, we discretize the output of the actor of Cacla by simply rounding it to the nearest allowed action in the action space of the MDP. This makes sure that Cacla does not have the advantage of being able to more finely determine its action, though since it is a feature of the design of the algorithm, this advantage is arguably neither unrealistic nor unfair. However, one of the purposes of this paper is to show that Cacla can find good solutions when the actions that can actually be performed must be chosen from a limited, finite set.

C. Function Approximation

In this paper we consider MDPs with continuous state spaces and therefore we need some kind of function approximation to be able to store the value functions used by the algorithms. For this, we use neural networks. All neural networks in our experiments have 15 hidden nodes. This number was varied in preliminary experiments, but the results seem to be relatively invariant to the precise number of hidden neurons used. Using much lower numbers resulted in lower performance levels, whereas using much more neurons increased the learning time. Numbers within the range of 10 to 20 hidden neurons behaved similarly and the major differences in performance result from the algorithm that is used. In short, neural networks with 15 hidden units are general enough function approximators for all the functions we try to approximate in our experiments. An in depth comparison for different numbers of hidden units and different topologies of neural networks is beyond the scope of this paper.

To avoid interference to the values of other actions when the value of an action is updated, a separate neural network is used to store the value of each action for the discrete algorithms. Cacla simply uses a single neural network as its actor.

D. Exploration Considerations

The simplest form of exploration is ϵ-greedy exploration, where the highest valued action is chosen with probability $(1 - \epsilon)$ and a random action is chosen with probability ϵ. However, this form of exploration uses very little of the available information about the values of states or state-action pairs. Therefore also Boltzmann exploration was tested, where the Q values or the preference values of the algorithms are used to calculate the probabilities as follows:

$$\pi_t(s_t, a) = \frac{e^{X_t(s_t,a)/\tau}}{\sum_b e^{X_t(s_t,b)/\tau}} \ , \tag{15}$$

where τ is a temperature parameter and X is P, R or Q, depending on the algorithm. Compared to ϵ-greedy exploration, Boltzmann exploration makes more use of the values found so far. Actions that seem more promising because of higher values have a higher probability of being selected.

It should be noted that unlike ϵ-greedy exploration, Boltzmann exploration cannot be used with Cacla, since the Cacla algorithm does not store action values. Since the Cacla algorithm can be viewed as a kind of hill climbing algorithm in the policy space, it is useful for Cacla to perform many exploratory actions, which are especially useful when they are relatively close to the present approximation of the optimal action. Therefore, Gaussian exploration seems a sensible choice for Cacla. Gaussian exploration draws a value stochastically from the Gaussian probability function $G(x, \mu, \sigma)$ centered around the output of the actor $Ac(s_t)$:

$$G(x, Ac(s_t), \sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-Ac(s_t))^2/2\sigma^2} \ , \tag{16}$$

where σ is the width of the Gaussian, which will be an exploration parameter. Then, the action from the action space that is closest to the resulting value is selected. If the actions have more than one dimension, the variance for each dimension could be chosen separately, but in this paper we only consider problems with one dimensional actions.

For instance, consider an action space containing the integers between 0 and 5. Then if the actor outputs the action

TABLE I

THE DIFFERENT PARAMETERS THAT NEED TO BE SET.

Parameter	Description
σ	Width of the Gaussian exploration
τ	Temperature of Boltzmann exploration
α	Learning rate of action values or actor
β	Learning rate of state value (if applicable)

3 and the Gaussian width σ is 2, we get the following action selection probabilities:

a	0	1	2	3	4	5
$\pi(s,a)$	0.01	0.06	0.24	0.38	0.24	0.07

Note that even though the actions are the same distance from action 3, action 5 has a slightly higher chance of being selected than action 1, because it is the result of rounding any value above 4.5 to the nearest available action, whereas action 1 only gets chosen when the result of adding Gaussian noise to action 3 results in a value between 0.5 and 1.5.

In all experiments all algorithms were tested with both Gaussian and ϵ-greedy exploration. Also, for all algorithms except Cacla experiments were run with Boltzmann exploration. In every experiment, Gaussian was significantly better than ϵ-greedy for Cacla and Boltzmann exploration was the best type of exploration for all the other algorithms. Therefore, in all experiments, the results are shown for Gaussian Cacla and the Boltzmann versions of all the other algorithms.

III. EXPERIMENTAL SETUP

To measure the performance of the algorithms, we used a cart pole balancing task and a mountain car task, which are explained below. For R-learning the best results were obtained with an optimistic initialization of ρ at $\rho_t = \max_t r_t$, which equals 100 for the mountain car and 1 for the cart pole. All algorithms except R-learning used a discount factor of 0.99. With this discount factor, there is no difference between an optimal solution in the average rewards setting and the discounted rewards setting for the tasks in this paper.

The parameters used for the experiments are given in Table I. Because the values of these parameters are not fully independent, for each problem we ran 100 experiments for a wide range of combinations of parameter settings. The best parameter setting of these experiments was then taken and run for a separate 300 trials to give the results in the sections below. Every parameter had as possible values 1×10^x, 2×10^x and 5×10^x, for $x \in [-6, 0]$ for the learning rates and $x \in [-3, 3]$ for the exploration parameters. Therefore, we step through the parameter space approximately with multiples of 2. A more fine grained resolution for these parameters can result in slightly better results. However, preliminary experiments indicate that all the conclusions in this paper continue to hold for more finely tuned parameter settings.

A. Mountain Car

For the mountain car problem [7] we used the same problem description as in the book by Sutton and Barto [10]. This means that the position of the car x and its velocity dx are updated as follows:

$$\begin{aligned} x_{t+1} &= x_t + dx_{t+1} \ , \\ dx_{t+1} &= dx_t + 0.001a_t - 0.0025cos(3x_t) \ . \end{aligned} \tag{17}$$

Furthermore, the position is bounded to $[-1.2, 0.5]$ where when it drops below -1.2 it is reset to -1.2 with zero velocity and the episode is considered a success when a position higher than 0.5 is reached.

The algorithms receive a reward of -1 on every time step, except when an episode ends in a success. Then a reward of $+100$ is received. If no success is obtained for 500 time steps, the episode is considered a failure and the episode also ends. Whenever an episode ends, the car is reset to the bottom of the track with zero velocity. The state vector given to the algorithms consists of the position and the velocity of the car. The available actions are -1, 0 and 1.

B. Cart Pole

For the cart pole task, we use commonly used system dynamics [1], [12]. The cart weighs 1.0 kg and the algorithms all push the cart with an integer amount of Newtons from the interval $[-10, 10]$. The pole is 1 m long and weighs 0.1 kg. The time steps between consecutive actions are 0.02 seconds. An episode is ended when either the cart hits one of the walls at 2.4 m in each direction, or when the pole drops further than 12 degrees from its upright position. An episode is considered a success and also ends when the pole is balanced for at least 40 seconds.

When an episode ends, the cart is reset at the center of the track with the pole tilted randomly between 0 and 3 degrees either to the left or to the right. The dynamics do not include friction, but are realistic in the other aspects. The state vector given to the algorithms consists of the position and velocity of the cart and the angle and angular velocity of the pole. On every time step the algorithms receive a reward of $+1$, except when an episode ends with a failure. Then a reward of -1 is received.

An important difference between the cart pole and the mountain car is that a poorly performing algorithm will get a lot of meaningful feedback on the cart pole, since the pole will drop quickly and will allow the algorithm to get information about good and bad situations to be in. In the mountain car such an algorithm will only observe rewards of -1 for all time steps, which by themselves do not carry much information. We will see that the two tasks are sufficiently different to require quite different settings of the parameters for the algorithms to reach the best performances.

IV. RESULTS

In this section we present the results of the different algorithms with various parameter settings on the two tasks. We will begin with the mountain car and will then discuss the results on the cart pole task. The results are obtained by running offline tests without exploration as well as observing the online performance of the algorithms during the training. We will look at performance measures that indicate how quickly the algorithms reach good solutions as well as measures that indicate how good the final solution is. We note that the best combination of exploration and learning rates for any algorithm can differ for all types of performance measures.

In the mountain car problem, we are interested in the number of steps before the goal is reached and we will show the results of the average number of steps during training, as well as the number of steps of the final solution found by each algorithm. In the cart pole problem, we will look at the number of seconds before the pole drops or the cart bumps against a wall. To give an indication of this, we will use the average number of failures per second to balance the pole. We will also look at the final performance after 2000 seconds of training. Also interesting for the cart pole is how much time it takes before the algorithms can balance the pole. For this, we looked at the number of seconds before the pole was balanced for the first time for at least 40 consecutive seconds.

After this, we include the results for an experiment to test how robust the algorithms are when some of the actions are no longer available. For this, we train the algorithms for 2000 seconds on the cart pole task, using an action space consisting of all the integer actions from the interval $[-10, 10]$. Then we exclude some actions from this interval and see how long it takes for each algorithm to adapt to the new situation.

We now continue with the results. In the tables containing the results also the parameters that were used to obtain these are given. The exploration parameter is σ in the case of Cacla and τ in the case of all the other algorithms, since as mentioned before the exploration methods corresponding to these parameters performed best. We show the mean performance over all 300 trials, which in the mountain car is the mean number of steps to reach the goal and in the cart pole is the mean number of times the pole is not balanced per second. In both cases, lower numbers are better and the algorithms are sorted by their performance. We also give the standard error on the means.

A. Mountain Car

Training in this task was limited to 1000 episodes, each of which lasted a maximum of 500 time steps. After every episode, a test episode was run without exploration and without updating any of the algorithms. This problem is arguably less suited to Cacla than the cart pole problem, because even though we can still assume an underlying continuous action space, there are only two actions of real interest: driving as fast to the left as possible and driving as fast to the right as possible. This means the ability of Cacla to finely tune its actor output becomes less important.

The average results over all the training episodes are given in Table II. We see that in offline performance Acla performs significantly better than Cacla, which in turn significantly outperforms the rest. In online performance, Cacla manages to outperform all the other algorithms with QV-learning a relatively close second. We see that in both cases the algorithms using state values outperform Q-learning and Sarsa by a significant margin.

TABLE II

AVERAGE NUMBER OF STEPS UNTIL GOAL IS REACHED: MEAN PERFORMANCE DURING TRAINING. AVERAGES OVER 300 TRIALS.

Offline performance					
	α	β	**exploration**	**mean**	**std error**
Acla	0.01	0.05	100.0	179.2	0.5
Cacla	0.00005	0.05	100.0	282.8	2.4
QV	0.00002	0.1	0.002	363.6	4.5
AC	0.001	0.2	1.0	456.4	2.1
Sarsa	0.0005	-	0.01	471.5	2.8
R	0.0002	0.01	100.0	471.6	3.2
Q	0.00002	-	1.0	479.8	3.1

Online performance					
	α	β	**exploration**	**mean**	**std error**
Cacla	0.0002	0.05	20.0	306.0	2.0
QV	0.00001	0.2	0.001	351.5	4.6
AC	0.001	0.2	1.0	428.4	3.3
Acla	0.0002	0.002	0.01	430.2	5.3
Sarsa	0.0005	-	0.01	461.2	3.9
Q	0.001	-	0.01	478.5	2.6
R	0.0001	0.005	0.01	483.4	2.5

TABLE III

AVERAGE NUMBER OF STEPS UNTIL GOAL IS REACHED: FINAL PERFORMANCE AFTER TRAINING. AVERAGED OVER 300 TRIALS.

Offline performance					
	α	β	**exploration**	**mean**	**std error**
Acla	0.005	0.05	100.0	132.2	0.9
Cacla	0.00002	0.05	100.0	187.3	3.1
QV	0.00002	0.1	0.002	273.4	9.2
AC	0.0005	0.2	1.0	416.4	8.5
R	0.01	0.05	2.0	461.5	5.3
Sarsa	0.0002	-	0.01	462.7	5.9
Q	0.00002	-	1.0	472.1	5.1

Online performance					
	α	β	**exploration**	**mean**	**std error**
Cacla	0.00005	0.02	50.0	222.4	4.4
QV	0.00002	0.1	0.002	309.9	7.6
AC	0.001	0.2	2.0	345.3	7.1
Sarsa	0.0002	-	0.01	453.6	6.0
Acla	0.0005	0.001	0.01	455.0	6.4
R	0.0001	0.005	0.01	475.2	4.0
Q	0.0001	-	0.01	495.9	1.2

Interestingly, we see that Cacla uses very large values for its Gaussian exploration in this problem. This is because for the mountain car only the two extreme actions are relevant. The idea is to drive as fast as you can to the left and then as fast as you can to the right. Cacla can learn to output actions far above 1 and far below -1, because the outputs are rounded to these actions anyway. Even in the online results, Cacla can perform well with very high exploration, while most other algorithms use quite low temperatures that translate into little exploration. This is also why there is such a large difference between the offline and online performance of Acla. The offline performance uses a high temperature, resulting in much exploration. This apparently allows Acla to learn good offline policies, but such a high temperature will not lead to good online performance.

The final performances after 1000 episodes are given in Table III. This table shows the performance of the last training and the last testing episode. In the offline case Acla reached the best performance, followed by Cacla. In

TABLE IV

AVERAGE RESULTS OVER THE WHOLE TRAINING RUN CONSISTING OF 2000 S OF TRAINING AND 4000 S OF TESTING. THE AVERAGE NUMBER OF FAILURES PER SECOND IS SHOWN FOR BOTH ONLINE AND OFFLINE PERFORMANCE. AVERAGED OVER 300 TRAILS.

Offline performance					
	α	β	**exploration**	**mean**	**std error**
Cacla	0.0005	0.002	10.0	0.107	0.003
AC	0.002	0.002	10.0	0.113	0.002
Acla	0.05	0.005	0.1	0.131	0.003
R	0.02	0.002	0.1	0.335	0.005
QV	0.005	0.005	1.0	0.229	0.005
Q	0.01	-	0.5	0.230	0.004
Sarsa	0.01	-	0.5	0.332	0.005
Online performance					
	α	β	**exploration**	**mean**	**std error**
Acla	0.05	0.005	0.1	0.103	0.002
Cacla	0.0005	0.002	20.0	0.141	0.003
AC	0.01	0.002	5.0	0.148	0.002
R	0.02	0.002	0.1	0.170	0.003
QV	0.005	0.005	1.0	0.206	0.004
Sarsa	0.01	-	0.5	0.316	0.005
Q	0.01	-	0.5	0.320	0.006

TABLE V

AVERAGE AMOUNT OF SECONDS BEFORE THE START OF THE FIRST PERFECT TEST AND TRAINING RUN. AVERAGED OVER 300 TRAILS.

Offline performance					
	α	β	**exploration**	**mean**	**std error**
Cacla	0.01	0.005	5.0	181.1	10.0
QV	0.005	0.005	1.0	232.0	7.2
Acla	0.05	0.005	0.1	236.4	8.3
R	0.01	0.0005	0.5	265.8	9.2
AC	0.01	0.005	2.0	359.6	13.6
Q	0.005	-	1.0	385.5	12.0
Sarsa	0.01	-	0.5	453.6	22.4
Online performance					
	α	β	**exploration**	**mean**	**std error**
Acla	0.05	0.005	0.1	191.0	5.7
Cacla	0.005	0.002	5.0	281.6	5.2
QV	0.005	0.005	1.0	340.8	9.6
R	0.01	0.0005	0.5	374.4	10.0
AC	0.01	0.005	2.0	439.8	13.5
Q	0.01	-	0.02	638.1	27.2
Sarsa	0.005	-	0.5	665.6	22.5

online performance Cacla outperforms the other algorithms, showing a similar picture as in the average results.

On a whole the results show that Cacla is a very reasonable choice to solve problems similar to the mountain car problem, despite the low number of available actions. Depending on the performance measure that you consider, Cacla performs best or second best to Acla. However, if one would combine the online and offline performance measures, Cacla is a better choice than Acla, because of the better performance in the online results. In general, algorithms that use state values outperform algorithms that use state-action values for all performance measures in this task.

We will now continue with the cart pole task, to see how many of the conclusions on the mountain car task continue to hold in this different problem.

B. Cart Pole

Table IV shows the average results of the algorithms on the cart pole task. For the offline performance, a test run of 40 seconds using the policy found so far was run without exploration after each 20 seconds of training. For the online performance simply the performance during training, including exploration, was used.

In the offline average results over all test runs, we see the algorithms fall more or less into three groups. The lowest performance comes from the group with the conventional state-action value based algorithms: Q-learning and Sarsa. Performing somewhat better are the state-action value based algorithms that use extra information in the form of average rewards or state values: R-learning and QV-learning. The best performance is observed for the algorithms that use Actor Critic methods: Actor Critic, Acla and Cacla. In a sense, these algorithms do not try to learn something about the whole state-action space, but are more focused on just finding the optimal policy. In the cart pole task this apparently translates into good average performance.

In the online results we observe more or less the same separation. We do note that the difference between the best performing algorithm and the worst performing algorithm is smaller than in the offline results. Apparently, most algorithms do not suffer from some exploratory steps, since except for Actor Critic and Cacla all algorithms in fact reach slightly better results in the online performance.

Apart from the average results during training, there are two other performance measures for the cart pole task that can be considered interesting. The first is the performance measure that measures how quickly each algorithm reached a perfect run for the first time. The first perfect offline run is defined as the first test run of 40 seconds in which the pole does not fall down and the cart does not bump against either side of the track. Similarly, the first perfect online run is defined as the first period of 40 seconds during training in which no such failure is observed. Table V shows the results of this measure.

We see that although Actor Critic reaches similar results as Acla and Cacla in average performance, the latter two algorithms reach a flawless performance a lot faster. In offline performance Cacla outperforms all the other algorithms, while in online performance the same holds for Acla. In both cases, the difference is statistically significant, as can be observed from the low standard error compared to the actual difference. Interestingly, Actor Critic only manages to outperform Q-learning and Sarsa, which again finish last. For this measure we see that for all algorithms except Acla, the online performance is worse than the offline performance by a considerable margin.

The last performance measure we consider measures the final performance, sorted by the percentage of successful runs. This performance measures captured how good the policy is that is found by each algorithm after 2000 seconds of training. This tells us how good the solution actually is that is found by each algorithm. In Table VI the percentage of successful runs is shown of the final 40 seconds of training and for the offline test run of 40 seconds after training has ended. The most interesting result from this table is that the performance of Cacla is apparently more reliable

TABLE VI

AVERAGE RESULTS OVER THE LAST TRAINING RUN CONSISTING OF 40 S OF TESTING WITHOUT EXPLORATION. AVERAGE FAILURES PER SECOND AND PERCENTAGE OF FLAWLESS RUNS OVER 300 TRIALS IS SHOWN.

Offline performance					
	α	β	**exploration**	**mean**	success
Cacla	0.0005	0.001	10.0	0.003	98.8 %
Acla	0.002	0.001	10.0	0.019	91.7 %
AC	0.002	0.002	0.5	0.037	82.6 %
QV	0.02	0.001	0.2	0.022	80.5 %
R	0.002	0.002	1.0	0.089	68.5 %
Sarsa	0.002	-	1.0	0.156	55.7 %
Q	0.05	-	0.01	0.131	54.8 %

Online performance					
	α	β	**exploration**	**mean**	success
Cacla	0.002	0.002	0.2	0.005	97.7 %
Acla	0.1	0.002	0.05	0.011	95.2 %
AC	0.002	0.002	5.0	0.012	87.7 %
R	0.02	0.001	0.1	0.030	79.3 %
QV	0.005	0.005	1.0	0.117	48.2 %
Q	0.005	-	0.2	0.128	37.6 %
Sarsa	0.005	-	0.01	0.150	35.5 %

over different trials, since it reaches a higher percentage of successful runs than all the other algorithms, with Acla being the best of the rest.

C. Cart Pole with Removed Actions

For the following results we removed some of the available actions after the algorithms were trained on the cart pole task. Three scenarios were tested, where after 2000 seconds of training a subset of the actions was made unavailable. In the first scenario *Even*, all the odd positive and negative integer forces were removed, leaving the even integers. In the second scenario *Bang*, we removed all actions except -10, 0 and 10, essentially testing the performance of the algorithms when only the actions of a bang-bang controller were allowed. For the third scenario *No Extreme* we only removed the most extreme options -10 and 10, since we observed that these were often used by all algorithms. In the third scenario therefore 19 of the original 21 actions are still available. For all discrete algorithms the state action values corresponding to the removed actions are simply not considered anymore and for Cacla the output of the actor gets scaled to the closest available action from the new action set.

After removing the actions, we first measured the performance with a test run of 40 seconds without exploration or updating the algorithms. The results are given in Table VII. The parameter settings are the same that were used for the best final offline performance after 2000 seconds of training, as shown in Table VI.

We see that Cacla manages to adapt very successfully to the changed situations. The percentage of successful test runs remain high and the average number of failures per second remains very small.

It is very interesting to view the differences between the three scenarios. We see that on average performance goes down significantly for all algorithms except Cacla when the odd actions are removed. However, Acla and Actor Critic still manage to reach a perfect run without additional training in more than half of the 300 trials. However, in the scenario

TABLE VII

MEAN AMOUNT OF FAILURES PER SECOND OF THE FIRST TEST RUN AFTER REMOVING ODD ACTIONS (*Even*), ALL EXCEPT -10, 0 AND 10 (*Bang*), OR JUST ACTIONS -10 AND 10 (*No Extreme*). AVERAGE MEANS AND PERCENTAGES OVER 300 TRIALS, SORTED BY PERFORMANCE BEFORE REMOVING THE ACTIONS.

	Even		*Bang*		*No Extreme*	
	mean	**success**	**mean**	**success**	**mean**	**success**
Cacla	0.002	98.8 %	0.004	96.0 %	0.003	98.5 %
Acla	0.117	77.3 %	0.209	44.1 %	1.028	3.8 %
AC	0.068	61.5 %	0.154	45.9 %	1.920	0.0 %
QV	2.967	11.8 %	2.894	0.6 %	4.136	0.0 %
R	2.071	22.2 %	3.702	2.5 %	4.187	0.1 %
Sarsa	3.512	10.7 %	2.411	0.2 %	4.107	0.0 %
Q	3.200	11.5 %	3.334	1.5 %	4.284	0.1 %

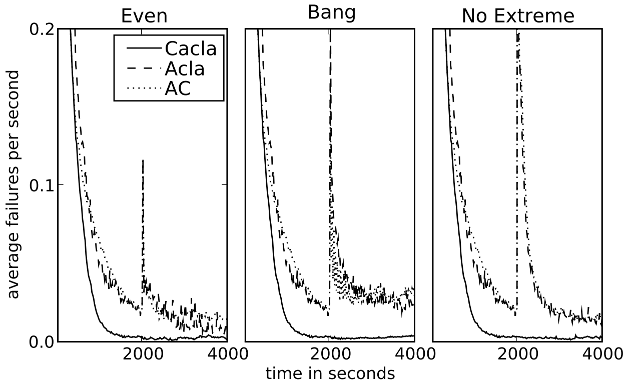

Fig. 1. The offline performance measured in average failures per second for Cacla, Acla and Actor Critic. After 2000 s the odd actions (top panel, scenario *Even*), all actions except -10, 0 and 10 (middle panel, scenario *Bang*), or -10 and 10 (bottom panel, scenario *No* 10 are removed.

where we only removed the actions corresponding to -10 N and 10 N performance of all discrete algorithms drops considerably. This is due to the fact that almost all policies found by the algorithms in the first 2000 seconds use these actions regularly. Using its ability to generalize, Cacla will immediately push with 9 N where it used to push with 10 N, but the other algorithms have to relearn which action then to take, since they regard all actions as qualitatively different options and apparently have not learned that 9 N is the second best option when 10 N becomes unavailable.

To get a better intuition of what happens to the performances, we include Fig. 1 with the offline performances of the best three algorithms: Cacla, Acla and Actor Critic. As explained above, after 2000 seconds actions are removed from the action space. In the left panel, we see that if the odd actions are removed, Acla and Actor Critic get a temporary setback, but quickly regain former performance levels. However, in the right panel we see that if the actions corresponding to -10 N and 10 N are removed they recover much slower. Note that all algorithms that are not shown in the figure perform much worse, as can be deducted from Table VII.

The results in this subsection show that Cacla can easily adapt to changing action spaces when some of the actions are removed. We expect this to also be the case if the action space is changed in other ways. It is non trivial how to adapt a conventional reinforcement learning algorithm such as Q-learning when for instance a different set of available actions is chosen from the continuous underlying action space, but Cacla can still simply use its generalization property and adapt with little problems as long as the actions come from the same range as the actions used for training. This is a useful property, also because often one will want to simulate a real-world problem and train an algorithm on this simulation. If then the actual problem turns out to use a different action space than the simulation, the training can still be useful. For completeness, we note that the online results that are not shown in this subsection are very similar to the offline results.

V. Conclusion

In this paper we compared the performance of the continuous action algorithm Cacla with several discrete reinforcement learning algorithms: Q-learning, Sarsa, Actor Critic, QV-learning, R-learning and a new version of the Acla algorithm. The performance was tested on discrete action MDPs, modeling a cart pole task and a mountain car task. We observed that the best performance was reached by either Cacla or Acla in both problems for all performance measures we looked at. Both online and offline performance was measured and in both cases Cacla performs well.

We also observed that with almost all performance measures we discussed, algorithms that in some way make use of the state values outperform the conventional algorithms Q-learning and Sarsa that only make use of state-action values. These algorithms are widely used, so it is interesting to note that their performance did not match the best performing algorithms in this paper. Also, it is quite easy to extend state values with eligibility traces to speed up learning. Although we have not tested this in the experiments in this paper, the expectation is that the algorithms using state values will then profit even more of a performance gain compared to the state-action based algorithms. This hypothesis should be tested in future research.

Perhaps the most important result in this paper is the observation that Cacla can easily handle the removal of some of the possible actions in the cart pole task. While the other algorithms had to spend some time learning new policies with the changed action spaces, Cacla could easily adapt by using its underlying continuous action space. It is easy to think of scenarios where this can be important. For instance, consider an application where a discrete algorithm such as Q-learning is trained on a simulation with a set of possible actions. Now suppose that the actual problem involves actions that are slightly different or a different number of possible actions. It is then non-trivial how the learned policy can be used on the actual problem, making the simulation less useful. On the other hand, Cacla should be able to adapt to the new set of actions easily, whether there are more, less or different actions available, as long as the available actions can be thought of as lying on the same continuous underlying action space as the ones that were used for training.

The idea used for the Cacla algorithm can not only be applied in the reinforcement learning domain. It would be interesting to see if similar good results can be obtained in classification, if the classes can also be considered a subset of a continuous underlying class. For instance, this approach seems promising for classification problems where the classes correspond to rankings, since here it is simple to assume a continuous underlying space, even if the actual space is not continuous.

Some reinforcement learning problems have inherently discrete actions. In such problems, is may be harder to apply the paradigm of an underlying continuous space. However, we have shown that the novel Acla algorithm performs well on the problems in this paper. Therefore, it would be interesting to see if Acla also outperforms many of the other available algorithms in other problems. Also extensions to batch learning can easily be made. For instance, one could construct a Neural Fitted Acla Iteration algorithm similar to the Neural Fitted Q Iteration algorithm [4].

References

[1] A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13:834–846, 1983.

[2] Sridhar Mahadevan. Average reward reinforcement learning: Foundations, algorithms, and empirical results. *Machine Learning*, 22:159, 1996.

[3] Martin Riedmiller. Neural fitted Q iteration - first experiences with a data efficient neural reinforcement learning method. In João Gama, Rui Camacho, Pavel Brazdil, Alípio Jorge, and Luís Torgo, editors, *Proceedings of the 16th European Conference on Machine Learning (ECML'05)*, pages 317–328. Springer, 2005.

[4] G.A. Rummery and M. Niranjan. On-line Q-learning using connectionist sytems. Technical Report CUED/F-INFENG-TR 166, Cambridge University, UK, 1994.

[5] A. Schwartz. A reinforcement learning method for maximizing undiscounted rewards. In *Machine Learning: Proceedings of the Tenth International Conference*, pages 298–305. Morgan Kaufmann, Amherst, MA, 1993.

[6] S. P. Singh and R. S. Sutton. Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22:123–158, 1996.

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

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

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

[10] H. van Hasselt and M. A. Wiering. Reinforcement learning in continuous action spaces. In *Proceedings of the IEEE International Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, pages 272–279, 2007.

[11] C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, England, 1989.

[12] A. P. Wieland. Evolving neural network controllers for unstable systems. In *International Joint Conference on Neural Networks*, volume 2, pages 667–673, Seattle, 1991. IEEE, New York.

[13] M. A. Wiering. QV(λ)-learning: A new on-policy reinforcement learning algorithm. In D. Leone, editor, *Proceedings of the 7th European Workshop on Reinforcement Learning*, pages 17–18, 2005.

[14] M. A. Wiering and H. van Hasselt. Two novel on-policy reinforcement learning algorithms based on TD(λ)-methods. In *Proceedings of the IEEE International Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, pages 280–287, 2007.