Actor-Critic Control with Reference Model Learning

Ivo Grondman [*] **Maarten Vaandrager Lucian Buşoniu** [*]
Robert Babuška [*] **Erik Schuitema** [*]

[*] *Delft Center for Systems and Control, Faculty 3mE, Delft University of Technology, Mekelweg 2, 2628 CD Delft, The Netherlands (e-mail: {i.grondman, i.l.busoniu, r.babuska, e.schuitema}@tudelft.nl)*

Abstract: We propose a new actor-critic algorithm for reinforcement learning. The algorithm does not use an explicit actor, but learns a reference model which represents a desired behaviour, along which the process is to be controlled by using the inverse of a learned process model. The algorithm uses Local Linear Regression (LLR) to learn approximations of all the functions involved. The online learning of a process and reference model, in combination with LLR, provides an efficient policy update for faster learning. In addition, the algorithm facilitates the incorporation of prior knowledge. The novel method and a standard actor-critic algorithm are applied to the pendulum swingup problem, in which the novel method achieves faster learning than the standard algorithm.

Keywords: learning control, reinforcement learning, actor-critic methods, local linear regression

1. INTRODUCTION AND RELATED WORK

Many processes in industry can benefit from control algorithms that learn to optimise a certain cost function. Reinforcement learning (RL) is such a learning method. The user sets a certain goal by specifying a suitable reward function for the RL controller. The RL controller then learns to maximise the cumulative reward received over time (the value function) in order to reach that goal. However, the controller typically starts learning without any knowledge and has to improve through trial and error. Because of this, the process goes through a long period of unpredictable and potentially damaging behaviour. This is usually unacceptable in industry, especially if a near-optimal controller is already available. The long period of trial and error learning must be considerably reduced for RL controllers to become useful in practice. In this paper, we introduce a novel algorithm that employs an efficient policy update which considerably reduces the learning time compared to standard actor-critic methods and also allows for inclusion of prior knowledge.

Actor-critic techniques are a class of RL methods which learn a separate actor and critic function. The critic is the value function approximator, and the actor is the policy approximator. The novel algorithm replaces the policy by learning a reference model which maps states to subsequent desired states. Together with the inverse of a learned process model, this is used to calculate the inputs to the system. Hence, the process model is not used to generate simulated experiences as most model-learning RL algorithms do (Sutton, 1992; Moore and Atkeson, 1993; Kuvayev and Sutton, 1996), but provides information on how to reach a desired state, given by the reference model.

The novel algorithm introduced here uses Local Linear Regression (LLR) as the function approximator. Memory-based learning methods, such as LLR and case-based reasoning (Gabel and Riedmiller, 2005), have successfully been applied to RL before, but mostly as an approximator for the value function. Our algorithm is similar to "learning from relevant trajectories" (Atkeson and Schaal, 1997), in which LLR is used to learn the process model of a robotic arm holding a pendulum, which is then employed to control the arm along a demonstrated trajectory that effectively swings up the pendulum. The main difference is that we do not make use of a demonstrated trajectory, but use a reference model which is learned and updated online.

2. REINFORCEMENT LEARNING

The RL problem can be described as a Markov decision process (MDP). In this paper, we use RL in a deterministic setting and hence we start with the deterministic MDP description. The MDP is defined by the tuple $M(X, U, f, \rho)$ where X is the state space, U is the action space, $f : X \times U \mapsto X$ is the state transition function and $\rho : X \times U \mapsto \mathbb{R}$ is the reward function.

The process to be controlled is described by the state transition function $f : X \times U \mapsto X$, which returns the state x_k that the process reaches from state x_{k-1} after applying action u_{k-1}. After each transition, the controller receives a scalar reward $r_k \in \mathbb{R}$, given by the reward function $r_k = \rho(x_{k-1}, u_{k-1})$. The actions are chosen according to the policy $\pi : X \mapsto U$. The goal in RL is then to find a policy, such that a discounted sum of future rewards is maximised. This sum (also called the return) is stored in a value function $V^\pi : X \mapsto \mathbb{R}$, which is defined as:

$$V^\pi(x) = \sum_{j=0}^{\infty} \gamma^j r_{k+j+1} \qquad \text{with } x_k = x \tag{1}$$

where $\gamma \in [0, 1)$ is the discount factor. The superscript π indicates that this is the sum of rewards collected when following the policy π, i.e. we have $u_k = \pi(x_k)$.

In continuous (or infinite discrete) state and action spaces, it is necessary to replace the exact value function V^π and the exact policy π with function approximators. This can be facilitated by using actor-critic methods.

3. ACTOR-CRITIC REINFORCEMENT LEARNING

Actor-critic techniques were introduced in Barto et al. (1983), and have been investigated often since then, e.g. in Konda and Tsitsiklis (2003); Berenji and Vengerov (2003); Peters and Schaal (2008). The actor-critic method is characterised by learning separate functions for the actor and the critic. The use of an actor allows for gradient-based policy updates which makes it possible to easily use continuous action spaces (Sutton et al., 2000).

In this paper we use a temporal difference based actor-critic method as our baseline to compare our new method to. We refer to this baseline as the standard actor-critic (S-AC) algorithm. Denote the approximate value function parameterised by θ with $V(x, \theta)$ and the approximate policy parameterised by ϑ with $\pi(x, \vartheta)$. The temporal difference error is defined as (Sutton and Barto, 1998):

$$\delta_k = r_k + \gamma V(x_k, \theta_{k-1}) - V(x_{k-1}, \theta_{k-1}) \tag{2}$$

Using the temporal difference, the gradient-descent update rule for the critic is:

$$\theta_k = \theta_{k-1} + \alpha_c \delta_k \left.\frac{\partial V(x,\theta)}{\partial \theta}\right|_{\substack{x=x_{k-1}\\ \theta=\theta_{k-1}}} \tag{3}$$

where $\alpha_c \in [0, 1]$ is the learning rate of the critic.

Using (3) to update the critic results in a one-step backup, whereas the reward received is often the result of a series of steps. Eligibility traces offer a quicker way of assigning credit to states visited earlier. The eligibility trace for a certain state x at time k is denoted[1] with $e_k(x)$:

$$e_k(x) = \begin{cases} 1 & \text{if } x = x_k \\ \lambda\gamma e_{k-1}(x) & \text{otherwise} \end{cases}$$

The trace decays with time by a factor $\lambda\gamma$, with $\lambda \in [0, 1)$ the trace decay parameter. This makes more recently visited states more eligible for receiving credit. All states along the trajectory now influence the update of θ according to the following equation:

$$\theta_k = \theta_{k-1} + \alpha_c \delta_k \sum_{x \in \mathcal{X}_v} \frac{\partial V(x,\theta)}{\partial \theta} e_k(x)$$

where $\mathcal{X}_v$ denotes the set of states visited during the current trial. The use of eligibility traces speeds up the learning considerably.

Reinforcement learning requires the use of exploration to keep trying new, possibly better, actions in the states encountered. With exploration, the control action u_k is different from the action $\pi(x_k, \vartheta_{k-1})$ indicated by the policy. This can be achieved by perturbing the latter with a zero mean random exploration term Δu_k:

$$u_k = \pi(x_k, \vartheta_{k-1}) + \Delta u_k$$

[1] Note the slight abuse of notation here. If the state space X is continuous, some mechanism has to be introduced such that there only exists a finite number of eligibility traces to update.

When the exploration Δu_k leads to a positive temporal difference, the policy is adjusted towards this perturbed action. Conversely, when δ_k is negative, the policy is adjusted away from this perturbation. This leads to the following update rule for the actor:

$$\vartheta_k = \vartheta_{k-1} + \alpha_a \delta_k \Delta u_{k-1} \left.\frac{\partial \pi(x,\vartheta)}{\partial \vartheta}\right|_{\substack{x=x_{k-1}\\ \vartheta=\vartheta_{k-1}}} \tag{4}$$

where $\alpha_a \in [0, 1]$ is the learning rate of the actor. The temporal difference is interpreted as a correction of the predicted performance, so that if the temporal difference is positive, the obtained performance is considered better than the predicted one.

The full implementation of the S-AC algorithm is shown in Algorithm 1.

Algorithm 1 Standard Actor-Critic (S-AC)

```
Input: $\gamma$, $\lambda$, $\alpha_c$, $\alpha_a$
 1: $e_0(x) = 0 \quad \forall x$
 2: Initialise $x_0$, $\theta_0$ and $\vartheta_0$
 3: Apply input $\pi(x_0, \vartheta_0) + \Delta u_0$
 4: $k \leftarrow 1$
 5: loop
 6:     Choose $\Delta u_k$ at random
 7:     Measure $x_k$, $r_k$
 8:     $u_k \leftarrow \pi(x_k, \vartheta_{k-1}) + \Delta u_k$
 9:     Apply $u_k$
10:     $\delta_k \leftarrow r_k + \gamma V(x_k, \theta_{k-1}) - V(x_{k-1}, \theta_{k-1})$
11:     $e_k(x) = \begin{cases} 1 & \text{if } x = x_k \\ \lambda\gamma e_{k-1}(x) & \text{otherwise} \end{cases}$
12:     $\theta_k \leftarrow \theta_{k-1} + \alpha_c \delta_k \sum_{x \in \mathcal{X}_v} \frac{\partial V(x,\theta)}{\partial \theta} e_k(x)$
13:     $\vartheta_k \leftarrow \vartheta_{k-1} + \alpha_a \delta_k \Delta u_{k-1} \frac{\partial \pi(x,\vartheta)}{\partial \vartheta}$
14:     $k \leftarrow k + 1$
15: end loop
```

4. LOCAL LINEAR REGRESSION

The algorithm presented in this paper uses Local Linear Regression (LLR) as a function approximator. LLR is a non-parametric memory-based method for approximating nonlinear functions. Memory-based methods are also called case-based, exemplar-based, lazy, instance-based or experience-based (Wettschereck et al., 1997; Wilson and Martinez, 2000). It has been shown that memory-based learning can work in RL and can quickly approximate a function with only a few observations (Gabel and Riedmiller, 2005). This is especially useful at the start of learning.

The main advantage of memory-based methods is that the user does not need to specify a global structure or predefine features for the (approximate) model. Instead of trying to fit a global structure to observations of the unknown function, LLR simply stores the observations in a memory. A stored observation is called a sample $s_i = [x_i^T \mid y_i^T]^T$ with $i = 1, \dots, N$. One sample s_i is a column vector containing the input data $x_i \in \mathbb{R}^n$ and output data $y_i \in \mathbb{R}^m$. The samples are stored in a matrix called the memory M with size $(n+m) \times N$ whose columns each represent one sample.

When a query x_q is made, LLR uses the stored samples to give a prediction $\hat{y}_q$ of the true output y_q. The prediction is computed by finding a local neighbourhood of x_q in the samples stored in memory. This neighbourhood is found by applying a weighted distance metric d_i (e.g. the 1-norm or 2-norm) to the query point x_q and the input data x_i of all samples in M. The weighting is used to scale the inputs x and has a large influence on the resulting neighbourhood and thus on the accuracy of the prediction.

By selecting a limited number of K samples with the smallest distance d, we create a subset $\mathcal{K}(x_q)$ with the indices of nearest neighbour samples. Only these K nearest neighbours are then used to make a prediction of $\hat{y}_q$. The set $\mathcal{K}_+(x_q)$ in the pseudocode is $\mathcal{K}(x_q)$, extended with the index where the sample representing x_q was inserted. The prediction is computed by fitting a linear model to these nearest neighbours. Applying the resulting linear model to the query point x_q yields the predicted value $\hat{y}_q$.

First the matrices X and Y need to be constructed using the K nearest neighbour samples:

$$X = \begin{bmatrix} x_1 & x_2 & \cdots & x_K \\ 1 & 1 & \cdots & 1 \end{bmatrix} \qquad Y = [\, y_1 \; y_2 \; \cdots \; y_K \,]$$

The last row of X allows for a bias on the output, making the model affine instead of truly linear.

The X and Y matrices form an over-determined set of equations for the model parameter matrix $\beta \in \mathbb{R}^{m \times (n+1)}$:

$$Y = \beta X$$

and can be solved (for example) by the method of least squares using the right pseudo inverse of X:

$$\beta = Y X^T (X X^T)^{-1}$$

At the start of a trial, the matrices X and Y do not yet form a fully determined set of equations. In this case, there are infinitely many solutions and β is chosen as the solution with the smallest norm.

Finally, the model parameter matrix β is used to compute the prediction for the query x_q:

$$\hat{y}_q = \beta x_q$$

As a result, the globally nonlinear function is approximated locally by a linear function.

Memory-based methods directly incorporate new observations, which makes it possible to get a good local estimate of the function after incorporating only a few observations. Note that if every observation were stored, the memory would grow indefinitely and so would the computational effort of finding $\mathcal{K}(x_q)$. One has to apply memory management to keep the memory from growing past a certain size. The exact description of the memory management algorithm used is outside the scope of this paper.

5. REFERENCE MODEL ACTOR-CRITIC

In this section an actor-critic algorithm is introduced, which uses a novel way of representing the actor. A learned reference model dictates a desired behaviour along which the process should be controlled, by using the inverse of a learned process model. Although it lacks an explicit actor, we refer to this method as Reference Model Actor-Critic (RMAC), as the reference model and process model together play the role of the actor. In the implementation of the algorithm we always use LLR to learn and approximate the functions and models involved.

Samples $s_i = [x_i^T \; u_i^T \mid {x'}_i^T]^T$ are held by the process memory M^{P}, where x' denotes the observed next state, i.e. $x' = f(x, u)$. The process model is updated by replacing samples that are deemed obsolete with new observations. The critic memory M^{C} holds samples $s_i = [x_i^T \mid V_i]^T$, while the reference model memory M^{R} holds samples $s_i = [x_i^T \mid \hat{x}_i^T]^T$, where $\hat{x}$ denotes a desired next state. During the learning process, the critic and reference model memory are not only updated by replacing samples, but also by adjusting the output parts of the nearest neighbour samples s_i that relate to the query point x_q. The method of updating them is explained in more detail later.

RMAC is different from the typical actor-critic methods in the sense that it does not learn a mapping from state x_k to action u_k. Instead it learns a reference model $R(x)$ that maps the state x_k to a desired state $\hat{x}_{k+1}$, i.e. $\hat{x}_{k+1} = R(x_k)$. The process is controlled towards this desired next state by using the inverse of the learned process model $x_{k+1} = \hat{f}(x_k, u_k)$. The reference model $R(x)$ and the inverse process model $u_k = \hat{f}^{-1}(x_k, x_{k+1})$ together act as a policy, by using the relation $u_k = \hat{f}^{-1}(x_k, R(x_k))$. The process model is given by

$$x_{k+1} = \hat{f}(x_k, u_k) = \underbrace{\left[\, \beta_x^{\mathrm{P}} \; \beta_u^{\mathrm{P}} \; \beta_b^{\mathrm{P}} \,\right]}_{\beta^{\mathrm{P}}} \cdot \begin{bmatrix} x_k \\ u_k \\ 1 \end{bmatrix}$$

By replacing x_{k+1} with the desired state $\hat{x}_{k+1}$ given by the reference model and inverting the process model we obtain the action u_k:

$$u_k = ({\beta_u^{\mathrm{P}}}^T \beta_u^{\mathrm{P}})^{-1} {\beta_u^{\mathrm{P}}}^T \cdot \left(R(x_k) - \beta_x^{\mathrm{P}} x_k - \beta_b^{\mathrm{P}} \right)$$

Since a local linear approximation of the process model is used, $f^{-1}(x_k, R(x_k))$ always exists as a linear function is always invertible, unless β_u^{P} is a zero vector, in which case the algorithm returns $u_k = 0$.

We can improve the $R(x)$ by adapting the desired state $\hat{x}$ of the nearest neighbour samples s_i $(i \in \mathcal{K}(x))$ towards higher state-values using the following gradient update rule:

$$\hat{x}_i \leftarrow \hat{x}_i + \alpha \left. \frac{\partial V}{\partial x} \right|_{x=x'} \tag{5}$$

But (5) eventually may lead to an infeasible reference model if the $\hat{x}_i$ are not kept within the reachable set $\mathcal{R}_x$, which is the set of all states that can be reached from the current state x within a single sampling interval:

$$\mathcal{R}_x = \{\, x' \in X \mid \exists u \in U \text{ with } x' = f(x, u) \}$$

It is not straightforward to determine this set because it depends on the current state, the (nonlinear) process dynamics and the action space U.

We approximate $\mathcal{R}_x$ as a convex hull by applying combinations of extremes of U to the learned process model $\hat{f}(x, u)$. The current state x_k and all possible combinations of extremes are put in a matrix U_R. Every column of U_R gives the current state x_k and a combination of maximum and minimum values for u. This way, the matrix U_R has a number of rows equal to the number of inputs of the learned process model $\hat{f}(x, u)$ and a number of columns equal to 2^m, with m the size of vector u. For example, with two input variables U_R would be

$$U_R = \begin{bmatrix} x_k & x_k & x_k & x_k \\ u_{1,\max} & u_{1,\max} & u_{1,\min} & u_{1,\min} \\ u_{2,\max} & u_{2,\min} & u_{2,\max} & u_{2,\min} \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

By applying U_R to the process model we obtain a matrix X_R containing the vertices of the convex hull as column vectors:

$$X_R = \beta^{\mathrm{P}} \cdot U_R$$

Given that we are using a locally linear model of the value function $V(x)$ bounded by a convex hull, we know that the optimum of $V(x)$ must then lie in one of the states found in X_R.[2] Denoting this set of reachable states with $\mathcal{X}_R$, we then calculate which of these states yields the highest value, using the local linear model of the value function:

$$V(x) = \beta^{\mathrm{C}} \cdot \begin{bmatrix} x \\ 1 \end{bmatrix}$$

The state x_r that corresponds to the highest value is then used to update the reference model $R(x)$:

$$x_r = \arg\max_{x \in \mathcal{X}_R} \beta^{\mathrm{C}} \cdot \begin{bmatrix} x \\ 1 \end{bmatrix}$$
$$\hat{x}_i \leftarrow \hat{x}_i + \alpha_r (x_r - \hat{x})$$

Because of the approximation of X_R, the reference model is updated using a desired state x_r that is the result of applying the extremes of u. Despite this, we can still achieve a smooth reference model and a smooth policy by using the learning rate α_r in the update of $R(x)$. However, it is likely that this approximation causes the algorithm to converge to a near-optimal solution at most and a more accurate calculation of the bounds could improve the performance.

In contrast to S-AC, the RMAC improves the reference model using (5) which does not involve random exploration to obtain the parameter increment. Instead, it improves the reference model using locally linear models estimated on the basis of previous experiences. However, random exploration is still needed to reduce the chance of the policy improvement getting stuck in a local optimum. Moreover, it improves the learned process model. The pseudocode for the RMAC method is found in Algorithm 2.

6. EXAMPLE: PENDULUM SWINGUP

To evaluate and compare the performance of our algorithm, we apply it to the task of learning to swing up a simulated inverted pendulum and compare it to the standard algorithm. The swingup task was chosen because it is a low-dimensional, but challenging, highly nonlinear control problem commonly used in RL literature. As the process has two state variables and one action variable it allows for easy visualization of the functions of interest. A picture of this system is shown in Figure 1.

The equation of motion of this system is:

$$J\ddot{\phi} = Mgl\sin(\phi) - \left(b + \frac{K^2}{R}\right)\dot{\phi} + \frac{K}{R}u$$

[2] When the function is nonlinear the optimum can lie inside the hull. With a linear function but nonlinear boundaries it lies on one of the edges.

Algorithm 2 Reference Model Actor-Critic (RMAC)

```
Input: $\gamma$, $\lambda$, $\alpha_c$, $\alpha_r$
 1: Initialise $x_0$, $\hat{x}_1$, $M^{\mathrm{C}}$, $M^{\mathrm{R}}$ and $M^{\mathrm{P}}$
 2: $V_0 = 0$, $\beta^{\mathrm{P}} = 0$
 3: $e_0(s_i) = 0 \quad \forall s_i \in M^{\mathrm{C}}$
 4: $\hat{x}_1 = R(x_0)$
 5: $u_0 \leftarrow \hat{f}^{-1}(x_0, \hat{x}_1) + \Delta u_0$
 6: $k \leftarrow 1$
 7: loop
 8:     Choose $\Delta u_k$ at random
 9:     Measure $x_k$, $r_k$
10:     $\hat{x}_{k+1} = R(x_k)$
11:     $u_k \leftarrow \hat{f}^{-1}(x_k, \hat{x}_{k+1}) + \Delta u_k$
12:     Apply $u_k$
13:     % Update process model
14:     Insert $[x_{k-1}^T \; u_{k-1}^T \mid x_k^T]^T$ in $M^{\mathrm{P}}$
15:     % Update reference model
16:     Select best reachable state $x_r$
17:     Insert $[x_{k-1}^T \mid x_r^T]^T$ in $M^{\mathrm{R}}$
18:     for $\forall i \in \mathcal{K}(x_{k-1})$ of $M^{\mathrm{R}}$ do
19:         $\hat{x}_i \leftarrow \hat{x}_i + \alpha_r(x_r - \hat{x}_k)$
20:     end for
21:     % Update critic
22:     $V_k \leftarrow V(x_k)$
23:     Insert $[x_{k-1}^T \mid V_{k-1}]^T$ in $M^{\mathrm{C}}$
24:     $\delta_k \leftarrow r_k + \gamma V_k - V_{k-1}$
25:     for $\forall s_i \in M^{\mathrm{C}}$ do
26:         $e_k(s_i) = \begin{cases} 1 & \text{if } i \in \mathcal{K}_+(x_{k-1}) \\ \lambda\gamma e_{k-1}(s_i) & \text{otherwise} \end{cases}$
27:         $V_i \leftarrow V_i + \alpha_c \delta_k e_k(s_i)$
28:     end for
29:     $k \leftarrow k + 1$
30: end loop
```

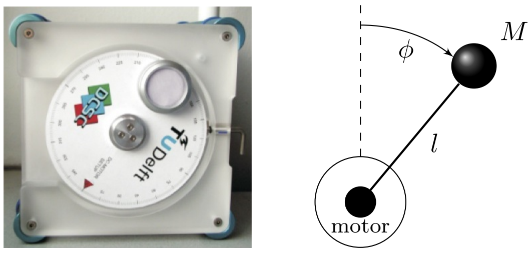

Fig. 1. The inverted pendulum setup.

where ϕ is the angle between the pendulum and the upright position. The model parameters are given in Table 1. The task is to learn to efficiently swing the pendulum from the upside-down position to the upright position and stabilise it in this position. The actuation signal u is

Table 1. Inverted pendulum model parameters

Model parameter	Symbol	Value	Units
Pendulum inertia	J	$1.91 \cdot 10^{-4}$	kgm^2
Pendulum mass	M	$5.50 \cdot 10^{-2}$	kg
Gravity	g	9.81	m/s^2
Pendulum length	l	$4.20 \cdot 10^{-2}$	m
Damping	b	$3 \cdot 10^{-6}$	Nms
Torque constant	K	$5.36 \cdot 10^{-2}$	Nm/A
Rotor resistance	R	9.50	Ω

limited to $u \in [-3, 3]$ V, making it impossible to directly move the pendulum to the upright position. Instead, the controller has to learn to increase the momentum of the pendulum by swinging it back and forth before it can push it up.

A continuous quadratic reward function ρ is used to define the swingup task. This reward function has its maximum in the upright position $[0\ 0]^T$ and quadratically penalises non-zero values of ϕ, $\dot{\phi}$ and u.

$$r_k = -x_{k-1}^T Q x_{k-1} - P u_{k-1}^2$$

with

$$x = \begin{bmatrix} \phi \\ \dot{\phi} \end{bmatrix} \qquad Q = \begin{bmatrix} 5 & 0 \\ 0 & 0.1 \end{bmatrix} \qquad P = 1$$

The standard actor-critic method S-AC and the novel method RMAC are applied to the simulated pendulum swingup problem described above. The algorithms run for 30 minutes of simulated time, consisting of 600 consecutive trials with each trial lasting 3 seconds. The pendulum needs approximately 1 second to swing up with a near-optimal policy. Every trial begins in the upside-down position with zero angular velocity, $x_0 = [\pi\ 0]^T$. With a *learning experiment* we denote one complete run of 600 consecutive trials.

The sum of rewards received per trial is plotted over the time which results in a learning curve. This procedure is repeated for 40 complete learning experiments to get an estimate of the mean and confidence interval of the learning curve.

6.1 Standard Actor-Critic

This section presents the results of applying the S-AC algorithm to the problem described above.

The tunable parameters are set to $\gamma = 0.97$, $\lambda = 0.65$, $\alpha_a = 0.005$ and $\alpha_c = 0.1$. The function approximator used for both actor and critic is tile coding as described in Sutton and Barto (1998). A number of 16 partitions, each consisting of a uniform grid of 7×7 tiles, are used. The partitions are equidistantly distributed over each dimension of the state space.

Exploration is done every third step by randomly perturbing the policy with normally distributed zero mean white noise with standard deviation $\sigma = 1$. The reason for exploring only once every three steps instead of every step is because this allows for large exploratory actions whilst giving the controller time to correct for suboptimal exploratory actions. Large exploratory actions appeared to be beneficial for learning. This can be explained by the fact that the representation of the value function by tile coding is not perfect. There is a persistent error in the approximation causing the temporal difference to continuously vary around a certain level. For small exploratory actions, their contribution to the resulting temporal difference is small compared to the contribution of the approximation error, causing the update of the actor by (4) to be very noisy.

The S-AC algorithm applied to the pendulum swingup task results in a learning curve shown as a dashed line in Figure 2. The method takes about 10 minutes of simulated time on average to converge. One striking characteristic of

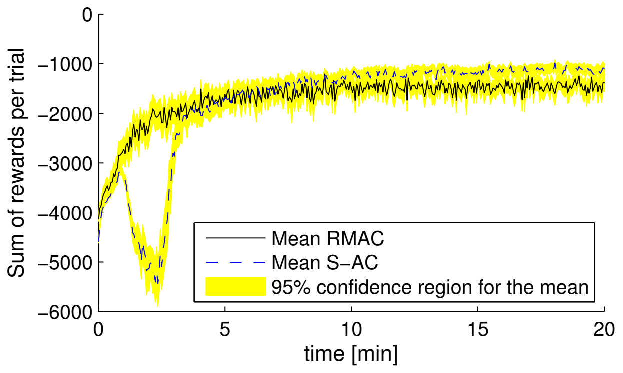

Fig. 2. Learning curves for S-AC and RMAC.

the S-AC learning curve is the short drop in performance in the first minutes. This can be explained by the fact that the value function is initialised to zero which is higher than the true value function. As a result, the algorithm collects a lot of negative rewards before it eventually learns the true value of “bad” states and adapts the actor to avoid these. In order to prevent this initial drop in performance, the value function can be initialised with low values, but this would decrease the overall learning speed as all new unvisited states would initially be assumed to be bad and would be avoided. The performance can be improved by increasing the number of partitions and number of tiles per partition in the tile coding, but this would decrease the learning speed.

The final approximations of the policy $\pi(x)$ after a representative learning experiment is shown in Figure 3.

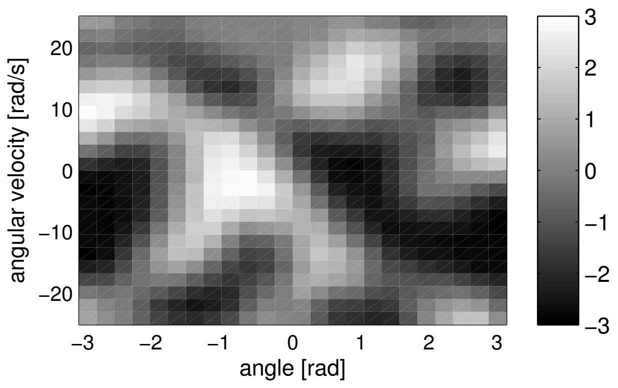

Fig. 3. Final actor $\pi(x)$ for the S-AC algorithm after one learning experiment.

6.2 Reference Model Actor-Critic

The RMAC algorithm was applied using $\gamma = 0.97$, $\lambda = 0.65$, just as before with the S-AC method. The other parameter settings used are shown in Table 2.

Table 2. Parameters for the RMAC method.

	Critic	**Process**	**Reference**
learning rate	0.1	-	0.005
memory size	2000	100	2000
nearest neighbours	20	9	20
input weighting	[1 .1]	[1 .1 1]	[1 .1]

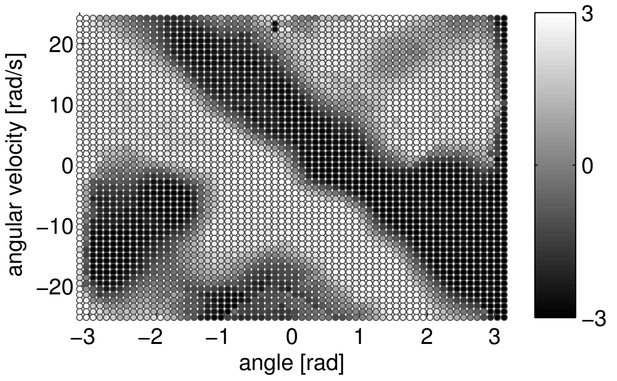

Fig. 4. Actor as given by the composition of the inverse process model with the reference model, evaluated at a rectangular grid of points in the state space.

The learning curve for the pendulum swingup task for the RMAC method is shown as a solid line in Figure 2. Figure 4 shows the actor as generated by the reference model and inverse process model, i.e. it shows the value of the composition $u = \hat{f}^{-1}(x, R(x))$ for a rectangular grid of points in the state space. Although the RMAC method is capable of faster learning, it does develop an actor that is less smooth than the one developed by the S-AC method (Figure 3), which explains the slightly better performance of the latter method at the end of learning.

The RMAC learns very quickly and converges to a good solution of the swingup task. The main reason for the fast learning is the fact that the method starts out by choosing desired states $\hat{x}$ that result from the extremes of u. Desired states that result from the extremes of u also result in large values for u and make the system explore a large part of the state space. This results in a fast initial estimate of the value function, which is beneficial for the learning speed.

7. CONCLUSIONS AND OPEN ISSUES

This paper introduced a novel actor-critic method, which uses LLR as a non-parametric, memory-based function approximator. It has been shown by simulation that this novel method is capable of fast learning. This increased learning speed can be attributed to the novel way in which the policy is updated by using a reference model, as well as to the use of LLR to approximate the functions.

In the pendulum swingup sample above, the memories were initialised as empty, but instead one can initialise them with previous measurements. This makes it easy to incorporate prior knowledge on the process dynamics, near optimal control policy or near optimal behaviour. For example, the reference model can be initialised with samples of the closed loop behaviour. This can be beneficial if the desired behaviour of the system is known but the control policy is yet unknown (which is often the case in supplying prior knowledge by imitation).

LLR seems very promising for use in fast learning algorithms, but a few issues prevent it from being used to its full potential. The first issue is how to choose the correct input weighting, which has a large influence on selecting the most relevant samples for regression. The second issue that has to be investigated more closely is memory management: different ways of scoring samples in terms of age and redundancy and thus deciding when to remove certain samples from the memory will also influence the accuracy of the estimates generated by local linear regression. Another issue is the high computational effort of searching through the memory for nearest neighbour samples. In this paper a simple sorting algorithm was used, but one can reduce the computational burden by using, for instance, k-d trees (Bentley and Friedman, 1979). Finally, the approximation of the reachable subset $\mathcal{X}_R$ may prove insufficiently accurate for more complex control tasks. A more reliable calculation of reachable states is the main improvement that could be made to this method.

REFERENCES

Atkeson, C.G. and Schaal, S. (1997). Robot Learning From Demonstration. In *Proceedings of the 14th International Conference on Machine Learning*, 12–20.

Barto, A.G., Sutton, R.S., and Anderson, C.W. (1983). Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 13(5), 834–846.

Bentley, J.L. and Friedman, J.H. (1979). Data Structures for Range Searching. *ACM Computing Surveys (CSUR)*, 11(4), 397–409.

Berenji, H.R. and Vengerov, D. (2003). A Convergent Actor–Critic-Based FRL Algorithm with Application to Power Management of Wireless Transmitters. *IEEE Transactions on Fuzzy Systems*, 11(4), 478–485.

Gabel, T. and Riedmiller, M. (2005). CBR for State Value Function Approximation in Reinforcement Learning. In *Proceedings of the 6th International Conference on Case-Based Reasoning*, 206–221.

Konda, V.R. and Tsitsiklis, J.N. (2003). On Actor-Critic Algorithms. *SIAM Journal on Control and Optimization*, 42(4), 1143–1166.

Kuvayev, L. and Sutton, R.S. (1996). Model-Based Reinforcement Learning with an Approximate, Learned Model. In *Proceedings of the 9th Yale Workshop on Adaptive and Learning Systems*, 101–105.

Moore, A.W. and Atkeson, C.G. (1993). Prioritized Sweeping: Reinforcement Learning with Less Data and Less Time. *Machine Learning*, 13, 103–130.

Peters, J. and Schaal, S. (2008). Natural actor-critic. *Neurocomputing*, 71, 1180–1190.

Sutton, R.S. (1992). Reinforcement Learning Architectures. *Proceedings of the International Symposium on Neural Information Processing.*

Sutton, R.S. and Barto, A.G. (1998). *Reinforcement Learning.* An Introduction. MIT Press.

Sutton, R.S., McAllester, D., Singh, S., and Mansour, Y. (2000). Policy Gradient Methods for Reinforcement Learning with Function Approximation. *Advances in Neural Information Processing Systems*, 12, 1057–1063.

Wettschereck, D., Aha, D.W., and Mohri, T. (1997). A Review and Empirical Evaluation of Feature Weighting Methods for a Class of Lazy Learning Algorithms. *Artificial Intelligence Review*, 11, 273–314.

Wilson, D.R. and Martinez, T.R. (2000). Reduction Techniques for Instance-Based Learning Algorithms. *Machine Learning*, 38, 257–286.