Efficient Model Learning Methods for Actor–Critic Control

Ivo Grondman, Maarten Vaandrager, Lucian Buşoniu, Robert Babuška, and Erik Schuitema

***Abstract*—We propose two new actor–critic algorithms for reinforcement learning. Both algorithms use local linear regression (LLR) to learn approximations of the functions involved. A crucial feature of the algorithms is that they also learn a process model, and this, in combination with LLR, provides an efficient policy update for faster learning. The first algorithm uses a novel model-based update rule for the actor parameters. The second algorithm does not use an explicit actor but learns a reference model which represents a desired behavior, from which desired control actions can be calculated using the inverse of the learned process model. The two novel methods and a standard actor–critic algorithm are applied to the pendulum swing-up problem, in which the novel methods achieve faster learning than the standard algorithm.**

***Index Terms*—Actor–critic, inverse model, local linear regression (LLR), machine learning algorithms, reinforcement learning (RL).**

I. Introduction and Related Work

MANY processes in industry can potentially benefit from control algorithms that learn to optimize a certain cost function. Reinforcement learning (RL) is such a learning method, based on ideas from animal learning psychology. The user sets a certain goal by specifying a suitable reward function for the RL controller, and the RL controller then learns to maximize 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 behavior. This is usually unacceptable in industry, particularly 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.

Manuscript received December 22, 2010; revised May 13, 2011; accepted September 8, 2011. Date of publication December 7, 2011; date of current version May 16, 2012. This work was performed while all authors were still with Delft University of Technology. This paper was recommended by Associate Editor A. Tayebi.

I. Grondman and R. Babuška are with the Delft Center for Systems and Control, Delft University of Technology, 2628 CD Delft, The Netherlands (e-mail: i.grondman@tudelft.nl; r.babuska@tudelft.nl).

M. Vaandrager is with Plotprojects, 1078 MN Amsterdam, The Netherlands (e-mail: maarten@vaandrager.com).

L. Buşoniu is with CNRS, Research Center for Automatic Control, University of Lorraine, 54500 Nancy, France and also with the Department of Automation, Technical University of Cluj-Napoca, 400020 Cluj-Napoca, Romania (e-mail: lucian@busoniu.net).

E. Schuitema is with the Delft Biorobotics Laboratory, Faculty of Mechanical Engineering, Delft University of Technology, 2628 CD Delft, The Netherlands (e-mail: e.schuitema@tudelft.nl).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

In this paper, we introduce two novel algorithms that employ an efficient policy update which increases the learning speed considerably compared to standard actor–critic methods. Actor–critic techniques are a class of RL methods which learn a critic function and a separate actor function. The critic is the value function approximator, and the actor is the policy approximator.

The first novel algorithm learns a process model and employs it to update the actor. However, instead of using the process model to generate simulated experiences as most model learning RL algorithms do [1]–[3], it uses the model to directly calculate an accurate policy gradient, which accelerates learning compared to other policy gradient methods.

The second novel algorithm learns not only a process model but also a reference model which represents desired behavior by mapping states to subsequent desired states. The reference model and inverse process model are then coupled to serve as an overall actor, which is used to calculate new inputs to the system.

The novel algorithms introduced here use local linear regression (LLR) to approximate the actor, critic, process model, and reference model. Memory-based learning methods have successfully been applied to RL before, mostly as an approximator for the value function [4], [5] and, in some cases, also for the process model [6], [7]. Although not exploited in this paper, one benefit of the memory-based function approximators is that they can easily be initialized with samples of prior knowledge. For example, the reference model memory could be initialized with samples of desired closed-loop behavior.

The second algorithm is similar to "learning from relevant trajectories" [8], 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. The use of LLR and a reference model are the essential factors in the new algorithms that speed up the learning.

II. 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 will introduce 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 maximized. This sum (also called the return) is stored in a value function $V^\pi : X \mapsto \mathbb{R}$, which is defined as

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

where $\gamma \in [0, 1)$ is the discount factor.

This function satisfies the Bellman equation [9]

$$V^\pi(x) = \rho\left(x, \pi(x)\right) + \gamma V^\pi(x') \tag{2}$$

where x' is given by the state transition function while using the policy π, i.e., $x' = f(x, \pi(x))$. The Bellman equation is the basis upon which RL can improve the policy π.

In continuous (or infinite discrete) state and action spaces, it is necessary to approximate the exact value function V^π and the exact policy π with function approximators.

III. Actor–Critic RL

Actor–critic techniques were introduced in [10] and have been investigated often since then (see, e.g., [11]–[20]). The actor–critic method is characterized by learning separate functions for the actor (the policy π) and the critic (the value function V^π). Actor–critic methods belong to the class of policy gradient methods. In these methods, the policy is represented by a differentiable parameterization, and gradient updates are performed to find the parameters that lead to (locally) maximal returns [21]. The critic takes the role of the value function and evaluates the performance of the actor, hereby helping with the estimation of the gradient to use for the actor's updates. The use of gradient-based policy updates makes actor–critic techniques suitable for continuous action spaces [22].

In this paper, a temporal-difference-based actor–critic method serves as a baseline to compare our new method to. We will refer to this baseline as the standard actor–critic (S-AC) algorithm. Denoting the approximate value function parameterized by the vector θ with $V(x, \theta)$, the temporal difference error is defined as [9]

$$\delta_k = r_k + \gamma V(x_k, \theta_{k-1}) - V(x_{k-1}, \theta_{k-1}). \tag{3}$$

This is the difference between the right-hand and left-hand sides of the Bellman equation (2). The goal is to make the approximation of the critic satisfy the Bellman equation. By using the temporal difference, the gradient-descent update rule for the critic parameter vector θ is

$$\theta_k = \theta_{k-1} + \alpha_c \delta_k \left.\frac{\partial V(x, \theta)}{\partial \theta}\right|_{\substack{x = x_{k-1} \\ \theta = \theta_{k-1}}} \tag{4}$$

where $\alpha_c > 0$ is the learning rate of the critic. This parameter update adapts the approximate value function such that the error between the approximated and real values at the state considered is minimized.

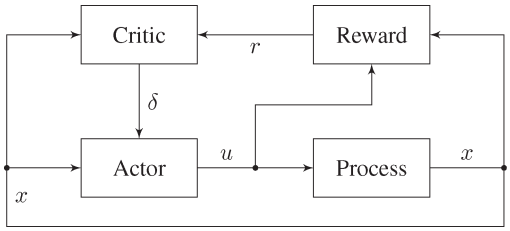


Fig. 1. Block diagram of the S-AC algorithm.

Using (4) 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 better way of assigning credit to states visited several steps earlier. The eligibility trace for a certain state x at time instant k is denoted[1] with $e_k(x)$

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

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

$$\theta_k = \theta_{k-1} + \alpha_c \delta_k \sum_{x_v \in \mathcal{X}_v} \left.\frac{\partial V(x, \theta)}{\partial \theta}\right|_{\substack{x = x_v \\ \theta = \theta_{k-1}}} e_k(x_v) \tag{5}$$

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

The approximate policy is parameterized by ϑ with $\pi(x, \vartheta)$ and indicates the action to take in a state x. RL 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 indicated by the policy. This can be achieved by perturbing the action with a zero-mean random exploration term Δu_k

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

When the exploration Δu_k leads to a positive temporal difference, the policy is adjusted toward 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:

$$\vartheta_k = \vartheta_{k-1} + \alpha_a \delta_k \Delta u_{k-1} \left.\frac{\partial \pi(x, \vartheta)}{\partial \vartheta}\right|_{\substack{x = x_{k-1} \\ \vartheta = \vartheta_{k-1}}} \tag{6}$$

where $\alpha_a > 0$ is the learning rate of the actor. The temporal difference is interpreted as a correction of the predicted performance; if the temporal difference is positive, the obtained performance is considered better than the predicted one.

Fig. 1 describes how different entities interact within the S-AC algorithm. The full implementation of the S-AC algorithm is shown in Algorithm 1.

[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.

Algorithm 1 S-AC

Input: γ, λ, α_c, α_a
$e_0(x) = 0 \quad \forall x$
Initialize x_0, θ_0 and ϑ_0
Apply random input u_0
$k \leftarrow 1$
loop
  Choose Δu_k at random
  Measure x_k, r_k
  $u_k \leftarrow \pi(x_k, \vartheta_{k-1}) + \Delta u_k$
  Apply u_k
  $\delta_k \leftarrow r_k + \gamma V(x_k, \theta_{k-1}) - V(x_{k-1}, \theta_{k-1})$
  $e_k(x) = \begin{cases} 1, & \text{if } x = x_k \\ \lambda\gamma e_{k-1}(x), & \text{otherwise} \end{cases}$
  $\theta_k \leftarrow \theta_{k-1} + \alpha_c \delta_k \sum_{x \in \mathcal{X}_v} (\partial V(x,\theta)/\partial\theta) e_k(x)$
  $\vartheta_k \leftarrow \vartheta_{k-1} + \alpha_a \delta_k \Delta u_{k-1} (\partial \pi(x,\vartheta)/\partial\vartheta)$
  $k \leftarrow k + 1$
end loop

IV. Function Approximation

In actor–critic methods, both the policy and the value function are represented using function approximation techniques. Two such function approximation techniques, LLR and tile coding, are used in this paper.

A. Local Linear Regression

The algorithms introduced in this paper use LLR as a function approximator. LLR is a nonparametric memory-based method for approximating nonlinear functions. Memory-based methods are also called case based, exemplar based, lazy, instance based, or experience based [23], [24]. It has been shown that memory-based learning can work in RL and can quickly approximate a function with only a few observations [4]. This is particularly 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 $s_i = [x_i^T | y_i^T]^T$ with $i = 1, \ldots, N$. 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 neighborhood of x_q in the samples stored in memory. This neighborhood 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—denoted with W—is used to scale the inputs x and has a large influence on the resulting neighborhood and thus on the accuracy of the prediction. Note that, in this paper, the input samples x_i are actually stored in the LLR memories as weighted samples Wx_i. Searching through the memory for nearest neighbor samples is computationally expensive. Here, a simple sorting algorithm was used, but one can reduce the computational burden by using, for instance, k-d trees [25].

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 neighbor samples. Only these K nearest neighbors are then used to make a prediction of $\hat{y}_q$. The prediction is computed by fitting a linear model to these nearest neighbors. 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 neighbor samples

$$Y = \begin{bmatrix} y_1 & y_2 & \cdots & y_K \end{bmatrix}$$
$$X = \begin{bmatrix} x_1 & x_2 & \cdots & x_K \\ 1 & 1 & \cdots & 1 \end{bmatrix}.$$

The last row of X consists of ones to allow for a bias on the output, making the model affine instead of truly linear.

The X and Y matrices form an overdetermined 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 least squares method using the right pseudoinverse of X

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

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

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.

The use of LLR comes with a few assumptions. The first and foremost one is that the approximated function should be smooth enough so that it can be captured by locally linear models. Any function with discontinuities or other nonsmooth behavior will be tough to approximate. This also depends on the maximum possible number of samples in the LLR memory. This number should be large enough so that the neighborhood in which a locally linear model is calculated is small enough, i.e., the linear model is indeed local enough. More specifically, when applying LLR in RL algorithms, the sampling time used should be small enough so that a locally linear model calculated at one time step is still good enough at the next time

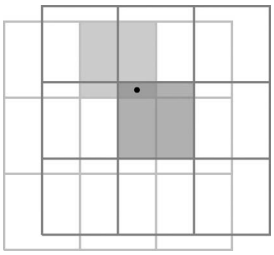


Fig. 2. Tile coding example. The dot represents a point in the state space. Two tilings each have one (shaded) tile to which that particular point belongs.

step. This is because we also use the model for predictions at the next time step.

B. Tile Coding

Tile coding is a classical function approximator commonly used in RL, also allowing for fast computation. It uses a limited number of tilings which each divide the space into a number of tiles. The distribution of the tiles is often the uniform gridlike distribution—which is the approach used in this paper—but any tile shape and distribution are possible. An illustration of a tile coding example is shown in Fig. 2.

A point in the state space either belongs to a tile or not, meaning that the tiles are, in fact, binary features [9]. The average of the parameters of the T tiles that the state belongs to is used to compute the prediction

$$\hat{y}_q = \frac{1}{T}\sum_{i=1}^{T}\theta_i.$$

LLR inserts new experiences (samples) directly into the (initially empty) memory, whereas tile coding starts with initial values of the parameters that are incrementally adjusted. LLR also allows for broad generalization over states at the start of learning when the number of collected samples is low. As the sample density grows, the neighborhood of the local model grows smaller, and the scope of generalization decreases. This causes LLR to be a better function approximator at the start of learning.

V. Efficient Policy Update Methods

In this section, two new actor–critic algorithms are introduced. The first algorithm is the model learning actor–critic (MLAC). The algorithm learns a process model in addition to the actor and critic. It is the actor update that makes this method different from actor–critic methods found in the literature. The update is done using a policy gradient which is calculated using a local gradient of the critic and a local gradient of the learned process model.

The second method learns a process model, a reference model, and a critic. The reference model takes the place of the actor, which means that there is no explicit actor function. This reference model represents a desired behavior along which the process is controlled by using the inverse of the learned process model. Although the scheme lacks an explicit actor, we refer to this method as reference model actor–critic (RMAC).

Both methods belong to the class of model learning algorithms (also called indirect methods) as opposed to direct algorithms [9]. In the implementation of the algorithms, we always use LLR to learn and approximate the functions and models involved.

The process model, actor, and critic are updated by inserting the last observed sample into the memory, as the most up-to-date knowledge should be incorporated in any approximation calculated from the memory. The critic memory M^C holds samples of the form $s_i = [x_i^T | V_i]^T$ with $i = 1, \ldots, N^C$, the actor memory M^A has samples $s_i = [x_i^T | u_i^T]^T$ with $i = 1, \ldots, N^A$, and the process memory M^P has samples $s_i = [x_i^T \;\; u_i^T | x_i'^T]^T$ with $i = 1, \ldots, N^P$, where x' denotes the observed next state, i.e., $x' = f(x, u)$. Finally, the reference model M^R has samples $s_i = [x_i^T | \hat{x}_i^T]^T$ with $i = 1, \ldots, N^R$, where $\hat{x}$ denotes a desired next state. During the learning process, the actor, critic, and reference model are updated by adjusting the output parts of the nearest neighbor samples s_i that relate to the query point x_q. The method of updating them is explained in more detail in the following sections.

A. Model Learning Actor–Critic

In addition to learning the actor and critic functions, the MLAC method learns an approximate process model $x' = \hat{f}(x, u)$. Having a learned process model available simplifies the update of the actor, as it allows us to predict what the next state x' will be, given some input u. Together with the approximate value function, this allows us to obtain information on the value $V(x')$ of the next state x'. This means that we can choose the input u such that $V(x')$ is optimal. However, since we assume that our action space is continuous, we cannot enumerate over all possible inputs u and therefore choose to put a policy gradient in place. By virtue of LLR, we can easily estimate the gradient of the value function with respect to the state x and the gradient of the process model with respect to the input u. Then, by applying the chain rule, we have a gradient of the value function with respect to the input u available and use this to update the actor.

Hence, the actor is updated by multiplying the local gradients of the value function and of the process model to obtain a gradient of the value function with respect to a chosen input u. By adjusting the input u in the direction given by this gradient and saturating the result element-wise such that the new input lies in the allowed input range $[u_{\min}, u_{\max}]$, the actor is trying to maximize $V(x')$

$$u_i \leftarrow \text{sat}\left\{u_i + \alpha_a \left.\frac{\partial V}{\partial x}\right|_{x=x'} \frac{\partial x'}{\partial u}\right\} \qquad \forall i \in \mathcal{K}(x). \tag{7}$$

Recall that x' is given by the state transition function $x' = f(x, u)$, which is approximated here by $\hat{f}$, based on the process model memory M^P.

The value function is approximated by LLR which estimates a local linear model on the basis of previous observations of $V(x)$. The local linear model is of the form

$$\begin{aligned} V(x) &= \beta^C \cdot \begin{bmatrix} x \\ 1 \end{bmatrix} \\ &= [\, \beta_x^C \quad \beta_b^C \,] \cdot \begin{bmatrix} x \\ 1 \end{bmatrix}. \end{aligned}$$

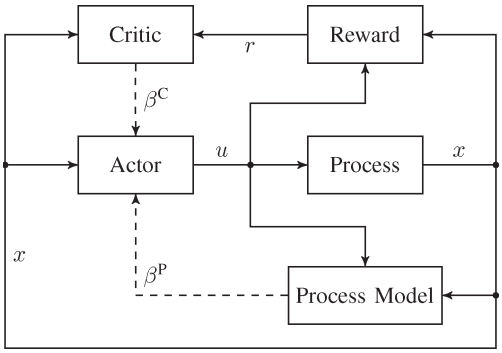


Fig. 3. Block diagram of the MLAC algorithm. The solid lines indicate actual signals. The dashed lines indicate the use of a local linear model or gradient from a particular entity.

This model has an input vector of length $n+1$ (n state dimensions plus a bias), a scalar output V, and a model parameter matrix β^C of size $1 \times (n+1)$. The gradient $\partial V/\partial x$ is the part of β^C that relates the input x to the output V. This part is denoted as β_x^C and has size $1 \times n$.

The gradient $\partial x'/\partial u$ can be found by LLR on previous observations of the process dynamics. The local linear process model is of the form

$$x' = \hat{f}(x,u) = \beta^P \cdot \begin{bmatrix} x \\ u \\ 1 \end{bmatrix}$$

$$= [\,\beta_x^P \quad \beta_u^P \quad \beta_b^P\,] \cdot \begin{bmatrix} x \\ u \\ 1 \end{bmatrix}.$$

This model has an input vector of length $n+m+1$ (with n being state dimensions and m as action dimensions plus a bias), an output vector x' of length n, and a model parameter matrix β^C of size $n \times (n+m+1)$. The gradient $\partial x'/\partial u$ is the part of β^P that relates u to x', denoted as β_u^P, and has size $n \times m$.

We can now use β_x^C, β_u^P, and (7) to improve the actor by adapting the nearest neighbor samples with

$$u_i \leftarrow \text{sat}\,\left\{u_i + \alpha \beta_x^C \beta_u^P\right\} \qquad \forall i \in \mathcal{K}(x).$$

Fig. 3 shows the scheme of MLAC, and the pseudocode is found in Algorithm 2. In the pseudocode, the set $\mathcal{K}_+(x_q)$ is $\mathcal{K}(x_q)$, extended with the index where the sample representing x_q was inserted.

Algorithm 2 MLAC

Input: γ, λ, α_c, α_a
Initialize x_0, M^C, M^A and M^P
$V_0 = 0$, $\beta^C = 0$
$e_0(s_i) = 0 \quad \forall s_i \in M^C$
Apply random input u_0
$k \leftarrow 1$
loop
  Choose Δu_k at random
  Measure x_k, r_k
  Obtain β^A from M^A for x_k
  $u_k \leftarrow \beta^A \cdot [x_k^T\ 1]^T + \Delta u_k$
  Apply u_k
  % Update process model
  Obtain β^P from M^P for x_{k-1}, u_{k-1}
  Insert $[x_{k-1}^T\ u_{k-1}^T | x_k^T]^T$ in M^P
  % Update actor
  Insert $[x_{k-1}^T | (u_{k-1} + \alpha_a \beta_x^C \beta_u^P)^T]^T$ in M^A
  for $\forall i \in \mathcal{K}(x_{k-1})$ of M^A **do**
    $u_i \leftarrow \text{sat}\,\left\{u_i + \alpha_a \beta_x^C \beta_u^P\right\}$
  end for
  % Update critic
  Obtain β^C from M^C for x_k
  $V_k \leftarrow \beta^C \cdot [x_k^T\ 1]^T$
  Insert $[x_{k-1}^T | V_{k-1}]^T$ in M^C
  $\delta_k \leftarrow r_k + \gamma V_k - V_{k-1}$
  for $\forall s_i \in M^C$ **do**
    $e_k(s_i) = \begin{cases} 1, & \text{if } i \in \mathcal{K}_+(x_{k-1}) \\ \lambda\gamma e_{k-1}(s_i), & \text{otherwise} \end{cases}$
    $V_i \leftarrow V_i + \alpha_c \delta_k e_k(s_i)$
  end for
  $k \leftarrow k+1$
end loop

Note that, with the S-AC algorithm, which uses (6), exploration is needed in order to update the actor. Because MLAC uses the process model gradient (7), it knows in what direction to update the actor such that higher state values will be encountered, without having to perform exploratory actions. As a result, the MLAC algorithm can estimate the policy gradient without exploration. Exploration is nevertheless needed because it reduces the chance that the policy improvement gets stuck in a local optimum. Moreover, it gives a more complete value function over the entire state space. The current policy only visits a part of the state space. Without exploration, only a part of the value function will then be correctly estimated. Finally, exploration improves the model of the process dynamics.

B. Reference Model Actor–Critic

RMAC is different from the typical actor–critic methods in the sense that it does not learn an explicit mapping from state x_k to action u_k. This means that an actor is no longer learned. Instead, RMAC learns a reference model that represents a desired behavior of the system, based on the value function. Similar to MLAC, this algorithm learns a process model, through which it identifies a desired next state x' with the highest possible value $V(x')$. The difference with respect to MLAC is that we now do not explicitly store the actor, mapping a state x onto an action u, but store the mapping from a state x to the desired next state x' in a reference model. Then, using the inverse of the learned process model, the action u is calculated.

The reference model $R(x)$ maps the state x_k to a desired next state $\hat{x}_{k+1}$, i.e.,

$$\hat{x}_{k+1} = R(x_k).$$

The process is controlled toward 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

$$x_{k+1} = \hat{f}(x_k, u_k) = \underbrace{\begin{bmatrix} \beta_x^P & \beta_u^P & \beta_b^P \end{bmatrix}}_{\beta^P} \cdot \begin{bmatrix} x_k \\ u_k \\ 1 \end{bmatrix}.$$

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

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

We improve $R(x)$ by adapting the desired state $\hat{x}$ of the nearest neighbor samples s_i ($i \in \mathcal{K}(x_{k-1})$) toward higher state values using the following gradient update rule:

$$\hat{x}_i \leftarrow \hat{x}_i + \alpha \left.\frac{\partial V}{\partial x}\right|_{x=x'}, \qquad i \in \mathcal{K}(x_{k-1}). \tag{8}$$

However, (8) eventually may lead to an infeasible reference model if $\hat{x}_i$ 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

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

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 extreme values of u 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. 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 being the size of vector u. For example, with two inputs, 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

$$X_R = \beta^P \cdot U_R.$$

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

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

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

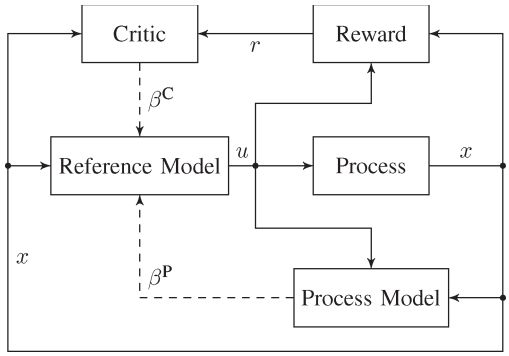


Fig. 4. Block diagram of the RMAC algorithm. The solid lines indicate actual signals. The dashed lines indicate the use of a local linear model or gradient from a particular entity.

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

$$x_r = \arg\max_{x \in \mathcal{X}_R} \beta^C \cdot \begin{bmatrix} x \\ 1 \end{bmatrix}$$

$$\hat{x}_i \leftarrow \hat{x}_i + \alpha_r (x_r - \hat{x}), \qquad i \in \mathcal{K}(x_{k-1}).$$

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

The aforementioned method to update samples already present in the reference model is also used to insert new samples into the reference model. Once the desired state x_r has been calculated for the state x_{k-1}, the sample $[x_{k-1}^T | x_r^T]^T$ is inserted into the reference model memory. This means that the reference model is completely learned and updated from scratch, since it can initialize itself this way online without using prior knowledge. However, initializing the reference model with samples learned from a demonstrated trajectory offline is still possible if they are available.

Learning the reference model online and from scratch might pose convergence issues, since another bootstrapping approximation comes into play. Nonetheless, we believe that convergence is likely to be achieved when following the same reasoning that S-AC algorithms use. In, e.g., [11] and [26], it is stated that, for those algorithms, convergence is not a problem as long as the actor is updated on a slower time scale than the critic. Now that the reference model is taking up the role of the actor, it is natural to conjecture that, when keeping the learning rate of the reference model below that of the critic, we should also have convergence.

In contrast to S-AC, the RMAC improves the reference model using (8) which does not involve exploration. Instead, it improves the reference model using locally linear models estimated on the basis of previous experiences. However, for the same reasons as with MLAC, exploration is still needed. Fig. 4 shows the scheme of RMAC, and the pseudocode is found in Algorithm 3.

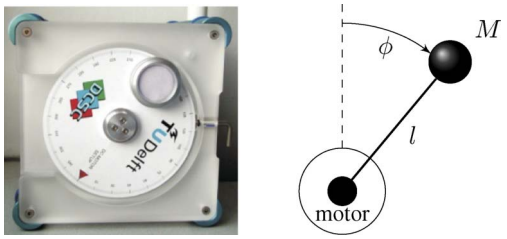


Fig. 5. Inverted pendulum setup.

TABLE I
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	$\mathrm{m/s}^2$
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

Algorithm 3 RMAC

Input: γ, λ, α_c, α_r
Initialize x_0, $\hat{x}_1$, M^C, M^R and M^P
$V_0 = 0$, $\beta^P = 0$
$e_0(s_i) = 0 \quad \forall s_i \in M^C$
Apply random input u_0
$k \leftarrow 1$
loop
  Choose Δu_k at random
  Measure x_k, r_k
  $\hat{x}_{k+1} = R(x_k)$
  $u_k \leftarrow \hat{f}^{-1}(x_k, \hat{x}_{k+1}) + \Delta u_k$
  Apply u_k
  % Update process model
  Insert $[x_{k-1}^T \; u_{k-1}^T | x_k^T]^T$ in M^P
  % Update reference model
  Select best reachable state x_r
  Insert $[x_{k-1}^T | x_r^T]^T$ in M^R
  for $\forall i \in \mathcal{K}(x_{k-1})$ of M^R **do**
    $\hat{x}_i \leftarrow \hat{x}_i + \alpha_r(x_r - \hat{x}_k)$
  end for
  % Update critic
  $V_k \leftarrow V(x_k)$
  Insert $[x_{k-1}^T | V_{k-1}]^T$ in M^C
  $\delta_k \leftarrow r_k + \gamma V_k - V_{k-1}$
  for $\forall s_i \in M^C$ **do**
    $e_k(s_i) = \begin{cases} 1, & \text{if } i \in \mathcal{K}_+(x_{k-1}) \\ \lambda\gamma e_{k-1}(s_i), & \text{otherwise} \end{cases}$
    $V_i \leftarrow V_i + \alpha_c \delta_k e_k(s_i)$
  end for
  $k \leftarrow k + 1$
end loop

VI. EXAMPLE: PENDULUM SWING-UP

To evaluate and compare the performance of our algorithms, we apply them to the task of learning to swing up an inverted pendulum and compare them to the standard algorithm. The swing-up 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 states and one action, it allows for easy visualization of the functions of interest. A photograph of this system is shown in Fig. 5.

The equation of motion of this system is

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

where ϕ is the angle of the pendulum measured from the upright position. The model parameters are given in Table I.

The task is to learn to swing the pendulum from the pointing-down position to the upright position as quickly as possible and stabilize it in this position. The (fully measurable) state x consists of the angle ϕ of the pendulum and the angular velocity $\dot{\phi}$ of the pendulum

$$x = \begin{bmatrix} \phi \\ \dot{\phi} \end{bmatrix}.$$

The actuation signal u is 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 swing-up task. This reward function has its maximum in the upright position $[0\ 0]^T$ and quadratically penalizes nonzero values of ϕ, $\dot{\phi}$, and u.

$$r_k(x_{k-1}, u_{k-1}) = -x_{k-1}^T Q x_{k-1} - P u_{k-1}^2 \tag{9}$$

with

$$Q = \begin{bmatrix} 5 & 0 \\ 0 & 0.1 \end{bmatrix} \quad P = 1.$$

The S-AC method and the two novel methods MLAC and RMAC are applied to the pendulum swing-up problem described earlier. The algorithms run for 30 min of simulated time, consisting of 600 consecutive trials with each trial lasting 3 s. The pendulum needs approximately 1 s 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, maximum, minimum, and confidence interval of the learning curve.

A. Standard Actor–Critic

This section presents the results of applying the S-AC algorithm to the swing-up problem.

The function approximator used is tile coding (Section IV-B), with 16 partitions, each being a uniform grid of 7×7 tiles. The

TABLE II
THE RL PARAMETERS AND THE PARAMETERS FOR LLR FOR THE S-AC, MLAC, AND RMAC METHODS

		S-AC	MLAC	RMAC
RL parameters				
sampling period	T_s	0.03	0.03	0.03
reward discount rate	γ	0.97	0.97	0.97
eligibility discount rate	λ	0.65	0.65	0.65
exploration variance	σ^2	1		
Process model parameters				
memory size	N^{P}		100	100
nearest neighbours	K^{P}		9	9
input weighting	W^{P}		[1 0.1 1]	[1 0.1 1]
Critic parameters				
learning rate	α_c	0.1	0.1	0.1
memory size	N^{C}		2000	2000
nearest neighbours	K^{C}		20	20
input weighting	W^{C}		[1 0.1]	[1 0.1]
Actor parameters				
learning rate	α_a	0.005	0.005	
memory size	N^{A}		2000	
nearest neighbours	K^{A}		9	
input weighting	W^{A}		[1 0.1]	
Reference model parameters				
learning rate	α_r			0.005
memory size	N^{R}			2000
nearest neighbours	K^{R}			20
input weighting	W^{R}			[1 0.1]

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 $\sigma = 1$

$$\Delta u \sim \mathcal{N}[0, 1].$$

The reason for exploring only once every three steps instead of every step is because this allows for large exploratory actions while 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 constant 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. This causes the update of the actor by (6) to be very noisy.

The S-AC algorithm, using parameter settings from Table II, applied to the pendulum swing-up task results in the learning curve as shown in Fig. 6. The final approximations of $V(x)$ and $\pi(x)$ after a representative learning experiment are shown in Figs. 7 and 8.

The method takes, on average, about 10 min to converge. One striking characteristic of the curve in Fig. 6 is the short drop in performance in the first minutes. This can be explained by the fact that the value function is initialized to zero, which is too

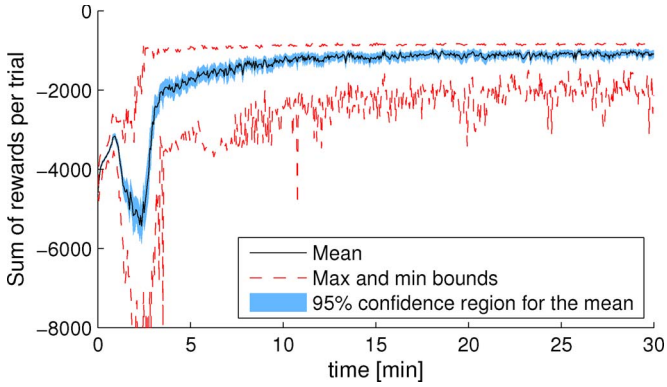


Fig. 6. Results for the S-AC algorithm. The mean, max, and min bounds and 95% confidence region are computed from 40 learning curves.

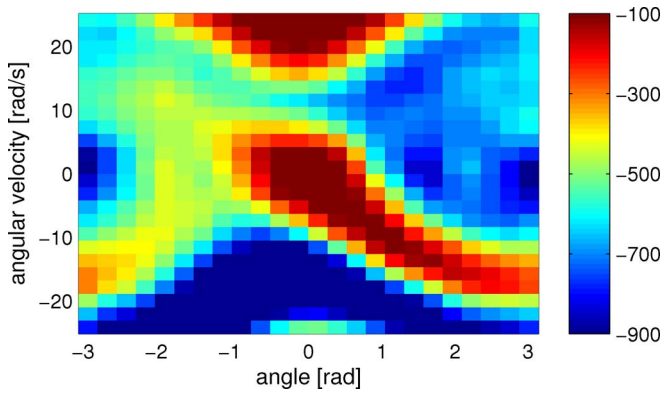


Fig. 7. Final critic $V(x)$ for the S-AC algorithm after one learning experiment.

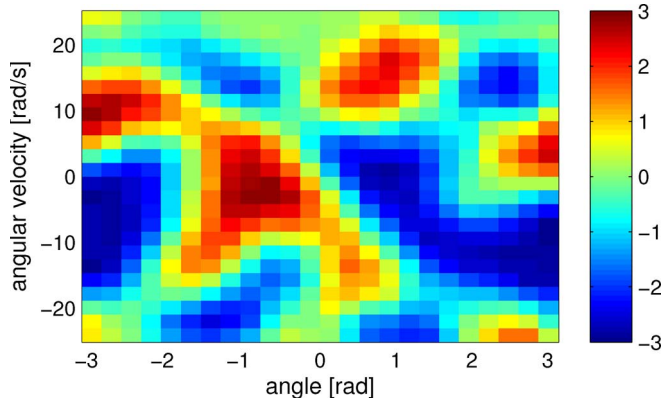


Fig. 8. Final actor $\pi(x)$ for the S-AC algorithm after one learning experiment.

optimistic compared to 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 initialized with low values, but this decreases the overall learning speed as all new unvisited states are initially assumed to be bad and are avoided. An example of this is shown in Fig. 9, where the value function has been initialized pessimistically, by using the infinite discounted sum of minimum rewards, i.e.,

$$\sum_{j=0}^{\infty} \gamma^j \min_{x,u} r(x,u) = \frac{1}{1-\gamma} \min_{x,u} r(x,u) \qquad (10)$$

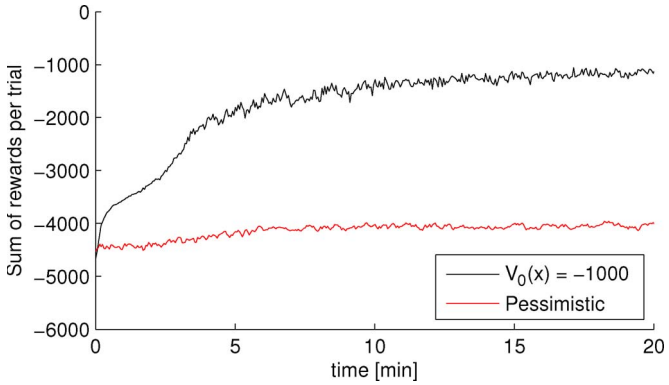


Fig. 9. Results for the S-AC algorithm with two initializations for the value function: $V_0(x) = -1000$ for all x and $V_0(x) = (1/1-\gamma)\min_{x,u} r(x,u) \approx -4050$ (pessimistic).

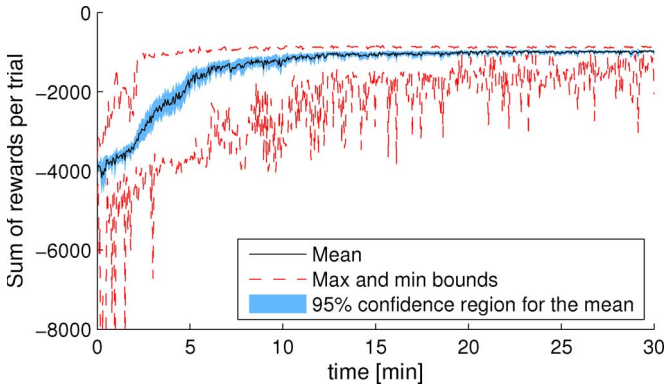


Fig. 10. Results for the MLAC algorithm. The mean, max, and min bounds and 95% confidence region are computed from 40 learning curves.

with r defined as in (9). Obviously, r has no minimum and tends toward $-\infty$ for $x \to \infty$, but assuming[3] that the angular velocity $|\dot{\theta}|$ will never exceed 8π rad $\cdot$ s^{-1} and using the facts that $|\theta| \leq \pi$ rad and $|u| \leq 3$ V, it is still possible to calculate the worst possible immediate reward that can be received, by using $x = [\pi\ 8\pi]^T$ and $u = 3$ in (9). This worst possible reward is then used as the minimum of r in (10). With $\gamma = 0.97$, the value function is then initialized with $V_0(x) \approx -4050$. In Fig. 9, a plot is also given for the case where the value function was initialized with $V_0(x) = -1000$ for all x. The drop in performance is gone, and learning is still quick. However, estimating an initial value that will achieve this sort of behavior is done by trial and error and is therefore very hard.

The final performance can be improved by increasing the number of partitions and number of tiles per partition in the tile coding, but this decreases the learning speed. The S-AC curve in Fig. 6 is used as a baseline to compare the novel methods to, because it was obtained with a nontuned reasonable initialization of the value function, as will most likely be done in applications.

B. Model Learning Actor–Critic

The MLAC algorithm was applied using the parameter settings in Table II. This results in the learning curve for the pendulum swing-up task shown in Fig. 10. Figs. 11 and 12 show the samples in the critic and actor memories after one full learning experiment, respectively.

[3]This assumption is justified by the fact that typical trajectories do not exceed this velocity.

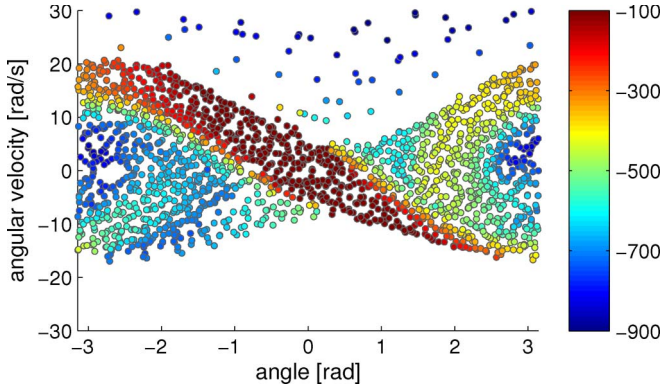


Fig. 11. Final critic $V(x)$ for the MLAC algorithm after one learning experiment. Every point represents a sample $[x|V]$ in the critic memory M^C.

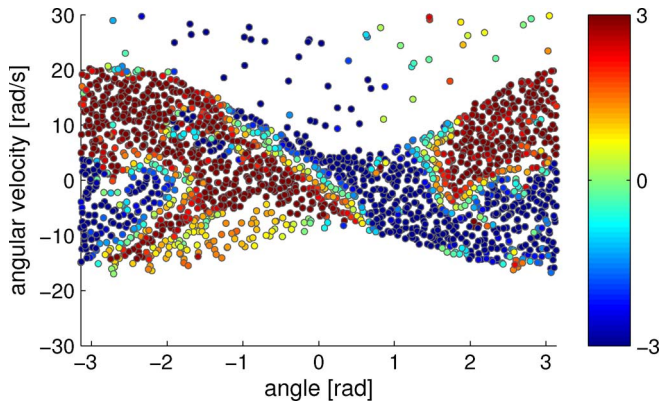


Fig. 12. Final actor $\pi(x)$ for the MLAC algorithm after one learning experiment. Every point represents a sample $[x|u]$ in the actor memory M^A.

There is a lack of samples in the lower regions of Figs. 11 and 12, because none of the trials in this particular learning experiment generated a trajectory through that region.

The MLAC method converges fast and to a good solution of the swing-up task. The fast learning speed can be attributed to the characteristics of LLR. LLR gives a good quick estimate at the start of learning by inserting observations directly into the memory and also allows for broad generalization over the states when the number of collected samples is still low. Finally, the update by (7) is not stochastic in contrast to the update of the S-AC by (6) which is typically based on random exploration Δu. This means that MLAC is less dependent on the proper tuning of exploration parameters, such as the standard deviation used in the generation of random exploratory actions.

C. Reference Model Actor–Critic

The RMAC algorithm was applied using the parameter settings in Table II, and this resulted in the learning curve for the pendulum swing-up task shown in Fig. 13. Fig. 14 shows the samples from the critic memory obtained after one representative learning experiment, whereas Fig. 15 shows the

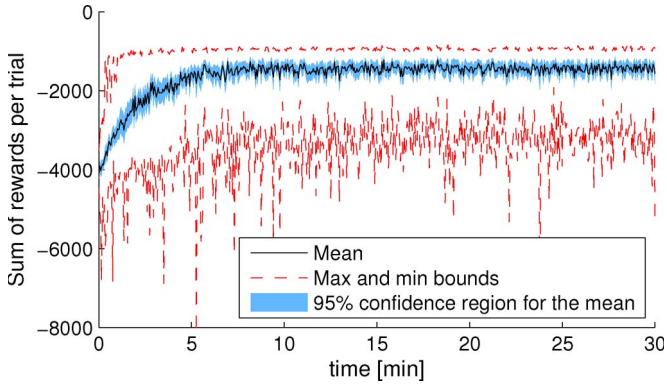


Fig. 13. Results for the RMAC algorithm. The mean, max, and min bounds and 95% confidence region are computed from 40 learning curves.

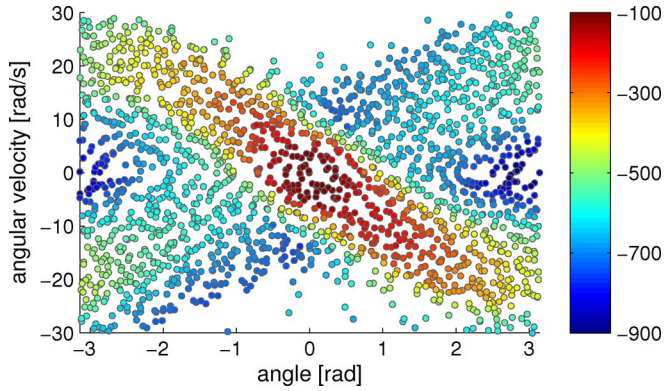


Fig. 14. Final critic $V(x)$ for the RMAC algorithm after one learning experiment. Every point represents a sample $[x|V]$ in the critic memory M^C.

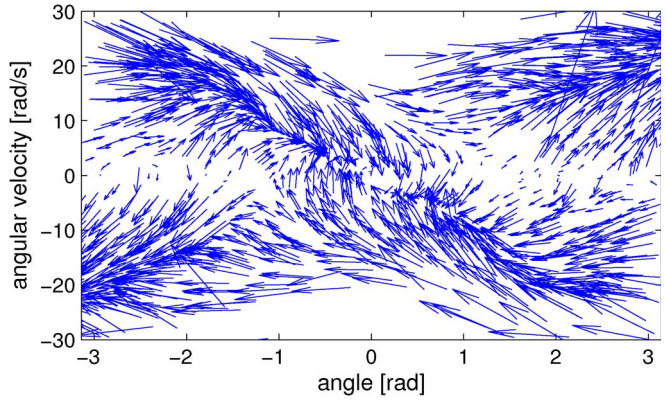


Fig. 15. Final reference model $R(x)$ for the RMAC algorithm after one learning experiment. Every arrow represents a sample $[x|\hat{x}]$ in the reference model memory M^R. The arrow points to the desired next state $\hat{x}$.

memory samples for the reference model, i.e., the mapping from states to their next respective desired states.

The RMAC method converges very quickly and to a good solution for the pendulum problem. The main reason for the fast convergence is the fact that the method starts out by choosing the desired states $\hat{x}$ that resulted 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 yields a fast initial estimate of the value function which is beneficial for the learning speed.

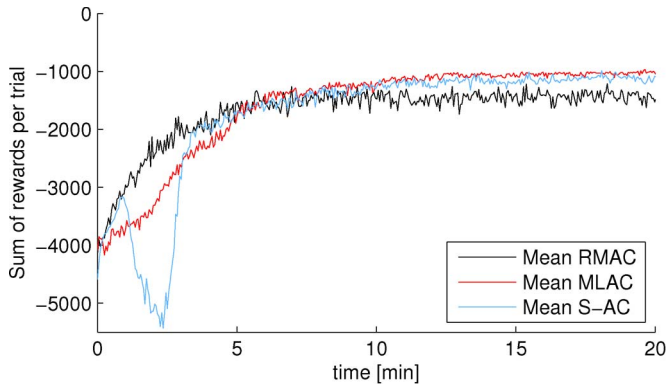


Fig. 16. Learning curves for S-AC, MLAC, and RMAC compared with each other.

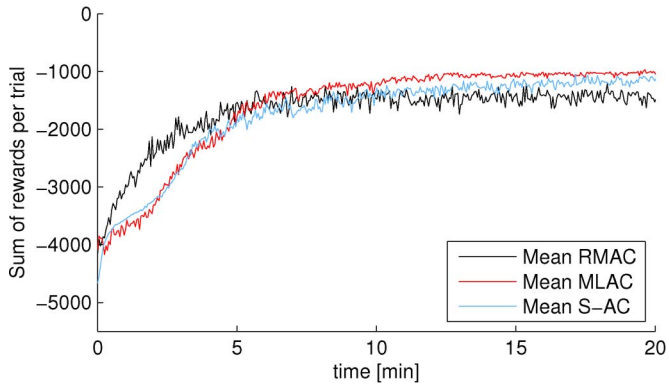


Fig. 17. Learning curves for S-AC, MLAC, and RMAC compared with each other, where the value function of S-AC was initialized with $V(x) = -1000$ for all x.

VII. Conclusion and Open Issues

This paper has introduced two novel actor–critic methods, which use LLR as a nonparametric memory-based function approximator. It has been shown by simulation experiments that these novel methods are capable of fast learning. Fig. 16 shows the learning curves of the S-AC, MLAC, and RMAC algorithms in one plot, allowing for a comparison in performance.

The most notable difference is that the algorithms using LLR learn faster than the algorithm using tile coding. For both MLAC and RMAC, this is because we have a process model and a critic function available. Combining these two, it is possible to calculate the inputs u that will yield high values at the next time step. This largely overcomes the problem of having to explore the effect of different inputs in certain states, and as a result, the algorithms are less likely to run into suboptimal states at the start of learning, like S-AC does. MLAC reaches the best solution, whereas RMAC converges fastest. The latter can be explained with the update of $R(x)$, which is on the basis of a predicted next state $\hat{x}_{k+1}$, initially calculated using the extremes of u. The result is a rapid exploration of the state space and estimation of the value function.

Using the initial value $V(x) = -1000$ for all x, as used before in Fig. 9, the comparison looks a bit different (see Fig. 17). In this case, MLAC and S-AC perform similarly, and

RMAC still learns the fastest in the early stages. However, it has to be noted that MLAC and RMAC are able to perform this way without the need for a good initialization of the value function, which is difficult to obtain in general.

In the pendulum swing-up experiment, the memories were initialized as empty, but instead, one can initialize them with previous measurements. This makes it easy to incorporate prior knowledge on the process dynamics, near-optimal control policy or near-optimal behavior, e.g., the RMAC reference model can be initialized with samples of the closed-loop behavior. This can be beneficial if the desired behavior of the system is known, but the control policy is yet unknown (which is often the case when supplying prior knowledge by imitation [15]). In both algorithms, the process model can be initialized with input/state/output samples of the open-loop system. This has the benefit that it is usually easy to conduct experiments that will generate these samples and that it is unnecessary to deduce an analytical model of the system, as the samples are used to calculate locally linear models that are accurate enough in their neighborhood.

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 (including the unit scaling of the inputs), which has a large influence on selecting the most relevant samples for regression. A second issue that must 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 LLR.

The experiment now only compares the performance of MLAC and RMAC with the S-AC algorithm. It should also be interesting to see how these methods compare to direct policy search methods. Finally, the approximation of the reachable subset $\mathcal{X}_R$ may not be sufficiently 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

[1] R. S. Sutton, "Reinforcement learning architectures," in *Proc. Int. Symp. Neural Inf. Porcess.*, 1992, pp. 211–216.

[2] A. W. Moore and C. G. Atkeson, "Prioritized sweeping: Reinforcement learning with less data and less time," *Mach. Learn.*, vol. 13, no. 1, pp. 103–130, Oct. 1993.

[3] L. Kuvayev and R. S. Sutton, "Model-based reinforcement learning with an approximate, learned model," in *Proc. 9th Yale Workshop Adapt. Learn. Syst.*, 1996, pp. 101–105.

[4] T. Gabel and M. Riedmiller, "CBR for state value function approximation in reinforcement learning," in *Proc. 6th Int. Conf. Case-Based Reason.*, 2005, pp. 206–220.

[5] G. Neumann and J. Peters, "Fitted Q-iteration by advantage weighted regression," in *Advances in Neural Information Processing Systems*, vol. 22. Cambridge, MA: MIT Press, 2009, pp. 1177–1184.

[6] A. Y. Ng, A. Coates, M. Diel, V. Ganapathi, J. Schulte, B. Tse, E. Berger, and E. Liang, "Autonomous inverted helicopter flight via reinforcement learning," in *Experimental Robotics IX*, vol. 21. Berlin, Germany: Springer-Verlag, 2006, pp. 363–372.

[7] J. Forbes and D. Andre, "Representations for learning control policies," in *Proc. 19th Int. Conf. Mach. Learn. Workshop Develop. Represent.*, 2002, pp. 7–14.

[8] C. G. Atkeson and S. Schaal, "Robot learning from demonstration," in *Proc. 14th Int. Conf. Mach. Learn.*, 1997, pp. 12–20.

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

[10] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-13, no. 5, pp. 834–846, Sep./Oct. 1983.

[11] V. R. Konda and J. N. Tsitsiklis, "On actor-critic algorithms," *SIAM J. Control Optim.*, vol. 42, no. 4, pp. 1143–1166, 2003.

[12] H. R. Berenji and D. Vengerov, "A convergent actor-critic-based FRL algorithm with application to power management of wireless transmitters," *IEEE Trans. Fuzzy Syst.*, vol. 11, no. 4, pp. 478–485, Aug. 2003.

[13] D. Vengerov, N. Bambos, and H. R. Berenji, "A fuzzy reinforcement learning approach to power control in wireless transmitters," *IEEE Trans. Syst., Man, Cybern. B, Cybern.*, vol. 35, no. 4, pp. 768–778, Aug. 2005.

[14] W. Usaha and J. A. Barria, "Reinforcement learning for resource allocation in LEO satellite networks," *IEEE Trans. Syst., Man, Cybern. B, Cybern.*, vol. 37, no. 3, pp. 515–527, Jun. 2007.

[15] J. Peters and S. Schaal, "Natural actor-critic," *Neurocomputing*, vol. 71, no. 7–9, pp. 1180–1190, Mar. 2008.

[16] Q. Yang, J. B. Vance, and S. Jagannathan, "Control of nonaffine nonlinear discrete-time systems using reinforcement-learning-based linearly parameterized neural networks," *IEEE Trans. Syst., Man, Cybern. B, Cybern.*, vol. 38, no. 4, pp. 994–1001, Aug. 2008.

[17] J. Valasek, J. Doebbler, M. D. Tandale, and A. J. Meade, "Improved adaptive–reinforcement learning control for morphing unmanned air vehicles," *IEEE Trans. Syst., Man, Cybern. B, Cybern.*, vol. 38, no. 4, pp. 1014–1020, Aug. 2008.

[18] S. Bathnagar, R. S. Sutton, M. Ghavamzadeh, and M. Lee, "Natural actor-critic algorithms," *Automatica*, vol. 45, no. 11, pp. 2471–2482, Nov. 2009.

[19] K. G. Vamvoudakis and F. L. Lewis, "Online actor-critic algorithm to solve the continuous-time infinite horizon optimal control problem," *Automatica*, vol. 46, no. 5, pp. 878–888, May 2010.

[20] B. Kim, J. Park, S. Park, and S. Kang, "Impedance learning for robotic contact tasks using natural actor-critic algorithm," *IEEE Trans. Syst., Man, Cybern. B, Cybern.*, vol. 40, no. 2, pp. 433–443, Apr. 2010.

[21] L. Buşoniu, R. Babuška, B. De Schutter, and D. Ernst, *Reinforcement Learning and Dynamic Programming Using Function Approximators*. Boca Raton, FL: CRC Press, 2010.

[22] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in Neural Information Processing Systems*, vol. 12. Cambridge, MA: MIT Press, 2000, pp. 1057–1063.

[23] D. Wettschereck, D. W. Aha, and T. Mohri, "A review and empirical evaluation of feature weighting methods for a class of lazy learning algorithms," *Artif. Intell. Rev.*, vol. 11, no. 1–5, pp. 273–314, Feb. 1997.

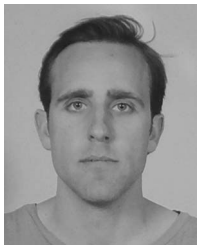
[24] R. Wilson and T. R. Martinez, "Reduction techniques for exemplar-based learning algorithms," *Mach. Learn.*, vol. 38, no. 3, pp. 257–286, Mar. 2000.

[25] J. Bentley and J. Friedman, "Data structures for range searching," *ACM Comput. Surv.*, vol. 11, no. 4, pp. 397–409, Dec. 1979.

[26] V. S. Borkar, "Stochastic approximation with two time scales," *Syst. Control Lett.*, vol. 29, no. 5, pp. 291–294, Feb. 1997.

Ivo Grondman received the B.Sc. degrees in applied mathematics and telematics from the University of Twente, Enschede, The Netherlands, in 2007 and the M.Sc. degree (with merit) in control systems from Imperial College London, London, U.K., in 2008. He is currently working toward the Ph.D. degree at the Delft Center for Systems and Control, Faculty of 3mE, Delft University of Technology, Delft, The Netherlands.

After graduation, he held a position with MathWorks, Cambridge, U.K., until 2009. His research interests include (approximate) dynamic programming and reinforcement learning, (optimal) control theory, and computational intelligence methods.



Maarten Vaandrager received the M.Sc. degree (*cum laude*) in systems and control from the Delft University of Technology, Delft, The Netherlands.

After graduation, he worked as a Technical Consultant with Stork FDO Inoteq until 2011. In 2011, he cofounded the software company Plotprojects, which develops smartphone applications with new location-based search services. His research interests include optimization techniques and learning algorithms for control applications.

Lucian Buşoniu received the M.Sc. degree (valedictorian) from the Technical University of Cluj-Napoca, Cluj-Napoca, Romania, in 2003 and the Ph.D. degree (*cum laude*) from the Delft University of Technology, Delft, The Netherlands, in 2009.

He is a research scientist with CNRS, Research Center for Automatic Control, University of Lorraine, Nancy, France. He is also with the Department of Automation, Technical University of Cluj-Napoca, Romania. His research interests include reinforcement learning and dynamic programming with function approximation, planning-based methods for nonlinear stochastic control, multiagent learning, and, more generally, intelligent and learning techniques for control. He has authored a book as well as a number of journal, conference, and chapter publications on these topics.

Dr. Buşoniu was the recipient of the 2009 Andrew P. Sage Award for the best paper in the IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS.

Robert Babuška received the M.Sc. degree (with honors) in electrical engineering from Czech Technical University, Prague, Czech Republic, in 1990 and the Ph.D. degree (*cum laude*) in control from the Delft University of Technology, Delft, The Netherlands, in 1997.

He is currently a full Professor with the Delft Center for Systems and Control, Delft University of Technology, Delft. His research interests include fuzzy system modeling and identification, data-driven construction and adaptation of neuro-fuzzy systems, and model-based fuzzy control and learning control. He is active in applying these techniques in robotics, mechatronics, and aerospace.

Erik Schuitema received the M.Sc. degree (with honors) in applied physics from the Delft University of Technology, Delft, The Netherlands, in 2006. He is currently working toward the Ph.D. degree in the Delft Biorobotics Laboratory, Faculty of Mechanical Engineering, Delft University of Technology, researching real-time reinforcement learning techniques for the control of a real bipedal walking robot.

His research interests include intelligent hardware and software for robotics, machine learning, autonomous agents, and software engineering.