Model Learning Actor–Critic Algorithms: Performance Evaluation in a Motion Control Task

Ivo Grondman[1], Lucian Buşoniu[2] and Robert Babuška[1]

***Abstract*— Reinforcement learning (RL) control provides a means to deal with uncertainty and nonlinearity associated with control tasks in an optimal way. The class of actor–critic RL algorithms proved useful for control systems with continuous state and input variables. In the literature, model-based actor–critic algorithms have recently been introduced to considerably speed up the the learning by constructing online a model through local linear regression (LLR). It has not been analyzed yet whether the speed-up is due to the model learning structure or the LLR approximator. Therefore, in this paper we generalize the model learning actor–critic algorithms to make them suitable for use with an arbitrary function approximator. Furthermore, we present the results of an extensive analysis through numerical simulations of a typical nonlinear motion control problem. The LLR approximator is compared with radial basis functions (RBFs) in terms of the initial convergence rate and in terms of the final performance obtained. The results show that LLR-based actor–critic RL outperforms the RBF counterpart: it gives quick initial learning and comparable or even superior final control performance.**

I. INTRODUCTION

Learning control can be used to optimize the control system's performance over a variety of conditions, which are difficult to predict or model for the purpose of off-line design. It can also deliver solutions to problems that we are currently not able to solve through conventional control design, due to the lack of suitable methods and tools. Reinforcement learning (RL) algorithms represent an important class of optimal control techniques which impose minimal assumptions on the process model properties and on the control task specifications. A general drawback of RL is its relatively slow convergence and therefore extensive learning times. To speed up learning, previous work [1] introduced two actor–critic reinforcement learning algorithms, which both use model learning capabilities and local linear regression (LLR) for all function approximations. Both methods belong to the class of model learning algorithms (also called indirect methods) as opposed to direct algorithms [2].

Model Learning Actor–Critic (MLAC) learns a process model and employs it to update the actor. Instead of using this process model to generate simulated experiences as most model learning RL algorithms do [3]–[5], it uses the model to directly calculate an accurate policy gradient, which accelerates learning compared to other policy gradient methods. Reference Model Actor–Critic (RMAC) also learns a process model, but in addition it also learns 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 goal of this paper is twofold. The first part of the paper generalizes the MLAC and RMAC algorithms of [1] to make them suitable for use with arbitrary function approximators. The second part then analyzes the performance of the two algorithms, using two different function approximators: a radial basis functions network and local linear regression. For the comparison we choose numerical simulations of the pendulum swing-up task, which is a highly nonlinear control problem commonly used in the literature [6], [7]. We are aware of the fact that the results do not necessarily carry over to other tasks (e.g., to higher-order systems). We do believe, however, that this represents a broad class of motion control problems characterized by dominant second-order dynamics, nonlinearity and a non-trivial solution, which is hard to find by alternative online learning methods.

II. REINFORCEMENT LEARNING

Reinforcement learning (RL) [2] can be used to solve problems modeled as Markov decision processes (MDPs). An MDP is a tuple $\langle X, U, f, \rho \rangle$, where X denotes the state space, U the action space, $f : X \times U \mapsto X$ the (deterministic) state transition function and $\rho : X \times U \mapsto \mathbb{R}$ 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+1} that the process reaches from state x_k after applying action u_k. After each transition to a state x_{k+1}, the controller receives an immediate scalar reward r_{k+1}, given by the reward function ρ, $r_{k+1} = \rho(x_k, u_k)$.

The goal in RL is to find the control policy $\pi : X \mapsto U$ that maximizes a function of the immediate rewards received while following the policy π. This function can simply be the sum or the average of all received immediate rewards, or a discounted sum, which is used in this paper. The value function $V^\pi : X \mapsto \mathbb{R}$ approximates the value of this discounted sum during learning:

$$V^\pi(x) = \mathrm{E}\left\{ \sum_{k=0}^{\infty} \gamma^k r_{k+1} \middle| x_0 = x, \pi \right\}$$

where $\gamma \in [0, 1)$ denotes the reward discount factor. The superscript π indicates that this is the discounted sum of rewards collected when following the policy π.

[1]I. Grondman and R. Babuška are with the Delft Center for Systems and Control of Delft University of Technology, The Netherlands ({i.grondman,r.babuska}@tudelft.nl).

[2]L. Buşoniu is with CNRS, Research Center for Automatic Control, University of Lorraine, Nancy, France and is also associated with the Department of Automation, Technical University of Cluj-Napoca, Romania (lucian@busoniu.net).

III. ACTOR–CRITIC ALGORITHMS

In real-life applications, such as robotics, processes usually have continuous state and action spaces, making it impossible to store exact value functions or policies for each separate state. Any RL algorithm used in practice will have to make use of function approximators for both value function and/or policy in order to cover the full continuous range of states and actions. Actor–critic algorithms [8], [9] facilitate the use of continuous state and action spaces in an easy way. As the policy (the actor) and value function (the critic) are stored separately, generating a control action does not—in contrast to critic–only methods—require an expensive (continuous) optimization procedure over the value function. Instead, control actions can be calculated directly from the learned policy.

Both the actor and the critic are usually parameterized functions. This allows both functions to easily operate on a continuous domain. The critic approximates and updates the value function using measured samples from interaction with the process. The value function is then used to update the actor's policy parameters in the direction of performance improvement.

Three actor–critic algorithms are evaluated in this paper. The descriptions of the algorithms here differ from the original ones in [1]. The focus is now more on the basic principles of the algorithms, rather than their implementation when using LLR. Moreover, they are now formalized in a more general way and are no longer tied to using local linear regression as the function approximator, but can use any function approximator that is linear in its parameters. For example: the actor, parameterized by $\vartheta \in \mathbb{R}^p$ and using basis functions $\psi(x)$, is defined as $\pi_\vartheta(x) = \vartheta^\top \psi(x)$. The critic, parameterized by $\theta \in \mathbb{R}^q$, is defined similarly and given by $V_\theta = \theta^\top \phi(x)$.

The critic part of all algorithms employs the same TD(λ) update

$$\delta_k = r_{k+1} + \gamma V_{\theta_k}(x_{k+1}) - V_{\theta_k}(x_k) \tag{1a}$$

$$z_k = \lambda\gamma z_{k-1} + \nabla_\theta V_{\theta_k}(x_k) \tag{1b}$$

$$\theta_{k+1} = \theta_k + \alpha_c \delta_k z_k \tag{1c}$$

This type of update, using eligibility traces $z_k \in \mathbb{R}^q$, is further explained in [2], [10]. The three algorithms differ in the way they represent and update the actor, which is further detailed in the remainder of this section.

A. Standard Actor–Critic

The Standard Actor–Critic (SAC) method uses the following heuristic estimate [11], [12] for the policy gradient $\nabla_\vartheta J$:

$$\nabla_\vartheta J(x_k) \approx \delta_k \Delta u_k \nabla_\vartheta \pi_\vartheta(x_k)$$

in which Δu_k is the random exploration term, drawn from a zero-mean normal distribution, that was added to the policy's output at time k. This results in the following update rule for the actor in SAC:

$$\vartheta_{k+1} = \vartheta_k + \alpha_a \delta_k \Delta u_k \nabla_\vartheta \pi_\vartheta(x_k) \tag{2}$$

The product of the exploration term Δu_k and the TD error from Equation (1a) serves as a sign switch for the gradient $\nabla_\vartheta \pi_\vartheta(x_k)$. When the exploration Δu_k leads to a positive TD error, the exploration is deemed beneficial to the performance and the policy is adjusted towards the perturbed action. Conversely, when δ_k is negative, the policy is adjusted away from this perturbation.

B. Model Learning Actor–Critic

In addition to learning the actor and critic functions, the Model Learning Actor–Critic (MLAC) method learns an approximate process model $x' = \hat{f}_\zeta(x, u)$. The process model is parameterized by $\zeta \in \mathbb{R}^{r \times n}$, where n is the state dimension. Having a learned process model available simplifies the update of the actor, as it allows to predict what the next state x' will be, given some input u. The value function then provides information on the value $V(x')$ of the next state x'. However, since the action space is assumed to be continuous, it is impossible to enumerate over all possible inputs u and therefore a policy gradient is put into place.

With appropriately chosen function approximators, the gradient of the value function with respect to the state x and the Jacobian of the process model with respect to the input u can be estimated. Then, by applying the chain rule, the Jacobian of the value function with respect to the input u becomes available. As the parameterization of the actor is known, this allows the policy gradient $\nabla_\vartheta J$ to be approximated by:

$$\nabla_\vartheta J(x_k) \approx \nabla_x V_\theta(x_k)^\top \nabla_u \widehat{f}_{\zeta_k}(x_k, u_k) \nabla_\vartheta \pi_\vartheta(x_k) \tag{3}$$

The process model itself is updated by applying a gradient descent update, using the error between the real output of the system and the model output and a learning rate α_p:

$$\zeta_{k+1} = \zeta_k + \alpha_p (x_{k+1} - \widehat{f}_{\zeta_k}(x_k, u_k)) \nabla_\zeta \widehat{f}_{\zeta_k}(x_k, u_k)$$

Note that with the SAC algorithm, which uses (2), exploration is needed in order to update the actor. Because MLAC uses the gradient in Equation (3) 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 gives a more complete value function over the entire state space as the current policy would only visit a part of the state space. Finally, exploration improves the model of the process dynamics.

C. Reference Model Actor–Critic

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. Instead of a policy, i.e. the actor, 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 an actor, mapping a state x onto

an action u, is not explicitly stored. Instead, the reference model is used in combination with the inverse of the learned process model to calculate the action u.

The parameterized reference model $R_\eta(x)$, with parameter $\eta \in \mathbb{R}^{s\times n}$, maps the state x_k to a desired next state $\widehat{x}_{k+1}$, i.e.

$$\widehat{x}_{k+1} = R_{\eta_k}(x_k)$$

The process is controlled towards this desired next state by using the inverse of the learned process model $x_{k+1} = \widehat{f}_{\zeta_k}(x_k, u_k)$. The reference model $R_{\eta_k}(x_k)$ and the inverse process model $u_k = \widehat{f}^{-1}_{\zeta_k}(x_k, x_{k+1})$ together act as a policy, by using the relation $u_k = \widehat{f}^{-1}_{\zeta_k}(x_k, R_{\eta_k}(x_k))$. This does imply that the process model, or more specifically the function approximator that represents it, is (locally) invertible. Here, invertibility is achieved by using a first order Taylor expansion of the process model around the point (x_k, u_{k-1}):

$$\widehat{x}_{k+1} = \widehat{f}_{\zeta_k}(x_k, u_{k-1}) + \nabla_u \widehat{f}_{\zeta_k}(x_k, u_{k-1})(u_k - u_{k-1})$$

From this equation, u_k can be directly calculated, given x_k, $\widehat{x}_{k+1}$ and u_{k-1}.

The introduction of a reference model also requires the introduction of an update rule for the reference model's parameters. A natural update rule for these parameters is to move them in the direction that will yield the highest value, i.e.

$$\eta_{k+1} = \eta_k + \alpha_r \nabla_x V(x_k)^\top \nabla_\eta R_{\eta_k}(x_k) \tag{4}$$

where $\alpha_r > 0$ is the learning rate of the reference model. Update (4) eventually may lead to an infeasible reference model if the output of $R_\eta(x)$ is 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:

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

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

To overcome this problem, it is assumed that the reachable set $\mathcal{R}_x$ can be defined by only using the extreme values of the action space U. Defining the set U_e as the finite discrete set containing all the combinations of extreme values[1] of U, the learned process model can be used to calculate the estimated next state when applying these extreme values:

$$\widehat{\mathcal{R}}_x = \left\{x' \in X | x' = \widehat{f}_\zeta(x, u), u \in U_e\right\}$$

Subsequently, the critic provides the estimated value of those next states and the state that yields the highest value is selected as the next desired state:

$$x_r = \arg\max_{x\in\widehat{\mathcal{R}}_x} V_\theta(x)$$

This procedure is justified by the assumption that if the sampling interval is short enough, both the process model and critic can be approximated locally by a linear function and linear functions always have their extreme values on the edges of their input domain.

[1]This requires the action space U to be a hyperbox.

Algorithm 1 Actor–Critic template

Input: γ, λ and learning rates α

$z_0 = 0$
Initialize x_0 and function approximators
Apply random input u_0
$k \leftarrow 0$
loop
  Measure x_{k+1}, r_{k+1}
  $\delta_k \leftarrow r_{k+1} + \gamma V_{\theta_k}(x_{k+1}) - V_{\theta_k}(x_k)$
  // SAC/MLAC: Choose action / update actor
  $u_{k+1} \leftarrow \pi_{\vartheta_k}(x_{k+1})$
  $\vartheta_{k+1} \leftarrow \vartheta_k + \alpha_a \nabla_\vartheta J(x_k)$
  // RMAC: Choose action / update reference model
  $\widehat{x}_{k+2} \leftarrow R_{\eta_k}(x_{k+1})$
  $u_{k+1} \leftarrow \widehat{f}^{-1}_{\zeta_k}(x_{k+1}, \widehat{x}_{k+2})$
  Select best reachable state x_r from x_k
  $\eta_{k+1} \leftarrow \eta_k + \alpha_r(x_r - \widehat{x}_{k+1})\nabla_\eta R_{\eta_k}(x_k)$
  // MLAC/RMAC: Update process model
  $\zeta_{k+1} \leftarrow \zeta_k + \alpha_p(x_{k+1} - \widehat{f}_{\zeta_k}(x_k, u_k))\nabla_\zeta \widehat{f}_{\zeta_k}(x_k, u_k)$
  // Update critic
  $z_k \leftarrow \lambda\gamma z_{k-1} + \nabla_\theta V_{\theta_k}(x_k)$
  $\theta_{k+1} \leftarrow \theta_k + \alpha_c \delta_k z_k$
  Choose exploration $\Delta u_{k+1} \sim \mathcal{N}(0, \sigma^2)$
  Apply $u_{k+1} + \Delta u_{k+1}$
  $k \leftarrow k + 1$
end loop

The state x_r is used to update the reference model:

$$\eta_{k+1} = \eta_k + \alpha_r(x_r - \widehat{x}_{k+1})\nabla_\eta R_{\eta_k}(x_k) \tag{5}$$

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 $R_\eta(x)$, a smooth reference model and a smooth policy can still be achieved. This approximation does mean, though, that the algorithm will not obtain a near-optimal solution and a more accurate calculation of the reachable set should improve the performance.

In contrast to SAC, the RMAC improves the reference model using (5) which does not involve exploration. Instead, it improves the reference model on the basis of previous experiences, but for the same reasons as with MLAC exploration is still needed.

A summarizing template for the three algorithms that were discussed in this section is given in Algorithm 1.

IV. APPROXIMATORS

In Section V, the algorithms described in the previous section are evaluated using two types of approximators: radial basis functions (RBFs) and local linear regression (LLR). This section describes the two approximators and also

discusses the caveats that have to be taken into consideration when using them with the algorithms.

A. Radial Basis Functions

The first type of approximator is a linear combination of normalized RBFs. The critic, for example, would be modeled by

$$V_\theta(x) = \theta^\top \phi(x)$$

where $\phi(x)$ is a column vector with the value of normalized RBFs:

$$\phi_i(x) = \frac{\widetilde{\phi}_i(x)}{\sum_j \widetilde{\phi}_j(x)} \tag{6}$$

with

$$\widetilde{\phi}_i(x) = e^{-\frac{1}{2}(x-c_i)^\top B^{-1}(x-c_i)} \tag{7}$$

where c_i are the centers of the RBFs and B is a diagonal matrix containing the widths of the RBFs.

For MLAC and RMAC, the gradient of the approximated functions is needed. With the normalization done in Equation (6), this gradient is:

$$\nabla_x \phi_i(x) = -\phi_i(x)\left[B^{-1}(x-c_i) + \frac{\sum_j \nabla_x \widetilde{\phi}_j(x)}{\sum_j \widetilde{\phi}_j(x)}\right]$$

The setup used for experiments described in Section V is a system with input saturation. This means that for the real system, the process gradient $\nabla_u f(x_k, u_k) = 0$ when u_k is outside of the allowed input range. As MLAC learns a process model, this input saturation needs to be dealt with when learning. Otherwise, a good policy will not be produced. In the RBF case, this problem is dealt with by setting the gradient $\nabla_u \widehat{f}_\zeta(x_k, u_k)$ in Equation (3) to zero when u_k is close to the input saturation bounds.

B. Local Linear Regression

The other type of function approximation used is local linear regression (LLR). Given a memory of a certain size, filled with input/output samples, it is possible to estimate the output for any arbitrary "query input", by:

1) Finding the k samples in the memory that are nearest to the query input, according to a (weighted) distance metric.
2) Fitting a linear model to these k samples by performing a least squares fit.
3) Applying this model to the query input.

The saturation issue with the process model learning in MLAC pictured earlier also holds in the LLR case. Here, however, a solution for keeping the values bounded is much more straightforward. As the LLR memories hold input/output samples, it is possible to saturate the output part of the actor's memory, such that it can never produce inputs u beyond the saturation bound. Because of this, setting $\nabla_u \widehat{f}(x_k, u_k) = 0$ when u_k is close to the input saturation bounds is not necessary here. A broader discussion on this approximator and its application to the algorithms in this paper is given in [1].

V. RESULTS

To evaluate and compare the performance of the algorithms with different function approximators, they are all applied to the task of learning to swing up an inverted pendulum as described in [1]. The swing-up must be done from the pointing-down position to the upright position as quickly as possible and the pendulum must be stabilized in this position. Using limited actuation makes 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. This task was chosen because it is a low-dimensional, but challenging, highly nonlinear control problem still commonly used in RL literature [6], [7]. A continuous quadratic reward function

$$r_{k+1}(x_k, u_k) = -x_k^\top \begin{bmatrix} 5 & 0 \\ 0 & 0.1 \end{bmatrix} x_k - u_k^2 \tag{8}$$

that has its maximum in the upright position $[0\ 0]^\top$ is used to define the swing-up task. This reward function quadratically penalizes non-zero values of the pendulum's angle φ and angular velocity $\dot{\varphi}$ and the control input u.

The three actor–critic methods, SAC, MLAC and RMAC are applied to the pendulum swing-up problem described above using two different function approximation techniques: local linear regression and radial basis functions.

The algorithms run for 30 minutes of simulated time, consisting of 600 consecutive episodes with each episode 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, $x_0 = [\pi\ 0]^\top$. One *learning experiment* consists of one complete run of 600 consecutive episodes. A set of 30 complete learning experiments is done per set of tuning parameters to get a good estimate of the mean performance.

Two performance measures have been used while testing each algorithm/approximator combination: quick initial learning and best performance. The optimal tuning for quick initial learning was decided on by measuring the average reward obtained in the first 50 episodes of all 30 learning experiments, which is equivalent to the first 2.5 minutes of simulated time. The optimal tuning for best performance was based on the average reward obtained in the last 50 episodes of all 30 learning experiments.

Tuning of all algorithm/approximator combinations was done by iterating over a grid of parameters, which included parameters of the algorithm (i.e. the learning rates) as well as parameters of the function approximator (neighborhood sizes in case of LLR and widths of RBFs, for example). The optimal parameters for both test scenarios are listed in Table I.

Most parameters listed in the tables are self-explanatory, but the parameters listed for the RBF function approximator need some additional comment. The number of RBFs is given by a vector. Each element corresponds to the number of RBFs used in a particular dimension of its input space. For example, if the critic model has a number of RBFs equal

TABLE I

THE OPTIMAL PARAMETERS WHEN TUNING FOR QUICK INITIAL LEARNING (REGULAR FONT) AND BEST PERFORMANCE (BOLD FONT).

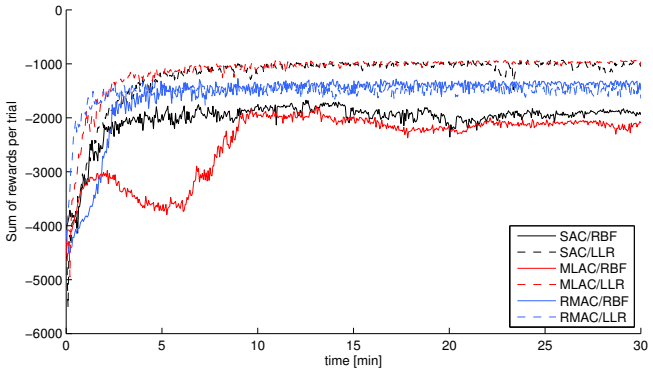
	SAC				MLAC				RMAC			
	RBF		LLR		RBF		LLR		RBF		LLR	
Actor / Reference model parameters												
learning rate $\alpha_{a/r}$	0.1	**0.05**	0.1	**0.05**	0.01	**0.001**	0.18	**0.12**	0.7	**0.5**	0.12	**0.04**
memory size $N^{a/r}$			2000	**2000**			2000	**2000**			2000	**2000**
nearest neighbors $k^{a/r}$			25	**20**			25	**25**			25	**15**
number of RBFs	$\begin{bmatrix}15\\10\end{bmatrix}$	$\begin{bmatrix}\mathbf{20}\\\mathbf{10}\end{bmatrix}$			$\begin{bmatrix}10\\10\end{bmatrix}$	$\begin{bmatrix}\mathbf{10}\\\mathbf{10}\end{bmatrix}$			$\begin{bmatrix}15\\10\end{bmatrix}$	$\begin{bmatrix}\mathbf{10}\\\mathbf{10}\end{bmatrix}$		
RBF intersection	0.5	**0.9**			0.7	**0.7**			0.7	**0.9**		
Critic parameters												
learning rate α_c	0.3	**0.4**	0.3	**0.3**	0.15	**0.1**	0.3	**0.3**	0.2	**0.15**	0.3	**0.2**
memory size N^c			2000	**1000**			2000	**2000**			2000	**2000**
nearest neighbors k^c			25	**10**			20	**15**			25	**15**
number of RBFs	$\begin{bmatrix}15\\10\end{bmatrix}$	$\begin{bmatrix}\mathbf{20}\\\mathbf{10}\end{bmatrix}$			$\begin{bmatrix}10\\10\end{bmatrix}$	$\begin{bmatrix}\mathbf{10}\\\mathbf{10}\end{bmatrix}$			$\begin{bmatrix}15\\10\end{bmatrix}$	$\begin{bmatrix}\mathbf{10}\\\mathbf{10}\end{bmatrix}$		
RBF intersection	0.5	**0.5**			0.9	**0.7**			0.5	**0.7**		
Process model parameters												
learning rate α_p					0.9	**0.5**			0.9	**0.7**		
RBF intersection					0.7	**0.9**			0.7	**0.7**		

to $[15\ 10]$, this means that 15 RBFs are used in the first space dimension φ and 10 in the second space dimension $\dot{\varphi}$, amounting to 150 RBFs in total. For the process model, the number of RBFs per dimension were kept equal to that of the actor and critic model, with the number of RBFs in the action space dimension always fixed at 10. The outer RBFs always have their centers at $\pm\pi$ in the φ dimension, $\pm 8\pi$ in the $\dot{\varphi}$ dimension and ± 3 in the action space U. The intersection height is the value of Equation (7) where two neighboring RBFs intersect and is deemed to be a more intuitive measure than the width of an RBF. The intersection height is chosen to be equal in all dimensions of the input space. This means that the widths of an RBF in the separate dimensions are not fixed and depend on the chosen number of RBFs: if the intersection height is kept fixed and more RBFs are used, the width of those RBFs decreases.

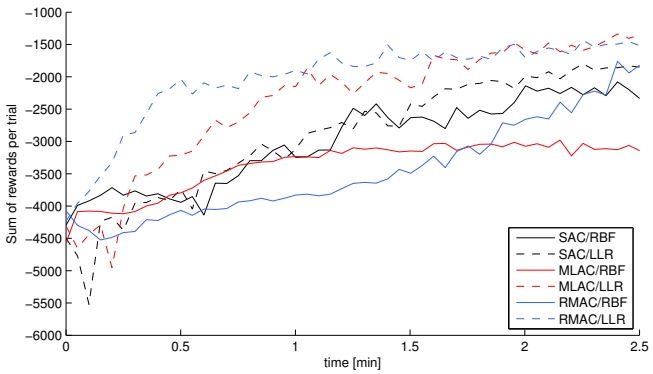
For an LLR approximation of the process model, the memory size and number of nearest neighbors were kept fixed at $N^p = 100$ and $k^p = 10$ for all experiments.

The first result in Figure 1 compares the performance of the initial learning of each algorithm, using the two different function approximation techniques. Figure 1a shows the complete learning curves, whereas Figure 1b zooms in on the first 2.5 minutes of learning, to get a better view on the initial learning phase.

It is clear from this picture that all methods learn the quickest when they use LLR as the function approximator. Furthermore, the MLAC and RMAC methods are performing better than SAC in this respect. When using RBFs, the MLAC and RMAC method do not perform better than SAC, which implies that they need a better function approximator to become truly powerful. In addition, the LLR implementations of SAC and MLAC also obtain a much better performance at the end of the learning experiment than their RBF counterparts, despite only tuning for quick initial learning. The RMAC/LLR combination shows the steepest learning curve at the start, but does not reach the

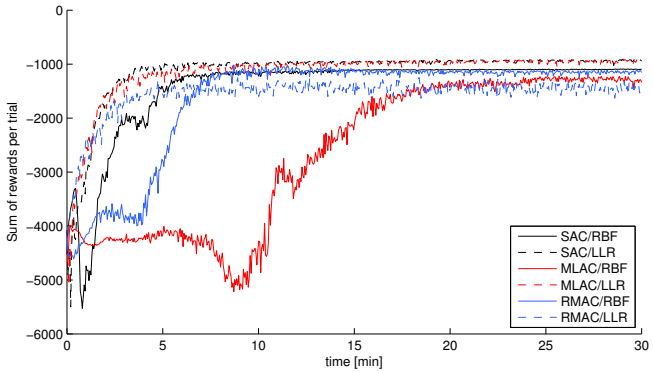


(a) The complete learning curves of all algorithms and function approximators.

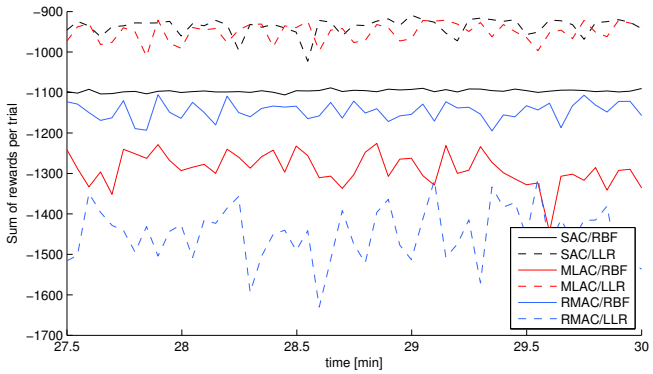


(b) The first 2.5 minutes of the learning curves.

Fig. 1. Comparison of different algorithms and function approximators when tuned for best initial learning in the first 50 episodes.



(a) The complete learning curves of all algorithms and function approximators.



(b) The last 2.5 minutes of the learning curves.

Fig. 2. Comparison of different algorithms and function approximators when tuned for best performance in the last 50 episodes.

best solution.

Figure 2 shows the comparison between the algorithms when they are tuned for best performance in the last 50 episodes, i.e. the tuning that will learn the best possible policy. The first plot shows the complete learning experiments, the second plot zooms in on the performance in the last 50 episodes.

Again, but with the exception of RMAC, the methods using LLR as the function approximator perform the best. The RBF implementations of MLAC and RMAC show a slightly worse performance than their SAC/RBF counterpart. Interestingly, the RMAC method is the only method here that performs better when using RBFs as the function approximator. A noticeable phenomenon when comparing Figure 1a and Figure 2a is that the initial performance of the RBF implementations has degraded significantly, whereas the LLR implementations still achieve quick initial learning.

VI. CONCLUSION

From the results in the previous section, local linear regression seems to be the approximator of choice when the learning should both be quick and deliver a good performance at the end of a learning experiment.

The MLAC and RMAC methods only really show their power when used in combination with local linear regression. This reinforces the statement in [1] that it is indeed the combination of both LLR and these new methods that will provide quick, stable and good learning. However, due to time limitations no tests were done using a mix of function approximators, e.g. RBFs for the actor and critic and LLR for the process model, which might yield even more powerful results. Another option is to try and combine the quick learning of RMAC/LLR with the good performance of MLAC/LLR, by starting with RMAC and switching to MLAC when the performance does not significantly increase anymore.

A crucial condition for the model learning algorithms to work is that the input saturation of a system should be dealt with when learning a process model. Simply ignoring this will not produce a well performing policy. This can be overcome by setting the process model's gradient with respect to the input to zero at the saturation bounds or by making sure that the actor can not produce output signals beyond the saturation bounds.

A persisting problem in dynamic programming and reinforcement learning is the proper tuning of the learning algorithms and, if applicable, function approximators. The quality of an RL algorithm should therefore not only be measured by looking at the performance of the policy it produces, but also by checking its robustness, i.e. how well it keeps performing when deviating from the set of optimally tuned parameters. This sensitivity analysis for our algorithms is left for future work.

REFERENCES

[1] I. Grondman, M. Vaandrager, L. Buşoniu, R. Babuška, and E. Schuitema, "Efficient Model Learning Methods for Actor-Critic Control," *IEEE Transactions on Systems, Man, and Cybernetics—Part B: Cybernetics*, vol. 42, no. 3, pp. 591–602, 2012.

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

[3] R. S. Sutton, "Reinforcement Learning Architectures," *Proceedings of the International Symposium on Neural Information Processing*, 1992.

[4] A. W. Moore and C. G. Atkeson, "Prioritized Sweeping: Reinforcement Learning with Less Data and Less Time," *Machine Learning*, vol. 13, pp. 103–130, 1993.

[5] L. Kuvayev and R. S. Sutton, "Model-Based Reinforcement Learning with an Approximate, Learned Model," in *Proceedings of the 9th Yale Workshop on Adaptive and Learning Systems*, 1996, pp. 101–105.

[6] J. Pazis and M. G. Lagoudakis, "Reinforcement Learning in Multidimensional Continuous Action Spaces," in *Proceedings of the IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning*, 2011, pp. 97–104.

[7] R. L. S. de Arruda and F. J. Von Zuben, "A Neural Architecture to Address Reinforcement Learning Problems," in *Proceedings of International Joint Conference on Neural Networks*, 2011, pp. 2930–2935.

[8] I. H. Witten, "An Adaptive Optimal Controller for Discrete-Time Markov Environments," *Information and Control*, vol. 34, pp. 286–295, 1977.

[9] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 13, no. 5, pp. 834–846, 1983.

[10] V. R. Konda and J. N. Tsitsiklis, "On Actor-Critic Algorithms," *SIAM Journal on Control and Optimization*, vol. 42, no. 4, pp. 1143–1166, 2003.

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

[12] L. Buşoniu, R. Babuška, B. De Schutter, and D. Ernst, *Reinforcement Learning and Dynamic Programming Using Function Approximators*, ser. Automation and Control Engineering Series. CRC Press, 2010.