Dyna-MLAC: Trading Computational and Sample Complexities in Actor-Critic Reinforcement Learning

Bruno Costa
Centro de Pesquisa de Energia Elétrica
Rio de Janeiro, Brazil
Email: brunosc@cepel.br

Wouter Caarls
Federal University of Rio de Janeiro
Rio de Janeiro, Brazil
Email: wouter@caarls.org

Daniel Sadoc Menasché
Federal University of Rio de Janeiro
Rio de Janeiro, Brazil
Email: sadoc@dcc.ufrj.br

***Abstract*—Sampling and computation budgets are two of the key elements that determine the performance of a reinforcement learning algorithm. In essence, any reinforcement learning agent must sample the environment and perform some computation over the samples to decide its best action. Although very fundamental, the trade-off between sampling and computation is still not well understood. In this paper, we explore this trade-off in an actor-critic perspective. First, we propose a new RL algorithm, Dyna-MLAC, which uses model-based actor-critic updates (MLAC) within the Dyna framework. Then, we numerically indicate that the convergence time of Dyna-MLAC is smaller than pre-existing solutions, and that Dyna-MLAC allows to efficiently trade number of samples and computation time.**

I. Introduction

Reinforcement Learning (RL) is a field of machine learning inspired by psychology and biology, concerned with how agents learn which actions to take in an environment in order to maximize some cumulative reward. At any point in time, the agent knows the current state and, after taking some action, it learns the resulting state and the obtained instantaneous reward. The tuple comprising the current state, action, resulting state and instantaneous reward is referred to as a *sample*. Given the current state and an action, the *state transition function* yields the resulting next state and an instantaneous reward. The transition function is unknown to the agent and typically stochastic.

Collecting and processing samples are two of the most fundamental activities performed by any reinforcement learning agent. In essence, the performance of any reinforcement learning algorithm must account for sampling and computation costs, also referred to as sampling and computational complexity [1]. Low sampling complexity algorithms are favored in situations where samples are costly or even dangerous to obtain, such as the control of manufacturing plants. Low computational complexity algorithms, on the other hand, may be preferred when samples are abundant (big data), or in *real-time* systems, where decisions have to be made within a short time interval.

Theoretical work on sampling complexity is focused on PAC (Probably Approximately Correct) algorithms [2], [3], which provide analytical bounds for the required number of samples. However, these bounds are very loose, and mostly apply to discrete state and action spaces (C-PACE [4] is a notable exception). In addition, the computational complexity for these algorithms is fixed.

Algorithms with low *empirical* sampling complexity are usually characterized by the re-use of samples, often in the form of a learned *process model* that approximates the state transition function. These algorithms are broadly classified as *model-based* solutions, e.g. PILCO [5], and have a very high computation cost. Conversely, algorithms which are computationally cheap, such as classical Q-learning [6], require a large number of samples to attain good performance. The required number of samples often precludes the use of such *model-free* solutions in realistic scenarios, and hampers their applicability.

In this work, the trade-off between computational complexity and sampling complexity is explored in an actor-critic perspective. Actor-critic algorithms use separate, explicit representations of the state-action mapping and expected cumulative reward in order to deal with continuous states and actions such as found in robotics applications [7]. The Dyna framework [8], a reinforcement learning framework that can scale smoothly between completely model-based and model-free modes, is a natural choice for investigating the trade-off. It was previously used for this purpose in previous work on discrete action spaces [9].

A *learning update rule* (or update rule, for short) determines how the solution must be modified as new samples are gathered. As Dyna learns a process model, a natural extension consists of its coupling with model-based learning update rules. As such, two algorithms are considered: 1) Dyna-SAC (Standard Actor-Critic), using standard temporal-difference update rules [10] and 2) Dyna-MLAC, using the model-based update rules of the MLAC (Model-Learning Actor-Critic) algorithm [11]. Learning a process model from the samples can be done using any supervised learning technique, like neural networks [12]. In this work, we use a memory based algorithm, Locally Linear Regression (LLR) [13].

In summary, the key question addressed in this paper is the following: *to what extent is it possible to trade number of samples and computation time within the Dyna actor-critic framework?* In answering this question, we provide the following contributions[1]:

1) *algorithm design*: we propose a version of Dyna-SAC using LLR and the new Dyna-MLAC algorithm, both with continuous action and state spaces. Our key insight consists of using LLR as function approximator and using MLAC updates within the Dyna framework;

[1]More details may be found in [14]

2) *convergence analysis*: we analyze the number of updates required in order to stabilize the learning curve. We verify that although Dyna-SAC and Dyna-MLAC achieve the same end performance after a significant number of updates per control step, Dyna-MLAC demands less iterations to converge.

II. Background

Our solution has two basic building blocks: actor-critic learning methods and locally linear regression.

A. Reinforcement Learning

Reinforcement Learning (RL) [15] agents are typically concerned with solving a Markov Decision Process (MDP). An MDP is a tuple (S, A, T, γ, D, R), where S denotes a set of states; A is a set of actions; $T = p(s'|s, a)$ is the state transition function which encodes the distribution of next states s' upon taking action a in state s; γ is a discount factor for future rewards; D is the initial-state distribution, from which the start state s_0 is drawn; and R is the reward function $R(s, a, s')$ upon transitioning from state s to s' through action a.

The policy $\pi : S \mapsto A$ determines what action to take in every possible state. The goal of the agent is to maximize the total expected reward in a possibly infinite-horizon task. The expected reward at time step t is the sum of all future intermediate rewards r:

$$R_t = \mathbb{E}\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots\} = \mathbb{E}\left\{\sum_{t=0}^{\infty} \gamma^t r_{t+1}\right\} \quad (1)$$

where the discount factor $\gamma \in (0, 1]$ weights the importance between future rewards and present rewards.

The value function $V^\pi : S \mapsto \mathbb{R}$ determines the expected sum of rewards obtained by following a given policy π with initial state s, and is defined as:

$$V^\pi(s) = \mathbb{E}_\pi \{R_t | s_t = s\} \quad (2)$$

The value function satisfies the Bellman equation [16]:

$$V^\pi(s) = \sum_{s'} T(s, \pi(s), s') \Big(r(s, \pi(s), s') + \gamma V^\pi(s') \Big) \quad (3)$$

Finding the optimal policy and associated value function is referred to as *solving* the MDP. The classical ways to solve an MDP [17] require full knowledge of all variables, which is hard, if not impossible, to obtain. Instead, reinforcement learning methods *sample* the environment without requiring full knowledge of all MDP variables, specially the state transition function T. In particular, *temporal difference* methods use samples of the right-hand side of (3) to iteratively approximate the Bellman equation.

Let δ_t be the *temporal difference error* (TD error) [10] of the expected reward after a transition from state s_{t-1} to state s_t at time t,

$$\delta_t = r_t + \gamma V(s_t) - V(s_{t-1}) \quad (4)$$

Actions associated to positive TD error (better than expected result) are iteratively reinforced as new samples are gathered.

B. Standard Actor-Critic

One class of temporal difference methods, known as actor-critic [18], iterates in searching the optimal value function and the optimal policy, where the first is performed by the critic and the latter by the actor. Actor-critic methods have a separate structure for the value function (critic) and for the policy (actor). Policies computed using the actor-critic approach usually admit compact representations and naturally extend to continuous state and action spaces.

The SAC algorithm is described in Algorithm 1. After initialization (lines 3-6), the agent samples its current state and instantaneous reward (line 8) and chooses an action to execute. Let a_t be the action executed by the actor at time t. To learn about the environment and to avoid local minima, a_t accounts for the policy learned so far and a white noise term Δ_t (zero-mean Gaussian). Then,

$$a_t = \pi(s_t) + \Delta_t \quad (5)$$

$\pi(s_t)$ and Δ_t are referred to as the exploitation and exploration components of the action, respectively (line 10).

The Standard Actor-Critic (SAC) updates are done using the temporal difference error. The temporal difference error is obtained according to (4) (line 11). Then, procedure SAC-Update is called to update the value function and the policy (lines 16-20 of Algorithm 1). The value function is updated towards minimizing the TD error (line 18), while the policy is adjusted towards the explored action only if the TD error was positive (and away from it otherwise, line 19). In Algorithm 1, α_a and α_c are the learning step for the actor and for the critic, respectively.

Algorithm 1 SAC algorithm

```
1:  procedure SAC
2:  Repeat forever:
3:      ∀s ∈ S : e(s) ← 0
4:      s_0 ← Initial state
5:      Apply random input a_0
6:      t ← 1
7:      loop until episode ends:
8:          Measure s_t and r_t
9:          Let Δ_t be a sample from a zero-mean Gaussian
10:         a_t ← π(s_t) + Δ_t                    ▷ Choose an action
11:         δ_t = r_t + γV(s_t) − V(s_{t−1})      ▷ Calculate td-error
12:         Call SAC-Update(δ_t, Δ_{t−1}, α_a, α_c, s_t)
13:         Apply a_t
14:         t ← t + 1
15: procedure SAC-UPDATE(δ_t, Δ_{t−1}, α_a, α_c, s_t)
16:     Update the eligibility trace e_t(s_t)
17:     for all s ∈ S do
18:         V(s) ← V(s) + α_c δ_t e_t(s)          ▷ Update the critic
19:     π(s_t) ← π(s_t) + α_a δ_t Δ_{t−1}         ▷ Update the actor
20:     Clamp π(s) to A
```

C. Eligibility Traces

Procedure SAC-Update updates all "eligible" elements of the value function (line 17-18). *Eligibility traces* are used to keep a history of the past visited states. They assign a weight to each state, in such a way that most recently visited states

are associated to greater weights, allowing for a refined way to credit past experiences. Let $\lambda \in [0, 1)$ be the eligibility decay rate. Let $\mathbf{e}_t$ be the *eligibility vector* at time t, $\mathbf{e}_t = (e_t(1), e_t(2), \ldots, e_t(|S|))$. $\mathbf{e}_t$ is initialized with zeros and, after each time step, it is updated as follows

$$e_t(s) = \begin{cases} \lambda\gamma e_{t-1}(s) & \text{if } s \neq s_t \\ 1 & \text{if } s = s_t \end{cases} \tag{6}$$

Weights decay by a factor of $\lambda\gamma$ at each step.

D. Locally Linear Regression

When searching for the policy π that maximizes the value function V, the policy and the value function are typically approximated using function approximators [19]. Function approximators for V and π are particularly useful to handle continuous state and action spaces. In our proposed solution, we use locally linear regression, a nonparametric, memory-based function approximator [13]. Although the hyperspace being approximated can be quite complex, if a small region is considered, it can usually be well approximated by a linear model. LLR stores samples, hence a memory-based approximator, to linearize the hyperspace around a point.

Building step: The building step in LLR is simply add the sample to the memory, taking account some kind of memory management [20] since the LLR memory is finite and the transitions can easily exceed it. Considering a memory of size N, let $\mathbf{m}_i$ be a stored sample, $\mathbf{m}_i = [\mathbf{x}_i, \mathbf{y}_i]$, where $i = 1, \ldots, N$. One sample $\mathbf{m}_i$ is a row vector containing the input data $\mathbf{x}_i \in \mathbb{R}^n$ and output data $\mathbf{y}_i \in \mathbb{R}^\ell$. The samples are stored in a matrix called the memory $\mathbf{M} \in \mathbb{R}^N \times \mathbb{R}^{n+\ell}$. Each row of the memory stores a sample.

Querying step: Given an input query $\mathbf{q} \in \mathbb{R}^n$ and memory $\mathbf{M}$, our goal is to determine the output $\hat{\mathbf{y}} \in \mathbb{R}^\ell$. To this aim, a linear model around the query is considered. First, the k-nearest neighbors of $\mathbf{q}$, denoted $\mathcal{K}_\mathbf{q}$, are found in the LLR memory. For performance purposes, the search is done with the help of a *k-d* tree [21].

Let $\mathbf{X}_\mathbf{q} \in \mathbb{R}^k \times \mathbb{R}^{n+1}$ and $\mathbf{Y}_\mathbf{q} \in \mathbb{R}^k \times \mathbb{R}^\ell$ be the input and output data matrices associated to the k-nearest neighbors of $\mathbf{q}$. Each row of $\mathbf{X}_\mathbf{q}$ contains input data corresponding to one of the k-nearest neighbors of $\mathbf{q}$ padded with a constant term equal to one, added to allow for a bias on the output. The bias makes the model affine instead of truly linear. The i-th row of $\mathbf{Y}_\mathbf{q}$ contains output data corresponding to the i-th row of $\mathbf{X}_\mathbf{q}$. Then, a linear model in the parameters $\boldsymbol{\beta} \in \mathbb{R}^{n+1} \times \mathbb{R}^\ell$ for a given input $\mathbf{q}$ is $\mathbf{X}_\mathbf{q}\boldsymbol{\beta} = \mathbf{Y}_\mathbf{q}$. The solution is obtained using the least square method and yields $\boldsymbol{\beta}$. The estimated output $\hat{\mathbf{y}}$ is given by:

$$\hat{\mathbf{y}} = [\mathbf{q}, 1]\boldsymbol{\beta} \tag{7}$$

Learning step: Learning consists in two steps: inserting a sample and updating the existing ones. Take for an example the actor updates in line 19. Each sample $\mathbf{m}_i$ stores a state $s \in S$ as the input $\mathbf{x}_i$ and an action $a \in A$ as the output $\mathbf{y}_i$.

The evaluation of the right hand side of line 19 involves a query of $\pi(s)$, solved using the nearest neighbors $\mathcal{K}_s$ and (7). Let $\hat{\mathbf{y}}$ be the obtained result. Then, a new sample $[s, \hat{\mathbf{y}} + \alpha_a\delta_t\Delta_{t-1}]$ is inserted into memory $\mathbf{M}$. Afterwards, the output of all samples in $\mathcal{K}_s$ is adjusted by adding $\alpha_a\delta_t\Delta_{t-1}$ to each of them as well. Similar steps are executed in the critic update (line 18), using a separate memory wherein each sample $\mathbf{m}_i$ stores a state $s \in S$ as the input $\mathbf{x}_i$, and the expected return as the output.

III. Algorithms

A. Model Learning Actor-Critic

The Model Learning Actor-Critic (MLAC) [11] extends Standard Actor-Critic (SAC) by considering a *process model*. A process model is a function $\hat{f}$ which approximates the state transition function, relating every state-action pair (s, a) to its corresponding predicted state s', $s' = \hat{f}(s, a)$. After every new measurement of s_t (line 8 of Algorithm 1), the process model must be updated accordingly. In this work, we consider process models approximated by LLR, analogous to the actor and critic.

Next, we assume that a process model is given and briefly introduce the MLAC actor update. As the LLR approximation of the value function gives us its gradient with respect to the state $\partial V/\partial s$, and the process model $\hat{f}$ gives us the gradient of the next state with respect to the action $\partial s'/\partial a$, we can use the chain rule to determine the gradient of the value of the next state with respect to the action, allowing us to update the actor towards maximizing the value of the next state $V(s')$.

As such, for every state s, the MLAC gradient-descent actor update is given as follows

$$\pi(s) \leftarrow \pi(s) + \alpha_a \frac{\partial V}{\partial s}\bigg|_{s=s'} \frac{\partial s'}{\partial a} \tag{8}$$

MLAC is obtained from Algorithm 1 by updating the process model after line 8 and substituting line 19 by (8).

In the original reference for MLAC, LLR was also used, hence the choice for this work. Not only the partial derivatives from (8) are easily obtained [11], but the authors have also evaluated the performance using radial basis functions [22] and achieved inferior results.

B. Dyna-SAC

The Dyna framework [8] was proposed as way to accelerate the learning process by using a model to simulate real-world interactions. Using a process model, Dyna updates both the actor and the critic in the same way as SAC, but using the learned model to simulate new samples which mimic real-world interactions. A number of updates using the learned model are done per control step, i.e., every time the agent learns using the real world, it also simulates a fixed number of interactions using the learned model. Dyna-SAC is shown in Algorithm 2, and uses the same SAC-Update procedure from Algorithm 1.

The simulated environment may be restarted in some situations (line 25), such as: 1) in an episodic task, i.e., a task that admits a terminal state, the simulated environment should be restarted every time a terminal state is reached or 2) the estimated variance of the predicted state becomes too high, indicating an inaccurate process model [13]. Note that the hyperspace region close to the initial state is typically the

Algorithm 2 Dyna-SAC algorithm

procedure DYNA-SAC
Repeat forever:
  $\forall s \in S : e(s) \leftarrow 0$
  $s_0 \leftarrow$ Initial state
  Apply random input a_0
  $t \leftarrow 1$
  $\overline{s}_0 \leftarrow$ Initial state
  $\overline{a}_0 \leftarrow$ random action
  $\overline{t} \leftarrow 1$
  loop *until episode ends*:
    Choose Δ_t at random
    Measure s_t and r_t
    Update the process model using $[s_{t-1}, a_{t-1}, s_t]$
    $a_t = \pi(s_t) + \Delta_t$ ▷ Choose an action
    $\delta_t = r_t + \gamma V(s_t) - V(s_{t-1})$ ▷ Calculate td-error
    Call SAC-Update(δ_t, Δ_{t-1}, α_a, α_c, s_t)
    for fixed number of updates per control step **do**
      Choose $\overline{\Delta}_{\overline{t}}$ at random
      $\overline{s}_{\overline{t}} \leftarrow \hat{f}(\overline{s}_{\overline{t}-1}, \overline{a}_{\overline{t}-1})$ ▷ Next simulated state
      $\overline{r}_{\overline{t}} \leftarrow R(\overline{s}_{\overline{t}-1}, \overline{a}_{\overline{t}-1}, \overline{s}_{\overline{t}})$ ▷ Transition reward
      $\overline{a}_{\overline{t}} \leftarrow \pi(\overline{s}_{\overline{t}}) + \overline{\Delta}_{\overline{t}}$ ▷ Next simulated action
      $\overline{\delta}_{\overline{t}} \leftarrow \overline{r}_{\overline{t}} + \gamma V(\overline{s}_{\overline{t}}) - V(\overline{s}_{\overline{t}-1})$
      Call SAC-Update($\overline{\delta}_{\overline{t}}$, $\overline{\Delta}_{\overline{t}-1}$, $\alpha_{sa}, \alpha_{sc}, \overline{s}_{\overline{t}}$)
      $\overline{t} \leftarrow \overline{t} + 1$
      if should restart simulated environment? **then**
        $\overline{s}_0 \leftarrow$ Initial state ▷ Restart model
        $\overline{a}_0 \leftarrow$ random action
        $\overline{t} \leftarrow 1$
    Apply a_t
    $t \leftarrow t + 1$

region that is sampled most often. Hence, the process model has higher confidence close to the initial state, which motivates the more intense usage of that region in the learning process.

The values for the learning steps α_{sa} and α_{sc} can be different from their non-Dyna version (α_a and α_c, respectively). Using a lower value makes the Dyna updates less important and can account for model error, for example. In this work, the values being used are the same for the Dyna and the non-Dyna, as shown in Table I.

C. Dyna-MLAC

By combining the Dyna framework with MLAC updates, the Dyna-MLAC algorithm is proposed. Dyna-MLAC is obtained from Algorithm 2, using the MLAC update described in Section III-A. Note that the MLAC update procedure relies on the process model already provided by the Dyna framework.

IV. Experiments

In this section, the performance of the four considered algorithms (SAC, MLAC, Dyna-SAC and Dyna-MLAC) will be evaluated using the pendulum swing-up environment[2]. Our goals are to 1) evaluate all four algorithms convergence to the same, optimal solution; 2) illustrate the trade-off between sampling and computation complexity, showing that Dyna-MLAC enables the trading between sampling and computation costs and 3) show that using the Dyna-MLAC updates we can extract more information from the process model achieving faster convergence.

To compare the algorithms, we plot the rise time against the computation time. The rise time is the number of episodes the system takes to converge. A trial is considered to have converged when the agent performed three episodes in a row with an accumulated reward sum greater than a fixed performance threshold. The computation time is given by the number of updates per control step. Note that SAC and MLAC have a fixed number of updates per control step equal to one. Therefore, our plots show straight lines when analyzing these two algorithms.

A. Pendulum Swing-up

The pendulum swing-up task, one of the benchmark problems described in [23], consists of a DC motor attached to a round plate. A weight of mass m is fixed at the border of the plate, creating a pendulum, as shown in Figure 1.

The goal is to swing up and balance the weight, but the motor does not have enough torque to do this immediately from the starting position; it will first have to rotate in the opposite direction to gain momentum. At every point in time t, the controller can change the voltage $a = u$ supplied to the motor. The system state $\boldsymbol{s} = [\theta, \dot{\theta}]$ is given by the angle and the angular velocity, where the initial state is $\boldsymbol{s}_0 = [\pi, 0]$. An episode takes 3 seconds, with a sampling time of 0.03 seconds, which leads to 100 control steps per episode. At every control step, the instantaneous reward is given by $r = -5\theta^2 - 0.1\dot{\theta}^2 - u$.

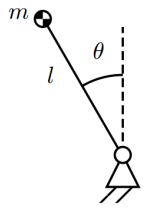


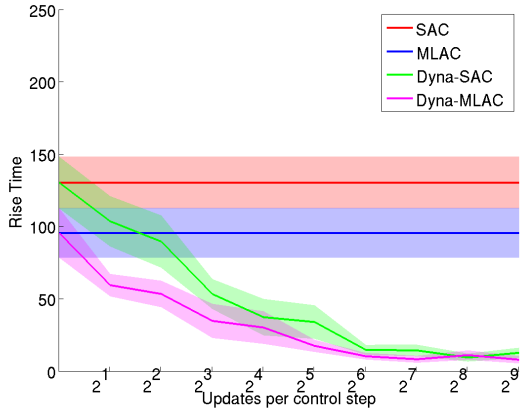
Figure 1: The pendulum swing-up environment.

B. Computational Budget for Model-Based Updates is Beneficial when Sampling is Costly

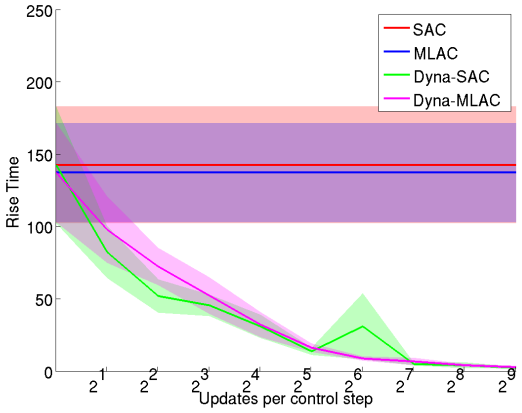
In this section, we consider the pendulum swing-up problem with maximum allowed voltage $u \in [-1.5, 1.5]$. In this setup, the weight must change direction twice before being able to balance at the top. Table I shows all the parameters used in the experiments. The SAC/Dyna-SAC and MLAC/Dyna-MLAC parameters are the same for a fair comparison. The other two parameters globally set across all four algorithms are: the eligibility decay rate $\lambda = 0.65$ and the reward discount rate $\gamma = 0.97$.

Figure 2a shows the rise time as a function of the computation time for all four algorithms. Under MLAC, Dyna-MLAC and Dyna-SAC, a process model is used when updating the

[2]All data used, along with the source code and results for this work are available at http://git.io/vmVLv



(a) Actions $A \in [-1.5, 1.5]$ and performance of -1700.



(b) Actions $A \in [-3, 3]$ and performance of -1000.

Figure 2: Rise time of all four algorithms on both versions of pendulum swing-up environment. The light area is the 95% confidence interval and the bold line is the mean. 32 runs were used to calculate the results.

approximator(s). The additional information extracted from the samples, stored in the process model, yields smaller rise times against SAC.

Under Dyna-SAC, a process model is used to simulate real-world interactions, which generate "virtual samples". These "virtual samples" are used to update the TD error, which in turn is used in the update rules. In that sense, Dyna-SAC implicitly uses the process model. The greater the number of "virtual samples" collected between two control steps, the smaller the rise time (green curve in Figure 2a). Under MLAC, in contrast, a process model is used explicitly by the actor update rule (recall the dependence of (8) on the process model s' through $\partial s'/\partial a$). Figure 2a allows us to compare the advantages and disadvantages of MLAC and Dyna-SAC in the way they make use of the process model. The rise time of MLAC is smaller than that of Dyna-SAC if the number of updates allowed per control step in Dyna-SAC is small, but greater otherwise (in Figure 2a, the green and blue curves cross roughly at 4 updates per control step).

Under Dyna-MLAC, the process model has a twofold role in the update rule, as it is used 1) to generate "virtual samples" that will impact the value function and 2) to determine $\partial s'/\partial a$. By extracting more information from the obtained samples, Dyna-MLAC shows the best performance among the studied algorithms. When the number of updates per control step is one, the rise time of Dyna-MLAC and of MLAC are equal. As the number of updates per control step increases, the rise time decreases, remaining always less than or equal to the rise time of Dyna-SAC.

Note that Dyna-SAC and Dyna-MLAC enable the trading between computational and sampling budgets. The greater the computational cost (updates per control step), the smaller the number of samples required to achieve convergence. However, when the number of updates per control step is greater than 2^6, the system is *saturated*, i.e., between every pair of control steps the policy converges to maximize performance on the current process model. After reaching the saturation regime, when no more information can be extracted either from the obtained real-world samples or from the learned transition model, the performance of Dyna-SAC and Dyna-MLAC is equal and additional computational budget will not reduce the rise time. The fidelity of the learned model thus poses a fundamental limit on the complexity trade-off.

	SAC	MLAC	Dyna-SAC	Dyna-MLAC
Actor Learning step	0.03	0.03	0.03	0.03
Actor Memory size	2000	2000	2000	2000
Actor # of Neighbors	10	10	10	10
Critic Learning step	0.2	0.3	0.2	0.3
Critic Memory size	2000	2000	2000	2000
Critic # of Neighbors	20	20	20	20
Process Model Memory size	-	100	100	100
Process Model # of Neighbors	-	10	10	10

Table I: Parameters used in algorithms

C. Model-Based Updates Are Not Always Necessary

Next, we consider a scenario where the motor voltage is controlled with $u \in [-3, 3]$. This allows the pendulum to be balanced with just one change of direction. All the other parameters are shown in Table I. Figure 2b shows the rise time as a function of the computation time for all four algorithms. As in the previous setup, both Dyna algorithms are faster than their non-Dyna counterparts. However, in this experiment Dyna-MLAC converges roughly as fast as Dyna-SAC.

To explain why Dyna-MLAC and Dyna-SAC have similar performance in this experiment, consider the left region of Figure 2a. When the number of updates per control step is equal to one, the gap between MLAC and SAC determines the advantage of Dyna-MLAC over Dyna-SAC , as in this case the performance of SAC (resp., MLAC) and Dyna-SAC (resp., Dyna-MLAC) are equal. In Figure 2b, the gap between SAC and MLAC is negligible. This is in agreement with [11], and explains why the performance of Dyna-MLAC and Dyna-SAC

is similar in this experiment.

D. The Effect of the LLR Memory Size

One of the most expensive steps in the algorithms considered in this paper is the search for the k-nearest neighbors. The computational complexity of this search is directly related to the size of the k-d tree memory. Given a finite amount of memory available to an agent, how should it be allocated to the critic, the actor and the process model?

Figure 3 shows the end performance of Dyna-MLAC as a function of the amount of memory allocated to the actor, critic and process model. The minimum memory unit is a sample. We consider 64 updates per control step and run the experiment described in Section IV-B for 20 episodes. If the memory capacity is smaller than 1000 samples, the end performance increases as additional memory capacity is provided. However, further increasing the memory capacity beyond 1000 samples does not impact end performance. Note that the process model memory capacity significantly impacts system performance if it is smaller than 60 samples. We also observe that the performance of the critic sharply increases when its memory capacity surpasses 125 samples. The performance of the actor, in contrast, smoothly increases as a function of its memory capacity.

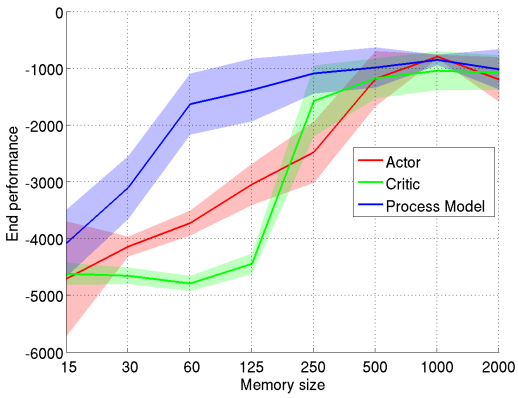


Figure 3: LLR memory size effect on Dyna-MLAC algorithm using 2^6 updates per control step.

V. Conclusion and Future Work

Sampling and computational complexity are in the essence of any reinforcement learning algorithm. Although very fundamental, the trade-offs involved are not well understood. In this paper, we provided new insights and algorithms that enable the trading between sampling and computational complexity under the actor-critic paradigm. Taking Dyna-SAC and MLAC as reference algorithms which bode well with sampling-constrained and computationally-constrained environments, respectively, we showed that the proposed Dyna-MLAC combines the best of the two solutions. In particular, given a certain sampling budget and feasible target rise time, the computational complexity of Dyna-MLAC can be tuned to reach the desired goals. Given the promising results presented in this paper, future work consists of further investigating under which conditions Dyna-MLAC outperforms its inspiring algorithms. The choice of LLR as the function approximator can also be a source of further investigation. Even though [22] measures the performance using radial basis functions with inferior results, other function approximators may fare better.

Acknowledgments: this work was partially supported by CAPES-Brasil (CsF program) and CNPq.

References

[1] V. Chandrasekaran and M. I. Jordan, "Computational and statistical tradeoffs via convex relaxation," *PNAS*, vol. 110, no. 13, pp. E1181–E1190, 2013.

[2] S. M. Kakade, "On the sample complexity of reinforcement learning," Ph.D. dissertation, University College London, 3 2003.

[3] M. G. Azar, R. Munos, and H. Kappen, "On the sample complexity of reinforcement learning with a generative model," Nov 2011.

[4] J. Pazis and R. Parr, "PAC optimal exploration in continuous space markov decision processes." in *Proc. AAAI*, 2013.

[5] M. P. Deisenroth and C. E. Rasmussen, "PILCO: A model-based and data-efficient approach to policy search," in *Proc. ICML*, 2011.

[6] C. J. C. H. Watkins and P. Dayan, "Technical note: q-learning," *Mach. Learn.*, vol. 8, no. 3-4, pp. 279–292, May 1992.

[7] J. Peters, S. Vijayakumar, and S. Schaal, "Natural actor-critic," in *Machine Learning: ECML 2005*. Springer, 2005, pp. 280–291.

[8] R. S. Sutton, "Integrated architectures for learning, planning, and reacting based on approximating dynamic programming," in *Proc. ICML*, 1990, pp. 216–224.

[9] W. Caarls and E. Schuitema, "Parallel online temporal difference learning for motor control," *IEEE Trans. Neural Netw. Learn. Syst.*, in press.

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

[11] I. Grondman, M. Vaandrager, L. Busoniu, R. Babuska, and E. Schuitema, "Efficient model learning methods for actor-critic control." *IEEE Trans. Syst., Man, Cybern., B*, vol. 42, no. 3, pp. 591–602, 2012.

[12] A. Galindo-Serrano and L. Giupponi, "Distributed q-learning for aggregated interference control in cognitive radio networks." *IEEE T. Vehicular Technology*, vol. 59, no. 4, pp. 1823–1834, 2010.

[13] C. G. Atkeson, A. Moore, and S. Schaal, "Locally weighted learning," *Artificial Intelligence Review*, pp. 11–73, 1997.

[14] B. S. C. da Costa, "Dyna-mlac: Trading between computational and sample complexities in actor-critic reinforcement learning," Master's thesis, Federal University of Rio de Janeiro, July 2015.

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

[16] R. Bellman, *Dynamic Programming*, 1st ed. Princeton, NJ, USA: Princeton University Press, 1957.

[17] M. L. Puterman, *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[18] 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, pp. 834–846, 1983.

[19] X. Xu, L. Zuo, and Z. Huang, "Reinforcement learning algorithms with function approximation: Recent advances and applications," *Information Sciences*, vol. 261, no. 0, pp. 1 – 31, 2014.

[20] M. Vaandrager, R. Babuska, L. Busoniu, and G. Lopes, "Imitation learning with non-parametric regression," in *Proc. AQTR*, May 2012, pp. 91–96.

[21] W. Cleveland and E. Grosse, "Computational methods for local regression," *Statistics and Computing*, vol. 1, no. 1, pp. 47–62, 1991.

[22] I. Grondman, L. Busoniu, and R. Babuska, "Model learning actor-critic algorithms: Performance evaluation in a motion control task," in *Proc. CDC*, Dec 2012, pp. 5272–5277.

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