UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
INSTITUTO TÉRCIO PACITTI DE APLICAÇÕES E PESQUISAS COMPUTACIONAIS
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

BRUNO SOUSA CAMPOS DA COSTA

DYNA-MLAC: TRADING BETWEEN COMPUTATIONAL AND SAMPLE COMPLEXITIES IN ACTOR-CRITIC REINFORCEMENT LEARNING

Rio de Janeiro
2015

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
INSTITUTO TÉRCIO PACITTI DE APLICAÇÕES E PESQUISAS COMPUTACIONAIS
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

BRUNO SOUSA CAMPOS DA COSTA

DYNA-MLAC: TRADING BETWEEN COMPUTATIONAL AND SAMPLE COMPLEXITIES IN ACTOR-CRITIC REINFORCEMENT LEARNING

Dissertação de Mestrado submetida ao Corpo Docente do Departamento de Ciência da Computação do Instituto de Matemática, e Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários para obtenção do título de Mestre em Informática.

Orientador: Daniel Sadoc Menasché

Co-orientador: Wouter Caarls

Rio de Janeiro
2015

CIP - Catalogação na Publicação

```
        Costa, Bruno Sousa Campos da
C837d      Dyna-MLAC: Trading Between Computational and
        Sample Com- plexities in Actor-Critic
        Reinforcement Learning / Bruno Sousa Campos da
        Costa. -- Rio de Janeiro, 2015.
           68 f.

           Orientador: Daniel Sadoc Menasché.
           Coorientador: Wouter Caarls..
           Dissertação (mestrado) - Universidade Federal
        do Rio de Janeiro, Instituto Tércio Pacitti de
        Aplicações e Pesquisas Computacionais, Programa de
        Pós-Graduação em informática, 2015.

           1. Inteligência artificial. 2. Aprendizado por
        reforço. 3. Ator-crítico. I. Menasché, Daniel
        Sadoc, orient. II. Caarls., Wouter, coorient.
        III. Título.
```

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a).

BRUNO SOUSA CAMPOS DA COSTA

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

Dissertação de Mestrado submetida ao Corpo Docente do Departamento de Ciência da Computação do Instituto de Matemática, e Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários para obtenção do título de Mestre em Informática.

Aprovado em: Rio de Janeiro, _____ de ______________ de _________.

Prof. Daniel Sadoc Menasché, Ph.D. (Orientador)

Prof. Wouter Caarls, Ph.D. (Co-orientador)

Prof. Adriano Joaquim de Oliveira Cruz, Ph.D.

Prof. Josefino Cabral Melo Lima, Docteur

Prof. Felipe Maia Galvão França, Ph.D.

Prof. Bruno Castro da Silva, Ph.D.

A todos que me apoiaram nessa jornada.

AGRADECIMENTOS

Foi uma jornada incrível. Sem dúvida, mesmo com todas as dificuldades, foi uma caminhada de muito conhecimento e aprimoramento pessoal.

Gostaria de agradecer, primeiramente, a Deus e a minha família, pelo apoio incondicional durante todo o trabalho. Em especial a minha namorada, Emily Rocha Fonseca, por tantas vezes me animar, me cobrar e por tantas vezes me ouvir falar sobre robôs e aprendizado.

Agradeço aos meus orientadores. Ao prof. Daniel Sadoc por ter aceitado ser meu orientador. Muito obrigado por todo o seu esforço e dedicação. Ao prof. Wouter Caarls, pela ajuda inestimável, sem a qual, esse trabalho não teria acontecido.

Meus agradecimentos também aos meus amigos que partilharam dessas dificuldades comigo, me incentivando e me dando forças para concluir esse trabalho. Muito obrigado a todos.

RESUMO

COSTA, Bruno Sousa Campos da. **Dyna-MLAC:** Trading Between Computational and Sample Complexities in Actor-Critic Reinforcement Learning. 2015. 68 f. Dissertação (Mestrado em Informática) - PPGI, Instituto de Matemática, Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2015.

Complexidade de amostragem e complexidade computacional são dois elementos chaves que determinam a performance dos algoritmos de aprendizado por reforço. Essencialmente, todo agente inteligente treinado utilizando algoritmos de aprendizado por reforço deve amostrar o ambiente e realizar alguma computação sobre as amostras para determinar a melhor ação. Apesar de ser um problema fundamental, o *trade-off* entre ambos ainda não é bem compreendido. Nesse trabalho, exploramos esse *trade-off* sob a perspectiva do esquema ator-crítico. Primeiro, apresentamos um novo algoritmo de aprendizado por reforço, o Dyna-MLAC, utilizando um modelo de transição para realizar o aprendizado (MLAC) e o *framework* Dyna. Então, indicamos numericamente que o tempo de convergência do Dyna-MLAC é menor que o das soluções já existente e que o Dyna-MLAC permite uma troca eficiente entre amostras e tempo computacional. Finalmente, investigamos o efeito de um conjunto de parâmetros na performance dos algoritmos estudados e como a alocação de memória também afeta a performance.

Palavras-chave: Inteligência Artificial, Aprendizado por reforço, Ator-crítico.

ABSTRACT

COSTA, Bruno Sousa Campos da. **Dyna-MLAC:** Trading Between Computational and Sample Complexities in Actor-Critic Reinforcement Learning. 2015. 68 f. Dissertação (Mestrado em Informática) - PPGI, Instituto de Matemática, Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2015.

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 work, 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. Finally, we also investigate the effect of a set of parameters and how memory allocation affects performance. We find that the performance is most sensitive to the amount of memory allocated to the critic.

Keywords: Artificial Intelligence, Reinforcement Learning, Actor-Critic.

LIST OF FIGURES

2.1 The policy iteration. Figures obtained from [17] 21

2.2 The cliff walking problem comparing Sarsa and Q-Learning. Figure obtained from [17] 26

2.3 Two main methods to implement Eligibility Traces: accumulating and replacing. Figure obtained from [17] 27

2.4 The gridworld updates when using Sarsa(λ). Figure obtained from [17] 28

4.1 A query example on a 2-tilings ($\tau = 2$) Tile Coding schema. . . . 36

4.2 The *sine* function approximated using LLR with 50 samples. The blue and the red lines approximately overlap each other. 39

4.3 LLR and LWR physical interpretation. The left figure represents LLR and it is constant weight while the right one is LWR using a distance weight. Figure obtained from [1] 40

4.4 Final critic and performance using the randomly strategy on LLR 41

4.5 Final critic and performance using the prediction strategy on LLR 42

4.6 Final critic and performance using the uniform strategy on LLR . 43

4.7 Comparison between the three different LLR memory strategies . 44

4.8 Comparison between Locally Linear Regression and Tile Coding on Standard Actor Critic 45

6.1 The pendulum swing-up environment. 53

6.2 Actions $A \in [-1.5, 1.5]$ and performance of -1700. 55

6.3 Dyna-MLAC performance surface. The green plane correspond to -1800 performance. 57

6.4 Contour plots of the performance curve for Dyna-MLAC (surface in Figure 6.3). It is possible to see the trade-off between episodes (sampling) and updates per control step (computation) given a required performance. 58

6.5 Actions $A \in [-3, 3]$ and performance of -1000. 59

6.6 Dyna-MLAC performance surface. The green plane correspond to -1000 performance. 60

6.7 Contour plots of the performance curve for Dyna-MLAC (surface in Figure 6.6). It is possible to see the trade-off between episodes (sampling) and updates per control step (computation) given a required performance. 61

6.8 Final critics $V(s)$ in the standard environment. Every point is a sample in the LLR memory. Both Dyna-SAC and Dyna-MLAC with 64 updates per control step 62
6.9 Log plot of different exploration rates for Dyna and Dyna-Mlac against the end performance 63
6.10 LLR memory size effect on Dyna-MLAC algorithm using 2^6 updates per control step. 64

LIST OF TABLES

6.1 Parameters used in algorithms 57

CONTENTS

1 INTRODUCTION 12

2 BACKGROUND THEORY 15

2.1 AGENTS AND ENVIRONMENTS 15

2.2 MARKOV DECISION PROCESS 16

2.3 REINFORCEMENT LEARNING 17

2.4 THE MDP IN THE VALUE FUNCTION 18

2.5 SOLVING THE PROBLEM 19

2.5.1 Model Based 20

2.5.2 Model Free methods 21

2.6 PROBLEMS USING REINFORCEMENT LEARNING 28

2.6.1 Exploration 28

2.6.2 Continuous space-state 30

2.6.3 Convergence 30

3 ACTOR CRITIC 31

3.1 STANDARD ACTOR-CRITIC ALGORITHM 32

4 FUNCTION APPROXIMATORS 34

4.1 TILE CODING 35

4.1.1 Building step 35

4.1.2 Querying step 35

4.1.3 Learning step 36

4.2 LOCALLY LINEAR REGRESSION 36

4.2.1 Building step 37

4.2.2 Querying step 37

4.2.3 Learning step 38

4.3 MEMORY MANAGEMENT 40

4.3.1 Randomly 40

4.3.2 Prediction 41

4.3.3 uniformly distributed 42

4.4 COMPARISON BETWEEN TILE CODING AND LOCALLY LINEAR REGRESSION 43

5 MODEL-BASED METHODS 46
5.1 MODEL LEARNING 46
5.1.1 What is the transition model 46
5.1.2 Why to learn a model 46
5.1.3 How to learn a model 47
5.2 MLAC 47
5.3 DYNA 49
5.3.1 Dyna-MLAC 51

6 EXPERIMENTS 52
6.1 THE PENDULUM SWING-UP 53
6.1.1 Computational Budget for Model-Based Updates is Beneficial when Sampling is Costly 54
6.1.2 Model-Based Updates Are Not Always Necessary 57
6.2 CONVERGENCE 59
6.3 THE EFFECT OF THE EXPLORATION RATE 60
6.4 THE EFFECT OF THE LLR MEMORY SIZE 61

7 CONCLUSIONS AND FUTURE WORK 65

REFERENCES 66

1 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 [5]. 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.

Algorithms which require a low number of samples are 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 [7], and have a very high computation cost. Conversely, algorithms which are computationally cheap, such as classical Q-learning [19], 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 [13]. The Dyna framework [16], 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.

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 [15] and 2) Dyna-MLAC, using the model-based update rules of the MLAC (Model-Learning Actor-Critic) algorithm [10].

Learning a process model from the samples can be done using any supervised learning technique, like neural networks [9]. In this work, we use a memory based algorithm, Locally Linear Regression (LLR) [1]. Given a fixed amount of memory, we study the impact of the split of this memory between the actor, the critic and the process model.

This trade-off is especially important considering time-sensitive problems, for example, a robotic arm used in a surgery. Not taking a action can be potentially worse than taking a slightly suboptimal decision.

In summary, the key question addressed in this dissertation, 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) *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;

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 fewer iterations to converge;

3) *algorithm parametrization*: we analyze the impact of the learning step, the exploration rate and the memory size used in LLR. We show how these parameters impacts differ between the actor, the critic and the process model.

The remainder of this work is organized as follows: Chapter 2, gives the background on Reinforcement Learning. In Chapter 3 the Standard Actor-Critic (SAC) algorithm is presented, along with the actor-critic theory. Chapter 4 presents the function approximator used in this paper, the Locally Linear Regression (LLR), comparing it against Tile Coding. Chapter 5 gives more information regarding the model based algorithms that will be used throughout this work. Chapter 6 details all experiments along with their results and further discussions. Finally, conclusions and future work are presented in Chapter 7.

2 BACKGROUND THEORY

In this section, we will discuss the inner working of Reinforcement Learning (RL). First, in Section 2.1, a brief description of agents and environments is given. From Section 2.2 onwards, solutions methods are considered. First, more theoretical and less practical ways are described in Section 2.2. Then, some practical methods are described in Section 2.5. At the end of this section, we show problems that arise from these solutions, but we delayed the discussion of the methods used in this work to Chapters 3 and 5.

2.1 Agents and Environments

Russel and Norvig [14] describe an agent as

> An agent is just something that acts (agent comes from the Latin agere, to do). Of course, all computer programs do something, but computer agents are expected to do more: operate autonomously, perceive their environment, persist over a prolonged time period, adapt to ration agent change, and create and pursue goals. A rational agent is one that acts so as to achieve the best outcome or, when there is uncertainty, the best expected outcome.

Our goal is to develop a rational agent, capable of learn a given task by trial and error.

2.2 Markov Decision Process

In this work, we are only going to consider problems with the Markov property. This means that if the current state is known, then the transition to the next state is independent of all previous transitions. The next one, however, can be stochastic. So, by this, the current state alone has enough information for the agent to decide the next best action. In this setup, the mathematical foundation used is known as Markov Decision Process.

Formally, the problem of finding a policy π, i.e. a function that tells the agent what to do in all possible situations, can be modeled using a Markov Decision Process (MDP). A MDP is a tuple (S, A, T, γ, D, R), where S denotes a set of states; A is a set of actions; $T = P_{sa}$ is a set of state transition probabilities (here, P_{sa} is the state transition distribution upon taking action a in state s); γ is a discount factor; D is the initial-state distribution, from which the start state s_0 is drawn; and R is the reward function $R(s, a, s')$, where s' is the next state.

The policy $\pi : S \mapsto A$ decides what action to take in every possible state. The goal of the agent is to maximize the total expected reward in possible infinite-horizon setup. So, the expected reward, after t time steps is:

$$R_t = \mathbb{E}\{r_{t+1} + \gamma_{t+2} + \gamma^2 r_{t+3} + \dots\} = \mathbb{E}\left\{\sum_{t=0}^{\infty} \gamma^t r_{t+1}\right\} \tag{2.1}$$

where $\gamma \in (0, 1]$ is a discount factor, dictating the future rewards importance in the problem. As $\gamma \to 1$, rewards further in the future have the same importance as the more immediate ones. On the other hand, considering $\gamma \to 0$, the farthest terms in Equation (2.1) can be ignored and does not count to the total expected reward.

2.3 Reinforcement Learning

Before we continue to solve the problem, we have to present more formally the value function. Let's say we know the transition matrix T and the reward function R of the MDP. If the agent has a stationary policy π, one can calculate the expected reward for every state s by simply following the given policy and estimate the outcome from Equation (2.1). Formally, we can define the value function V as:

$$V^{\pi}(s) = \overset{\pi}{\mathbb{E}}\{R_t | s_t = s\} \tag{2.2}$$

The value function is one of the most important functions in this work. But, another very useful one is the action-value function, known as Q. Because the agent has to learn the best action in every state, for a class of algorithms it is more useful to define a function Q given by:

$$Q^{\pi}(s,a) = \overset{\pi}{\mathbb{E}}\{R_t | s_t = s, a_t = a\} \tag{2.3}$$

which is also dependent on a given policy π.

The last important notion before we move to solving the problem is the optimal value function V^{π^*}:

$$V^{\pi^*}(s) = \max_{\pi} V^{\pi}(s) \tag{2.4}$$

The same holds for the action-value function Q:

$$Q^{\pi^*}(s,a) = \max_{\pi} Q^{\pi}(s,a) \tag{2.5}$$

2.4 The MDP in the value function

In Section 2.2, the Markov property was introduced along with the MDP concept. In Section 2.3, the value function and the action-value function were introduced. In this section, the connection between them is presented in a way that finding the optimal value for the value function means finding an optimal policy in the underlying MDP.

Let's recall our definition of the value function V:

$$V^{\pi}(s) = \overset{\pi}{\mathbb{E}}\{R_t | s_t = s\} \tag{2.6}$$

The Bellman equation [2] can be obtained from V as:

$$\begin{aligned}
V^{\pi}(s) &= \overset{\pi}{\mathbb{E}}\{R_t | s_t = s\} \\
&= \overset{\pi}{\mathbb{E}}\left\{\sum_{t=0}^{\infty} \gamma^t r_{t+1} | s_t = s\right\} \\
&= \overset{\pi}{\mathbb{E}}\left\{r_{t+1} + \gamma \sum_{t=0}^{\infty} \gamma^t r_{t+2} | s_t = s\right\} \\
&= \overset{\pi}{\mathbb{E}}\{r_{t+1} + \gamma V^{\pi}(s') | s_t = s, s_{t+1} = s'\} && (2.7) \\
&= \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma V^{\pi}(s')\right] && (2.8)
\end{aligned}$$

where $T(s, a, s')$ and $R(s, a, s')$ are variables from the underlying MDP.

Further on, we can combine the optimal value function Equation (2.4) with (2.7) to get the optimal value function in terms of ifself:

$$\begin{aligned} V^*(s) &= \max_\pi V^\pi(s) \\ &= \max_\pi \overset{\pi}{\mathbb{E}} \{r_{t+1} + \gamma V^\pi(s') | s_t = s, s_{t+1} = s'\} \end{aligned} \tag{2.9}$$

So, if the optimal value function is known, the optimal policy is the argument that maximizes Equation (2.9):

$$\begin{aligned} \pi^* &= arg \max_\pi V^\pi(s) \\ &= arg \max_\pi \overset{\pi}{\mathbb{E}} \{r_{t+1} + \gamma V^\pi(s') | s_t = s, s_{t+1} = s'\} \end{aligned} \tag{2.10}$$

The policy selects the action for each the according function is maximum, therefore, this policy is called a greedy policy.

2.5 Solving the problem

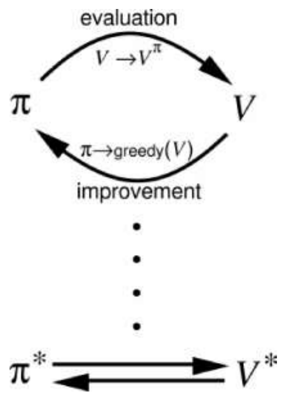
Now we are ready to look into ways of solving the problem. First, we will see the model based algorithms(Section 2.5.1). These are the ones that need all the information from the MDP and are very computational expensive, but we will see that the main ideas behind them are present in all of the others. Then we will take a look at model free algorithms (Section 2.5.2). They solve some of the main problems we found in model based, but lack the speed we found in before. Finally, we will see the model learning (Section 5.1) algorithms, that learn a model as it goes.

2.5.1 Model Based

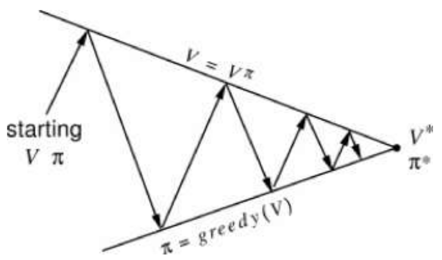
A model based learning controller knows all the variables that fully describe the problem, i.e., the state is fully observable and the transition matrix T and the reward function R are known. In this situation, and by looking at Equation (2.8), it is easy to notice that the only unknown variable is V^*, which leads to a system of equations that can be solved. The assumptions of knowing everything unfortunately makes its usability very limited.

2.5.1.1 Policy Iteration

The process known as policy iteration is composed of two steps: first the policy is evaluated (policy evaluation) and then it is improved (policy improvement). Starting with a random policy π_0, the algorithm evaluates the corresponding value V^{π_0} using Equation (2.7). This is the policy evaluation step. Once we have V^{π_0}, we can obtain the next best policy π_1 in a greedy fashion, according to Equation (2.10). This is the policy improvement step. In Figures 2.1a and 2.1b we can see how this process iteratively gets to the optimal solution.



(a) Policy iteration scheme.



(b) Policy iteration in a 2-d space.

Figure 2.1: The policy iteration. Figures obtained from [17]

2.5.2 Model Free methods

In the last section, we solved a MDP problem, i.e., obtained the optimal policy. For practical applications, however, an requirement of full knowledge of the environment in the policy iteration restricts its applicability. In general, one does not have such knowledge, making the use of policy iteration very rare. Fortunately, there are other ways to solve this problem without such knowledge. In this section, we will look into two methods: Monte Carlo (Section 2.5.2.1) and Temporal Difference (Section 2.5.2.2). Both are considered to be on-line methods, opposed to policy iteration that is a off-line method.

2.5.2.1 Monte Carlo

Monte Carlo algorithms estimates the value-function based on trials, simulated or obtained from a real environment. One requirement here is that the problem must be *episodic*, i.e., it must have an end. This method works by averaging the sample returns, the same way Monte Carlo simulations are used to obtain relevant statistics in others fields [8].

The episodic requirement exists because Monte Carlo methods only learn based on a complete return of the task, while in Temporal Difference (Section 2.5.2.2), the learning is done using partial returns.

This method works similar to Policy Iteration (Section 2.5.1.1). The main difference here is that the expected value, used to compute the value function Equation (2.7) is estimated after every simulation step. As in Policy Iteration, a two step algorithm is used here: first, we evaluate the policy and them we update it. The problem here is, as we only follow one possible path to perform the evaluation, the policy improvement can't be all greedy, otherwise, we wouldn't explore the space state. To solve this, we will use an ϵ-greedy policy instead. This exploration trade-off is further analyzed in Section 2.6.1.

The main Monte Carlo algorithm to update the value function is described in Algorithm 1.

2.5.2.2 Temporal Difference

Two algorithms were presented in the past sections: a) policy iteration, which requires full knowledge of the environment but works without the need of sampling from the environment, and b) Monte Carlo simulations, which don't require any knowledge of the environment, but delays the learning until the end of the episode. Temporal Difference method has the same requirements as Monte Carlo, but it estimate the value-function in a step-by-step fashion, while Monte Carlo does in an episode-by-episode way. This means that, after each step, we update the value function following Equation (2.11):

Algorithm 1 ϵ-greedy Monte Carlo algorithm

```
1: procedure MONTECARLO
2:     $Q(s,a) \leftarrow$ arbitrary
3:     $Returns(s,a) \leftarrow$ empty list
4:     $\pi \leftarrow$ an arbitrary $\epsilon$-soft policy
5: Repeat forever:
6:     Generate an episode using $\pi$
7:     for all pair s,a appearing in the episode: do
8:         $G \leftarrow$ return following the first occurrence of s, a
9:         Append G to Returns(s, a)
10:        $Q(s,a) \leftarrow$ average$(Returns(s,a))$
11:    for all s in the episode: do
12:        $a^* \leftarrow \arg\max_a Q(s,a)$
13:        for all $a \in A(s)$ do
14:            $\pi(a|s) = \begin{cases} 1-\epsilon+\frac{\epsilon}{|A(s)|} & \text{if } a = a^* \\ \frac{\epsilon}{|A(s)|} & \text{if } a \neq a^* \end{cases}$
```

$$
\begin{aligned}
V(s_t) &\leftarrow (1-\alpha_t)V(s_t) + \alpha_t(r_{t+1} + \gamma V(s_{t+1})) \\
V(s_t) &\leftarrow V(s_t) + \alpha_t \delta_t \qquad (2.11)
\end{aligned}
$$

where α is the learning step and

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \qquad (2.12)$$

is the so-called Temporal Difference (TD) error.

The TD error interpretation is the difference between the predicted value of a state s_t and the real reward r_{t+1} plus the predicted value following the next state $V(s_{t+1})$. If the agent has full knowledge of the value function, then the TD error is zero. There are two classical algorithms that uses TD error to update the value function: SARSA (Section 2.5.2.3) and Q-Learning (Section 2.5.2.4).

2.5.2.3 SARSA

State-action-reward-state-action (SARSA) is an algorithm used to estimated the action-value function. It works, basically, using the same update rule as in Equation (2.11):

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right] \tag{2.13}$$

This update is done after every iteration, as we can see in Algorithm 2. The action used to update is the same that was chosen, therefore, this method is considered to be *on-policy*, as it learns the same policy that it follows.

Algorithm 2 Sarsa algorithm

```
1: procedure SARSA
2:     $Q(s, a) \leftarrow$ arbitrary
3: Repeat forever:
4:     Initialize $s_t$
5:     Choose $a_t$ from $s_t$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
6:     for all step of the episode do
7:         Take action $a_t$, observe $r_{t+1}$, $s_{t+1}$
8:         Choose $a_{t+1}$ from $s_{t+1}$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
9:         $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$
10:        $s_t \leftarrow s_{t+1}; a_t = a_{t+1}$
```

2.5.2.4 Q-Learning

Q-Learning is another TD method to learn the action-value function $Q(s, a)$. Differently from Sarsa, the approximated action-value is not necessarily the same followed by a greedy policy. Q-Learning uses the following update rule:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right] \qquad (2.14)$$

This update is considered to be *off-policy*, as opposed to Sarsa that is an *on-policy* method.

The main difference between both can be seen in the following example obtained from [17]. The Cliff Walking problem is a common example of a grid world problem. It has a starting state (S) and a goal state (G), as in Figure 2.2. Every transition has a penalty of -1, except in the region marked as “The Cliff”, which has a penalty of -100 and take the agent back to the start position. The agent can move up, down, left and right.

The lower part of the figure shows the performance that each of the algorithms achieved. Both Sarsa and Q-Learning agents use an ϵ-greedy action selection, with $\epsilon = 0.1$, but this randomness has different consequences in each algorithm. While Q-Learning has learned the optimal path, because of the ϵ-greedy action selection, sometimes, the agent falls the cliff, and so it has a lower on-line performance than Sarsa, that learned the longer, but safer path. Considering the policy without the exploration, Q-Learning would have a better performance.

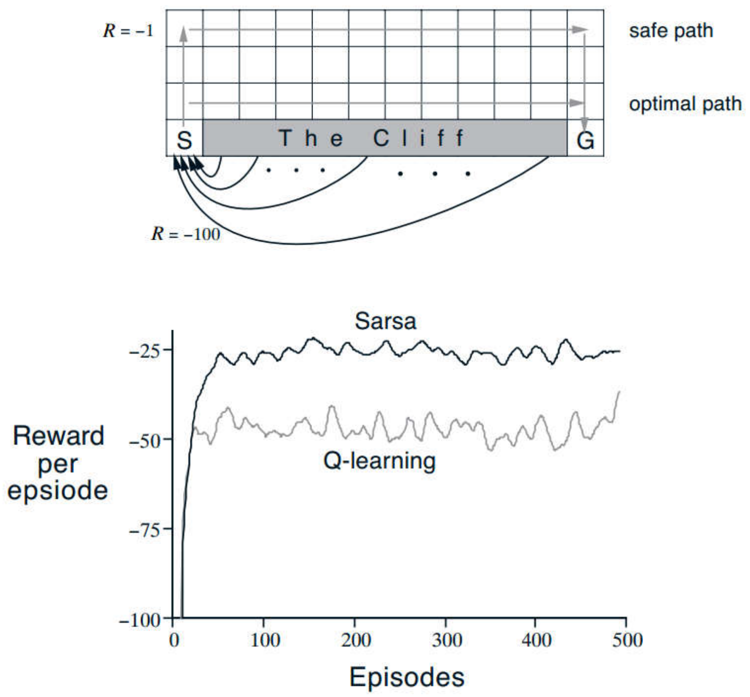


Figure 2.2: The cliff walking problem comparing Sarsa and Q-Learning. Figure obtained from [17]

2.5.2.5 Eligibility Traces

Eligibility traces are used to indicate the path the agent has followed, i.e., all the states that was visited so far, allowing for a refined way to credit past experiences.

Every state is assigned a variable representing the eligibility trace $e_t(s)$. Let $\lambda \in [0, 1)$ be the eligibility decay rate. At every time step, it is decayed of a factor $\lambda\gamma$, where λ is the eligibility decay rate and γ is the discount factor from the underlying MDP (Section 2.2). The decay is important to make the past states less relevant to the update. There are two main ways to update the traces: accumulating or replacing the traces. In Figure 2.3, we can see the main difference between both. On accumulating, the trace value is incremented by 1 after every visit, while in replacing it's capped at 1.

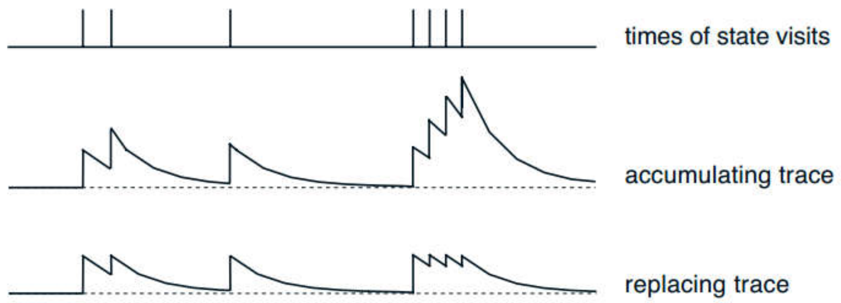


Figure 2.3: Two main methods to implement Eligibility Traces: accumulating and replacing. Figure obtained from [17]

Replacing traces are used in this work and the eligibility trace update becomes:

$$e_t(s) = \begin{cases} \lambda\gamma e_{t-1}(s) & \text{if } s \neq s_t \\ 1 & \text{if } s = s_t \end{cases} \tag{2.15}$$

The update rule 2.11, considering traces, is:

$$V(s_t) = V(s_t) + \alpha_t \delta_t e_t(s_t) \tag{2.16}$$

This technique is so important that when Sarsa or Q-Learning are combined with eligibility traces, they are knows as Sarsa(λ) and Q(λ) respectively.

Let us consider a grid-world problem, similar to one from Section 2.5.2.4, but all states have a reward of 0 except on the goal state that has a reward of 100. On Figure 2.4, we can compare Sarsa with and without the use of eligibility traces.

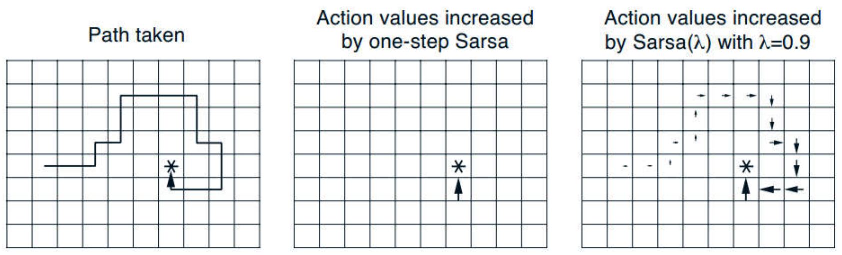

Path taken
Action values increased by one-step Sarsa
Action values increased by Sarsa(λ) with λ=0.9

Figure 2.4: The gridworld updates when using Sarsa(λ). Figure obtained from [17]

2.6 Problems using Reinforcement Learning

So far, we only mentioned a few problems one might deal with while using reinforcement learning to solve a problem. In the section, we'll delve into a few of these problems, why they happen and possible solutions. We also describe how we solve each of them in this work.

2.6.1 Exploration

In many problems showed so far, the model is unknown, therefore, the agent must explore. In fact, most of the convergence proofs rely on constant exploration of all state-space. The exploration-exploitation trade-off has been around for decades, and mathematicians have been working on it [11]. Usually, a method selecting random actions is used to guarantee the required exploration, where ϵ-greedy and Softmax are the two mostly used.

2.6.1.1 ϵ-greedy

One way to tackle the exploration-exploitation problem is ϵ-greedy. An exploration rate ϵ is given and used to select a random action (exploration) instead of the current best action so far (greedy - exploitation). The exploration rate does not have to be constant throughout the entire learning episode, but can be gradually decreased for less exploratory actions as learns proceeds. The action selections is given according to Algorithm 3.

Algorithm 3 ϵ-greedy action selection

```
1: procedure ϵ-GREEDY
2:     $Q(s,a) \leftarrow$ the current action-value function
3:     $\rho \leftarrow$ a randomly generated number between $[0,1]$
4:     $\pi(a|s) = \begin{cases} \text{best action } a^* \text{ from } Q(s,a) & \text{if } \rho > \epsilon \\ \text{action selected following a uniform distribution} & \rho \leq \epsilon \end{cases}$
```

2.6.1.2 Softmax

One problem in ϵ-greedy action selection is using a uniform distribution to select a random action. Another approach is to select a random action a in state s following some probability $p(a|s)$. In Softmax method, $p(a|s)$ is chosen following the Boltzmann distribution:

$$p(a|s) = \frac{e^{\frac{Q(s,a)}{\tau}}}{\sum\limits_{a' \in A} e^{\frac{Q(s,a')}{\tau}}} \tag{2.17}$$

where τ is a temperature parameter. The higher τ value, the greater probability to choose an exploratory action. The τ parameter can also be lowered as the learning process evolves.

2.6.2 Continuous space-state

In this work only continuous state and action spaces are being considered. In these scenarios, the exploration-exploitation trade-off is tackled differently. After an action is selected greedily, a Gaussian noise is added to account for the exploration. This is controlled using an exploration rate e, indicating how often this noise is added to the action. Using an exploration rate of 1 means add the noise every time to the selected greedy action.

2.6.3 Convergence

So far, only discrete tabular version of the algorithms have been showed. In these circumstances, a convergence proof does exist, and the optimal policy is obtained if we follow a few requirements, such as constant exploration and learning parameter $\alpha \to 0$ as steps $\to \infty$.

3 ACTOR CRITIC

Over the years, several types of reinforcement learning algorithms have been presented, but they all can be divided in three categories [12]: actor-only, critic-only and actor-critic. Actor and critic are synonyms for the policy and the value function, respectively.

Actor-only methods work with a parametrized family of policies. The updates are done performing gradient descent updates over the actor parameters through simulation. In this type of algorithms, the spectrum of continuous actions can be generated. A possible drawback of these algorithms is that the gradient estimators may have a larger variance.

Critic-only methods rely exclusively on the value function. All the methods presented in this work so far are considered critic-only. The temporal difference learning usually has a lower variance in the estimate of the total expected reward R_t as in Equation (2.1). The most usual way to determine a policy using these methods is selecting greedy actions as in Equation (2.10). However, this is usually very computation expensive, specially if continuous action are being used. Therefore, these methods usually discretize the continuous action space, where the optimization procedure becomes a matter of enumeration. Obviously, this approach undermines the ability of using continuous actions and thus of finding the true optimum.

Actor-critic methods combines the best of the actor-only and the critic-only algorithms. Having a separate structure working as the policy, allows the agent to rapidly decides the action to take, even on continuous action space. The critic's estimate of the total expected reward allows the gradient updates of the actor to have a low variance.

3.1 Standard Actor-Critic Algorithm

The Standard Actor-Critic (SAC) algorithm described in this section is the foundation for this work. The critic update is based on the TD error update from Section 2.5.2.2, and the actor update

The SAC algorithm is described in Algorithm 4. 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,

$$a_t = \pi(s_t) + \Delta_t \tag{3.1}$$

$\pi(s_t)$ 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 TD error is obtained according to (2.12) (line 11). Then, procedure SAC-Update is called to update the value function and the policy (lines 16-20 of Algorithm 4). The value function (critic) is updated towards minimizing the TD error (line 18), while the policy (actor) is adjusted towards the explored action only if the TD error was positive (and away from it otherwise, line 19). In Algorithm 4, α_a and α_c are the learning step for the actor and for the critic, respectively.

Algorithm 4 SAC algorithm

```
1:  procedure SAC
2:  Repeat forever:
3:      $\forall s \in S : e(s) \leftarrow 0$
4:      $s_0 \leftarrow$ Initial state
5:      Apply random input $a_0$
6:      $t \leftarrow 1$
7:      loop until episode ends:
8:          Measure $s_t$ and $r_t$
9:          Let $\Delta_t$ be a sample from a zero-mean Gaussian
10:         $a_t \leftarrow \pi(s_t) + \Delta_t$                         ▷ Choose an action
11:         $\delta_t = r_t + \gamma V(s_t) - V(s_{t-1})$               ▷ Calculate td-error
12:         Call SAC-Update($\delta_t$, $\Delta_{t-1}$, $\alpha_a$, $\alpha_c$, $s_t$)
13:         Execute $a_t$
14:         $t \leftarrow t + 1$
15: procedure SAC-Update($\delta_t$, $\Delta_{t-1}$, $\alpha_a$, $\alpha_c$, $s_t$)
16:     Update the eligibility trace $e_t(s_t)$
17:     for all $s \in S$ do
18:         $V(s) \leftarrow V(s) + \alpha_c \delta_t e_t(s)$              ▷ Update the critic
19:     $\pi(s) \leftarrow \pi(s) + \alpha_a \delta_t \Delta_{t-1}$                ▷ Update the actor
20:     Clamp $\pi(s)$ to $A$
```

4 FUNCTION APPROXIMATORS

Considering continuous state and action spaces, as in this work, the policy and the value function must be approximated using some kind of function approximators [20].

The goal is to approximate an unknown function from a set of samples. Typically, a sample $\mathbf{m}$ is given by $\mathbf{m} = [\mathbf{x}, \mathbf{y}]$ where $\mathbf{x}$ is the input and $\mathbf{y}$ is the output. Considering the actor approximator, for example, the input is a state s_i and the output is the action a_i to take in the given state s_i.

Let $\hat{f}$ be the approximator. $\hat{f}$ can be found to minimize the error e given a query point $\mathbf{q}$:

$$e = \left\| \mathbf{y_q} - \hat{f}(\mathbf{x_q}) \right\|_2^2 \tag{4.1}$$

where $\left\|\cdot\right\|_2^2$ is the Euclidean norm.

In this work we are going to use the Locally Linear Regression (LLR) method. We compared LLR (Section 4.2) against another commonly used alternative approach for approximating functions, Tile Coding, discussed in Section 4.1. In Section 4.4 it is showed that LLR learns faster.

4.1 Tile Coding

This is a classical approximator in RL, which allows for fast computations. It works using a fixed number of tilings dividing the space into a number of tiles. Each tile has a value θ associated with it. These tiles are usually distributed in a grid-like, uniform way, but this is not a requirement. Actually, any tile shape and distribution is possible. A illustrative image on the process is show on Figure 4.1.

4.1.1 Building step

To build the Tile Coding approximator, it's necessary to allocate memory for all tilings and initialize all the associated θ values to some random value.

4.1.2 Querying step

Each query point $\mathbf{q}$ either belongs to a tile or not, so tiles are in fact, binary features. By having many tilings superposed at slightly different positions, we can query for a point by averaging the values on the selected tiles as in Equation (4.2).

$$\hat{\mathbf{y}}_{\mathbf{q}} = \frac{1}{\tau} \sum_{i=1}^{\tau} \theta_i \tag{4.2}$$

where τ is the total number of tilings and θ_i is the value associated with each tile.

The output $\hat{\mathbf{y}}$ is the sum of the associated value of all the "activated" tiles, i.e., the tiles where the query point q superposes (the shaded squares on Figure 4.1).

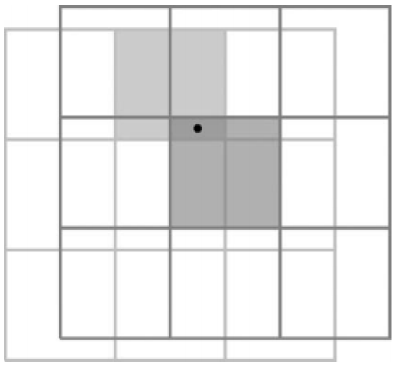


Figure 4.1: A query example on a 2-tilings ($\tau = 2$) Tile Coding schema.

4.1.3 Learning step

The learning step in Tile Coding consists of updating the θ values towards the $\mathbf{y}$ value. Say a query point $\mathbf{q}$ activates some set of tiles Θ. If the output $\hat{\mathbf{y}}$ has an associated error e (Equation (4.1)) the approximator should be adjusted towards minimizing this error, i.e., the θ values of the set Θ should be adjusted, following Equation (4.3).

$$\theta_i = \theta_i + \alpha_{tc}\frac{e}{T} \qquad \forall \theta \in \Theta \tag{4.3}$$

where $\alpha_{tc} \in (0, 1]$ is the learning step for Tile Coding.

4.2 Locally Linear Regression

Locally linear regression is a nonparametric, memory-based function approximator [1]. Although the function 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 region around a query point.

4.2.1 Building step

The building step in LLR is simple, as the sample only has to be added to the memory, although some kind of memory management (see Section 4.3) is necessary since the LLR memory is finite and the transitions can easily exceed the available memory. 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.

4.2.2 Querying step

Given an input query $\mathbf{q} \in \mathbb{R}^n$ and the 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 searched in the LLR memory. For performance purposes, the search can be done with the help of a k-d tree [6].

Let $\mathbf{X_q} \in \mathbb{R}^k \times \mathbb{R}^{n+1}$ and $\mathbf{Y_q} \in \mathbb{R}^k \times \mathbb{R}^\ell$ be the input and output data matrices associated to the k-nearest neighbors of $\mathbf{q}$. Each row of $\mathbf{X_q}$ 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 $\mathbf{Y_q}$ contains output data corresponding to the i-th row of $\mathbf{X_q}$.

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:

$$\mathbf{X_q}\boldsymbol{\beta} = \mathbf{Y_q} \tag{4.4}$$

The solution of (4.4) is obtained using the least square method and yields $\boldsymbol{\beta}$. The estimated output $\hat{\mathbf{y}}$ is given by:

$$\hat{\mathbf{y}} = [\mathbf{q}, 1]\boldsymbol{\beta} \tag{4.5}$$

4.2.3 Learning step

The learning step using LLR is divided in two steps: inserting a sample and updating the existing ones. Consider, for example, the actor updates in line 19 in Algorithm 4. 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 $\pi(s)$. Using the nearest neighbors $\mathcal{K}_s$ and (4.5), the query is resolved. 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 in Algorithm 4), 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. The effect of memory allocation is studied in Section 6.4.

Figure 4.2 is an example of the function *sine* being approximated by a LLR. The blue dots are the retained memory on LLR after the training, the red line is the *sine* function and the blue one is the predicted one based on the memory.

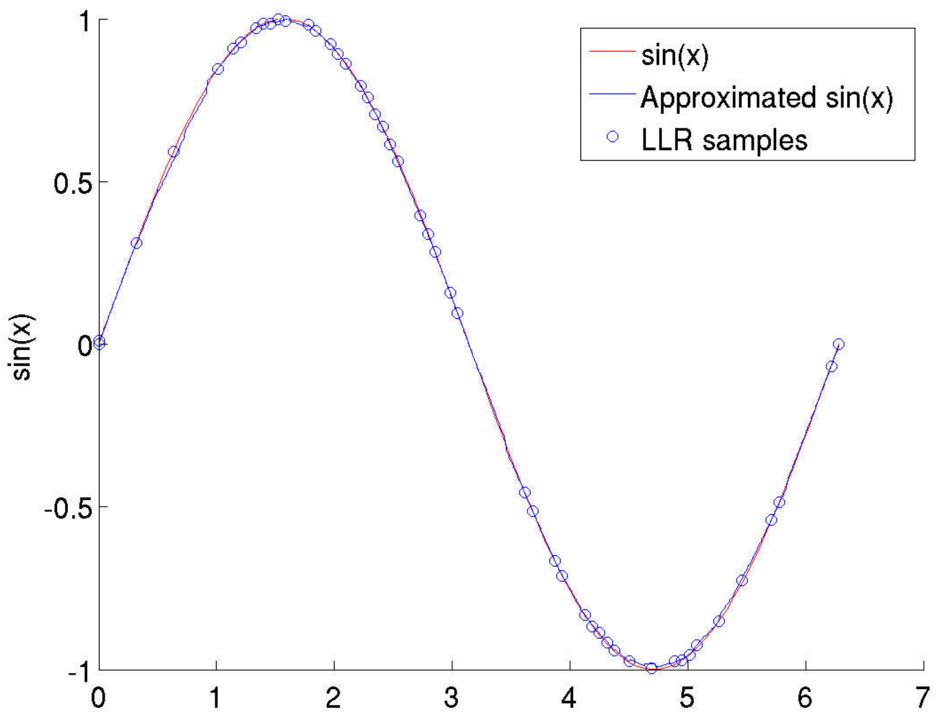


Figure 4.2: The *sine* function approximated using LLR with 50 samples. The blue and the red lines approximately overlap each other.

Analyzing Figure 4.2 we can conclude: a) memory is a limited resource in a computer, and because of that, one must use some kind of memory management (Section 4.3) to decide which observations to keep on which to throw away; b) more points are grouped in the round area, where the linearization is harder.

LLR is actually a specific case of Locally Weighted Regression (LWR), described in [1]. The difference here is that one can give more importance (weight) to the closest neighbors than the farthest ones. The physical interpretation here is given in Figure 4.3. The one on the left is LLR and the one on the right is LWR.

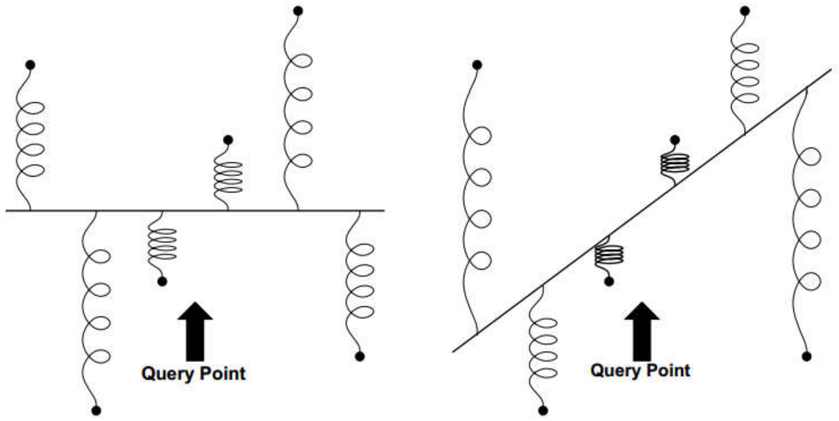


Figure 4.3: LLR and LWR physical interpretation. The left figure represents LLR and it is constant weight while the right one is LWR using a distance weight. Figure obtained from [1]

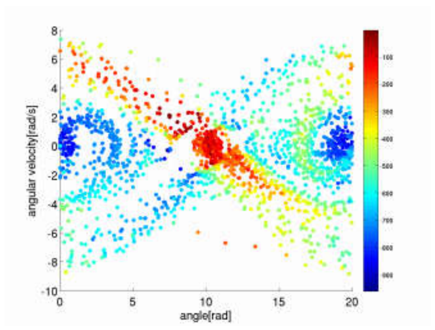
In this work, we use LLR to approximate the actor and the critic and LWR to approximate the process model, when it is needed.

4.3 Memory Management

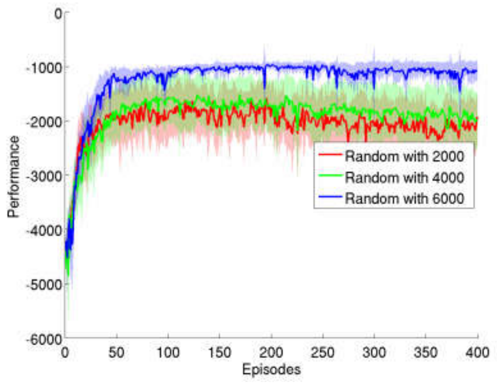
Managing the memory usage in LLR [18] is a key process, with impact in computational performance and accuracy of the approximator. We compared three ways to manage the memory: randomly, prediction and uniformly distributed and show that, on this particular problem, the uniformly distributed works better.

4.3.1 Randomly

Every time we have to remove a point from the memory to add another new one, we just remove a random point to make room to the new one. This is the least computational effort way to solve the problem. We can see on Figure 4.4a, once the system has converged and uses more and more often the best path, most of the memory observations get concentrated around this path. On Figure 4.4b we are comparing different memory sizes for this strategy, where a performance around -1000 is achieved only using 6000 samples.



(a) Final critic



(b) Performance on Standard Actor-Critic

Figure 4.4: Final critic and performance using the randomly strategy on LLR

4.3.2 Prediction

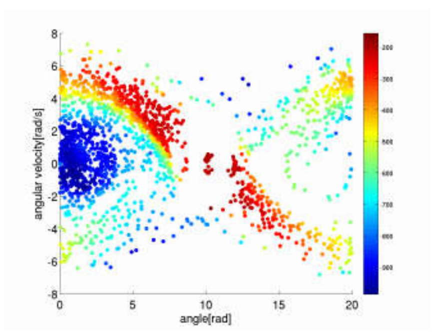
This method works by having a relevance value ϵ_i associated with every entry in the LLR memory $\mathbf{M}$ to describe how useful a given point is globally. So, every time we have to remove an entry, the one with lowest relevance is deleted.

Every time a sample $\mathbf{m}_i$ is used to predict some query $\mathbf{q}$, the associated ϵ_i value is updated with the difference from the model:

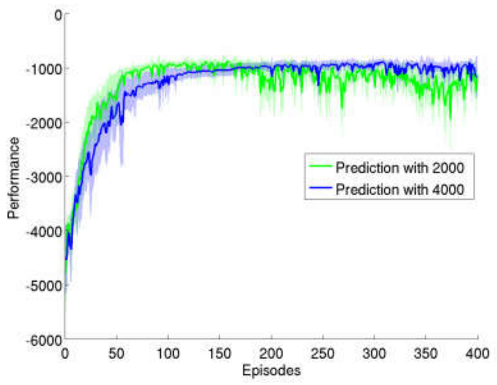
$$\epsilon_i = \eta\epsilon_i + (1-\eta)\left\|\mathbf{y}_i - \hat{f}(\mathbf{x}_i)\right\|_2^2 \tag{4.6}$$

where $\eta \in (0, 1]$.

On Figure 4.5a, we can see that the points get concentrated around the edges as we expected. On Figure 4.5b we can see that this strategy uses less samples than the random one, but a steady performance of -1000 is only achieved using 4000 samples.



(a) Final critic



(b) Performance on Standard Actor-Critic

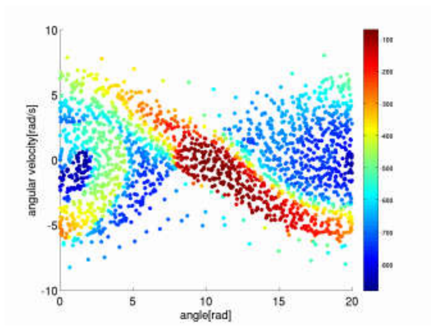
Figure 4.5: Final critic and performance using the prediction strategy on LLR

4.3.3 uniformly distributed

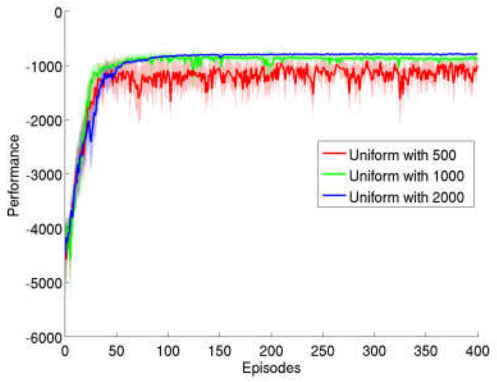
Here, the goal is to cover the most out of the space. As in the prediction method, every sample has an ϵ value associated as the relevance. The difference here is ϵ accounts for the mean Euclidean distance to the k-nearest neighbors $\mathcal{K}_i$ of the sample.

As before, every time we have to delete a sample, the one with lowest relevance is chosen. In this method, this means to delete the one that has its neighbors closer.

On Figure 4.6a, we can see that the space is much better covered using this strategy, while on Figure 4.6b we can see this strategy requires much fewer samples than the previous ones.



(a) Final critic



(b) Performance on Standard Actor-Critic

Figure 4.6: Final critic and performance using the uniform strategy on LLR

Finally, on Figure 4.7 is possible to see the tests that were made to decide the best strategy to this work. Uniform is as fast as the others, has a bit higher end performance and requires less memory. Besides, given the randomness nature of our study, fill the hyperspace is advised as the actor need to know what to do in every situation.

4.4 Comparison between Tile Coding and Locally Linear Regression

Given the different function approximators available, we decided to compare LLR against Tile Coding in order to decide which one to use. On Figure 4.8 it's possible to see that not only the LLR achieves a higher end performance but it also learns faster. When the number of samples is low, LLR generalizes better by not trying to estimate a large number of parameters from few samples, as Tile Coding

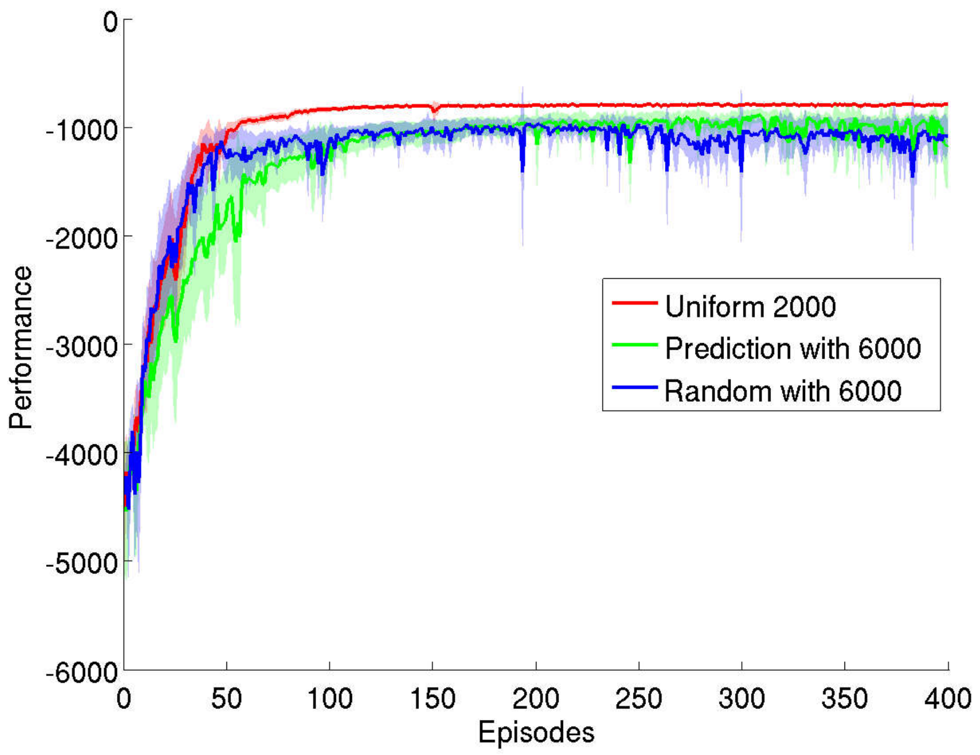


Figure 4.7: Comparison between the three different LLR memory strategies

would.

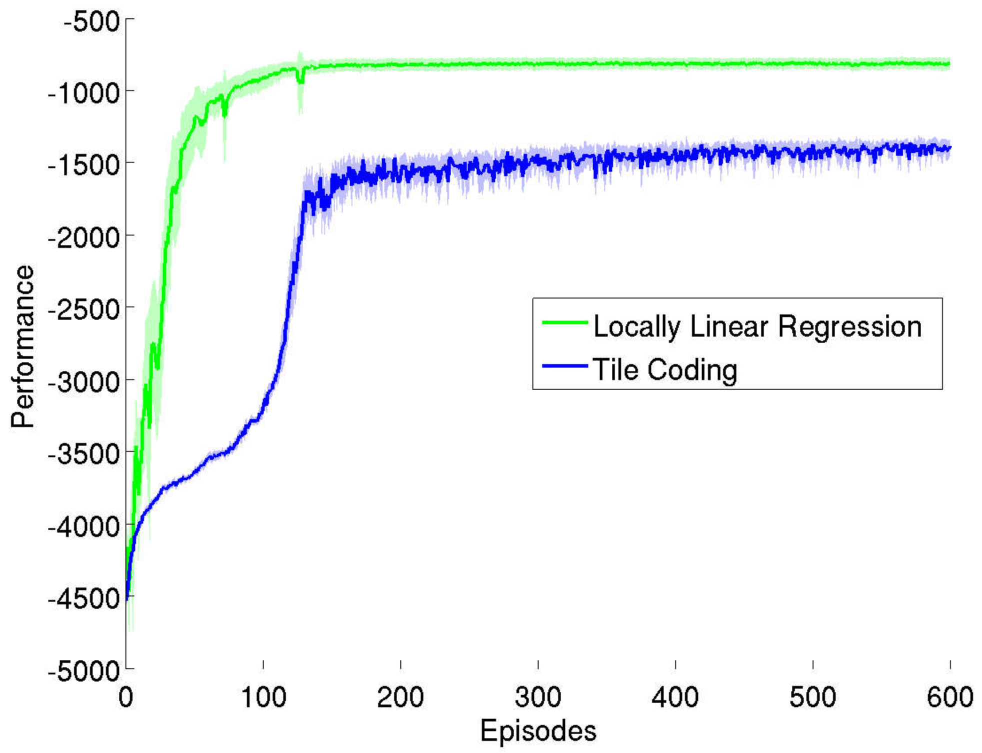


Figure 4.8: Comparison between Locally Linear Regression and Tile Coding on Standard Actor Critic

5 MODEL-BASED METHODS

5.1 Model Learning

In this section, we will describe what exactly is a transition model, why and how we can learn it.

5.1.1 What is the transition model

The transition model is a mathematical description of the environment, e.g., given an observation, the transition model is able to predict the next state and the reward associated. More precisely, the transition model has to approximate the underlying transition matrix T and the reward function R, described in Section 2.2.

5.1.2 Why to learn a model

Real experiences can be considered expensive. For example, if a robot is learning how to walk, every time it falls, it can break, which makes every real experience financially expensive, and if it takes a few minutes to fix it, it is a time consuming process.

In Standard Actor-Critic (Section 3.1), we use each experience to perform only one update. Even using eligibility traces, which updates many states, every transition is used only once. The main motivation to learn a transition model is to reuse past experiences in a way that new, unseen experiences, could be predicted.

5.1.3 How to learn a model

The transition model is a approximation of the transition matrix T, but as we are not considering the stochastic possibility, we are actually approximating the process model $x' = \hat{f}(x, a)$ which means $\hat{f}$ output the next state x' given the current state x and the chosen action a. In this work we are using the Locally Weighted Regression as function approximator (Section 4). The learning process is supervised, so we need real transitions to learn the model. Basically, we add another step in the standard actor critic algorithm (Section 3.1). After each update, we add the observed transition into the model.

In this work we are going to compare two different ways to use it: the Model Learning Actor-Critic (Section 5.2) and the Dyna framework (Section 5.3).

5.2 MLAC

The Model Learning Actor-Critic (MLAC) [10] extends Standard Actor-Critic (SAC) by considering a *process model*. A process model is a function $\hat{f}$ which approximates the state transition model, 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 4), the process model must be updated accordingly. In this work, we consider process models approximated by LLR, analogous to the actor and critic.

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 $V(s')$.

After action a_t is executed the gradient of the value function with respect to a is given by $\partial V/\partial a|_{a=a_t} = \partial V/\partial s|_{s=\tilde{s}'}\partial s/\partial a|_{a=\tilde{a}}$, where $\tilde{s}' = \hat{f}(s,\tilde{a})$. As such, for every state s, the MLAC gradient-descent actor update is given as follows

$$\pi(s) \leftarrow \pi(s) + \alpha_a \frac{\partial V}{\partial s}\bigg|_{s=\tilde{s}'} \frac{\partial s}{\partial a}\bigg|_{a=\tilde{a}} \tag{5.1}$$

MLAC Algorithm, described in Algorithm 5, is obtained from SAC (Algorithm 4) by updating the process model (line 9) and substituting the actor update (line 20-21) by (5.1).

Algorithm 5 MLAC algorithm

```
 1: procedure MLAC
 2: Repeat forever:
 3:     $\forall s \in S : e(s) \leftarrow 0$
 4:     $s_0 \leftarrow$ Initial state
 5:     Apply random input $a_0$
 6:     $t \leftarrow 1$
 7:     loop until episode ends:
 8:         Measure $s_t$ and $r_t$
 9:         Update the process model using $[s_{t-1}, a_{t-1}, s_t]$
10:         Let $\Delta_t$ be a sample from a zero-mean Gaussian
11:         $a_t \leftarrow \pi(s_t) + \Delta_t$                         ▷ Choose an action
12:         $\delta_t = r_t + \gamma V(s_t) - V(s_{t-1})$               ▷ Calculate td-error
13:         Call MLAC-Update($\delta_t$, $\Delta_{t-1}$, $\alpha_a$, $\alpha_c$, $s_t$)
14:         Apply $a_t$
15:         $t \leftarrow t + 1$
16: procedure MLAC-Update($\delta_t$, $\Delta_{t-1}$, $\alpha_a$, $\alpha_c$, $s_t$)
17:     Update the eligibility trace $e_t(s_t)$
18:     for all $s \in S$ do
19:         $V(s) \leftarrow V(s) + \alpha_c \delta_t e_t(s)$              ▷ Update the critic
20:     $\pi(s) \leftarrow \pi(s) + \alpha_a \frac{\partial V}{\partial s}\big|_{s=\tilde{s}'} \frac{\partial s}{\partial a}\big|_{a=\tilde{a}}$       ▷ Update the actor
21:     Clamp $\pi(s)$ to $A$
```

5.3 Dyna

The Dyna framework [16] 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 6, and uses the same SAC-Update procedure from Algorithm 4.

Note on line 13 of Algorithm 6, the current reward is not used to update the process model. This is because the reward is not being approximated in this work. Instead, we are using the true reward function R to while processing the updates for Dyna (line 20). In real applications, the reward function is usually built by hand, by an expert, so it is available to use in Dyna updates.

The simulated environment should 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 [1]. Note that the underlyng region close to the initial state is typically the 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,

Algorithm 6 Dyna-SAC algorithm

```
1: procedure DYNA-SAC
2: Repeat forever:
3:     $\forall s \in S : e(s) \leftarrow 0$
4:     $s_0 \leftarrow$ Initial state
5:     Apply random input $a_0$
6:     $t \leftarrow 1$
7:     $\overline{s}_0 \leftarrow$ Initial state
8:     $\overline{a}_0 \leftarrow$ random action
9:     $\overline{t} \leftarrow 1$
10:    loop until episode ends:
11:        Choose $\Delta_t$ at random
12:        Measure $s_t$ and $r_t$
13:        Update the process model using $[s_{t-1}, a_{t-1}, s_t]$
14:        $a_t = \pi(s_t) + \Delta_t$                                  ▷ Choose an action
15:        $\delta_t = r_t + \gamma V(s_t) - V(s_{t-1})$                ▷ Calculate td-error
16:        Call SAC-Update($\delta_t$, $\Delta_{t-1}$, $\alpha_a$, $\alpha_c$, $s_t$)
17:        for fixed number of updates per control step do
18:            Choose $\overline{\Delta}_{\overline{t}}$ at random
19:            $\overline{s}_{\overline{t}} \leftarrow \hat{f}(\overline{s}_{\overline{t}-1}, \overline{a}_{\overline{t}-1})$                ▷ Next simulated state
20:            $\overline{r}_{\overline{t}} \leftarrow R(\overline{s}_{\overline{t}-1}, \overline{a}_{\overline{t}-1}, \overline{s}_{\overline{t}})$              ▷ Transition reward
21:            $\overline{a}_{\overline{t}} \leftarrow \pi(\overline{s}_{\overline{t}}) + \overline{\Delta}_{\overline{t}}$                ▷ Next simulated action
22:            $\overline{\delta}_{\overline{t}} \leftarrow \overline{r}_{\overline{t}} + \gamma V(\overline{s}_{\overline{t}}) - V(\overline{s}_{\overline{t}-1})$
23:            Call SAC-Update($\overline{\delta}_{\overline{t}}, \overline{\Delta}_{\overline{t}-1}, \alpha_{sa}, \alpha_{sc}, \overline{s}_{\overline{t}}$)
24:            $\overline{t} \leftarrow \overline{t} + 1$
25:            if should restart simulated environment? then
26:                $\overline{s}_0 \leftarrow$ Initial state                ▷ Restart model
27:                $\overline{a}_0 \leftarrow$ random action
28:                $\overline{t} \leftarrow 1$
29:        Apply $a_t$
30:        $t \leftarrow t + 1$
```

the values being used are the same for the Dyna and the non-Dyna, as shown in Table 6.1.

5.3.1 Dyna-MLAC

By combining the Dyna framework with MLAC updates, the Dyna-MLAC algorithm is proposed. Dyna-MLAC is obtained from Algorithm 6, using the MLAC update described in Section 5.2. Note that the MLAC update procedure relies on the process model already provided by the Dyna framework.

6 EXPERIMENTS

In this chapter, the performance of the four considered algorithms (Standard Actor-Critic, Model Learning Actor-Critic, Dyna-SAC and Dyna-MLAC) will be evaluated using the pendulum swing-up environment. 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.

To guarantee a fair comparison between Dyna-MLAC and Dyna-SAC, we use the same set of parameters for the two (including same reward function $\hat{r}$ and learning steps), and consider the pendulum swing-up problem as described next.

6.1 The Pendulum Swing-Up

Considering the pendulum swing-up task, which is one of the benchmark problems described in [4]. The environment 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 6.1.

The task 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 torque u_t applied to the pendulum.

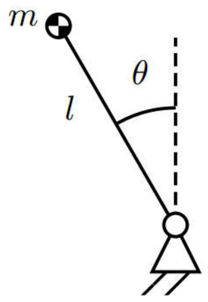


Figure 6.1: The pendulum swing-up environment.

Let $\boldsymbol{s}$ be the system state. Let θ be the angle of the pendulum. Then, the system state is given by the angle and the angular velocity $\boldsymbol{s} = [\theta, \dot{\theta}]$, where the initial state is $\boldsymbol{s}_0 = [\pi, 0]$.

The following equation characterizes the dynamics of the system given a fixed torque a, the current angle and the current velocity:

$$\ddot{\theta} = \left(mgl\sin(\theta) - \left(b + \frac{K^2}{R} \right) \dot{\theta} + \frac{K}{R} a \right) \frac{1}{J} \tag{6.1}$$

The constants are defined as $J = 1.91 \cdot 10^{-4} kgm^2$, $m = 0.055kg$, $g = 9.81m/s^2$, $l = 0.042\ m$, $b = 3 \cdot 10^{-6} Nms/rad$, $K = 0.0536Nm/A$ and $R = 9.5\Omega$.

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 $r = -5\theta^2 - 0.1\dot{\theta}^2 - a$.

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

To balance the pendulum at the top, a number of changes in direction might be required. In this section, we consider the pendulum swing-up problem with maximum allowed torque $a \in [-1.5, 1.5]$. In this setup, the weight must change direction twice before being able to balance at the top.

Table 6.1 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 6.2 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 approximator(s). The additional information extracted from the samples, stored in the process model, yields faster convergence 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 calculate the

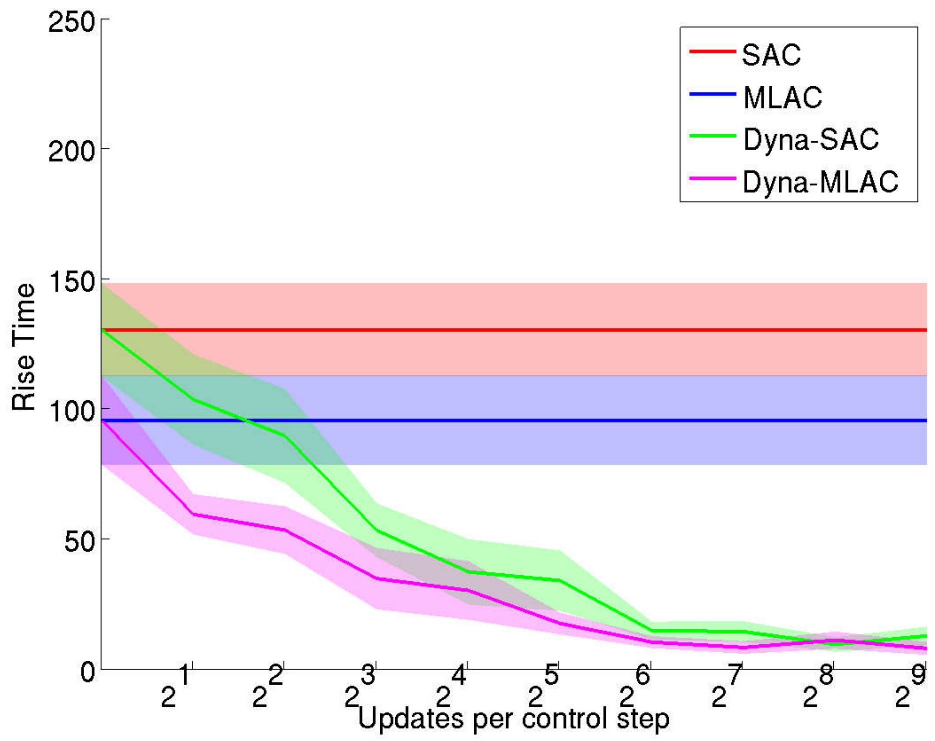


Figure 6.2: Actions $A \in [-1.5, 1.5]$ and performance of -1700.

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 6.2). Under MLAC, in contrast, a process model is used explicitly by the actor update rule (recall the dependence of (5.1) on the process model s' through $\partial s'/\partial a$). Figure 6.2 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 6.2, 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.

This trade-off can be better visualized in Figures 6.3 and 6.4. Figure 6.3 shows the surface obtained from plotting all the individual curves for every experiment, where the green plane correspond to the desired performance of -1800. The contour in Figure 6.4 shows exactly the trade-off discussed. Given a target performance (represented by one of the contours lines), one can decide to take more episodes (sampling) or update per control step (computation) and get the same result.

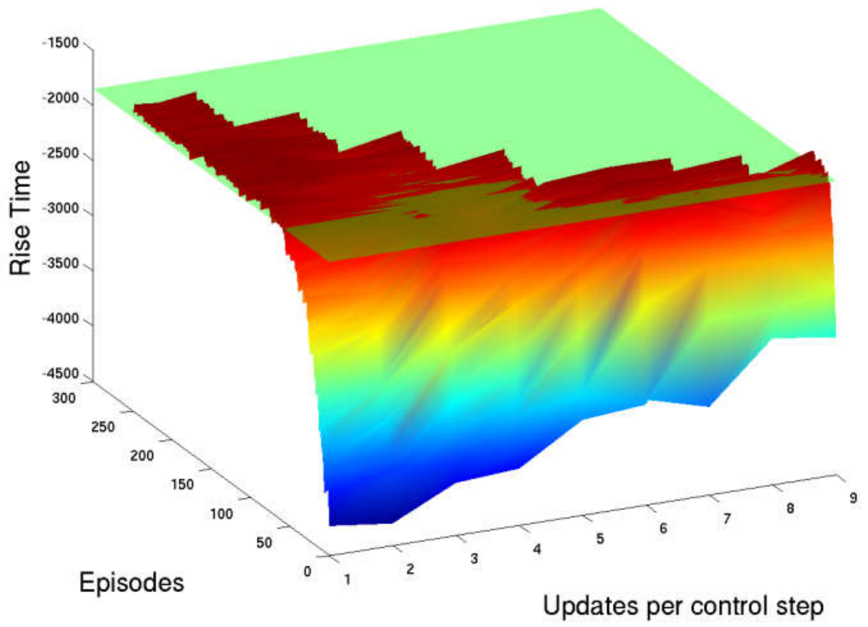


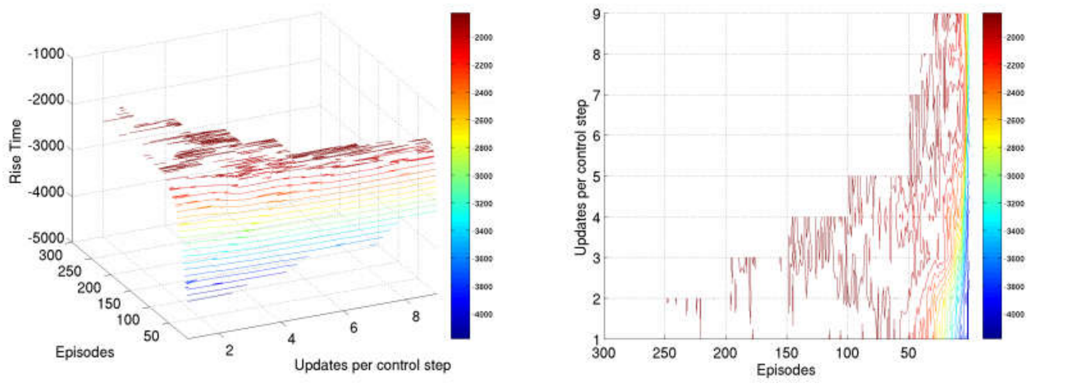
Figure 6.3: Dyna-MLAC performance surface. The green plane correspond to -1800 performance.

	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 6.1: Parameters used in algorithms

6.1.2 Model-Based Updates Are Not Always Necessary

Next, we consider a scenario where the motor voltage is controlled with $a \in [-3, 3]$. This allows the pendulum to be balanced with just one change of direction.



(a) Side-view (b) Upper-view

Figure 6.4: Contour plots of the performance curve for Dyna-MLAC (surface in Figure 6.3). It is possible to see the trade-off between episodes (sampling) and updates per control step (computation) given a required performance.

All the other parameters are shown in Table 6.1.

Figure 6.5 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 6.2. 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 6.5, the gap between SAC and MLAC is negligible. This is in agreement with [10], and explains why the performance of Dyna-MLAC and Dyna-SAC is similar in this experiment.

The same trade-off can also be visualized in this scenario in Figures 6.6 and

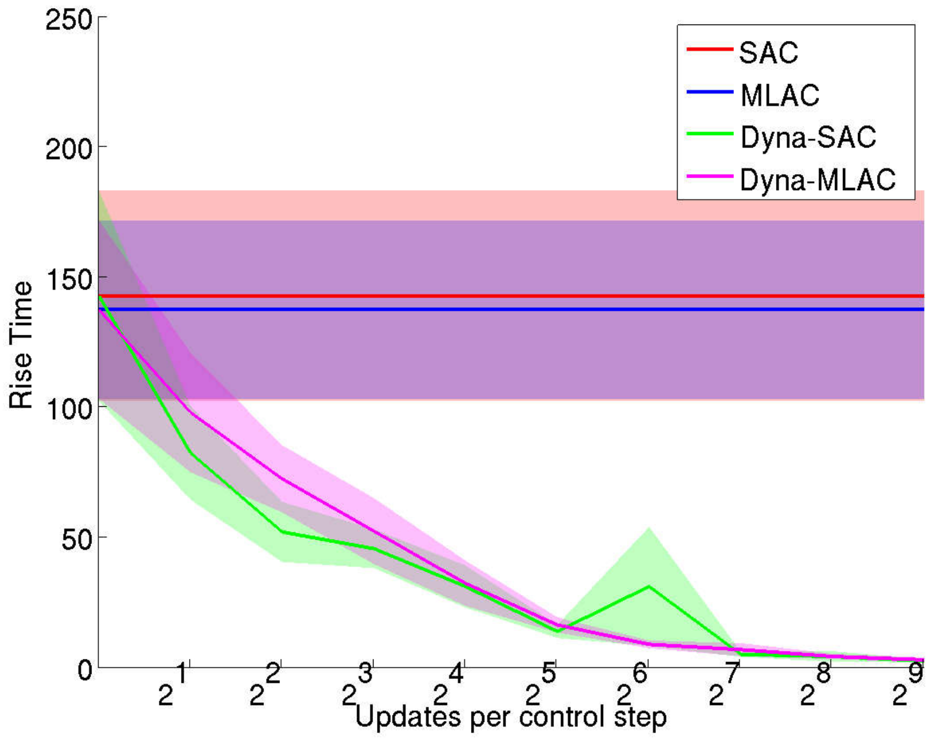


Figure 6.5: Actions $A \in [-3, 3]$ and performance of -1000.

6.7. Figure 6.6 shows the surface obtained from plotting all the individual curves for every experiment, but the green plane correspond to the desired performance of -1000 instead.

6.2 Convergence

The value function of the critic is shown in Figure 6.8. For each algorithm, the figure shows the final values stored in the LLR memory. Each dot corresponds to a pair $(\theta, \dot{\theta})$. Red dots are associated to greater rewards, and blue dots to smaller rewards. Note that all four algorithms converged to roughly the same solution.

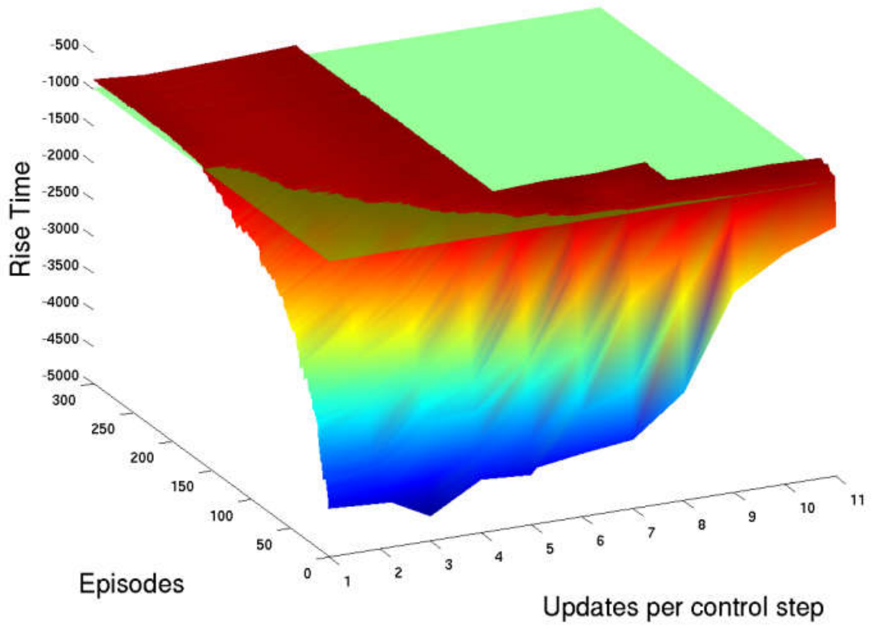
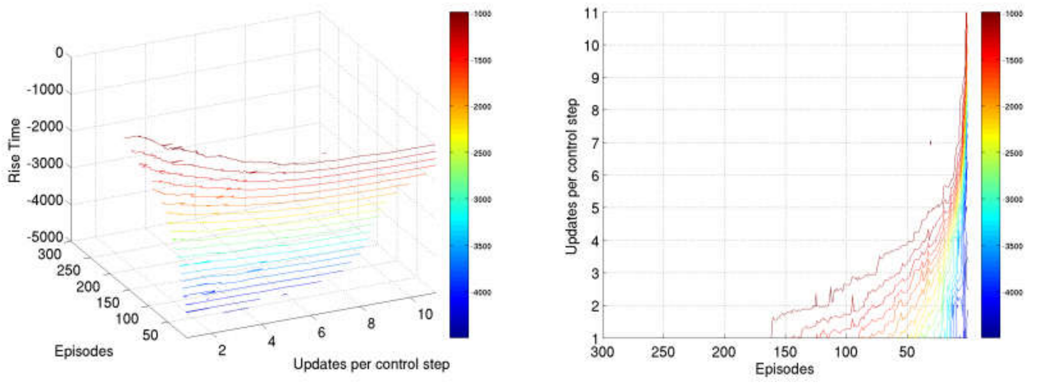


Figure 6.6: Dyna-MLAC performance surface. The green plane correspond to -1000 performance.

6.3 The Effect of the Exploration Rate

The exploration-exploitation trade-off is a known problem in every Reinforcement Learning algorithm. In this work, the exploration rate is a way to guarantee the exploration step, but what is its effect on the Dyna-SAC and on Dyna-MLAC algorithms?

Figure 6.9 shows the end performance of Dyna-SAC and Dyna-MLAC with 64 updates per control step as a function of the exploration rate. As in the Dyna framework we have two different exploration rates, one on the real environment and the other on the simulated environment, the "model" on the plot states for the



(a) Side-view (b) Upper-view

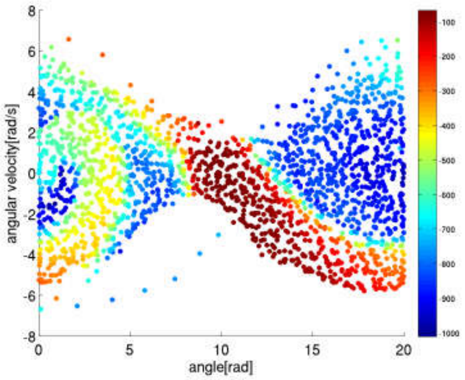
Figure 6.7: Contour plots of the performance curve for Dyna-MLAC (surface in Figure 6.6). It is possible to see the trade-off between episodes (sampling) and updates per control step (computation) given a required performance.

simulated environment.

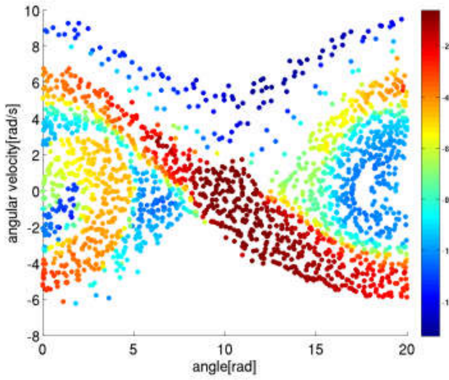
As we are doing 64 updates per control step, the effect of the exploration rate on the simulated environment is small compared to the effect on the real environment. Notice the importance of exploration on Dyna-SAC (blue line), as its performance drops if the exploration rate on the real environment is high. The same effect is not observable on Dyna-MLAC (red line), since the actor updates are towards the best action using the process model, the exploration is less relevant here.

6.4 The Effect of the LLR Memory Size

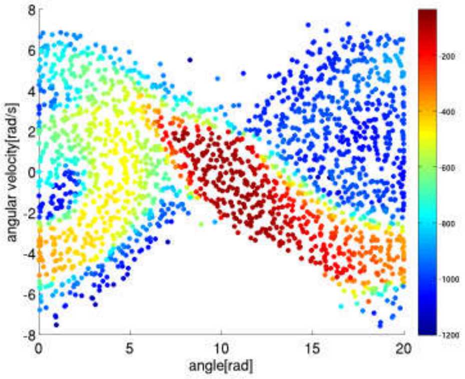
One of the most expensive steps in the algorithms considered in this work is the search for the k-nearest neighbors [3]. The computational complexity of this search is directly related to the size of the k-d tree memory. Given a finite amount



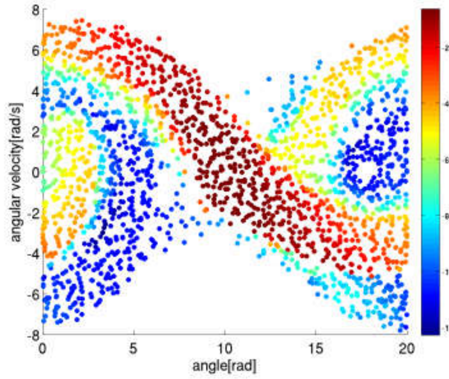
(a) SAC



(b) MLAC



(c) Dyna-SAC



(d) Dyna-MLAC

Figure 6.8: Final critics $V(s)$ in the standard environment. Every point is a sample in the LLR memory. Both Dyna-SAC and Dyna-MLAC with 64 updates per control step

of memory available to an agent, how should it be allocated to the critic, the actor and the process model?

Figure 6.10 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 6.1.1 for 20 episodes. If the memory capacity is

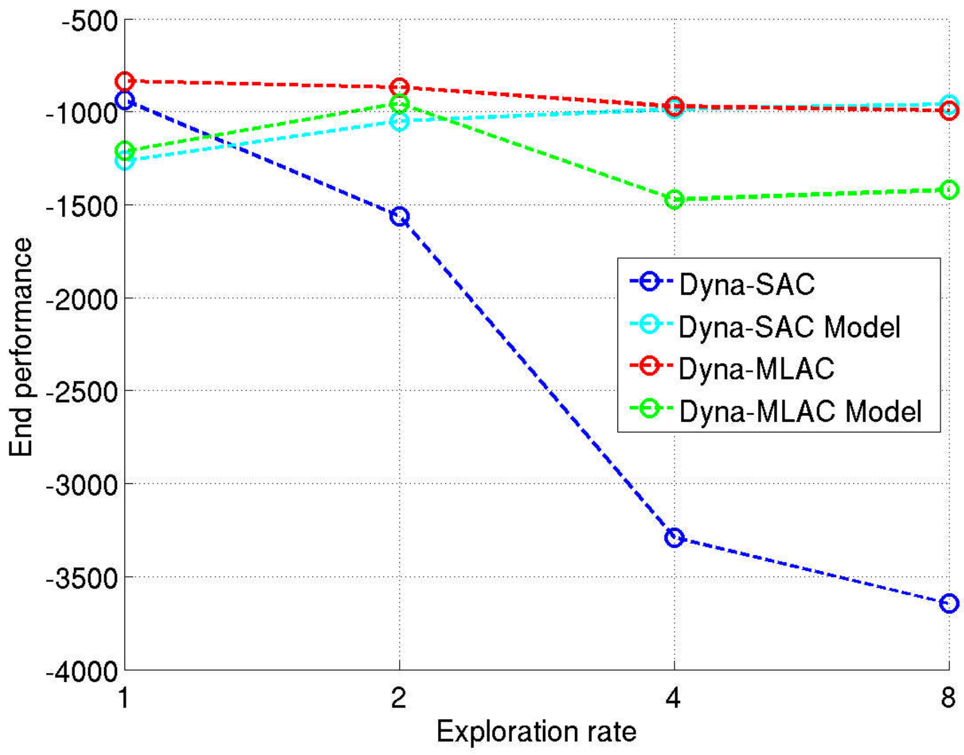


Figure 6.9: Log plot of different exploration rates for Dyna and Dyna-Mlac against the end performance

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.

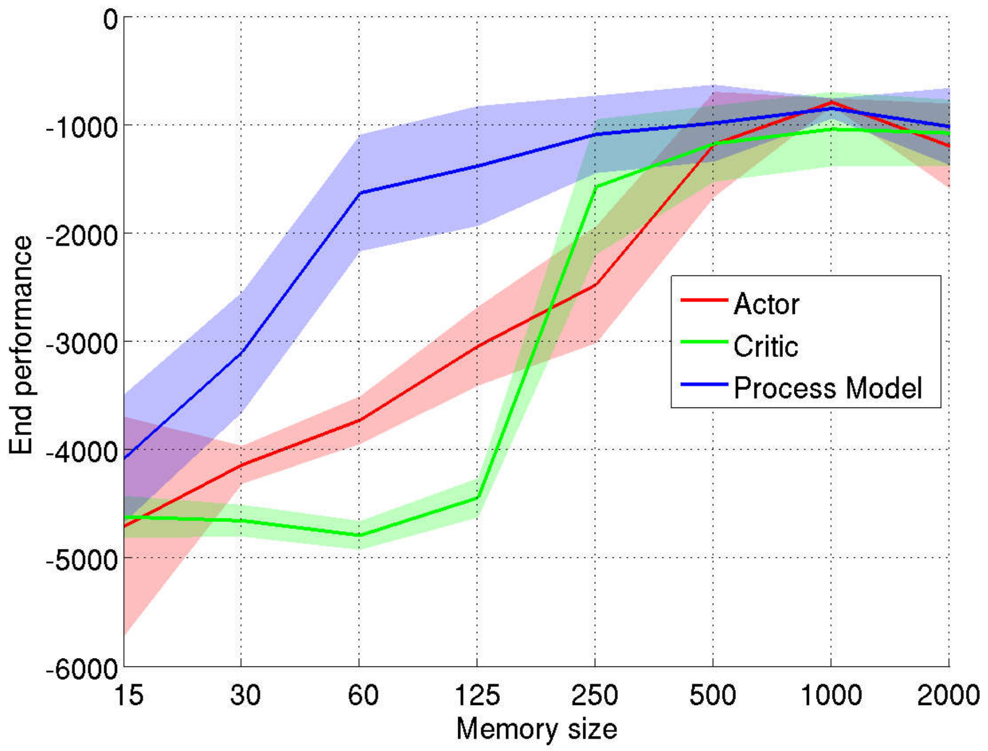


Figure 6.10: LLR memory size effect on Dyna-MLAC algorithm using 2^6 updates per control step.

7 CONCLUSIONS 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 work, 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 work, future work consists of further investigating under which conditions Dyna-MLAC outperforms its inspiring algorithms.

This shows that Dyna with a limited amount of updates per control step does not fully explore the information available in the process model, while MLAC can extract more information from it. A formal analysis showing this is left as a future work.

Another interesting future work, is knowing whether Dyna-MLAC is also faster in more complex environments, and the effect of a defective or very non-linear model, in both Dyna and MLAC updates.

Acknowledgments: this work was partially sponsored by CAPES (BJT/CSF) and CNPq.

REFERENCES

[1] ATKESON, C. G.; MOORE, A.; SCHAAL, S. Locally weighted learning. **Artificial Intelligence Review**, Dordrecht, p.11–73, 1997.

[2] BELLMAN, R. **Dynamic Programming**. 1.ed. Princeton: Princeton University Press, 1957.

[3] BROWN, R. A. Building a Balanced k-d Tree in $O(kn \log n)$ Time. **Journal of Computer Graphics Techniques**, v.4, n.1, p.50–68, 2015.

[4] BUŞONIU, L. et al. **Reinforcement Learning and Dynamic Programming Using Function Approximators**. Boca Raton: CRC Press, 2010.

[5] CHANDRASEKARAN, V.; JORDAN, M. I. Computational and statistical tradeoffs via convex relaxation. **Proceedings of the National Academy of Sciences**, v.110, n.13, p.E1181–E1190, 2013.

[6] CLEVELAND, W.; GROSSE, E. Computational methods for local regression. **Statistics and Computing**, London, v.1, n.1, p.47–62, 1991.

[7] DEISENROTH, M. P.; RASMUSSEN, C. E. PILCO: a model-based and data-efficient approach to policy search. In: INTERNATIONAL CONFERENCE ON MACHINE LEARNING. **Proceedings. . .** 2011. p.465–472.

[8] FISHMAN, G. S. **Monte Carlo**: concepts, algorithms, and applications. New York: Springer, 1996. (Springer series in operations research).

[9] GALINDO-SERRANO, A.; GIUPPONI, L. Distributed Q-Learning for Aggregated Interference Control in Cognitive Radio Networks. **IEEE Transactions on Vehicular Technology**, New York, v.59, n.4, p.1823–1834, 2010.

[10] GRONDMAN, I. et al. Efficient Model Learning Methods for Actor-Critic Control. **IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics**, New York, v.42, n.3, p.591–602, 2012.

[11] KATEHAKIS, M.; VEINOTT, J. A. The multi-armed bandit problem: decomposition and computation. **Mathematics of Operations Research**, Providence, v.12, n.2, p.262–268, February 1987.

[12] KONDA, V. R.; TSITSIKLIS, J. N. Actor-Critic Algorithms. In: SOLLA, S. A.; LEEN, T. K.; MULLER, K. R. (Ed.). **Advances in Neural Information Processing Systems 12**. Cambridge: MIT Press, 1999. p.1008–1014.

[13] PETERS, J.; VIJAYAKUMAR, S.; SCHAAL, S. Natural actor-critic. In: GAMA, J. (Ed.). **Machine Learning**: ECML 2005. Berlin: Springer, 2005. p.280–291. (Lecture Notes in Artificial Intelligence, 3720).

[14] RUSSELL, S. J.; NORVIG, P. **Artificial intelligence**: a modern approach. 3rd ed. Upper Saddle River: Prentice Hall, 2010.

[15] SUTTON, R. S. Learning to predict by the methods of temporal differences. **Machine Learning**, Boston, v.3, n.1, p.9–44, 1988.

[16] SUTTON, R. S. Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming. In: PORTER, B. W.; MOONEY, R. J. (Ed.). **Proceedings of the Seventh International Conference on Machine Learning**. Burlington: Morgan Kaufmann, 1990. p.216–224.

[17] SUTTON, R. S.; BARTO, A. G. **Reinforcement Learning**: an introduction. Cambridge: MIT Press, 1998.

[18] VAANDRAGER, M. et al. Imitation learning with non-parametric regression. In: IEEE INTERNATIONAL CONFERENCE ON AUTOMATION QUALITY AND TESTING ROBOTICS, 2012, Cluj-Napoca. **Proceedings. . .** New York:IEEE, 2012. p.91–96.

[19] WATKINS, C. J. C. H.; DAYAN, P. Technical Note: q-learning. **Machine Learning**, Boston, v.8, n.3-4, p.279–292, May 1992.

[20] XU, X.; ZUO, L.; HUANG, Z. Reinforcement learning algorithms with function approximation: recent advances and applications. **Information Sciences**, New York, v.261, n.0, p.1 – 31, 2014.