Motion Control using Reinforcement Learning

handed in
PRACTICAL COURSE

Leandro Pereira

Human-centered Assistive Robotics
Technical University of Munich

Univ.-Prof. Dr.-Ing. Dongheui Lee

Supervisor: Dr. Alejandro Agostini

TECHNISCHE UNIVERSITÄT MÜNCHEN

Human-centered Assistive Robotics

UNIV.-PROF. DR.-ING. DONGHEUI LEE

February 11, 2021

P R A C T I C A L   C O U R S E
for
Leandro Pereira, Mat.-Nr. 03739993

Motion Control using Reinforcement Learning

Problem description:

Reinforcement learning (RL) algorithms [2] are an appealing alternative to classical control mechanisms for controlling simple actuators. These algorithms permit learning control strategies automatically, without the need of knowing the actuator's dynamics and avoiding the fine tuning of controllers parameters. However, RL algorithms requires that every possible situation is experienced several times in order to learn a (sub-) optimal control policy. This is clearly a limitation in continuous environments, typical of motion control problems, where experiencing all the possible situations is unfeasible. To overcome this limitation, RL approaches are complemented with function approximation methods that permits inferring the consequences of actions in unexperienced situations from the experienced ones (generalization) [1]. This project comprises the combination of function approximation and RL methods for the control of simple actuators.

Tasks:

- Literature research.
- Task 1: Implementation of a simulator for an underactuated inverted-pendulum.
- Task 2: Implementation of a function approximation method based on variable resolution.
- Task 3: Combination of variable resolution FA with Q-learning, SARSA, and actor-critic.
- Task 4: Control of an inverted-pendulum using the RL approach of Task 3.

Bibliography:

[1] Alejandro Agostini and Enric Celaya. Online reinforcement learning using a probability density estimation. *Neural computation*, 29(1):220–246, 2017.

[2] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

Supervisor: Dr. Alejandro Agostini

(D. Lee)
Univ.-Professor

Contents

Chapter 1

Introduction

Machine Learning (ML) aims to recognise patterns and predict events based on previous observations. In this project we address a specific machine learning method: reinforcement learning (RL). In RL an agent learns by interacting with its environment, observing the results of these interactions, and receiving a reward (positive or negative) accordingly. There are many applications for reinforcement learning in the real world, but one of the main areas where there is a lot of research is in robotics. The robot tries to learn a task by repeatedly trying various actions until it successfully performs the task, such as walking, running or moving objects.

In many tasks that we would like to apply RL the state space is continuous, for instance, the case of robotics. Using conventional RL methods in these cases would lead to an infinite number of states which would make it impossible to find the optimal policy in finite time and with finite data. The key to solve this problem is using generalization in which we approximate a function by experiencing only some states and generalize them in order to build an approximation of the entire function.

In this project, it is implemented 3 different methods for controlling an inverted pendulum that can be seen as a robotic arm with only 1 degree of freedom. To solve the problem of the continuous state and action spaces it is used variable resolution to approximate our functions which is a very simple method in which the space is discretized following certain conditions. At the end of the implementation of the three methods with variable resolution, a comparison of the performance between the different methods is made.

Chapter 2

Reinforcement Learning

Reinforcement Learning (RL) is an area of machine learning where an intelligent agent ought to take actions in an environment in order to maximize the cumulative reward. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them. The focus of the agent is on finding a balance between exploration and exploitation. Reinforcement learning is one of the three basic machine learning paradigms alongside with supervised learning and unsupervised learning.

In contrast to *supervised learning*, RL doesn't require a dataset labelled by an external supervisor. In supervised learning, each example is a pair consisting of an input object (typically a vector) and a desired output value (also called the supervisory signal). A supervised learning algorithm analyzes the training data and produces an inferred function, which can be used for mapping new examples.
The other method, *unsupervised learning*, doesn't require a labelled dataset, instead has the purpose of searching for previously undetected patterns. RL could be confused as unsupervised learning but this tries to maximize a the cumulative reward instead of trying to find undetected patterns.
The formalization of the problem of reinforcement learning lies in the Markov Decision Processes (MDP), as models the environment, and Dynamic Programming (DP). The difference between classical dynamic programming and RL algorithms is that the latter does not assume the existence of a previous MDP model of the environment.

RL is composed essentially by two main elements: the agent and the environment and four sub elements: a policy, a reward function, a value function and, in some cases, a model of the environment.
The **policy** is responsible for defining the behavior of the learning agent at a given time. It maps the states of the environment to actions to be made once in those states. It gives the probabilities of taking the action a when in a state s. The policy

might be defined as basic function or a query table, while in other cases it might include extensive calculation, for example, a search process. The policy is enough to determine the behavior of the system.
The **reward function** defines the reward that the agent receives when it executes a particular action being in a given state. On each time step, an action is performed in the environment and this returns a number representing the reward obtained. The main goal of the agent in the learning process is to maximize the accumulated reward. This reward is the way to characterize good and bad actions taken in a particular state. If the action performed is followed by a low reward, then the policy must learn that in the future this is not the best action to take in the specific state.
The **value function** determines what is good in the long run, in contrast to the reward signal that reveals what is a good in two consecutive time steps. The value of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state, just by following the policy learnt. This considers the states that are likely to be followed in the next iterations and the reward in those states. For instance, a state can yield a low immediate reward even though have a high value because in the future the states can yield high rewards.
Some reinforcement learning method consider another element, the **model of the environment**. This simulates the behavior of the environment that allows inferences to be made about how the environment will behave. Methods that use this this model are called model-based methods in contrast to the methods model-free that must use exploration in order to learn the model of the environment.

2.1 Markov Decision Processes and Dynamic Programming

An (MDP) is a discrete-time stochastic control process. It is describes by a set of actions, A, a set of states, S, and two probabilities described as:
$P_a(s, s') = Pr(s_{t+1} = s'|s_t = s, a_t = a)$ is the probability that action a in state s at time t will lead to state s' at time $t + 1$,
$R_a(s, s')$ is the immediate reward (or expected immediate reward) received after transitioning to state s' from state s, due to action a.

At each time step t, the agent is in some state $s \in S$, and the decision maker selects an action $a \in A$ that is available in state s. At time step $t+1$ as consequence of the action executed, the agent receives a numerical reward $R_a(s, s')$.

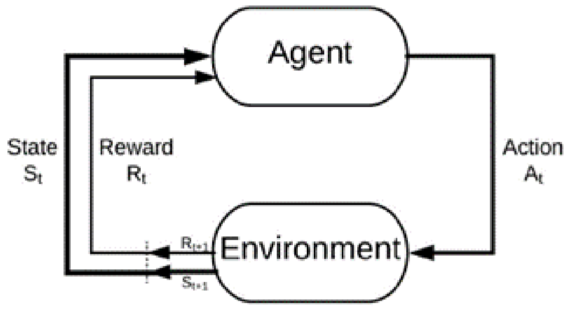

Figure 2.1: The agent-environment interaction in a Markov decision process

2.2 Problems in Reinforcement Learning

One of the difficulties that emerge in reinforcement learning, and not in other types of learning, is the trade-off between exploration and exploitation. To get a lot of reward, a reinforcement learning agent should favor action that it has attempted before and discovered to be effective in creating reward. However, to find such actions, it needs to attempt actions that it has not chosen previously. The agent needs to exploit what it has just experienced to obtain reward, however it additionally needs to explore to make better action selections in the future.
The problem is that neither exploration nor exploitation can be pursued uniquely without failing at the task. The agent should attempt different actions and continuously favor those that return the best reward.

In conventional RL methods, only Markov Decision Processes with finite states and actions are considered. However, in real world applications the states and actions are infinite because they are continuous time problems. A discretization of the entire space is not a solution as this would lead to a huge number of states, moreover some regions are more important than others, and therefore these need a fine grained discretization to get good results. There are several possible methods to solve this problem. In this project we implemented function approximation with variable resolution. Chapter 3 gives a more detailed explanation of this method.

2.3 Reinforcement Learning Methods

2.3.1 Q-Learning

Q-Learning is one of the most popular Reinforcement Learning methods due to its simplicity. It is a *model-free* method, which means it does not require a model of the environment. It aims to find an optimal policy in the sense of maximizing the expected value of the total reward over all successive steps. The updating formula of the Q-values is given by the equation 2.1.

$$Q(s,a) \leftarrow Q(s,a) + \eta \left(r(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a) \right) \tag{2.1}$$

In this method the action-value function, Q, directly approximates the optimal action-value function $q*$, independently of the policy followed, which means this a *off-policy* RL method. This allows this method to have a more flexible definition of the exploration-exploitation strategy since this does not affect the updating of the Q-function and it also allows a faster convergence to the optimal policy. However, the *off-policy* nature can lead to abrupt changes in the policy followed by the agent and, therefore, in its behavior which in some applications can lead to damage of the equipment. In these cases other methods would be preferred.

2.3.2 SARSA

SARSA interacts with the environment and updates the policy based on actions taken, in contrast to Q-learning this is a *on-policy* learning algorithm. Its name reflects the process for update the Q-value, since it depends on the current state of the agent (S), the action of the agent chooses (A), the reward R generated for choosing this action, the state S that the agent enters after taking that action and the next action the agent chooses in the new state (A). This sequence of actions generates the acronym SARSA. In SARSA the Q-values are updated in a similar way as in Q-Learning but here the update depends on the next state and next action given by the policy instead of following the greedy-policy as described in the previous section. The update is given by equation 2.2.

$$Q(s,a) \leftarrow Q(s,a) + \eta \left(r(s,a) + \gamma Q^{\pi}(s',a') - Q(s,a) \right) \tag{2.2}$$

2.3.3 Actor-Critic

The actor-critic method has a separate structure to explicitly represent the policy learnt independent of the value function. The policy structure is know as the actor responsible to select the action to execute and the estimated value function is knows as the critic, because it "criticizes" the actions made by the actor. The learning is *on-policy*, the critic must learn and critique the policy followed by the action. The output of the critic gives a TD error that is used for the learning of both actor and critic. The implementation of this method is explained in section 4.4.

2.4 Hyper-parameters

All the above methods require hyper-parameters that must be selected so that the policy suits the desired value as soon as possible and in a stable way. In the implementation of these methods three main parameters have been used: the learning rate (α), discount factor (γ) and the initial conditions such as variance.

2.4.1 Learning Rate (α)

The discount factor determines the importance of future rewards, if the factor is 1 the agent will consider all future rewards with the same weight, if it is 0 the agent will only consider the current reward. In the case of the implementation of this project, being a continuous problem, the discount factor should be closer to 1 than 0 because it's easily executed a big amount of steps and quickly we could forget the future rewards which is not the intended.

2.4.2 Discount factor (γ)

The discount factor determines the importance of future rewards, if the factor is 1 the agent will consider all future rewards with the same importance, if it is 0 the agent will only consider the current reward. In the case of the implementation of this project, being a continuous problem, the discount factor should be closer to 1 than 0 because they are easily executed enough steps and quickly we could forget the future rewards that is not the intended.

2.4.3 Initial conditions

In all methods of this project, it was necessary to define initial conditions for state-action space. One of the parameters used to determine the probability of exploration or exploitation is the variance of the estimated Q-value in the state-action space. A large variance means that the values used to estimate the real value are quite dispersed, which will increase the probability of exploration, on the other hand, a very low value determines that the estimate made is perfect and therefore no exploration is necessary. In the case of the actor-critic the greedy-policy was used and thus had to be defined a ϵ which determines the probability of exploitation. This value is updated and converges to 0 to allow convergence of the algorithm.

Chapter 3

Function Approximation

A state is represented by the combination of observable features or variables. This means every time a feature or a variable has a new value, it results in a new state. In the conventional RL methods the value function, or state-action function, has a tabular representation which means all the possible actions must be experienced several times for the agent be able to learn. This representation only covers problems with finite discrete states and actions. In many tasks on which we apply RL the states and actions are continuous so we have an infinite number of possible actions and/or states, therefore would be impossible find an optimal policy in a reasonable time and not only would be a problem of time but also the memory needed to store all the variables would be impracticable. In many of our tasks many states will never have been seen before so we can use the features of the state to generalize the estimation of the value at states that have similar features. This is the problem of *generalization*. This can be solved using function approximation (FA), which generally aims to select a function f using historical or available observations from the domain. In the case of RL, FA aims to find a generalized state space that is a good approximation of the larger set containing unseen states. There are different function approximation algorithms used in RL context, in this problem it was used Variable Resolution.

3.1 Variable Resolution

One of the simplest function approximation methods is to represent each of the states as a finite portion and then treat the problem as in a tabular case. In the context of RL, Variable resolution (VR) is a method that aims to represent the value function, or the action-value function, using a partition of the state-action space. In the continuous space, it is created a bounded area which aggregates all the intermediate values, all the different states inside the defined area make a part.

In the case of the approximation of the action-value function, the pair (state, action) is the input of the function and the output is the approximated Q-Value. When the approximation of a part is not good enough, defined by certain conditions, it is divided into two halves along the variable with larger dimension. Figure 3.1 shows an example of the state-action space representation after a sufficient approximation has been made.

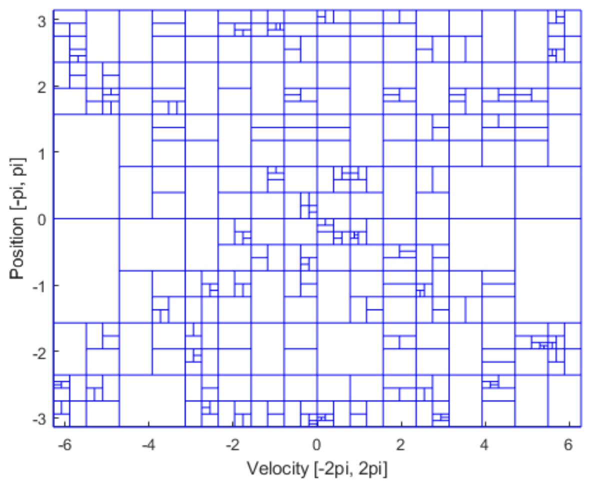

Figure 3.1: Representation of the state space with two variables using VR

Chapter 4

Implementation

4.1 Simulator

To evaluate the performance of all the methods implemented an inverted pendulum with limited torque was used. In the end of the learning process the agent should be able to swing and stabilize the pendulum in the upright position and stay there for an indefinite period. The policy learnt is not straightforward due to the limitation of the torque, the policy must learn to swing the pendulum in order to make its kinetic energy overcome the load torque and reach the upright position. The image 4.1 shows the schematic of the implemented pendulum.

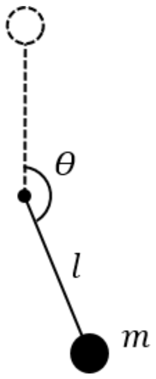

Figure 4.1: Pendulum schematic

$$ml^2 = -\mu\dot{\theta} + mgl\sin\theta + \tau \tag{4.1}$$

The dynamics of the pendulum are defined by the equation 4.1. Where θ is the angular position, m is the mass of the pendulum considering the center of mass in

Variable	**Value**
θ	$[-\pi, \pi]$
$\dot{\theta}$	$[-2\pi, 2\pi]$
τ	$[-5, 5]$
m	1
l	1
g	9.8
μ	0.01
dt	0.001

Table 4.1: Parameters used in the pendulum implementation

the distal end with respect to the joint, l is the length, g is the gravity constant, μ is the friction factor and τ is the input torque. The values used are presented in the table 4.1. The implementation was made in MATLAB using the Euler method with a differential time of 0.001 seconds and an actuation interval of 0.1 seconds. It was also created a plotting function to visualize the behavior of the pendulum during the training and testing process. The generated image is shown below in figure 4.2.

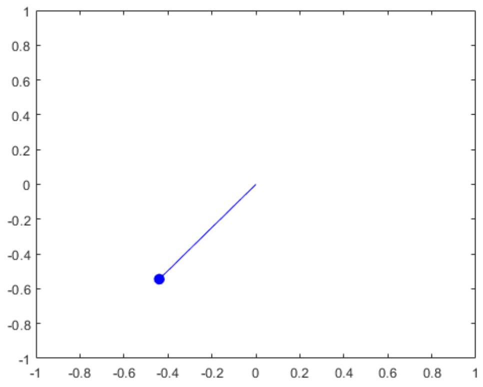

Figure 4.2: Pendulum plot generate by the plotting function in MATLAB

4.2 Q-Learning with Variable Resolution

To implement the Variable Resolution state-aggregation technique was necessary to define the partition, consisted in a grid formed by N^D parts where D represents the dimension of the state-action space and N the number of divisions made in the range of each variable. In this case the dimension of the state-action space is three since we have two variables to represent the state and one to represent the action.

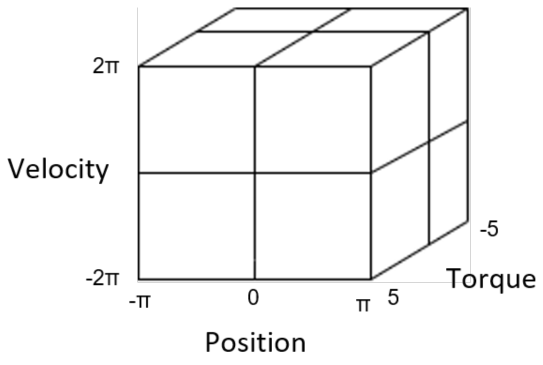

Figure 4.3: State-Action-Space representation after initialization for $N = 2$

In each part i, there is an estimation of the action-value $Q_i = \hat{Q}(s, a)$, that is assigned to all action-value pairs belonging to the part. To represent the partition, it was used an array of structures. Each element of the array represented a state-action pair. The structure created stored the Q-value, the range of the state covered by the part, in this case, the range of velocity and position, and the range of actions covered. It was also added a field to store an estimation of the variance of Q in the part and a variable to store the number of samples experienced so far in the part. All values are initialized with chosen parameters in order to optimize convergence, the values used are described in table 4.2.

Variable	**Value**
γ	0.98
S^2	500
a	0.1
b	10
thr_{err}	0.1
thr_n	50

Table 4.2: Hyperparameters used in Q-Learning algorithm

In the beginning is defined the current position as π and the velocity as zero, so, the training episode starts with the pendulum stopped and pointing down. At this point, an action a must be chosen following the exploration-exploitation strategy. To obtain the action to perform it is calculated an action value $\hat{q}_{rand}(s, a')$ for each evaluated action using the formula 4.2.

$$\hat{q}_{rand}(s, a') = N_i(q; \overline{Q}_i, S_i)|(s, a') \tag{4.2}$$

It is used a normal distribution with mean $\hat{Q}_i$ and variance S_i belonging to the part i. Then is select the action with maximum value, $\hat{Q}_{rand}(s, a)$. Since the action is not a defined value but instead a range of values it is selected, as the action to execute, a random value between the action limits of the part. In this project, the action is the torque applied to the inverted pendulum. After the selected action be executed a reward is received, which evaluates how good or bad was the decision taken. This reward is calculated by the equation 4.3.

$$r(s,a) = -|\theta| \tag{4.3}$$

with θ as the final position of the pendulum after the execution of the action. In this implementation the action is executed for 0.1 seconds. The reward is then used to calculate the value $q(s,a)$,given by the equation 4.4

$$q(s,a) = r(s,a) + \gamma \max_{a'} Q(s,a') \tag{4.4}$$

where s' in the state observed after executing the action and $\hat{Q}(s',a)$ is the approximation of the action-value at (s',a), given by the part containing (s',a). The $\gamma \in [0,1]$ is the discount factor which regulates the influence of future rewards by introducing some forgetting of past observations. In this approach, the calculated value $q(s,a)$ is used to update Q_i using the incremental formula 4.5 for discounted average estimations.

$$Q_i = Q_i + \eta_i(q(s,a) - Q_i) \tag{4.5}$$

Where η_i is the learning coefficient. In this implementation is used the formula 4.6.

$$\eta_i = \frac{1}{an_i + b} \tag{4.6}$$

to represent the learning coefficient, where n_i is the number of samples experienced so far in the part, and $a \in [0,1]$ and $b \geq 0$. These values are hyper-parameters defined to optimize the convergence. The variance of the part containing the state-action pair experienced is also updated using an equivalent incremental formula 4.7.

$$S_i^2 = S_i^2 + \eta_i((q(s,a) - Q_i)^2 - S_i^2) \tag{4.7}$$

In the end it is necessary to verify if the approximation of the state-space is good enough, if not we have to split the corresponding part. For that it was considered that a poor approximation of a Q-value is carried out when the inequality 4.8 is fullfilled. This means the part must be split when the error between the approximated value and the current q value is high.

$$(q(s,a) - \hat{Q}(s,a))^2 \geq th_{err} \tag{4.8}$$

It was also considered whether the number of samples experienced in a part is higher than a certain threshold to be considered as a confident estimation.

$$n_i > th_n \tag{4.9}$$

When both of the condition are fulfilled the part containing the sample (s, a) is divided in two halves along the dimension where the length of the variable is the biggest in relation to the total span of the variable, therefore, previously, the variables were normalized between 0 and 1 in order to make a fair comparison between them.

Algorithm 1 Q-Learning with VRFA

```
Initialize partition and hyper-parameters
Initialize current state (vel = 0, pos = $\pi$)
loop
    Select action $a$ following the exploration-exploitation strategy
    Execute the action $a$, get the reward $r(s,a)$ and observe the new state $s_{t+1}$
    Estimate maximum $Q_{max} = \max_{a'} Q(s', a')$
    Generate $q(s,a) = r(s,a) + \gamma Q_{max}$
    Calculate learning rate: $\alpha \Longleftarrow 1/(\mathrm{a}\ \mathrm{n}_i + b)$
    $Q_i \Longleftarrow Q_i + \alpha(q(s,a) - Q_i)$
    $S_i^2 \Longleftarrow S_i^2 + \alpha((q(s,a) - Q_i)^2 - S_i^2)$
    if $(q - \bar{Q}_i)^2 \geq th_{err}$ and $n_i \geq th_n$ then
        Split $X_i$ in two halves along the dimension with largest size
        Initialize estimations of the Q-Value and variance in each partition
    end if
    $s \Longleftarrow s'$
end loop=0
```

4.3 SARSA

To represent the action-value function in SARSA it was used VRFA as in Q-Learning. The difference is that this is a *on-policy* algorithm as described in section 2.3.2 which changes the updating equation of the Q-Value. In this case the Q-function is updated using the formula 4.10

$$Q^{\pi}(s,a) \leftarrow Q^{\pi}(s,a) + \eta\left(r(s,a) + \gamma Q^{\pi}(s',a') - Q^{\pi}(s,a)\right) \tag{4.10}$$

For the exploration-exploitation strategy was used the ϵ-greedy-policy. It was defined a global ϵ that was decremented over time following the formula 4.11

$$\epsilon = \frac{1}{\beta \cdot t + 1/\epsilon_0} \tag{4.11}$$

where ϵ_0 is the initial value for ϵ and β is the decay rate over time t. In the end is verified the approximation of the state-action value function in order to divide a part if the approximation does not full-fill the criteria, the same as in Q-Learning. Contrarily to Q-Learning, in this method is not necessary to update the variance of the samples in a part because this value is not used for exploration, instead, here is used the ϵ-greedy algorithm.

The hyper-parameters used in this algorithm are shown in table 4.3.

Variable	**Value**
ϵ_0	0.5
β	0.000001
a	0.001
b	5
γ	0.92
thr_{err}	0.1
thr_n	200

Table 4.3: Hyperparameters used in SARSA algorithm

Algorithm 2 SARSA with VRFA

```
Initialize partition and hyper-parameters
Initialize current state (vel = 0, pos = $\pi$)
Select action $a$ following the current policy
**loop**
    Execute the action $a$, get the reward $r(s, a)$ and observe the new state $s'$
    $\epsilon \Longleftarrow 1/(\beta \cdot t + 1/\epsilon_0)$
    Choose $a'$ in the new state $s'$ using the $\epsilon$-greedy strategy
    Generate $q(s, a) = r(s, a) + \gamma Q^{\pi}(s', a')$
    Calculate learning rate: $\alpha \Longleftarrow$1/(a n$_i$ + $b$)
    $Q_i \Longleftarrow Q_i + \alpha(q(s, a) - Q_i)$
    **if** $(q - \bar{Q}_i)^2 \geq th_{err}$ **and** $n_i \geq th_n$ **then**
        Split $X_i$ in two halves along the dimension with largest size
        Initialize estimations of Q-Value in each partition
    **end if**
    $s \Longleftarrow s'$
    $a \Longleftarrow a'$
**end loop**=0
```

4.4 Actor-Critic

In actor critic it is necessary to approximate two functions: the actor and the critic. For that it was used the same method as before but in this case it was created two arrays of structures. The actor was represented by an array of structures containing the state, the action to perform in that state and its variance. The critic was represented as an action-value function similarly as in Q-Learning. The learning algorithm starts in the same way as in the other methods, the initial state is defined and 500 interaction are run. Firstly, a random action is select by the actor following a exploration-exploitation strategy. It is used a normal distribution with mean in the action of the corresponding current state. The action selected is given by equation 4.12

$$a_{rand}(s) = N_i(a; \overline{A}_i, S_i)|(s) \subset X_i \tag{4.12}$$

Subsequently, the action select is executed and the reward and the new state of the environment is stored. It is then calculated the Q-value which is given by the critic in the new state-action pair. The new action is obtained by the actor following the policy in the new state. This value is used to estimate the value $q(s_t, a_t)$ as in equation 4.13

$$q(s_t, a_t) = r(s, a) + \gamma Q^{\pi}(s_{t+1}, a_{t+1}) \tag{4.13}$$

To update the critic the sample $(s_t, a_t, q(a_t, s_t))$ is used in equation 4.14.

$$Q_i(s_t, a_t) = Q_i(s_t, a_t) + \eta_i(q(s_t, a_t) - Q_i(s_t, a_t)) \tag{4.14}$$

The learning rate (η_i) is calculated using the formula 4.15 used in the previous methods rewritten here

$$\eta_i = \frac{1}{an_i + b} \tag{4.15}$$

where a and b are hyper-parameters and n_i is the number of samples experienced in part i. In order to check if the action performed by the actor was a good decision we must compare the sample $q(s, a)$ and the Q-Value given by the critic. If the q is bigger than the Q-value the action selected was a better choice than the action before which means we must select the a_{target} as the action selected by the actor. If the Q-value given by the critic is bigger than the q we must select the a_{target} as the action that gives the highest reward, in other words, the action from the current state that has the biggest Q-Value in the critic. The a_{target} value is used to update the actor function using the formula 4.16.

$$a(s_t) = a(s_t) + \eta_i(a_{target} - a(s_t)) \tag{4.16}$$

In the end of the iteration it was evaluated the approximation of the critic and the actor. Similarly to the previous methods, the action-value function of the critic was divided using the criteria 4.17 and 4.18.

$$(q - \bar{Q}_i)^2 \geq th_{err} \tag{4.17}$$

$$n_i \geq th_n \tag{4.18}$$

To evaluate the approximation of the actor a similar approach was used but instead of approximate the Q value, it was approximated the action. Therefore, the conditions 4.19 and 4.20 were used to determine whether the part had to be divided or not.

$$(a_{target} - \bar{A}_i)^2 \geq th_{err} \tag{4.19}$$

$$n_i \geq th_n \tag{4.20}$$

Algorithm 3 Actor-Critic with VRFA

```
Initialize partition and hyper-parameters
Initialize current state (vel = 0, pos = $\pi$)
**loop**
  Select a random action $a$ from the actor for state $s_t$
  Execute the action $a_t$, get the reward $r(s_t, a_t)$ and observe the new state $s_{t+1}$
  Generate $q(s_t, a_t) = r(s_t, a_t) + \gamma Q^{\pi}(s_{t+a}, a_{t+1})$
  Calculate learning rate: $\alpha \Longleftarrow 1/(\mathrm{a}\ \mathrm{n}_i + b)$
  Update the Q-function: $Q(s_t, a_t) \Longleftarrow Q(s_t, a_t) + \alpha(q(s_t, a_t) - Q(s_t, a_t))$
  Generate $a_{target}$
  Update actor policy: $A(s_t) = A(s_t) + \alpha(a_{target} - A(s_t))$
  Update actor variance: $S_i^2 \Longleftarrow S_i^2 + \alpha((a_{target} - A_i)^2 - S_i^2)$
  **if** $(q - \bar{Q}_i)^2 \geq th_{err_{crit}}$ **and** $n_i \geq th_{n_{crit}}$ **then**
    Split $X_i$ in two halves along the dimension with largest size
    Initialize estimations of Q-Value in each partition
  **end if**
  **if** $(a_{target} - \bar{A}_i)^2 \geq th_{err_{act}}$ **and** $n_i \geq th_{n_{act}}$ **then**
    Split $Y_i$ in two halves along the dimension with largest size
  **end if**
  $s_t \Longleftarrow s_{t+1}$
**end loop**=0
```

Chapter 5

Evaluation

5.1 Q-Learning Results

Q-Learning algorithm was trained during 300 episodes, with each episode running 500 iterations. Each iteration correspond to an action executed in the environment and 0.1 seconds elapsed. So the algorithm was trained for 15000 seconds which corresponds to 4 hours and approximately 10 minutes. After each training episode was performed a testing episode in order to verify the convergence of the agent. The reward accumulated during the testing episode was stored in an array which allowed plotting the graph of figure 5.1 of the total reward accumulated over time in each testing episode.

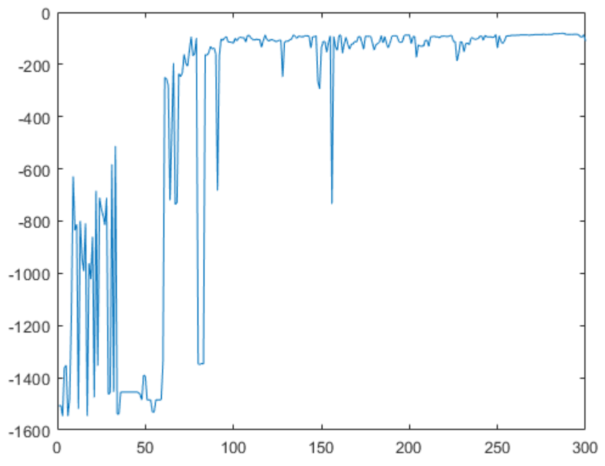

Figure 5.1: Plot of the accumulated reward for 300 episodes using the Q-Learning algorithm

We can observe a stable and good convergence after 250 episodes, at the end of the training the accumulated reward is approximately -80.

By observing the pendulum animation we can say that the agent learned to take the pendulum to the upright position without a big oscillation and after reach the final position it was able to maintain it there with a very small oscillation even though this oscillation around the zero position could be even smaller if the hyper-parameters were chosen more accurately.

At the end of the training, the state-action space was divided into approximately 6200 parts. A bigger division was made around the speed and zero position because these are states in which the agent will be when the pendulum is balanced in the upward position. Therefore, it is necessary to have a greater precision of the actions taken because a small error in the executed action can make the pendulum fall to the rest position. In left plot in figure 5.2 is shown the division of the space for a torque range that contains the value 0. That is, all the states for which the action range of the part contains the value 0. In this figure we can see that there are small divisions around the position and zero velocity, but also for the combination of big speed and the position near the rest point. In the other hand, for action $\tau = 5$ the state-action space doesn't need to be so divided since this action is more executed to get the necessary velocity to reach the upward position but doesn't require a big precision. The plot on the right of figure 5.2 represents the entire state-space at the end of the training, that is all actions are represented, each point represents a part located in the mean values of the state.

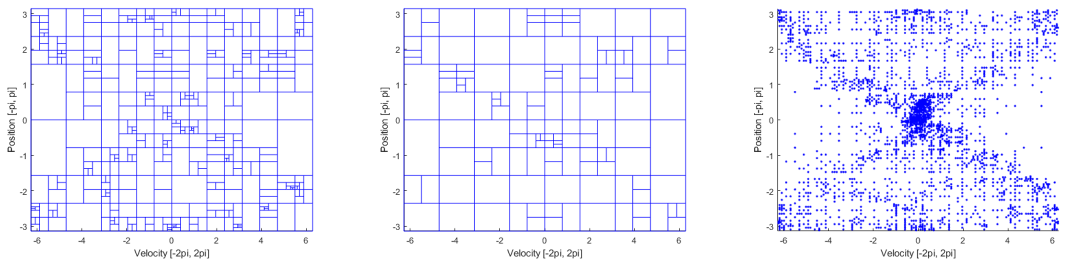

Figure 5.2: Representation of the state-space for states containing the action $\tau = 0$ (left), representation of the state-space for states containing the action $\tau = 5$ (center) and representation of the entire state-action space for any action (right)

5.2 SARSA Results

SARSA requires a longer training time than Q-Learning to converge, so this algorithm was trained during 3000 episodes which is equivalent to 150000 seconds or 41 hours and 40 minutes. The process performed was similar to Q-Learning to calculate the accumulated reward over time in order to perceive the learning done by the agent. In this case the test episode was only executed every 10 training episodes, since as the algorithm learns more slowly there are no significant changes at each episode, it also allowed to accelerate the total training process. Figure 5.3 shows the evolution of the accumulated reward over time.

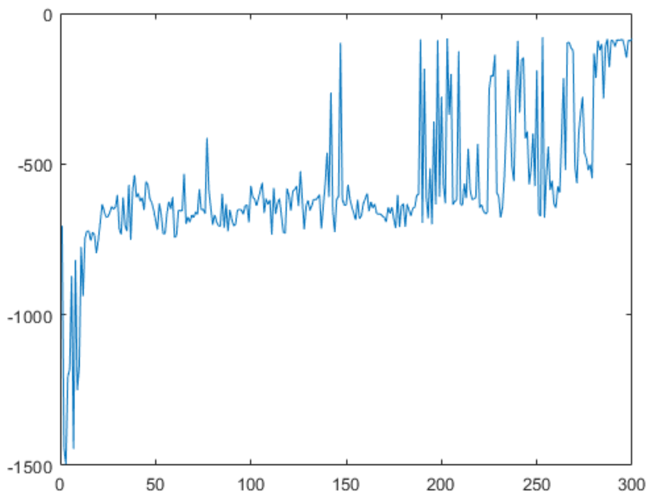

Figure 5.3: Plot of the accumulated reward for 3000 episodes using SARSA algorithm

We can see from the graph that it does not have a good stable convergence for a long time, the training process was terminated as soon as convergence to a sufficiently good value was achieved. The state-action space plot in figure 7.1 shows that this division is not done in the best way possible. It can be seen that there is a greater partitioning of states near zero speed and π and $-\pi$ position. This is not necessary since these states represent the pendulum when it is at rest position, instead, the division should be greater close to zero velocity and zero position. These problems could be solved by improving the hyper-parameters of the algorithm.

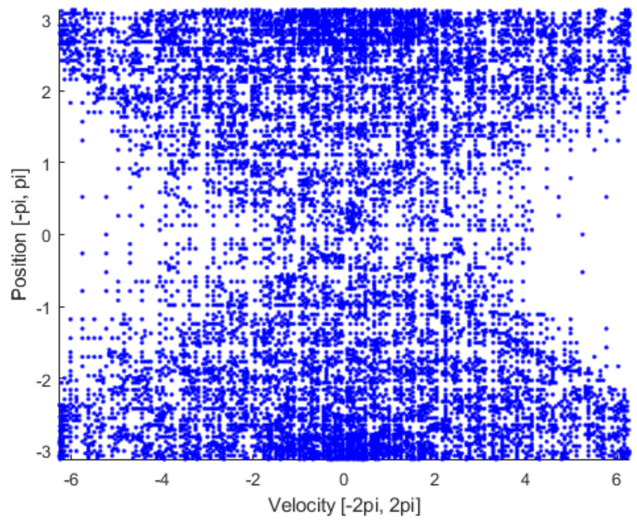

Figure 5.4: Representation of the entire state-action space for any action

Nevertheless, the method was able to control the pendulum in the desired way and without much oscillation when balanced in the upright position.

5.3 Actor-Critic Results

The Actor-critic algorithm is constituted by two functions, one to represent the actor, responsible for choosing the actions, and the critic, responsible for criticizing the actions taken by the actor and in this way improving the policy learned. This makes the number of hyper-parameters selected almost double and as such it is necessary to tune more parameters which makes the problem more complex. That said, in this implementation it was not possible to find the values that lead to the convergence of the policy learned.

Chapter 6

Discussion

The main difference between SARSA and Q-Learning is that SARSA follows the policy it is learning (*on-policy*) while Q-Learning can follow any policy that meets the convergence requirements (*off-policy*). Q-Learning has greater per-sample variance than SARSA which can result in convergence problems. SARSA is a more conservative method, i.e. if there is a risk of a large negative reward near the optimal path, Q-Learning would tend to trigger that reward when exploring while SARSA tends to avoid dangerous optimal paths and only learns more gradually to use the reward when the parameters of exploration are low. This presents an advantage for SARSA in training cases that are not simulated, but instead are trained in the real world, as more conservative learning will avoid greater risks to the equipment that may be costly.

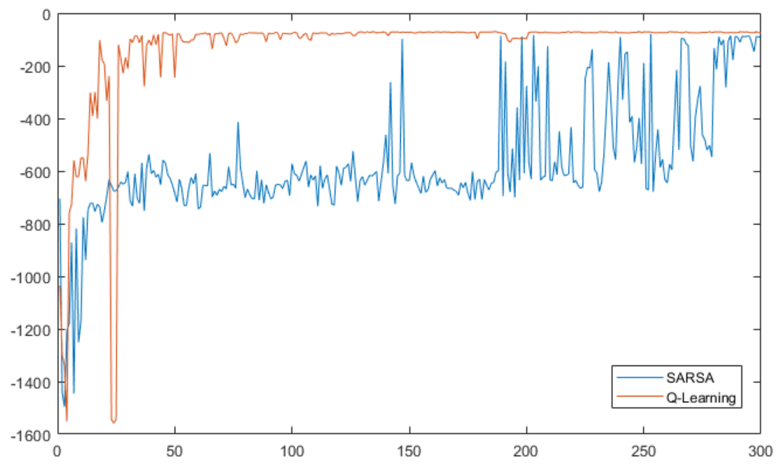

Figure 6.1: Comparison between Q-Learning and SARSA of the accumulated reward after 3000 episodes

Figure 6.1 shows a comparison between the accumulated reward of SARSA and Q-Learning after 3000 episodes.It is clear that Q-Learning reaches convergence much faster than SARSA and still converges to a higher value (higher reward), which means that the policy learned by Q-Learning is closer to the optimal policy than the policy learned by SARSA.

As mentioned before, training the RL agent requires a lot of time and computing power, which is inversely proportional to the training time. When applying the Q-Learning algorithm in the pendulum environment, the training time was reasonable since only 300 episodes were required to achieve convergence. On the other hand, SARSA was a much slower method because it had to be trained longer and had a more complex algorithm than the previous one.In order to choose the correct hyper parameters the training was run multiple times which led to a total of several days of training until the desired convergence was achieved. In actor-critic it was not possible to select the desired parameters leading to convergence in the desired time. Having a larger computational capacity would allow training to be performed faster which would help find the solution in a shorter time.

Chapter 7

Conclusion

From this work we can conclude the following points:

- Reinforcement Learning can be applied in many different cases from continuous or discrete problems. It is possible to obtain in both cases satisfactory results that allow the control of the agent in the desired way;
- For each problem it is necessary to choose the algorithm that best performs the desired task and adapt the hyper parameters in order to achieve convergence. It is also necessary to choose a reward function for each different RL application.
- In real world applications like robotics, the number of states is very large or infinite in case it is continuous, in this case it is necessary to use some method to deal with this problem, in this project VRFA was used. However there are other methods that don't require the state-action space to be discrete.
- RL may take a long time to be trained, in some cases it is necessary to wait days to achieve convergence. As it is necessary to choose the right parameters, the training may have to be done multiple times, which will further increase the total time to achieve the desired result.
- SARSA is a more conservative method having a more stable behavior throughout the training while Q-Learning may have a more random and unpredictable behavior.
- Q-Learning achieves convergence faster than SARSA and therefore may be favorable in scenarios that are intended to simplify the problem and there are no risks due to unpredictable behavior.
- Variable resolution is a simple method for overcoming the continuous state and action space problem, and it shows good results in this implementation.

7.1 Future Work

In the implementation of the RL methods in this project, VR was used to approximate continuous functions, however, there are other methods that could have been used for function approximation such as neural networks. RL still presents some problems as described in [1], which means that a lot of research can still be done in this area as a goal to achieve increasingly intelligent and autonomous systems. In recent years with the advancement of computational capacity the interest for machine learning has increased. RL, being an area of machine learning, was no different, the increase in the number of articles published in recent years shows an increasing interest in RL.

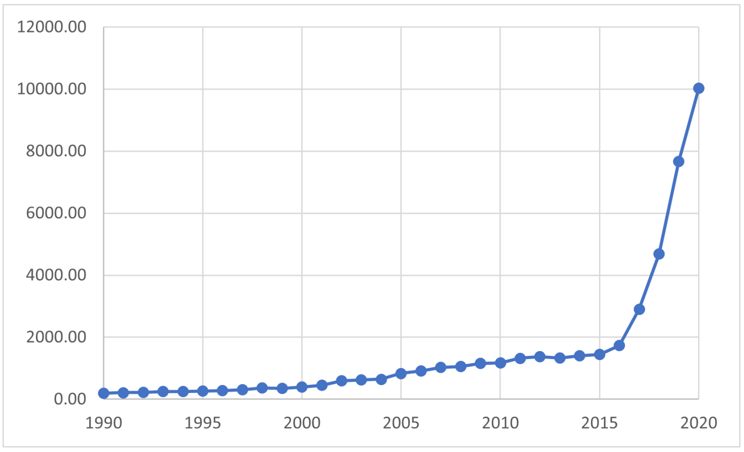

Figure 7.1: Evolution of the number of published papers about RL published over the years

Bibliography

[1] G. Dulac-Arnold, D. Mankowitz, and T. Hester, "Challenges of real-world reinforcement learning," 2019.

[2] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning.* Cambridge, MA, USA: MIT Press, 1st ed., 1998.

[3] A. W. Moore and C. G. Atkeson, "The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces," *Machine Learning*, vol. 21, pp. 199–233, Dec 1995.

[4] R. Munos and A. Moore, "Variable resolution discretization in optimal control," *Machine Learning*, vol. 49, pp. 291–323, Nov 2002.

[5] A. G. Agostini, "Q-learning with a degenerate function approximation," 2011.

[6] A. Agostini and E. Celaya, "Online reinforcement learning using a probability density estimation," *Neural Computation*, vol. 29, pp. 1–27, 10 2016.

[7] H. Van Hasselt and M. Wiering, "Reinforcement learning in continuous action spaces," pp. 272 – 279, 05 2007.