Experiments with Solving Mountain Car Problem Using State Discretization and Q-Learning

Amelia Bădică[1], Costin Bădică[1](✉), Mirjana Ivanović[2], and Doina Logofătu[3]

[1] University of Craiova, Craiova, Romania
costin.badica@edu.ucv.ro
[2] University of Novi Sad, Novi Sad, Serbia
[3] Frankfurt University of Applied Sciences, Frankfurt am Main, Germany

Abstract. The aim of this paper is to explore the model of the Mountain Car Problem. We provide insight into the physics behind the model. We present some experimental results obtained by numerically simulating the model. We also propose a reinforcement learning approach for deriving an optimal control policy combining model discretization and Q-learning.

Keywords: Dynamic system · Mountain car problem · Q-learning · SARSA

1 Introduction

The Mountain Car Problem is a standard benchmark for experimenting with reinforcement learning algorithms. Our aim is to provide new theoretical and experimental insights into the Mountain Car Problem. Although this problem has a tradition of more than three decades in the literature of reinforcement learning, we could not find in the literature a complete and physically accurate explanation of its dynamic model.

The main contributions of our work are outlined as follows: i) accurate presentation of the physics details of developing the dynamic model of the Mountain Car Problem; ii) new insights into random walk experimental results obtained for the Mountain Car Problem; iii) detailed investigation and comparison of standard Q-learning and SARSA algorithms for solving the digitized version of the Mountain Car Problem.

We start in Sect. 2 with a brief overview of related works on developments of the Mountain Car model in the reinforcement learning literature. In the first half of Sect. 3 we present the detailed development of the Mountain Car model starting from the first principles from physics. In the second half of Sect. 3 we present new insights into experimental results that we obtained by performing a sequence of random walks on the Mountain Car Model. Section 4 is dedicated to

https://doi.org/10.1007/978-3-031-21743-2_12

introduce the Q-learning and SARSA algorithms, our proposed implementations as well as a discussion of the experimental results obtained. For the experiments we have used a digitized (i.e. finite state) version of the Mountain Car Problem. The last section concludes and points to future works.

2 Related Works

Although the Mountain Car Problem has been proposed more than 30 years ago as a benchmark problem for optimal control and reinforcement learning algorithms, we could not find in the scientific literature a complete and physically accurate explanation of its dynamic model.

The Mountain Car Problem was firstly proposed in Moore's PhD thesis [3, Chapter 4.3] and was referenced in one of his early papers [4]. In these works, the hill-shaped driving track of the car is described by a polynomial function that defines the height of the hill depending on the horizontal vehicle location. Moreover, the model is using a continuous model for the speed of the car depending on the pedal height.

The Moore's proposed model is referenced in the experimental part of [5] as benchmark for the reinforcement learning with replacing eligibility traces approach. However, the details of the Mountain Car problem considered in this work are not given in [5].

Another dynamic model of the Mountain Car Problem is presented in [6, Chapter 4.5.2]. This model is closer to the model that we have used in our paper. The shape of the hill is the same, but the values of the other parameters differ. Moreover, although some physics details are included in the model presentation, its complete physics explanation is missing.

OpenAI Gym is a software framework for bench-marking reinforcement learning algorithms [1]. On one hand it includes a number of dynamic models, and on the other hand it allows definition of new models. Those models are defined as environment model. The idea is that reinforcement learning algorithms are encapsulated into agents that interact with the environment through a standard programming interface. OpenAI Gym includes two Mountain Car models: "Mountain Car v0" that is similar to the model that we used in our paper and "Mountain Car Continuous v0" that differs from our model by the fact that the action space is continuous, rather than discrete.

Mountain Car Problem was used in many online tutorials that introduce the practical programming of reinforcement learning algorithms. One such a video tutorial is [8]. Here the author presents an approach for solving the Mountain Car Problem included in OpenAI Gym using Q-learning and state discretization. We have used it as a base implementation for our experiments. Nevertheless, we extended this implementation in many directions (see Sect. 4).

Standard reference [7, Chapter 10.1] also includes a model of the Mountain Car Problem that is similar to the one used in our paper. However, the details of the physics of the model are missing. The model is introduced as a mathematical object, suitable for experimenting with various reinforcement learning algorithms.

3 Modeling the Mountain Car Problem

3.1 Physics of the Mountain Car Problem

The Mountain Car problem assumes an autonomous car driving on a one-dimensional track that follows a mountain range described by the equation:

$$y = \sin \omega x \quad \text{for} \quad x \in [-\frac{3\pi}{2\omega}, \frac{\pi}{2\omega}] \tag{1}$$

Note that Eq. (1) describes a function that spans a full cycle of "sinus" function thus modeling a valley between a left and right hill. The car is supposed to start its driving episode at initial position x_0 somewhere in the valley in the vicinity of $-\frac{\pi}{2\omega}$ and its goal is to climb the rightmost hill. The goal can be described as the car reaching a location $x \geq x_g$, where the "goal position" x_g is a real value in the left vicinity of $\frac{\pi}{2\omega}$.

The car engine is controlled by a thruster that can either thrust left or right with equal force or no thrust at all. The force of the thruster is not strong enough to defeat gravity and accelerate up the slope to the top of the right hill. The solution assumes the movement of the car in the opposite direction to the goal (i.e. to the left) to accumulate enough inertia to defeat the slope, even if slowing down up to the top of the hill.

The standard physical environment of the Mountain Car is shown in Fig. 1. In this case $\omega = 3.0$ and the curve describing the mountain range $y = \sin 3x$ is defined on interval $[-\frac{\pi}{2}, \frac{\pi}{6}] \approx [-1.57, 0.52]$. Note that in the standard Mountain Car model, as defined in [7, Chapter 10.1] and [1], this interval is set to $[-1.2, 0.6]$. If the position of the car reaches a value below the lower bound then the location is clipped to the lower bound and the car velocity is reset to 0. The goal of the car is to reach the top of the hill at a location $x \geq 0.5$. The initial position x_0 is defined in the vicinity of $-\frac{\pi}{6} \approx -0.52$, while the initial velocity of the car is set to 0. The current implementation from OpenAI Gym assumes that x_0 is a uniformly distributed random number in interval $[-0.6, -0.4]$.

The dynamics of the car is described with the following equation in vector format:

$$m \cdot \overrightarrow{a_t} = \overrightarrow{F_t} + \overrightarrow{G} + \overrightarrow{F_f} \tag{2}$$

Projecting Eq. (2) on the movement direction we obtain:

$$m \cdot a_t \cos\theta = F_t \cdot \cos\theta - m \cdot g \cdot \sin\theta - k_f \cdot v_t \cdot \cos\theta \tag{3}$$

Note that g denotes the gravitational constant, m is the mass of the car and k_f represents the friction coefficient. F_t is the force of the thruster and it represents the input that controls the dynamics of the model. Simplifying Eq. (3) by $m \cdot \cos\theta$ we obtain:

$$a_t = \frac{F_t}{m} - g \cdot \tan\theta - \frac{k_f}{m} \cdot v_t \tag{4}$$

Note that $\tan\theta$ is the slope of the tangent to the curve representing the mountain range. If $y = f(x)$ is the equation of this curve, it follows that $\tan\theta =$

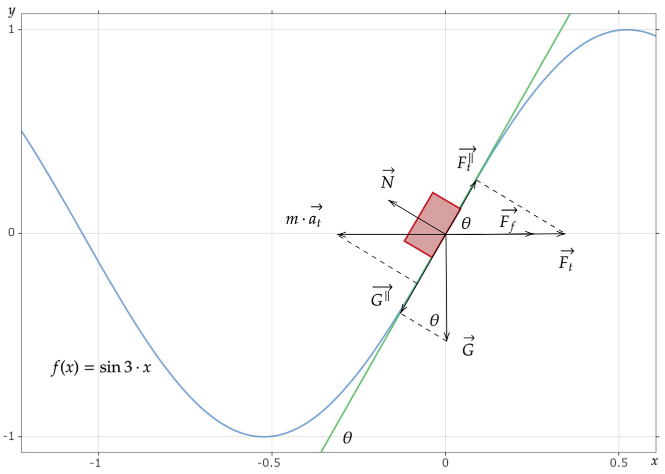

Fig. 1. Mountain car physical environment.

$\frac{df(x)}{dx}(x_t) = f'(x_t) = \omega \cdot \cos \omega x_t$. Substituting $v_t = \dot{x}_t$ and $a_t = \ddot{x}_t$, Eq. (4) becomes:

$$\ddot{x}_t = \frac{F_t}{m} - g \cdot \omega \cdot \cos \omega x_t - \frac{k_f}{m} \cdot \dot{x}_t \tag{5}$$

Differential Eq. (5) gives the dynamics of the car under the control of the thrust force. The system state is captured by the pair $(x_t, v_t) = (x_t, \dot{x}_t)$ containing the location and the velocity of the car in horizontal direction. The initial state is $(x_0, 0)$ where x_0 is the initial location of the car in the vicinity of $-\frac{\pi}{2\omega}$. This equation is usually presented in discretized form (6) with a time step $h > 0$ using Euler's numerical integration method.

$$\begin{aligned} \dot{x}_{t+h} &= \dot{x}_t + h \cdot (\tfrac{F_t}{m} - g \cdot \omega \cdot \cos \omega x_t - \tfrac{k_f}{m} \cdot \dot{x}_t) \\ x_{t+h} &= x_t + h \cdot \dot{x}_t \end{aligned} \tag{6}$$

Note that Eqs. (6) still represent a discrete time continuous model, as the time is discrete but the states are continuous.

The aim of the Mountain Car problem is to reach the goal location on the top of the right hill. Therefore each action of the car is rewarded accordingly. In the weaker version of the problem it is assumed that the car does not know the goal. So the only way to perceive the goal is when it was reached. Therefore the car is rewarded a small negative value (usually -1) for each reached state that does not achieve the goal.

3.2 Model Exploration Using Random Walk and Numerical Simulation

In this section we provide an experimental investigation of the Mountain Car model as defined by Eq. (6) using numerical simulation. We have performed experiments using our own hand-crafted Mountain Car model. We chose this approach after noticing that the simulation using our model was faster than the one provided by OpenAI Gym.

We set the Mountain Car model parameters to obtain a mathematical model similar to that included in OpenAI Gym [1] and standard reference ([7, Chapter 10.1]):

$$\begin{array}{lll} \omega = 3.0 & m = 1000.0 & g = 0.0025/3.0 \\ k_f = 0 & h = 1.0 & x_g = 0.5 \\ F_t \in \{-1, 0, 1\} & x_t \in [-1.2, 0.6] & v_t \in [-0.07, 0.07] \end{array} \tag{7}$$

Note the bounds of position and velocity in Eqs. (6). While the bounds of location are pretty obvious, taking into account Eq. (1), the bounds of velocity are far from obvious. Therefore we decided to evaluate them experimentally by performing a series of random walks in the Mountain Car model given by Eqs. (6). Each random walk represents an episode of the simulation, in which the car starts from the initial state and drives until reaching the goal state.

The simulation is described by Algorithm 1. This algorithms takes the Mountain Car environment Env and the total number $ITMAX$ of episodes and returns the bounds of position and velocity $(pmin, pmax, vmin, vmax)$, as well as the vector $Episodes$ containing the length of each episode. We assume that the API of the Mountain Car environment supports the following methods:

– *reset()* that resets the environment to its initial state as a random value in the vicinity of $-\frac{\pi}{2\omega}$. This method returns the state and the value of *done* that is reset to *False*.
– *step(action)* that takes an action from the action space and determines the next state of the environment. If the goal is reached, flag *done* is set to $True$.
– *random_action()* that returns an action of the car by sampling the action space.
– Note that methods *reset()* and *step(action)* return also the reward, but this is not used in Algorithm 1.

We have implemented Algorithm 1 in Python 3.7.3 on an x64-based PC with a 2 cores/4 threads Intel i7-5500U CPU at 2.40 GHz and running Windows 10.

The total simulation time for 2000 episodes was 1057.793 s, i.e. approximately 18 min. Episode lengths were distributed in the interval [865, 324776] with an average value of 42209.101 and standard deviation of 37989.213.

Figure 2 presents the 80 bins histogram of episode lengths. It is interesting to observe that the shape of this histogram suggests a log-normal distribution of the episode length [2]. This observation is also supported by the high value obtained for the standard deviation that is typical for log-normal distribution. However, observe that the shape is clipped and highly skewed to the left side, as there is a strictly positive minimum length of an episode.

The minimum and maximum bounds of velocity that were recorded during this simulation were $vel\_min = -0.06587993$ and $vel\_max = 0.05977063$, thus

Algorithm 1. $MountainCarRandomWalk(Env, ITMAX)$ algorithm for determining the position and velocity bounds in Mountain Car problem using a series of random walks. The algorithms returns also the vector of lengths of each driving episode from initial state to the goal state.

```
Require: $Env$. The environment model of the Mountain Car problem.
         $ITMAX$ The number of episodes.
Ensure: $Episodes$. Vector of lengths of each episode.
        $pmin, pmax, vmin, vmax$. Position and velocity bounds.
 1: $Episodes \leftarrow []$
 2: $(pmin, pmax, vmin, vmax) \leftarrow (0, 0, 0, 0)$
 3: for $k = 1, ITMAX$ do
 4:     $(p, v, done) \leftarrow Env.reset()$
 5:     $count \leftarrow 0$
 6:     while $\neg done$ do
 7:         $action \leftarrow Env.random\_action()$
 8:         $(p, v, done) \leftarrow Env.step(action)$
 9:         $(pmin, pmax) \leftarrow (\min(pmin, p), \max(pmax, p))$
10:         $(vmin, vmax) \leftarrow (\min(vmin, p), \max(vmax, p))$
11:         $count \leftarrow count + 1$
12:         $Episodes.append(count)$
13:     end while
14: end for
15: return $Episodes, pmin, pmax, vmin, vmax$
```

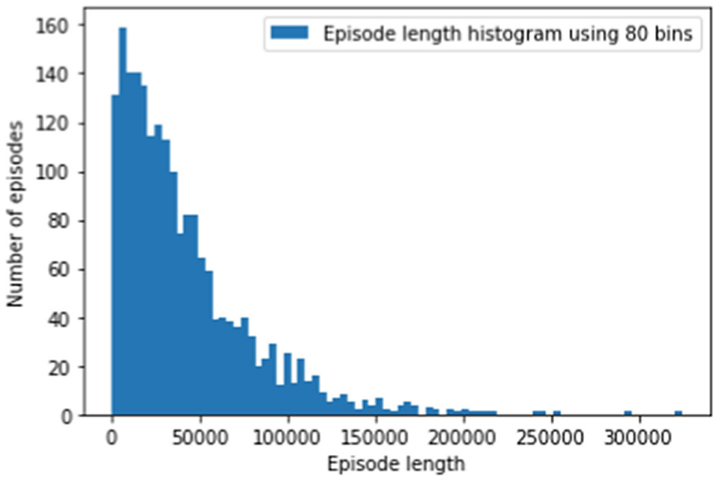

Fig. 2. The 80 bins histogram of episode length using data from 2000 episodes.

confirming the bounding box $[-0.07, 0.07]$ set for the velocity in the Mountain Car model. The bounds recorded for the position were $pos\_min = -1.2$ and $pos\_max = 0.52564615$, again confirming expectations.

4 Optimal Control Using State Discretization and Q-Learning

4.1 Q-Learning and SARSA Algorithms

We model our autonomous car as an intelligent agent that is able to perceive its environment through sensors and act upon that environment through actuators, as shown in Fig. 3. In particular we are interested to employ the technique of reinforcement learning to let the car "learn" an optimal acting policy through self-driving episodes. Reinforcement learning aims to optimize agent action based on action feedback as punishments and/or rewards. An agent percept in this model will consist of a pair $state, reward$. The $state$ represents the agent observation of the environment state, while $reward$ is a real value that locally estimates the "goodness" of that state from the agent perspective.

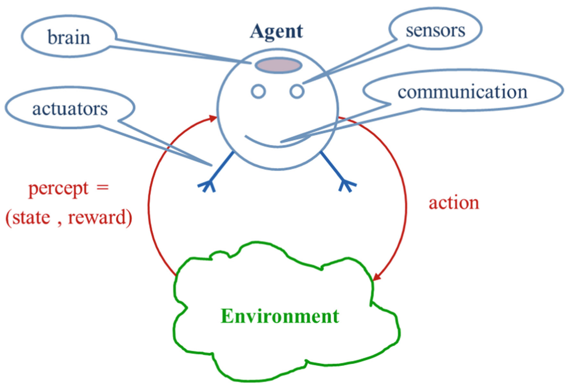

Fig. 3. Elements of an intelligent agent.

The agent strategy specifies what the agent must do in each observed environment state. If agent takes action a in state s then environment will transit in state s'. We assume that the environment is Markovian, i.e. its next state depends only on its current state and agent action. If E denotes the set of environment states and A denotes the set of agent actions, the strategy is called *policy* and it is defined as function π mapping each state to an action:

$$\pi : E \to A \tag{8}$$

Let us assume that the reward of the agent by taking action a in state s is denoted as $R(s, a)$ and that the initial state of the environment is s_0. An agent following a given policy π will generate the following agent run:

$$r = s_0 \xrightarrow{a_0=\pi(s_0)} s_1 \xrightarrow{a_1=\pi(s_1)} \ldots \xrightarrow{a_n=\pi(s_n)} s_{n+1} \ldots \tag{9}$$

The utility perceived by the agent for the run $[s_0, a_0, s_1, a_1, s_2, a_2 \ldots]$ accumulates all the rewards that were perceived by the agent in each visited state. We assume a potentially infinite horizon of the agent, i.e. the length of the agent run is unbounded. As there are goal states, the agent run is terminated when a goal state is reached. In order to guarantee the convergence of the agent utility in the presence of unbounded runs, we assume a discounted additive utility of the agent, defined as follows:

$$U^\pi(s = s_0) = U([s_0, a_0, s_1, a_1, \ldots, s_n, a_n, \ldots]) = \sum_{n \geq 0} \gamma^n \cdot R(s_n, a_n) \tag{10}$$

Equation (10) suggests the recursive definition of $U^\pi(s)$ for each transition $s \xrightarrow{a=\pi(s)} s'$ given by Eq. (11).

$$U^\pi(s) = R(s, a) + \gamma \cdot U^\pi(s') \tag{11}$$

Assuming that the discount factor is $0 < \gamma < 1$ and the set of rewards $\{R(s, a)\}_{s \in E, a \in A}$ is bounded, the utility of the agent is well defined by Eq. (10).

The aim of reinforcement learning is to determine the optimal policy π that maximizes the value of $U^\pi(s)$ for all states s. The corresponding maximum value $U(s)$ represents the utility of the agent in state s by running its optimal policy.

$$U(s) = \max_{\pi(s)=a \in A} U^\pi(s) \tag{12}$$

Maximizing for π in Eq. (11) we obtain recurrence (13) for agent utility. This is a model-based equation, as it explicitly uses the transition model $s \xrightarrow{a} s'$ of the environment. In a realistic setting, this model is not available to the agent.

$$U(s) = \max_{a \in A, s \xrightarrow{a} s'} (R(s, a) + \gamma \cdot U(s')) \tag{13}$$

Q-learning aims to determine the values $Q(s, a)$ that estimate the utility of the agent when taking action a in environment state s. Then the optimal policy is defined as:

$$\pi(s) = \operatorname*{argmax}_{a \in A} Q(s, a) \tag{14}$$

We can derive a model-based recursive equation defining Q values, following Eq. (13). However, as we assume that the transition model of the environment is not available, for the computation of Q values we shall use temporal difference learning. If the currently observed state is s then the agent will choose a certain action a using a given learning strategy. This produces reward $R(s, a)$ and next state s'. Then the value $Q(s,a)$ is updated using Eq. (15). Hyper-parameter α (learning rate) can be set to a constant small positive value or it can be computed as the harmonic sequence $\alpha_n = A/(B + C \cdot n)$ (A, B, C are positive constants) depending on the number n of times action a was taken in state s. Note that for $C = 0$ the learning factor is constant $\alpha = A/B$.

$$\begin{aligned} Q(s,a) &= (1-\alpha) \cdot Q_{OLD}(s,a) + \alpha \cdot Q_{NEW}(s,a) \\ &= Q_{OLD}(s,a) + \alpha \cdot (Q_{NEW}(s,a) - Q_{OLD}(s,a)) \end{aligned} \tag{15}$$

We consider two related algorithms that follow Eq. (15) for updating Q values for each observed transition:

– Q-learning that computes Q_{NEW} using Eq. (16).

$$Q_{NEW}(s,a) = R(s,a) + \gamma \cdot \max_{a' \in A} Q(s', a') \tag{16}$$

– SARSA that computes Q_{NEW} using Eq. (17). Here a' is the agent action in state s' determined using the learning strategy.

$$Q_{NEW}(s,a) = R(s,a) + \gamma \cdot Q(s', a') \tag{17}$$

The learning algorithm follows an epsilon-Greedy strategy that mixes exploration and exploitation. Exploration favors random actions, while exploitation favors the locally best actions based on current Q value. The choice between the two possibilities is determined by a Bernoulli experiment with a probability $0 \leq \varepsilon \leq 1$. As the learning proceeds, ε is decreased so exploitation (i.e. Greedy choice) is favored in the limit.

4.2 State Discretization

State discretization strategy follows the idea from [8]. Its effect is to transform our discrete time continuous model (6) into a finite state model. The size of the resulting model is controlled by the discretization step.

Let us consider a state variable with domain $[a, b)$ and a natural number $n \in \mathbb{N}$ representing the number of states of the finite state representation. The discretization step is computed as $h = (b - a)/n$. Each value $x \in [a, b)$ is mapped to a natural number $digi(x) \in \{0, 1, \ldots, n-1\}$ according to Eq. (18).

$$digi(x) = k \in \{0, 1, \ldots, n-1\} \iff x \in I_k = [a + k \cdot h, a + (k+1) \cdot h) \tag{18}$$

Discretization in Q-learning is useful to control the size of the table $Q[s, a]$. In the Mountain Car problem there are two state variables: position and velocity. If the finite state representation uses n_p positions and n_v velocities then the size of Q table is $3 \cdot n_p \cdot n_v$.

State discretization is used as follows. Before performing the update of the Q values, the observed state is digitized. Then the action determination according to the learning strategy and the update itself will use the digitized value of the state. This basically means that for all states $s \in I_k$ for which $digi(s) = k$ the algorithm will determine the same value in the Q table.

4.3 Experimental Results

We have implemented the Q-learning and SARSA algorithms presented in this section. The starting point of our implementation is [8]. We have extended the implementation in many directions, as follows:

- We have added the SARSA algorithm, not present in [8].
- We have added our own implementation of the Mountain Car model, following the complete physical model developed in Sect. 3. Using our model, the learning runs considerably faster than using the OpenAI Gym provided model.
- We have added two additional visualizations of the optimal policy and of the model exploration during learning process.
- We have added a procedure for evaluating the policy computed by the algorithms.

A training session using either Q-learning or SARSA involves performing a sequence of N_E episodes. Each episode is a sequence of agent steps, following the learning algorithm. An episode can finish either by reaching a maximum preset number of steps N_{LE} or by reaching the goal state. Differently from [8], we imposed a significantly larger upper bound of each episode of the learning session. This decision was taken based on the observation that more than 50% of the episodes of a series of random walks actually reached the goal state after a reasonable number of steps (below 30000).

The probability ε of selecting exploration versus exploitation starts from an initial value ε_s and it is decreased at a constant rate after each episode until it reaches the minimum value $\varepsilon_f < \varepsilon_s$. The rate of decrease is $\Delta\varepsilon = (\varepsilon_s - \varepsilon_f)/N_E$.

The policy determined by the learning algorithm was evaluated by running a fixed number of episodes N_{TE}, each with a maximum number of steps N_{TS}. Then we determined the percent of episodes that reached the goal state *acc*, as well as their average length *len*. For each episode we recorded the value of the accumulated rewards. Then we plotted the moving average of each N_m consecutive accumulated rewards.

The description and the values of the parameters that we have used in our experiments are summarized below:

- $N_E = 4000$. Number of learning episodes.
- $N_{LE} = 30000$. Maximum number of steps of a learning episode.
- $\varepsilon_s = 0.6$. $\varepsilon_f = 0.01$. Maximum (initial) and minimum (final) value of the probability of selecting exploration versus exploitation during learning.
- $N_{TE} = 1000$. Maximum number of episodes for testing the policy computed by the learning algorithm.
- $N_{TS} = 500$. Maximum number of steps of running each episode for testing the policy computed by the learning algorithm.
- $n_p, n_v \in \{10, 20, 30, 40\}$. Number of discretization steps for position and velocity.
- $\gamma = 0.99$. Discount factor.

- $A = 1.0$, $B = 10.0$, $C = 0.01$. Parameters of the harmonic learning factor α_n.
- $N_m = 50$. Number of rewards used to compute and plot the moving average of the accumulated rewards per each learning episode.

Our experimental results are presented in Tables 1 and 2. Each table entry is defined for a specific value of the discretization parameters n_l and n_v. Topmost halves of each table refer to Q-learning, while bottom half of each table refers to SARSA.

Table 1 presents the accuracy (percentage of successful test episodes) and the average length of the car trajectory (from successful episodes) generated by the policy computed by the algorithm. First of all observe that all the 32 algorithm runs produced an accuracy higher than 73%, only 3 runs produced an accuracy below 80%, 6 runs produced an accuracy below 90% and 24 runs produced an excellent accuracy higher than 95%. In 9 runs the accuracy was actually 100%.

The average of the length of the shortest trajectory was 140.99 and it was obtained by SARSA algorithm for $n_p = 20$ and $n_v = 30$ (shown in bold underlined in Table 1). Combined with the excellent accuracy 99.90%, clearly this case can be considered as providing the best result among all the cases. Note also that close and consistent results for $n_p = 20$ and $n_v = 30$ were also obtained by the Q-learning algorithm.

Table 1. Accuracy (*acc*) and average solution length (*len*) obtained with Q-learning (top) and SARSA (bottom) algorithms.

n_p/n_v	10	20	30	40
10	80.10%, 169.75	96.20%, 150.54	99.20%, 154.68	100%, 243.46
20	90.10%, 172.05	100%, 157.97	99.60%, 147.26	100%, 160.73
30	94.40%, 158.01	98.40%, 179.66	100%, 171.51	100%, 145.14
40	77.30%, 157.77	97.80%, 150.23	99.30%, 147.64	99.30%, 195.60
10	81.00%, 174.16	96.70%, 154.68	99.30%, 155.96	100%, 151.28
20	86.00%, 168.38	100%, 156.47	99.90%, **<u>140.99</u>**	73.50%, 154.38
30	95.50%, 165.31	98.50%, 157.80	100%, 168.21	100%, 149.57
40	76.30%, 208.01	97.90%, 147.78	99.70%, 176.92	95.10%, 184.15

Table 2 presents the total number of iterations performed by each experimental case during learning. Note that the values were different as we did each training episode until the goal state was reached, by setting a high value for the maximum number of iterations per episode (to force closing really long-running training episodes). The shortest number of iterations was done for training the Q-learning algorithm for $n_p = 20$ and $n_v = 30$. Moreover, in this case the accuracy of the solution was 100%. However, as regarding its optimality, The average of the length of the shortest trajectory was 157.97 (see cell in bold underline in Table 2), i.e. higher than the best case 140.99.

In our opinion, the differences between the results obtained by Q-learning and SARSA were not significant. Therefore, we also checked if there is some statistical similarity in the results produced by the algorithms. We found a positive correlation (0.983) between the number of iterations produced by each algorithm, as well as a positive correlation 0.686 between the accuracy of the algorithms. However, the average lengths of the trajectories generated by the solution policies were uncorrelated (0.038).

We also checked if there are any correlations between the results within each algorithm. Among all the situation, we found interesting the negative correlation between the number of iterations and the accuracy of the solution in both algorithms (-0.46 for Q-learning and -0.33 for SARSA). One possible explanation could maybe that the higher number of iterations indicates some difficulties in the learning process, difficulties that are also explained by the lower accuracy of the solution.

As part of our numerical simulation program for learning the optimal policy and testing the solution, we have also developed three visualizations for presenting: i) the convergence of the accumulated rewards during learning; ii) the optimal policy; iii) the exploration done by the car agent during the learning process.

Table 2. Number of iterations performed by Q-learning (top) and SARSA (bottom) algorithms.

n_p/n_v	10	20	30	40
10	1332618	1161777	1245702	1285570
20	1306056	**1156701**	1178533	1244548
30	1474829	1236062	1272904	1373627
40	1563608	1347271	1394002	1510739
10	1346977	1182048	1227957	1323528
20	1347295	1176168	1215110	1291964
30	1442344	1258570	1312203	1411848
40	1585404	1358674	1437671	1548015

Figure 4 presents the accumulated reward convergence process for the case that we considered the best: SARSA with $n_p = 20$ and $n_v = 30$.

Figure 5 presents the policy computed by SARSA algorithm with $n_p = 20$ and $n_v = 30$. The color codes represents actions performed by the policy in each state, while small rectangle patches represent the digitized states of the Mountain Car model.

Figure 6 is an example plot showing how the car agent is exploring the environment by taking push right moves. Similar visualizations were produced for no push, push left as well as total number of actions taken by the agent car in each state of the environment.

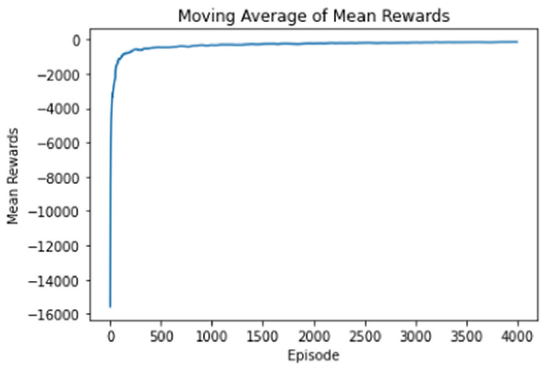

Fig. 4. Convergence of accumulated rewards during learning in SARSA with $n_p = 20$ and $n_v = 30$, plotted as the moving average of results obtained for $N_m = 50$ consecutive episodes.

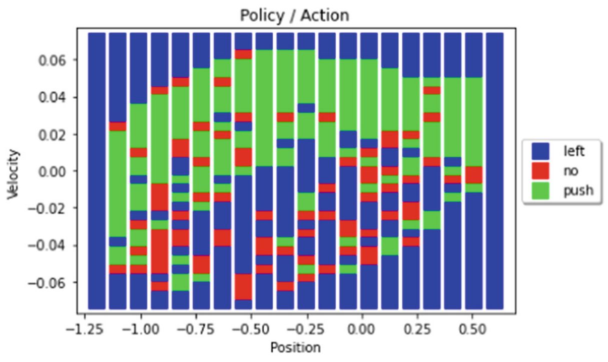

Fig. 5. Policy computed by SARSA with $n_p = 20$ and $n_v = 30$.

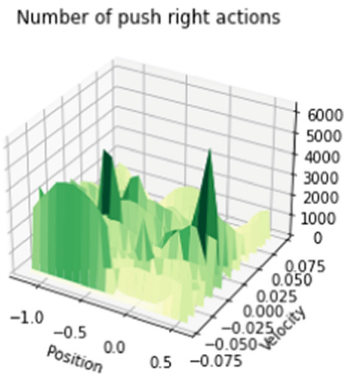

Fig. 6. Number of push right actions per state during SARSA learning for $n_p = 20$ and $n_v = 30$.

5 Conclusions and Future Work

In this paper we provided a theoretical and experimental analysis of the Mountain Car Problem. We have explained the physics of the dynamic model of the Mountain Car. We have used the model to perform a sequence of random walks in order to better understand the problem. Then we have shown how the problem can be solved by combining state discretization with Q-learning and SARSA algorithms. We provide detailed experimental results regarding the computational effort incurred by these algorithms, as well as the accuracy and optimality of the solutions produced by the algorithms. As future work, this research can be extended by: i) incorporating different reinforcement learning algorithms for the Mountain Car problem; ii) providing similar analysis for other standard benchmark problems in reinforcement learning.

References

1. Brockman, G., et al.: Openai gym. arXiv preprint arXiv:1606.01540 (2016)
2. Limpert, E., Stahel, W.A., Abbt, M.: Log-normal Distributions across the sciences: keys and clues: on the charms of statistics, and how mechanical models resembling gambling machines offer a link to a handy way to characterize log-normal distributions, which can provide deeper insight into variability and probability-normal or log-normal: that is the question. BioScience **51**(5), 341–352 (2001). https://doi.org/10.1641/0006-3568(2001)051[0341:LNDATS]2.0.CO;2
3. Moore, A.W.: Efficient memory-based learning for robot control. Ph.D. thesis, Computer Laboratory, University of Cambridge, Cambridge CB3 0FD, United Kingdom (October 1990)
4. Moore, A.W.: Variable resolution dynamic programming: efficiently learning action maps in multivariate real-valued state-spaces. In: Birnbaum, L.A., Collins, G.C. (eds.) Machine Learning Proceedings 1991, pp. 333–337. Morgan Kaufmann, San Francisco (1991). https://doi.org/10.1016/B978-1-55860-200-7.50069-6
5. Singh, S.P., Sutton, R.S.: Reinforcement learning with replacing eligibility traces. Mach. Learn. **22**(1), 123–158 (1996). https://doi.org/10.1023/A:1018012322525
6. Sugiyama, M.: Statistical Reinforcement Learning. Modern Machine Learning Approaches. Chapman and Hall/CRC, Boca Raton (2015)
7. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction, 2nd edn. The MIT Press, Cambridge (2020)
8. Tabor, P.: Q learning with just NumPy. Solving the mountain car. Tutorial (2019). https://www.youtube.com/watch?v=rBzOyjywtPw&t=3s. Accessed 7 Jan 2022