Reinforcement Learning to solve Inverted Pendulum Problem

Md Raisul Kibria

INTRODUCTION

A classic control problem called the inverted pendulum includes a pole connected with a hinge/joint over a cart which can freely move on a singular axis. Due to gravitational pull, the pole will always end up falling in either side. The goal of the problem is to move around the cart to balance the pole preventing it from falling over a target step in time. The state space includes two parameters – the angle of the pole (θ), and the angular velocity (ω). The angle of the pole can be at a continuous range of $[-\frac{\pi}{2}, \frac{\pi}{2}]$ radians, and the angular velocity can range between $[-\pi, \pi]$ rad/sec. The overall problem can be simplified to a few parameters, and two equations that represent the system and evolution of the state space. These parameters are the masses of the cart (M, kg) and the pole (m, kg), the length of the pole (d, m) and the time step ($\Delta t, sec.$). The internal equations for the system are:

$$\theta_{t+1} = \theta_t + \omega_{t+1}\Delta t \tag{1}$$

$$\omega_{t+1} = \omega_t + \frac{g \sin\theta_t - \alpha m d \omega_t^2 \frac{\sin 2\theta_t}{2} + \alpha \cos\theta_t \alpha_t}{\frac{4d}{3} - \alpha m d (\sin\theta_t)^2} \Delta t \tag{2}$$

Where, $\alpha = \frac{1}{M+m}$, and α_t is action applied at time t, which is applying a force in Newtons on the cart towards either side or remaining stationary which is then transferred to the pole attached to it. For this specific implementation, the given parameters which are used across the experiments are:

$Pole\ mass, m = 1.5\ kg$

$Cart\ mass, M = 7\ kg$

$Pole\ length, d = 0.5\ m$

$Time\ step, \Delta t = 0.1\ sec.$

The goal is to prevent the pendulum to fall during a 10 sec. simulation (100 steps).

The problem can be addressed through an intelligent agent that uses reinforcement learning algorithms to learn a good policy such that the agent can complete its mission (achieve the target of preventing falling). As part of the task requirement, the problem has been adapted for the Monte-Carlo for control and the temporal differencing equivalent – the SARSA algorithm. Both variants of the SARSA algorithm – SARSA (0) and SARSA (λ) has been implemented and different parameters has been explored to analyze the effects on the agent. Finally, a comparative analysis has been presented among the experimented algorithms with a discussion. For easier structuring, the sections follow the given tasks in order.

EXPERIMENTATION

Task 1: Analyzing given code

inverted_pendulum.py:

Models the environment of the problem following the standards of other libraries like the OpenAI Gym. The class InvertedPendulum initiates the internal parameter and provides the following methods:

- *Reset:* The method clears out all data of state of the environment recorded in any previous runs. This includes clearing out the list of the angular states of the pendulum and setting the angular velocity to 0. If *exploring_starts* is set, the agent is enabled to explore from randomized initial angular positions. The initial position is sampled from $\mathcal{N}(0, 0.1)$, so mostly from a vertical position.
- *Step:* The step method models the internal functionalities of the pendulum by applying the input action α_t through equation (1) and (2) to the systems. Here, the overall problem is abstracted down and simplified to only three actions of $[-50N, 0N, 50N]$ force. The step function also determines whether the system has reached a terminal step (the pendulum has fall on either side) and uses a flag *done* to record that. Along with the new state, the method also calculates the reward for the input action. In this case, the ideal function for calculating the reward is the cosine of the angle, as in the specified range, it rewards the vertically aligned position with the maximum reward (Fig. 1).

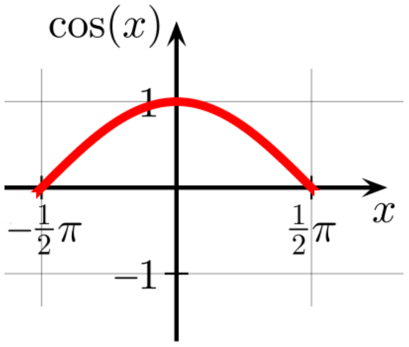


Fig. 1: Cosine of angular position as Reward function

- *Render:* Creates an animation across the recorded angular states of the system. The visualization is good way to observe how the system behaves in the given iteration (Example: Fig. 2).

random_agent_inverted_pendulum.py: This script initializes the environment with the input parameters and creates an experimental episode with taking random actions in each step until a terminal step is reached or the loop of 100 steps (10sec.) is completed. As the system does not use any learning mechanism or reward function, often the experiment ends in a short number of steps. A sample run is presented in Fig. 2.

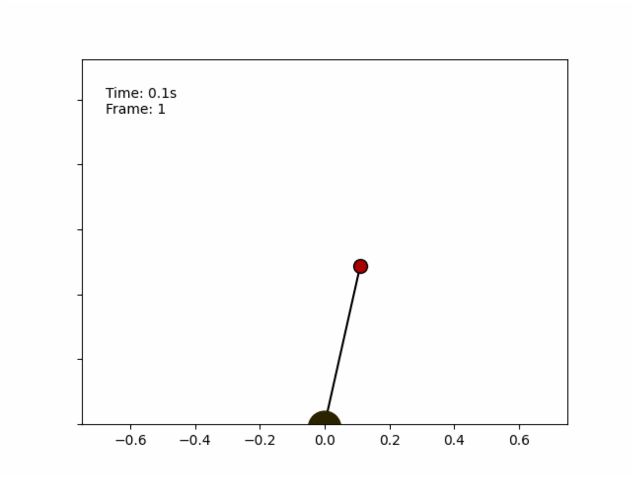


Fig. 2: Sample output for a Random Agent

montecarlo_control_inverted_pendulum.py:

The Monte-Carlo for control algorithm is an active reinforcement learning method. The algorithm uses a state-action function instead of a utility function. As the state space of the problem is continuous, the script discretizes the space into (12x12) bins for both state variables (θ, ω). The bins (1-2, 2-3, ..) are as following:

	1	2	3	4	5	6	7	8	9	10	11	12
Velocity State Bins	-3.14	-2.57	-1.99	-1.43	-0.86	-0.28	0.28	0.85	1.43	1.99	2.57	3.14
Position State Bins	-1.57	-1.28	-0.99	-0.71	-0.43	-0.14	0.14	0.43	0.71	0.99	1.28	1.57

The policy matrix, and the state-action matrix are initialized with random actions and values. The implementation of the algorithm is done in three phases for each episode:

i. *Running the episode:* The initial episodes work as the random agent, as the agent here follows random actions dictated by the policy and keeps track of the evolution for the episode until terminal state is reached or the episode is completed.
ii. *Policy Update:* The implementation is for the first visit MC. Hence, the state-action matrix is updated only during the first visit to a state during an episode following the equation:

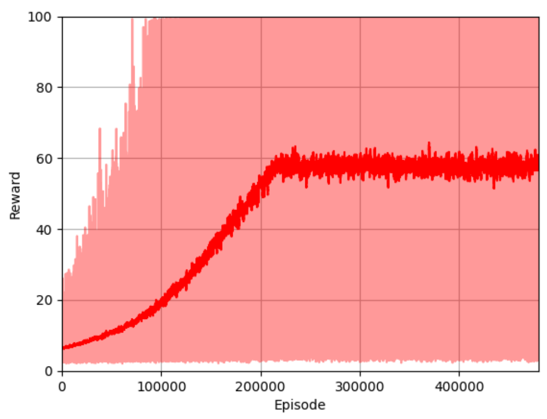
$$return = \sum_{i}^{N} \gamma^{i}\ epsiode[i].reward$$

$$Q(s,a) = Q(s,a) + return$$

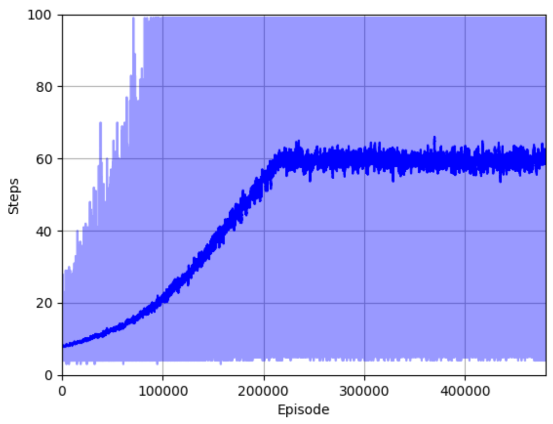
Along with the Q-matrix, another matrix called the *running mean matrix* is maintained for the next phase.

iii. *Policy improvement:* For each visited state in the episode, the policy matrix is updated by following a greedy method while selecting the best action from the running mean of the state-action matrix.

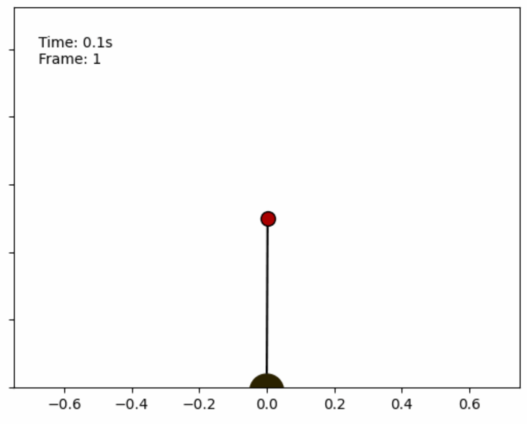
The implementation uses an ϵ-greedy strategy with an exponential decay to enable randomness while exploring the policy to find if there exists any better policy (optimal). The results for a run with 500,000 episodes are presented in Fig. 3.



(a) Reward curve



(b) Steps curve



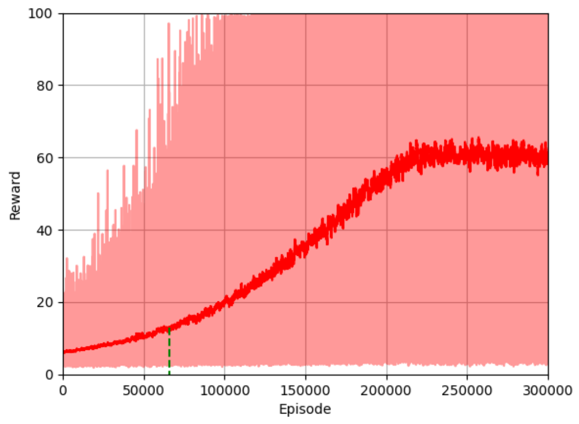
(c) Simulation

Fig. 3: 500000 episodes of Monte Carlo for control for default parameters

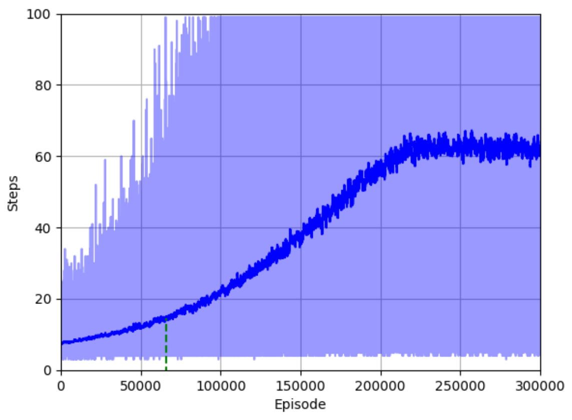
It can be clearly observed that the agent, in contrast to the random one, has learned a good policy and always tries to balance the pole in the reverse direction of the gravitational pull to prevent falling. Around 220,000 episodes, the policy starts providing stable rewards with more frequent success. Hence, to reduce time, all further experiments are run for **300,000 episodes**.

Task 2: Updating the Monte-Carlo Script with target parameters

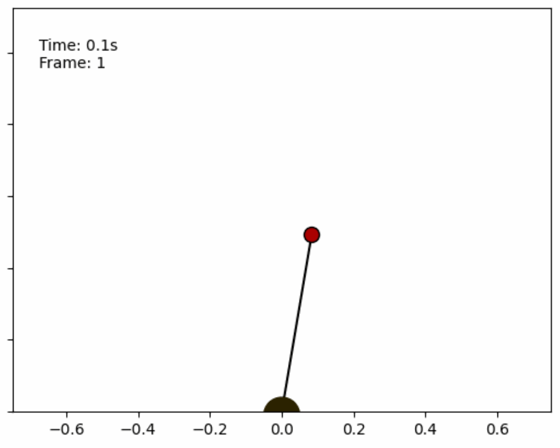
In order to initialize the environment with the target parameters mentioned in the introduction section, a dictionary called *PARAMS* is used which keeps track of all the different parameters with their default values. The code is also updated to maintain certain target metrics for future comparisons. As specified earlier, this time the agent was run for 300,000 episodes and the results are demonstrated in Fig. 4.



(a) Reward curve



(b) Steps curve



(c) Simulation

Fig. 4: 300000 episodes of Monte Carlo for control for target parameters

In contrast to the default parameters, after approximately 220,000 episodes, the reward on average stabilizes around 60. Around the same number of stages, the exponential decay also is fixed to the minimum value (0.1). So, as the random exploration stops, the agent can reach a more stable policy. More than half of the length of the episode the pendulum is balanced in a vertical position or $\pm 53°$ $(acos(.6) = 53°)$ range for the full episode. The simulation for the last episode, however, is for a shorter one. The random exploration with exponential decay is enabled here as well.

Task 3: Implementing SARSA (0)

SARSA (0) is a time differencing algorithm equivalent to the Monte-Carlo method for control. The algorithm learns an action-value function iteratively by updating the Q-values based on the observed rewards and the Q-values of subsequent state-action pairs. It uses a combination of exploration and exploitation to balance learning and taking actions based on the current policy.

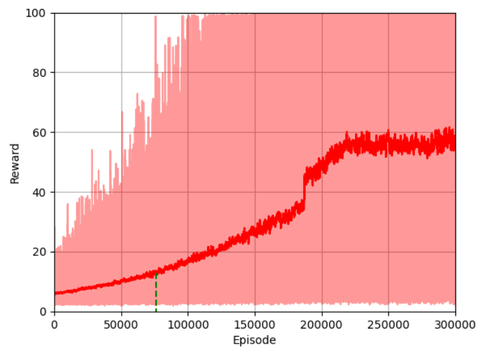
To implement the algorithm, the Monte-Carlo script is changed as SARSA (0) does not require separate phases. The state-action matrix (Q) and the policy matrix is updated at each stop during an episode. Although majority of the rest of the functions are remained similar, an *update_state_action_matrix* function is introduced for the algorithm. The function updates the state-action matrix according to the following equations:

$$\delta = r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$$

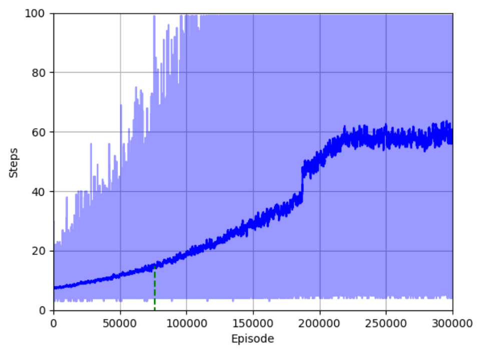
$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \times \delta$$

Here, $\gamma\ is\ set\ to\ 0.99\ and\ learning\ rate\ \alpha\ is\ set\ to\ 0.001$.

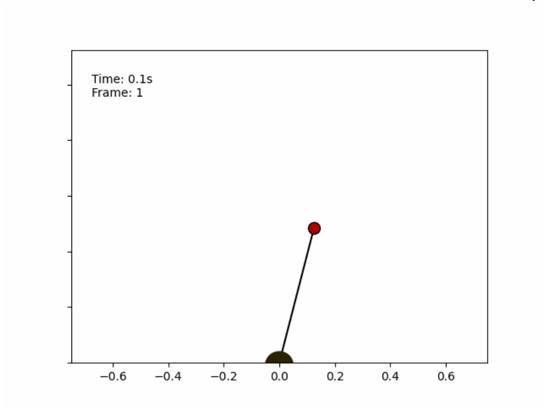
With the updated state-action matrix, the policy matrix is updated following a greedy strategy. In this case, the same ϵ-greedy strategy is used as the MC method. Other utilities such as arguments, restructuring and metric tracking is introduced in the script. After a sample run, the results for the SARSA (0) are as demonstrated in Fig. 5.



(a) Reward curve



(b) Steps curve



(c) Simulation
Fig. 5: SARSA (0) sample run

From the results, we can observe similarity to the MC method as the cumulated rewards reaches a mean value just below 60 after 200,000 episodes. The specific number of episodes again corresponds to the ϵ decay reaching the minimum possible value. However, unlike the MCs gradual increase throughout the reward curve, there is a straight rise around 190,000 episodes in the average value. It could correspond to finding a policy much better than above which might result in different episodes from the stochasticity in the system. The simulation also shows the robustness of the agents as it is able to recover from very big swings.

Task 4: Implementing SARSA (λ)

The SARSA(λ) algorithm combines the concepts of TD learning with eligibility traces to efficiently update the Q-values and capture the influence of past actions. It allows for more effective credit assignment, accounting for the contribution of multiple state-action pairs in the learning process.

The rest of the algorithm is similar to the SARSA (0) method, except the use of eligibility traces in updating the Q-values, and simultaneously updating the eligibility traces. It follows the equations:

Incorporating replacing and clearing of traces,

$$E_t(s,a) = \begin{cases} 1; for\ s_t, a_t \\ 0; for\ s_t, not\ a_t \\ \gamma\lambda E_t(s,a); otherwise \end{cases}$$

Updating state-action matrix,

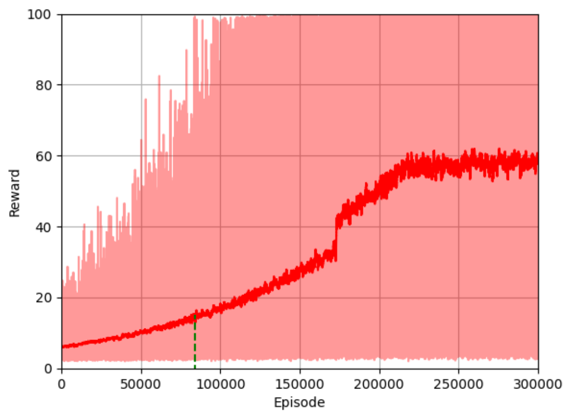
$$\delta = r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$$

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \times \delta \times E_t(s,a); \forall(s,a)$$

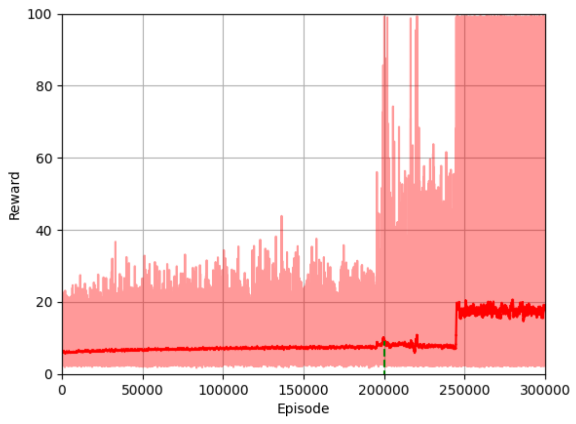
The *update_state_action_matrix* is updated to first incorporate the replacing of traces of the target state-action pair and clearing the traces of all other actions. Then, the *trace_matrix* is used to update the state-action matrix. Finally, the eligibility *trace_matrix* is decayed according to the decay factor $\gamma(= 0.99)$ and the lambda λ value. The same ϵ decay method was used for this experimentation. The lambda λ parameter is tested for the system with keeping other parameters same.

Effect of λ value:

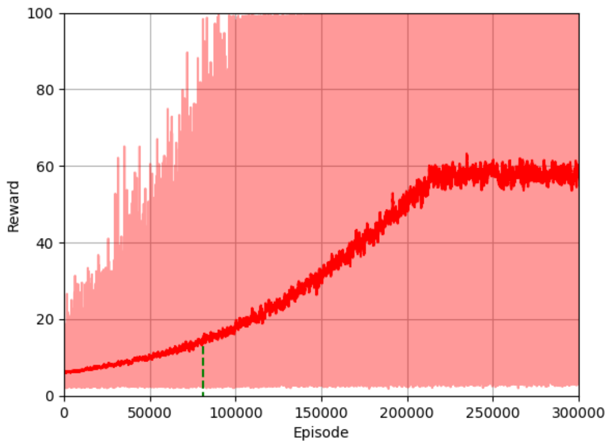
λ is the eligibility trace decay parameter that determines the influence of past actions on the updates. It ranges between 0 and 1, with 0 indicating a short trace and 1 indicating a long trace. Here, three values for lambda have been experimented for a very short trace (0.01), medium value (0.5) and a maximum trace (1). As the reward and step graph are very similar, for brevity, only the reward graphs are presented in Fig. 6.



(a) $\lambda = 0.01$



(b) $\lambda = 1$



(c) $\lambda = 0.5$

Fig. 6: Effect of Different Lambda Values

We can observe the consideration of different length of trace (λ) values in the experiments significantly impacts the outcome. A very short trace (0.1) similar effects in the reward curve as SARSA(0). However, the addition of trace, results in a slightly higher average reward. In case of medium trace, the running mean stabilizes with the epsilon value, and before that there is a gradual increase, same as the MC method. Thus, a medium trace demonstrates a stable and slightly better learning behavior. A λ value of 1 implies giving maximum attention to historical actions, which in this case, negatively effects the learning. As other states get updated in large amounts, the agent diverges from a good policy. Other than very high relevance to historical actions, for a larger trace, the convergence is smoother.

Task 5: Incorporating ϵ- greedy strategies

The three ϵ- greedy strategies discussed are:

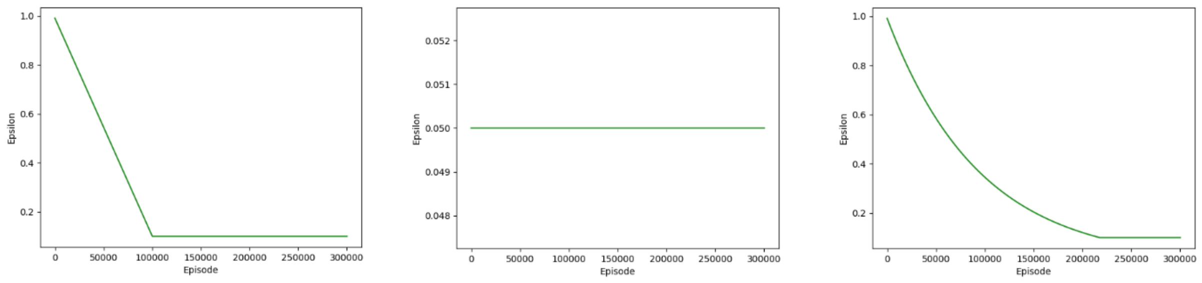
i. *ϵ with linear decay from 0.99 to 0.1*: A decay operation that linearly decreases the ϵ value across specified number of episodes before settling to a minimum value of 0.1 (to maintain some randomness). Here, the number of steps is set to 100,000.

ii. *ϵ fixed to a certain value:* a constant value to always have certain randomness in the greedy strategy.

iii. *ϵ with exponential decay* according to,

$$\epsilon(ep) = \max\{\mu_{\epsilon}, \epsilon \cdot d^{ep}\}$$
$$\text{where, } d=0.9 \text{ and } \mu_{\epsilon} = 0.1$$

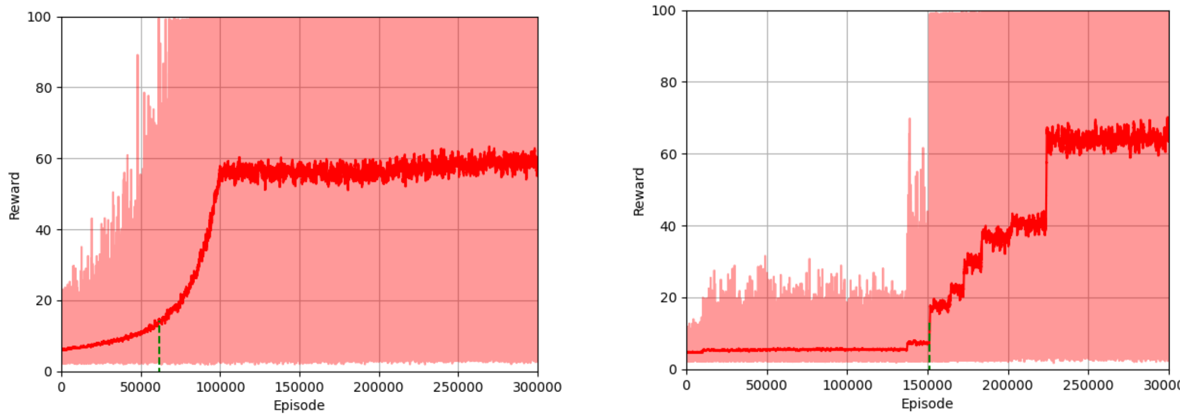
The different strategies and their value curves across the 300,000-episode runs are presented in Fig. 7.



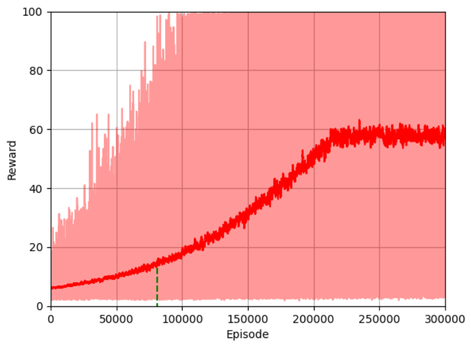
(a) Linear Decay (b) Fixed Value ($\epsilon = 0.05$) (c) Exponential Decay

Fig. 7. Different ϵ- greedy strategies

The exponential decay is already implemented in the script of the MC method. A parameter (an argument to the script) can be used to choose between the target ϵ-greedy strategy. Here, we present the output for different strategies with other parameters kept constant in Fig. 8.



(a) Linear Decay (b) Fixed Value ($\epsilon = 0.05$)



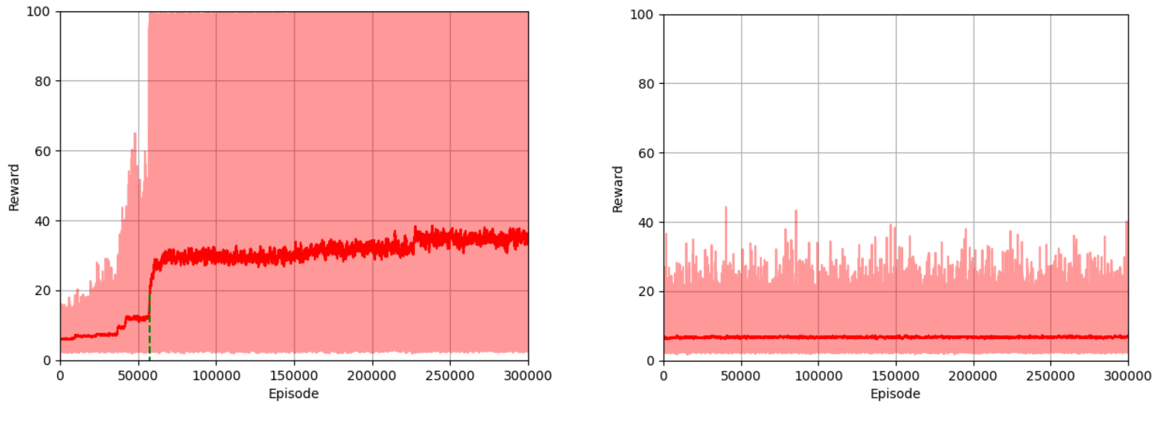
(c) Exponential decay

Fig. 8. Effect of different ϵ-greedy strategy with $\lambda = 0.5$

For different ϵ-strategies we can see the same phenomenon observed in other experiments. As soon as, the ϵ values settle to the minimum thus making random explorations unlikely, the reward curve soon reaches a stable mean. For the linear decay, as the length of the scale has to be prespecified (100,000 in this case) we can see the reward curve reaching the previous average of near 60 reward just after 100,000 episodes, and right to 60 near the 230,000 mark. In case of both linear and exponential decay, the stable mean is almost same, and before reaching that, the graph follows a steady increase. However, in case of the fixed value of 0.05, we can see step like patterns throughout the curve (not smooth due to the stochasticity across the scale). But it also manages to reach a much higher mean of around 70 (70% of the time the pendulum remains vertical). The fixed value works best in this case, as the very small randomness help improve current policy in a greedy fashion, while very small chances enable exploration, resulting in bumps when it finds a much better policy. As the search is confined but with small chances of exploration, the best policy can be reached with more confidence (stable state-action matrix). In this case again, the highest mean is reached after 230,000 episodes.

Effect of ϵ value for fixed strategy:

As the fixed strategy provides the best outcome, different values are tested with more randomness (0.9) and limited randomness (0.2). The results are presented in Fig. 9.



(a) $\epsilon = 0.2$ (b) $\epsilon = 0.9$

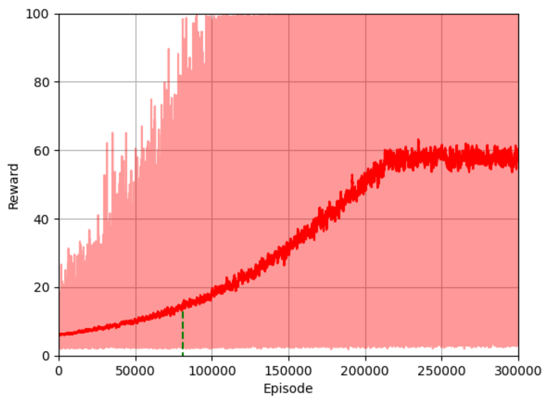
Fig. 9: Fixed strategy with different ϵ

The outcomes of the fixed strategy with very little randomness cannot be reproduced with other values. With $\epsilon = 0.2$, thus limited exploration, we can see that the ramps are not present anymore. Although it reaches a stable policy much faster (~60,000 episodes), it is not as good as other methods (mean below 40). However, the results are much worse with completely random strategy (not greedy anymore). Across the tested number of episodes, the system cannot get to a policy providing the maximum reward at all.

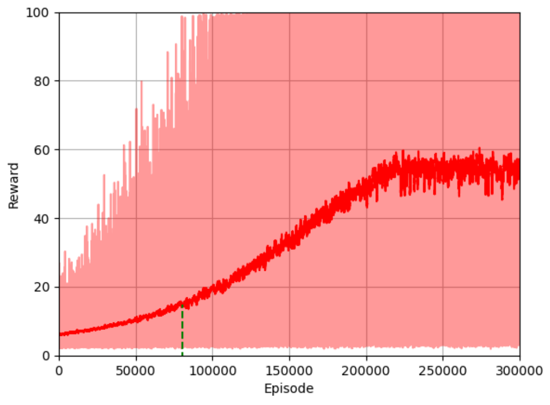
Effect of learning rate α:

The learning rate decides how much to rely on newer information with respect to current knowledge. A very high learning rate can result in unstable and erratic learning, causing the algorithm to overshoot and fail to converge to optimal values. On the other hand, a low learning rate, although, can provide more stability and prevent overshooting, it may result in slower learning and longer convergence times.

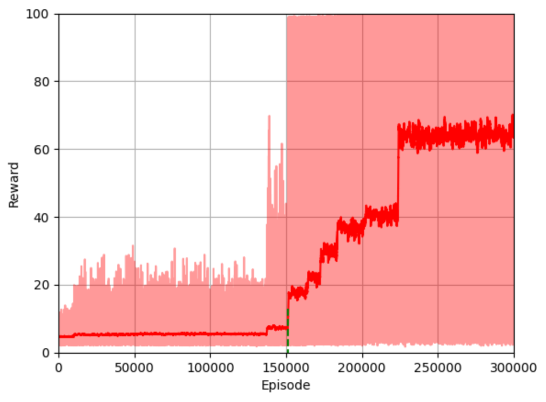
In this experiment, two different learning rates are experimented for exponential decay and fixed epsilon strategies. A smaller rate of 0.001 is used for majority of the experiments, and a higher learning rate 0.1 is tested. The results are presented in Fig. 10.



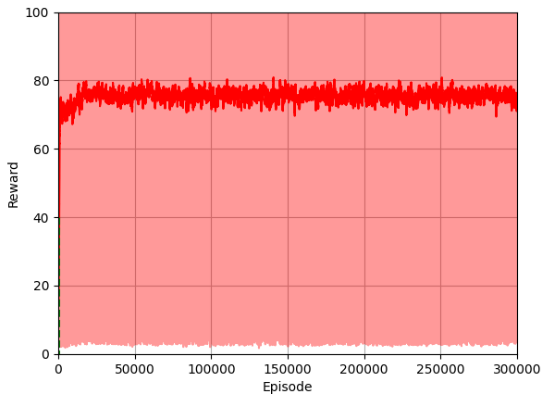
(a) Exponential ϵ Decay (α=0.001)



(b) Exponential ϵ Decay (α=0.1)



(a) ϵ=0.05 (α=0.001)



(b) ϵ=0.05 (α=0.1)

Fig. 10: Effect of learning rate

We can see considerable effects of higher learning rate. With the exponential decay, although the outcomes are similar, the curve for higher rate sees much more fluctuation. The high-rate results in drastically swinging around the found policy. In case, of fixed ϵ, the higher learning rate results in the algorithm to overshoot towards expected Q-values. In this case, it results in the algorithm finding a good enough policy quite early. The high learning rate allowed the agent to quickly update its Q-values based on the observed rewards, thus the rapid convergence to an initial policy with high rewards. The agent learns from each experience more aggressively, quickly incorporating new information into its Q-value estimates. As the fixed ϵ, limits random exploration, the agent demonstrates much more stability in the found policy.

Task 6: Comparative Analysis

For comparison across different algorithms and parameters, a set of metrics are defined. As the system is built with a stochastic nature, and the optimal policy is not known, the metrics used only provide an abstract picture and are not definitive. The presented metrics are,

- *Mean number of steps:* for a fixed total number of episodes, higher mean in general should refer to a better learned system.
- *Median of steps*: The median provides a good representation of the distribution.
- *Longest Streak of success:* The number of steps the policy gives success continuously.
- *Success ratio:* Ratio of completed episode to the total.
- *First Success:* The first time the agent completes its mission.

Algorithm	Mean Steps	Median	Longest Streak	Success ratio	First Success
Monte-Carlo for control	37.74	23	9 [262880:262889]	14.11%	65553
SARSA (0) – alpha 0.001	33.27	20	8 [279727:279735]	10.08%	75850
SARSA (λ) – exponential decay – alpha 0.001 – lambda 0.5	35.16	21	11 [228779:228790]	11.89%	81031
SARSA (λ) – exponential decay – alpha 0.001 – lambda 0.01	34.48	21	9 [274432:274441]	11.36%	83856
SARSA (λ) – exponential decay – alpha 0.001 – lambda 1	10.61	8	2 [245010:245012]	.42%	200178
SARSA (λ) – fixed decay 0.2– alpha 0.001 – lambda 0.5	29.14	20	4 [253629:253633]	3.73%	57600
SARSA (λ) – fixed decay 0.05 – alpha 0.001 – lambda 0.5	28.30	9	13 [277186:277199]	11.74%	151497
SARSA (λ) – fixed decay 0.9– alpha 0.001 – lambda 0.5	8.19	7	0 [-1:-1]	0.0%	None
SARSA (λ) – linear decay– alpha 0.001 – lambda 0.5	44.95	33	9 [145776:145785]	20%	61404
SARSA (λ) – exponential decay – alpha 0.1 – lambda 0.5	34.6	21	9 [293212:293221]	10.89%	80124
SARSA (λ) – fixed decay 0.05– alpha 0.1 – lambda 0.5	76.37	99	25 [192568:192593]	57.24%	403

DISCUSSION AND CONCLUSION

From the comparative table, it is evident that the SARSA (λ) method with high learning rate and lower stochasticity in action sampling reaches the best possible outcomes. Almost 57% of the total 300,000 episodes, end up in the agent completing its mission. The relatively smaller action-state space of the problem favors the scenario. However, as high learning rate may not be suitable due to the erratic learning in different cases, the method may not be very reliable in every use-case. The second-best method is SARSA (λ) with linear decay and with fixed value of epsilon to 0.05. In this case, the mean number of steps is above 40 and 1/5 of the total episodes result in success. The eligibility trace results in SARSA (λ) getting better convergence compared the SARSA (0) the The worst possible setting for the agent is with SARSA (λ) and a fixed epsilon of 0.9. The agent fails to achieve any success due to the extreme randomness of action sampling. Finally, compared to SARSA, Monte-Carlo also provides very good solutions in the same number of episodes being overall the third best in terms of the compared metrics.

Throughout these tasks, an intelligent agent was designed to solve a problem that is although simple, difficult to learn efficiently. Different techniques like discretizing the state-space or enabling random initializations are introduced to make the problem manageable. Active reinforcement learning algorithms such as the Monte-Carlo for control or different versions of SARSA are adapted to approximate the problem. In addition, several internal parameters are extensively explored and compared to find an optimal setting for the solution.

The rest of the graphs along with the source code is available here: https://github.com/raisul-kibria/pendulum_learning_agent