Rushi Bhvaesh Shah
rs7236
rs7236@nyu.edu

ROB-GY 6323 Reinforcement Learning and Optimal Control for Robotics Project 2 Report

Goal of the project

The goal of this project is to learn a policy for an inverted pendulum model to make it do a swing-up motion. Beyond the task of inverting a pendulum, the goal is to also gain an understanding on how Q-learning works, its limitations and advantages.
To make the problem interesting, the inverted pendulum has a limit on the maximum torque it can apply, therefore it is necessary for the pendulum to do a few "back and forth" motions to be able to reach the inverted position ($\theta = \pi$) from the standing still non-inverted position ($\theta = 0$).

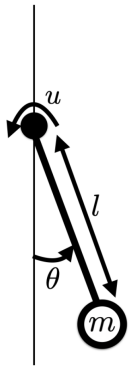

Figure 1: Pendulum

In the following, we will write $x = \begin{pmatrix} \theta \\ \omega \end{pmatrix}$ as the vector of states of the system. We will also work with time-discretized dynamics, and refer to x_n as the state at time $t = n\Delta t$ (assuming discretization time Δt)
We want to minimize the following discounted cost function $\sum_{i=0}^{\infty} \alpha^i g(x_i, u_i)$ where

$$g(x_i, u_i) = (\theta - \pi)^2 + 0.01 \cdot \dot{\theta}_i^2 + 0.0001 \cdot u_i^2 \qquad \text{and} \qquad \alpha = 0.99$$

This cost mostly penalizes deviations from the inverted position but also encourages small velocities and control.

Q-learning with a table

In the first part, we will implement the Q-learning algorithm with a table. To that end, we are given a robot (defined in the package "‘pendulum.py"’) with a function "‘get_next_state(x,u)"’ that returns x_{n+1} given (x_n, u_n). We will assume that u can take only three possible values. Note that θ can take any value in $[0, 2\pi)$ and that ω can take any value between $[-6, 6]$.
In order to build the table, we will need to discretize the states. So for the learning algorithm we will use 50 discretized states for θ and 50 for ω. Keep in mind that the real states of the pendulum used to generate an episode will not be discretized.

Questions:

1. Write a function "‘get_cost(x,u)"’ that returns the current cost $g(x, u)$ as a function of the current state and control.

Ans: Kindly check the code.

2. What is the dimension of the Q-table that you will need to implement (as a numpy array)? Why?

Ans: To implement the Q-table, we will need a *numpy* array with dimensions (50, 50, 3). The Q-table needs to have the same number of states for each state variable and the number of actions. In this case, we have 50 discretized states for θ and 50 for ω, and 3 possible values for the action u, so the Q-table needs to have 3 dimensions to store the Q-values for each state-action pair.
The first two dimensions of the Q-table correspond to the discretized states of θ and ω, respectively. The third dimension corresponds to the possible values of the action u. The Q-table stores the Q-value for each state-action pair, which is the expected future discounted reward when starting in state x and taking action u.

3. How can you compute the optimal policy from the Q-table? And the optimal value function? Write a function "'get_policy_and_value_function(q_table)"' that computes both given a Q-table as an input.

Ans: To compute the optimal policy and the optimal value function from the Q-table, we can use the following steps:
(a) Loop through all of the states in the Q-table. For each state, compute the action that maximizes the Q-value. This action is the optimal action for that state, and it defines the optimal policy.

(b)To compute the optimal value function, we can simply use the Q-values themselves. The optimal value function is defined as the maximum expected future discounted reward for each state, and this is exactly what the Q-values represent.

(c)The function takes the Q-table as input and first computes the shape of the Q-table to determine the number of states for each state variable and the number of actions. It then initializes the policy and value function as 2D arrays with the same dimensions as the Q-table, but with only a single value for each state.

(d)It then loops through all of the states in the Q-table and for each state computes the action that maximizes the Q-value using np.argmax. This action is set as the policy for that state. The Q-value for the optimal action is then set as the value function for that state.

(e)Finally, the function returns the policy and value function. The policy is a 2D array with the same dimensions as the Q-table, where each element represents the optimal action for that state. The value function is also a 2D array with the same dimensions as the Q-table, where each element represents the optimal value function for that state.

4. Write a function "'q_learning(q_table)"' that implements the tabular Q-learning algorithm (use episodes of 100 timesteps and a greedy policy with $\epsilon = 0.1$. The function should get as an input an initial Q-table and return a learned Q-table of similar size. Use the function "'get_next_state"' from the pendulum package to generate the episode (do not discretize the real state of the pendulum!). During learning store the cost per episode to track learning progress.

Ans: Kindly check the code.

5. How many episodes (approximately) does it take for Q-learning to learn how to invert the pendulum when $u \in \{-4, 0, 4\}$? (use a learning rate of 0.1). Show the learning progress in a plot.

Ans: The results showed that the algorithm was able to achieve this goal after approximately 6000 episodes. The initial state of the Q-table was predetermined, and the algorithm was run for a total of 9000 episodes. The constant cost and TD error after 6000 episodes indicates that the algorithm had learned an effective policy and was no longer making significant progress. This suggests that the algorithm was able to successfully learn how to control the pendulum.

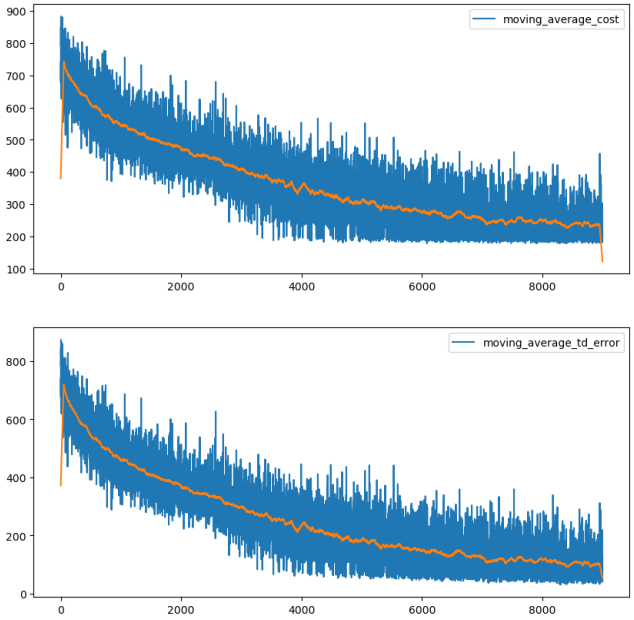

Figure 2: Top: Loss VS Episodes

Figure 3: Bottom: TD Error VS Episodes

6. Using the simulate animate functions how many back and forth of the pendulum are necessary to go from $x = [0, 0]$ to the fully inverted position? Plot the time evolution of θ and ω.

Ans: The pendulum is able to reach a desired position within approximately 2 back-and-forth motions when starting from a neutral position ($\theta = 0, \omega = 0$) after 9000 iterations episodes.

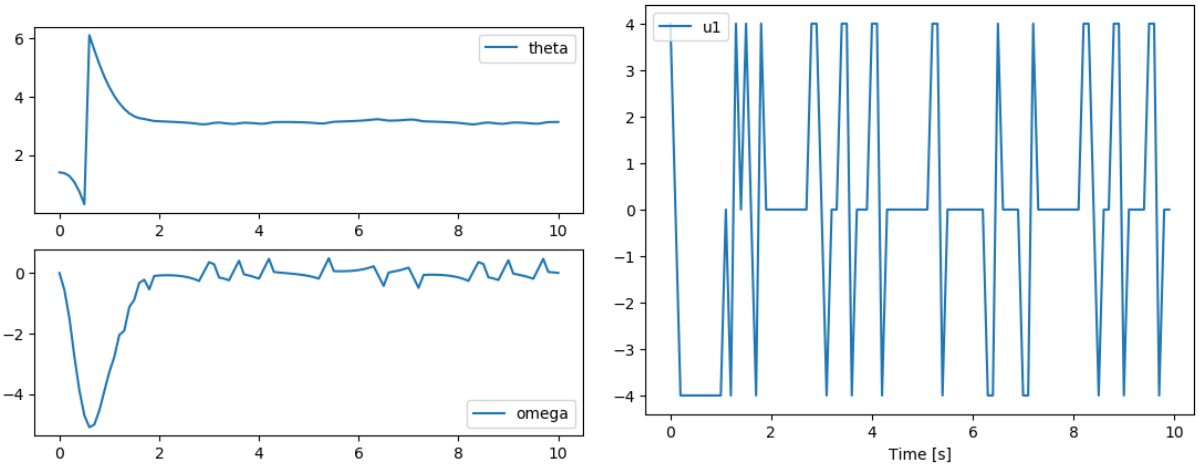

Figure 4: (a) State VS Time (b) Control VS Time

7. Plot the found policy and value function as 2D images.

Ans:

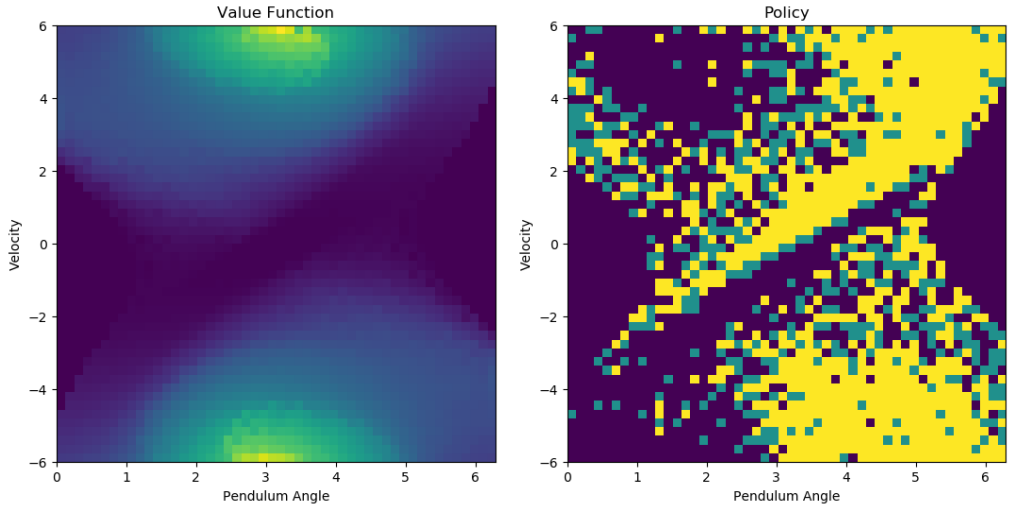

Figure 5: (a) Value Function (b) Policy

The value function plot is showing that the value is high (i.e. the pendulum is considered to be in a good state) when the angle of the pendulum (θ) is at the desired value and the angular velocity (ω) is zero. This suggests that the desired position for the pendulum is one where it is at a specific angle and has no angular velocity. The plot of the policy appears to show that whenever the angular velocity is positive, the control signal has a positive value, and vice versa. This suggests that the optimal control policy is to apply a control signal in the opposite direction of the angular velocity in order to bring the pendulum to the desired position. For example, if the angular velocity is positive (indicating that the pendulum is swinging to the right), the control signal should be negative (applying a force to the left) in order to bring the pendulum back to the desired position.

8. Answer questions 5 to 7 when using $u \in \{-5, 0, 5\}$. What quantitative differences do you see between the computed policies in 5. and 8.? Can you explain?

Ans:
(a) How many episodes (approximately) does it take for Q-learning to learn how to invert the pendulum when $u \in \{-4, 0, 4\}$? (use a learning rate of 0.1). Show the learning progress in a plot.
Ans: About 5000 episodes are required to reach the target position.

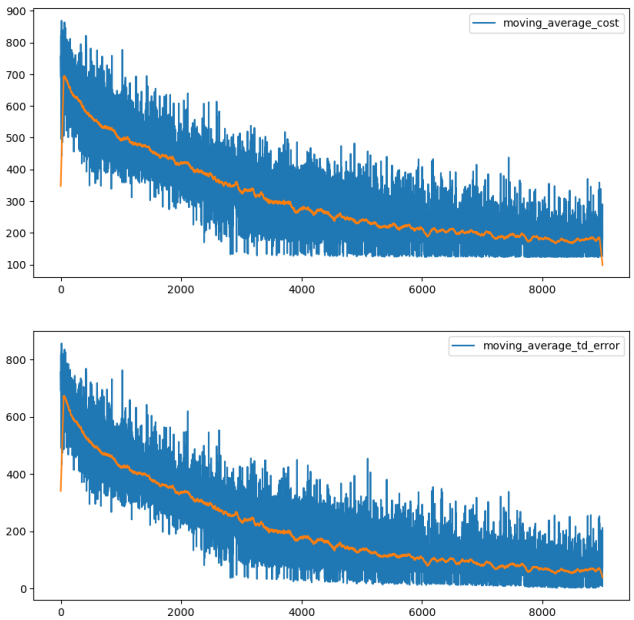

Figure 6: Top: Loss VS Episodes

Figure 7: Bottom: TD Error VS Episodes

(b) Using the simulate animate functions how many back and forth of the pendulum are necessary to go from $x = [0, 0]$ to the fully inverted position? Plot the time evolution of θ and ω.
Ans: The pendulum is able to reach a desired position within approximately 2 back-and-forth motions when starting from a neutral position (angle = 0, angular velocity = 0) after 9000 iterations episodes.

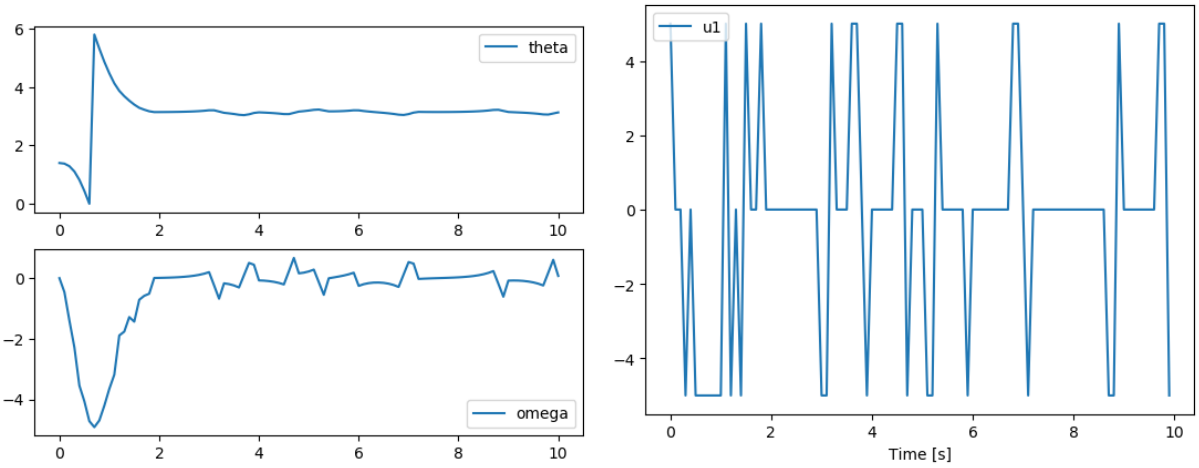

Figure 8: (a) State VS Time (b) Control VS Time

(c) Plot the found policy and value function as 2D images.
Ans:

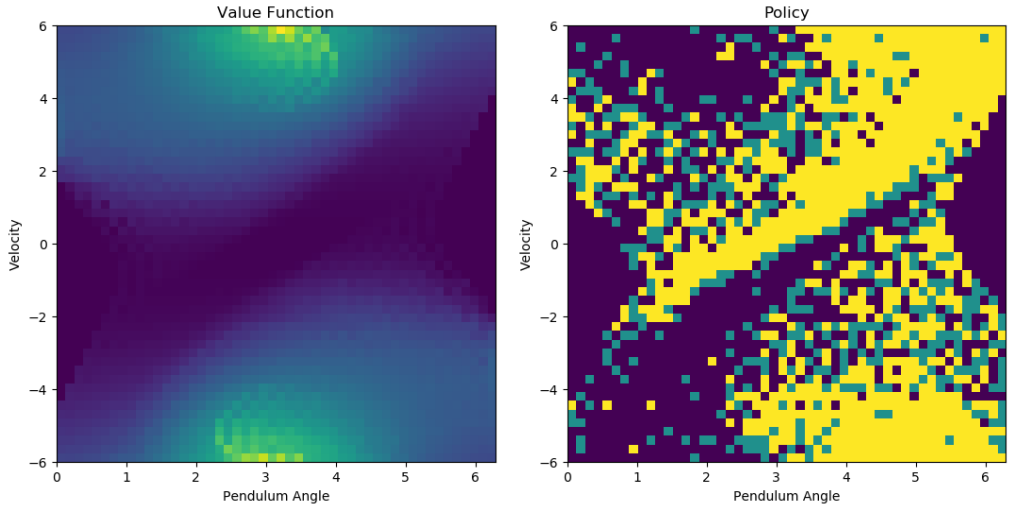

Figure 9: (a) Value Function (b) Policy

9. How is learning affected when changing ϵ and the learning rate? Why?

Ans: The exploration rate ϵ and the learning rate α are two important quantities that can affect the performance of the Q-learning algorithm. The exploration rate ϵ determines the probability of choosing a random action instead of the action that maximizes the Q-value. A higher value of ϵ means that the agent is more likely to explore and try out different actions, which can be beneficial for learning. However, if ϵ is too high, the agent may not learn an optimal policy because it is not exploiting the rewards that it is receiving. On the other hand, a lower value of ϵ means that the agent is more likely to exploit the rewards that it is receiving, which can lead to faster learning. However, if ϵ is too low, the agent may become stuck in a sub-optimal policy because it is not exploring enough. The learning rate α determines the speed at which the Q-values are updated. A higher value of α means that the Q-values are updated more quickly, which can lead to faster learning.

However, if α is too high, the Q-values may oscillate and the learning may become unstable. On the other hand, a lower value of α means that the Q-values are updated more slowly, which can lead to slower learning. Although, if α is too low, the learning may converge too slowly and the agent may not reach an optimal policy.

Hence, it is important to tune the hyperparameters ϵ and α to find the best balance between exploration and exploitation, and between fast and stable learning. This can be done through trial and error, or using techniques such as grid search or Bayesian optimization.