COMPSCI 687 Homework 5 - Fall 2024

Due **November 26**, 2024, 11:55pm Eastern Time

SHREYA BIRTHARE

1 Instructions

This homework assignment consists only of a programming portion. While you may discuss problems with your peers (e.g., to discuss high-level approaches), you must write your own code from scratch. The exception to this is that you may re-use code that you wrote for the previous homework assignments. **We will run an automated system for detecting plagiarism. Your code will be compared to the code submitted by your colleagues, and also with many implementations available online.** In your submission, do explicitly list all students with whom you discussed this assignment. Submissions must be typed (handwritten and scanned submissions will not be accepted). You must use LaTeX. The assignment should be submitted on Gradescope as a PDF with marked answers via the Gradescope interface. The source code should be submitted via the Gradescope programming assignment as a .zip file. Include with your source code instructions for how to run your code. You **must** use Python 3 for your homework code. You may not use any reinforcement learning or machine learning-specific libraries in your code, e.g., TensorFlow, PyTorch, or scikit-learn. You *may* use libraries like numpy and matplotlib, though. The automated system will not accept assignments after 11:55pm on November 26. The tex file for this homework can be found here.

Programming Assignment [100 Points, total]

In this assignment, you will implement three algorithms based on Temporal Difference (TD): TD-Learning for policy evaluation, and SARSA and Q-Learning for control. You will deploy all three algorithms on the Cat-vs-Monsters domain. You will also deploy SARSA and Q-Learning on the Inverted Pendulum Swing-Up domain. **Notice that you may not use existing RL code for this problem—you must implement the learning algorithms and the environments entirely on your own and from scratch.** The complete definition of the Cat-vs-Monsters domain can be found in Homework 3. The complete definition of the Inverted Pendulum Swing-Up domain can be found in Homework 2.

General instructions:

- You are free to initialize the q-function in any way that you want. More details about this later.
- Whenever displaying values (e.g., the value of a state), use 4 decimal places; e.g, $v(s) = 9.4312$.
- Whenever showing the value function and policy learned by your algorithm, present them in a format resembling that depicted in Figure 1.

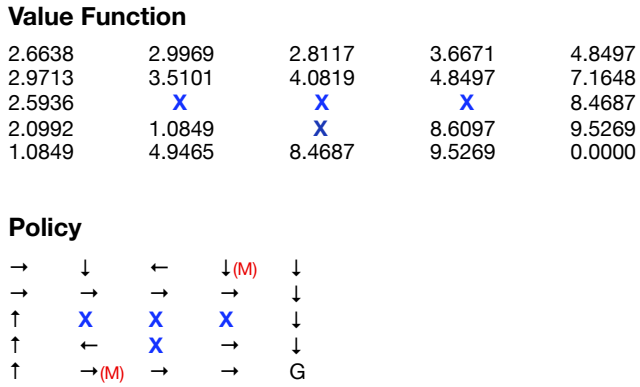

Figure 1: Optimal value function and optimal policy for the Cat-vs-Monsters domain.

1. TD-Learning on the Cat-vs-Monsters domain [20 Points, total]

First, implement the TD-Learning algorithm for policy evaluation and use it to construct an estimate, $\hat{v}$, of the value function of the optimal policy, π^*, for the Cat-vs-Monsters domain. **Remember that both the optimal value function and optimal policy for the Cat-vs-Monsters domain were identified in a previous homework by using the Value Iteration algorithm. For reference, both of them are shown in Figure 1.** When sampling trajectories, set d_0 to be a uniform distribution over $\mathcal{S}$ to ensure that you are collecting returns from all states. Do not let the agent be initialized in a Forbidden Furniture location. Notice that using this alternative definition of d_0 (instead of the original one) is important because we want to generate trajectories from all states of the MDP, not just states that are visited under the optimal deterministic policy shown in Figure 1.

You should run your algorithm until the value function estimate does not change significantly between successive iterations; i.e., whenever the Max-Norm between $\hat{v}_{t-1}$ and $\hat{v}_t$ is at most δ, for some value of δ that you should set empirically. For each run of your algorithm, count how many episodes were necessary for it to converge to an estimate of v^{π^*}. You should run your algorithm, as described above, 50 times and report:

(Question 1a. 2 Points) the value of α that was used. You should pick a value of α that causes the algorithm to converge to a solution that is close to v^{π^*}, while not requiring an excessively large number of episodes.

I used $\alpha = 0.05$ that resulted in max-norm difference from optimal value function of 0.324766452910056

(Question 1b. 9 Points) the average value function estimated by the algorithm—i.e., the average of all 50 value functions learned by the algorithm at the end of its execution. Report, also, the Max-Norm between this average value function and the optimal value function for the Cat-vs-Monsters domain.

See Figure 2

(Question 1c. 9 Points) the average and the standard deviation of the number of episodes needed for the algorithm to converge.

See Figure 2

```
******** TD(0) Results ********
Step size (alpha): 0.05
Average value function:
[[2.57689945 2.94936797 2.73430937 3.34233355 4.62226352]
 [2.92017038 3.47198838 4.05922131 4.77991045 7.20378061]
 [2.51610129 0.         0.         0.         8.43594195]
 [1.99876111 0.85255582 0.         8.49809292 9.41378751]
 [0.87748029 4.72251056 8.34175815 9.37925267 0.        ]]
Max-norm difference from optimal value function: 0.324766452910056
Average episodes to converge: 7487.92
Standard deviation of episodes: 6383.217821255985
```

Figure 2: Average Value function, Max Norm, Mean and Std Dev of number of episodes using TD(0) for 50 runs

2. SARSA on the Cat-vs-Monsters domain [20 Points, total]

Implement the SARSA algorithm and use it to construct $\hat{q}$, an estimate of the optimal **action-value function**, q^{π^*}, of the Cat-vs-Monsters domain; and to construct an estimate, $\hat{\pi}$, of its optimal policy, π^*. You will have to make four main design decisions: *(i)* which value of α to use; *(ii)* how to initialize the q-function; you may, e.g., initialize it with zeros, with random values, or optimistically. Remember that $q(s_\infty, \cdot) = 0$ by definition; *(iii)* how to explore; you may use different exploration strategies, such as ϵ-greedy exploration or softmax action selection; and *(iv)* how to control the exploration rate over time; you might choose to keep the exploration parameter (e.g., ϵ) fixed over time, or have it decay as a function of the number of episodes.

(Question 2a. 2 Points) Report the design decisions *(i, ii, iii, and iv)* you made and discuss what was the reasoning behind your choices.

(i) I used $\alpha = 0.005$ I set it to a small constant value as a small learning rate ensures stability and gradual updates to the Q-function. This avoids large fluctuations in Q-values, especially when there are noisy rewards or sparse rewards like reaching the goal or encountering monsters. I tried with $\alpha = 0.5, 0.05$ too but i found the i got the best result when $\alpha = 0.005$
(ii) I initialized my q-function with all zeros. Zero initialization is simple and assumes no prior knowledge about the environment. It avoids introducing bias through random or optimistic initialization. This aligns well with SARSA, where the agent learns by interacting with the environment. While randomly initializing Q-values can introduce variability in learning, it risks misleading the agent to favor certain actions prematurely, especially if the random values are far from the true Q-values. This could lead to slower convergence. Assigning high initial Q-values encourages exploration but may cause the agent to overestimate actions that are not optimal, requiring additional corrections over time. Since SARSA is policy-dependent, optimistic initialization can lead to suboptimal policies during early stages.
(iii) I used ϵ-greedy exploration. It balances exploration (trying new actions) and exploitation (choosing the current best-known action). By choosing random actions with probability $\epsilon = 0.1$, the agent explores the environment adequately while still favoring actions with higher Q-values.
(iv) I have kept ϵ fixed over time. Since the environment includes stochastic transitions (due to probabilistic actions), maintaining a constant exploration rate ensures the agent continues to explore throughout the learning process. Decaying ϵ could prematurely limit exploration, leaving parts of the environment underexplored.

(Question 2b. 7 Points) Construct a learning curve where you show, on the x axis, the total number of actions taken by the agent, from the beginning of the training process (i.e., the total number of steps taken across all episodes up to that moment in time). On the y axis, you should show the number of episodes completed up to that point. If learning is successful, this graph should have an increasing slope, indicating that as the agent takes more and more actions (and thus executes more learning updates), it requires fewer actions/timesteps to complete each episode. Your graph should, ideally, look like the figure shown in Example 6.5 of the RL book (2nd edition). To construct this graph, run your algorithm 20 times and show the average curve across those runs.

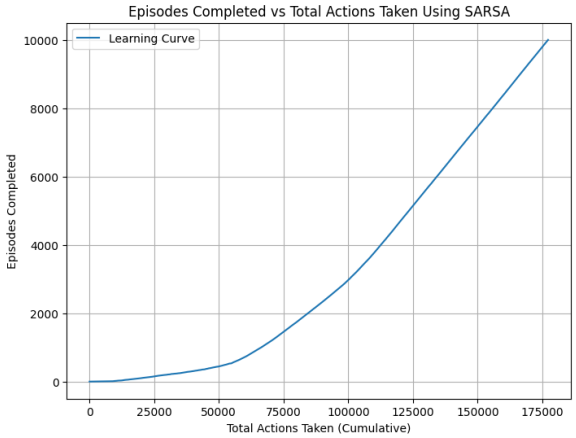

Figure 3: Learning Curve, Total Actions Taken v/s Episodes completed for 20 runs using SARSA algorithm

(Question 2c. 8 Points) Construct another learning curve: one where you show the number of episodes on the x axis, and, on the y axis, the mean squared error between your current estimate of the value function, $\hat{v}$, and the true optimal value function, v^{π^*}. The mean squared error between two value functions, v_1 and v_2, can be computed as $\frac{1}{|\mathcal{S}|}\sum_s (v_1(s) - v_2(s))^2$. Remember that SARSA estimates a q-function ($\hat{q}$), however, not a value function ($\hat{v}$). To compute $\hat{v}$, remember that $\hat{v}(s) = \sum_a \pi(s,a)\hat{q}(s,a)$. Remember, also, that the SARSA policy π, at each given moment, depends on your exploration strategy. Let us say, for example, that you are using ϵ-greedy exploration. Then, in the general case when more than one action

may be optimal with respect to $\hat{q}(s,a)$, the ϵ-greedy policy would be as follows:

$$\pi(s,a) = \begin{cases} \frac{1-\epsilon}{|\mathcal{A}^*|} + \frac{\epsilon}{|\mathcal{A}|} & \text{if } a \in \mathcal{A}^* \\ \frac{\epsilon}{|\mathcal{A}|} & \text{otherwise,} \end{cases} \tag{1}$$

where $\mathcal{A}^* = \arg\max_{a \in \mathcal{A}} \hat{q}(s,a)$. To construct the learning curve, run your algorithm 20 times and show the average curve across those runs; i.e., the average mean squared error, computed over 20 runs, as a function of the number of episodes.

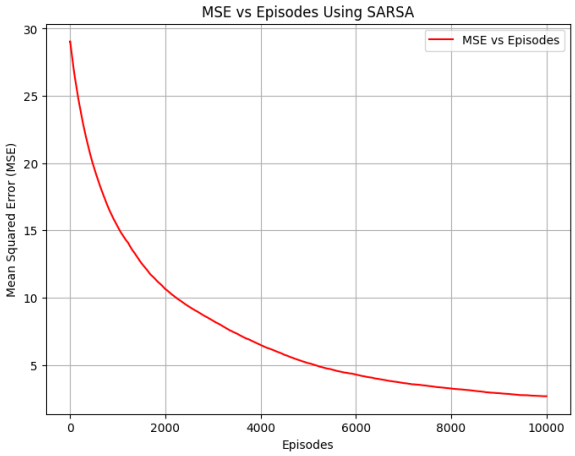

Figure 4: Learning Curve, Average MSE for 20 runs using SARSA algorithm

(Question 2d. 3 Points) Show the *greedy policy* with respect to the q-values learned by SARSA.

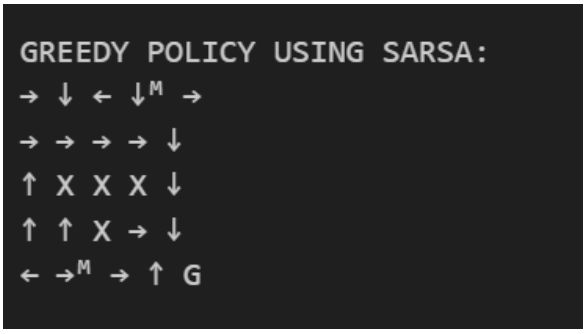

Figure 5: Final Greedy Policy using SARSA algorithm

3. Q-Learning on the Cat-vs-Monsters domain [20 Points, total]

Answer the same questions as above (*2a, 2b, 2c, and 2d*), but now using the Q-Learning algorithm; i.e., you will be using Q-Learning to solve the Cat-vs-Monsters domain. When computing $\hat{v}$, for question *3c*, you should compute the value function of the greedy policy: $\hat{v}(s) = \max_a \hat{q}(s,a)$. The amount of points associated with questions *3a, 3b, 3c, and 3d*, will be the same as those associated with the corresponding items of Question 2.

(i) I set the learning rate $\alpha = 0.005$. In Q-learning, a small α ensures that the Q-values are updated

gradually, preventing drastic changes to the value estimates and promoting stable convergence. This choice strikes a balance between ensuring steady learning progress and maintaining stability in the Q-value updates. A smaller α avoids erratic updates, which can occur with larger learning rates, helping the agent converge to the optimal policy more reliably.

(ii) I initialized the Q-function with zeros for all state-action pairs to avoid any initial bias. In Q-learning, starting with zero values means that the agent begins with no preconceived notions about the quality of actions in any state, enabling it to learn the optimal policy from scratch. While random initialization could introduce variance, potentially overestimating some actions and slowing down convergence, zero initialization ensures that all actions are treated equally at the beginning, leading to a more stable learning process. Optimistic initialization could promote exploration but may lead to inefficient exploration in environments with sparse rewards, making zero initialization a more stable choice.

(iii) I used the ϵ-greedy exploration strategy, which is standard in Q-learning. This strategy balances exploration and exploitation by selecting a random action with probability ϵ (exploration) and the best-known action with probability $1 - \epsilon$ (exploitation). The ϵ-greedy approach is effective in ensuring that the agent explores enough of the state space to avoid getting stuck in suboptimal policies while exploiting learned knowledge to maximize rewards. It's simple and efficient, making it ideal for Q-learning, where the goal is to iteratively improve Q-values based on experience.

(iv) I kept the exploration rate fixed at $\epsilon = 0.1$ throughout the training process. In Q-learning, maintaining a fixed ϵ ensures a consistent level of exploration during the entire learning process. While decaying ϵ over time can encourage more exploitation as the agent becomes more confident in its policy, a fixed ϵ ensures that the agent continues to explore sufficient actions, preventing premature convergence to suboptimal policies. This strategy helps strike a balance between exploration and exploitation throughout training.

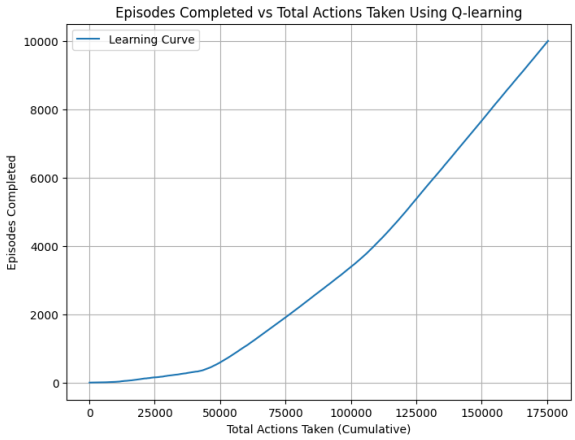

Figure 6: Learning Curve, Total Actions Taken v/s Episodes completed for 20 runs using Q-Learning algorithm

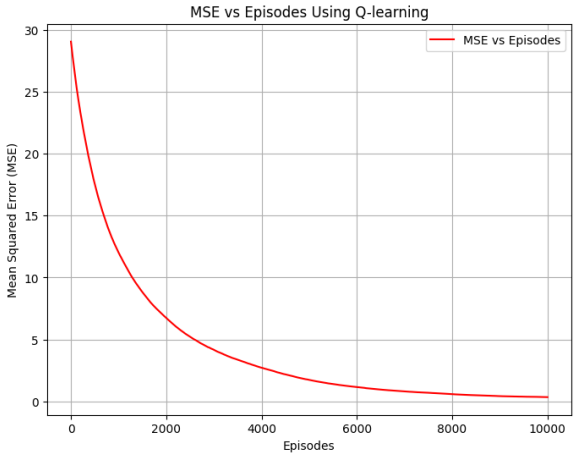

Figure 7: Learning Curve, Average MSE for 20 runs using Q-Learning algorithm

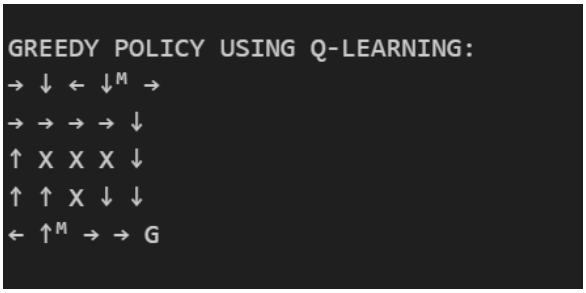

Figure 8: Final Greedy Policy using Q-Learning algorithm

4. SARSA on the Inverted Pendulum Swing-Up domain [20 Points, total]

You will now use the SARSA algorithm to solve the Inverted Pendulum Swing-Up domain. Notice that in the Inverted Pendulum Swing-Up domain, states are continuous. You will first have to discretize them to be able to deploy the standard/tabular version of SARSA[1]. You will have to make five main design

[1] **How to discretize the continuous states of the Inverted Pendulum Swing-Up domain:**

- Suppose you are discretizing a continuous state $s = (\omega, \dot{\omega})$, where $\omega \in [-\pi, \pi]$ is the angle of the pendulum relative to the upright position and $\dot{\omega}$ is pendulum's angular velocity.
- First, recall that to ensure that ω is in the specified interval, it has to be normalized according to the equation $\big((\omega + \pi) \bmod (2\pi)\big) - \pi$, where mod is the modulo operator used to get the remainder of a division.
- Similarly, $\dot{\omega}$ needs to be clipped to the interval from -MAX_SPEED to -MAX_SPEED via the command $\texttt{clip}\big(\dot{\omega}, \texttt{-MAX\_SPEED}, \texttt{MAX\_SPEED}\big)$
- After normalizing ω and clipping $\dot{\omega}$ as above, in order to ensure that they lie in the correct intervals, suppose you choose to discretize each component of the state into 3 bins.
- To do so, you will break down the interval of values of ω (from $-\pi$ to π) into three contiguous bins: the first bin would correspond to values of ω from -3.1415 to -1.0471; the second bin would correspond to values of ω from -1.0471 to 1.0471; and the third bin, to values of ω from 1.0471 to 3.1415.
- You would perform a similar discretization process over the state variable $\dot{\omega}$.
- Then, the final discretized version of a continuous state $s = (\omega, \dot{\omega})$ would correspond to a pair $(b_\omega, b_{\dot{\omega}})$ indicating the number of the bin, b_ω, on which the ω variable fell; and the number of the bin, $b_{\dot{\omega}}$, where the $\dot{\omega}$ variable fell.
- Notice that if you discretize a 2-dimensional continuous state (such as that of the Inverted Pendulum Swing-Up domain) into N bins, you will end up with N^2 discrete states.

decisions, here: the same four design decisions as in Question 2, and also an additional design decision (design decision v): the level of discretization to be used; i.e., the number of bins used to discretize the continuous state of the MDP.

(Question 4a. 5 Points) Report the design decisions *(i, ii, iii, iv, and v)* you made and discuss what was the reasoning behind your choices.

(i) I chose $\alpha = 0.1$ as a moderate/small α ensures stable learning while still allowing updates to reflect new information. Higher values risk oscillations, while lower values slow convergence. For the pendulum task, a higher α could lead to oscillations in Q-values due to the continuous nature of the state-action space, while a lower value would slow down convergence, making the learning process inefficient. This choice strikes a balance, enabling the agent to adapt its policy efficiently as it explores the dynamics of the pendulum environment.
(ii) I have initialized Q-function with zeros. While optimistic initialization (setting higher initial Q-values) encourages exploration, it was not chosen here to avoid potential overestimation of rewards early in training. In the pendulum task, starting with zero values ensures the agent learns realistic Q-values based on actual returns without being influenced by inflated expectations. This approach enables sufficient exploration of the pendulum's state-action space while allowing the agent to favor actions with higher Q-values, balancing exploration and exploitation effectively.
(iii) I have used ϵ-greedy exploration It balances exploration (trying new actions) and exploitation (choosing the current best-known action). By choosing random actions with probability $\epsilon = 0.05$, the agent explores the environment adequately while still favoring actions with higher Q-values. *(iv)* I am using fixed ϵ. It ensures a consistent balance between exploration and exploitation throughout training. This approach suits the pendulum's relatively smooth and predictable dynamics, as it maintains steady exploration of the discretized state-action space, preventing the agent from prematurely converging to suboptimal policies.
(v) The level of discretization was set to 75 bins each for the angle and angular velocity and 10 bins for the action space. This fine-grained discretization ensures that the SARSA algorithm captures the continuous dynamics of the pendulum accurately while balancing computational efficiency. Using fewer bins could oversimplify the state space, leading to suboptimal policies due to insufficient resolution. Conversely, excessively fine discretization would increase computational overhead without significant benefits for this problem. The chosen levels provide an appropriate trade-off, capturing key nuances of the pendulum's motion and enabling the agent to learn a precise and effective policy.

(Question 4b. 15 Points) Construct a learning curve where you show the number of episodes on the x axis, and, on the y axis, the return achieved by the agent when using the policy it learned after a given number of episodes. To construct this graph, run your algorithm 20 times and show the average curve across those runs (i.e., the average return achievable by the agent's policies, computed across 20 runs, as a function of the number of episodes). Show, also, the standard deviation associated with each of these points. Your graph should, ideally, look like that in Figure 10.

- Using a coarser discretization results in fewer states, but where each discrete state is a "less accurate" representation of the true underlying continuous state. Using a finer discretization results in a "more accurate" representation of the true underlying continuous state, but generates a larger state space.

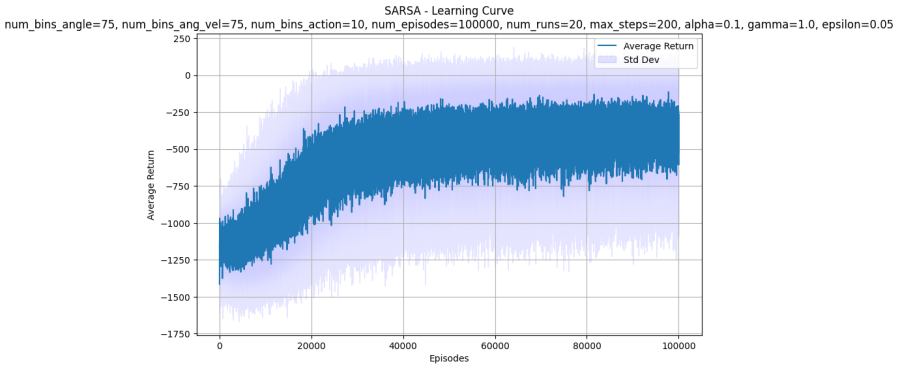

Figure 9: Learning Curve using SARSA algo for 20 runs

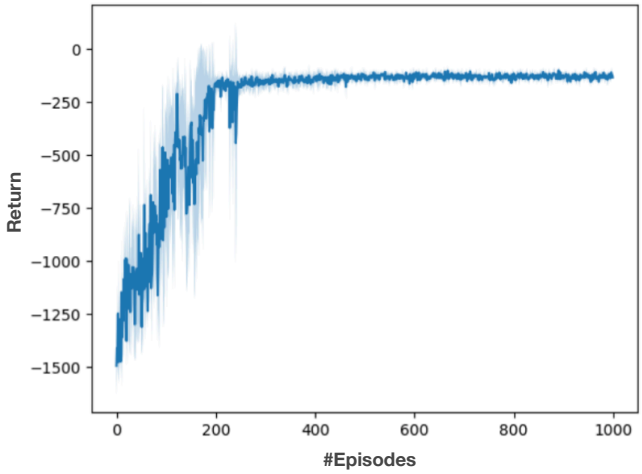

Figure 10: Example of a learning curve: average return (and the associated standard deviation) as a function of the number of episodes.

5. Q-Learning on the Inverted Pendulum Swing-Up domain [20 Points, total]

Answer the same questions as above (*4a* and *4b*), but now using the Q-Learning algorithm; i.e., you will be using Q-Learning to solve the Inverted Pendulum Swing-Up domain. The amount of points associated with questions *5a* and *5b*, will be the same as those associated with the corresponding items of Question 4.

(i) I chose $\alpha = 0.1$ to balance stability and adaptability in Q-Learning updates. In the pendulum task, where the state-action space is continuous and complex, a higher α could lead to erratic Q-value updates, especially in the early stages of learning, while a lower α would result in slow policy improvement. This moderate choice ensures that the Q-values converge effectively without oscillations, allowing the agent to track the reward structure of the pendulum environment as it learns to stabilize it.

(ii) The Q-function was initialized with zeros. This initialization prevents overoptimistic exploration, which can lead to biased learning early on in the pendulum's state space. Zero initialization ensures that the agent focuses on realistic Q-value estimation based on actual returns rather than being misled by inflated values. This is crucial in the pendulum task, as overestimating Q-values might prioritize unproductive exploration of irrelevant states.

(iii) I used ϵ-greedy exploration to manage the trade-off between exploration and exploitation in Q-Learning. With $\epsilon = 0.05$, the agent explores the pendulum's state-action space effectively while predominantly exploiting its current best-known policy. The low ϵ value is suitable for the pendulum's relatively

predictable dynamics, ensuring that sufficient exploration occurs without excessive randomness, which could slow convergence.
(iv) I opted for a fixed $\epsilon = 0.05$ to maintain a consistent exploration rate throughout training. This is well-suited to Q-Learning for the pendulum task, as the continuous state and action spaces necessitate ongoing exploration to prevent the agent from converging prematurely to a suboptimal policy. Fixed ϵ ensures that the agent continues exploring less-visited parts of the discretized space, aiding in thorough policy refinement.
(v) The state space was discretized into 75 bins each for the angle and angular velocity, and the action space into 10 bins. In Q-Learning, this discretization level balances the need to capture the pendulum's continuous dynamics accurately and computational feasibility. Fewer bins would result in coarse state-action resolution, impairing the agent's ability to learn precise policies, while more bins would increase computational demands unnecessarily. This choice ensures that the agent learns an effective policy for stabilizing the pendulum without undue computational overhead.

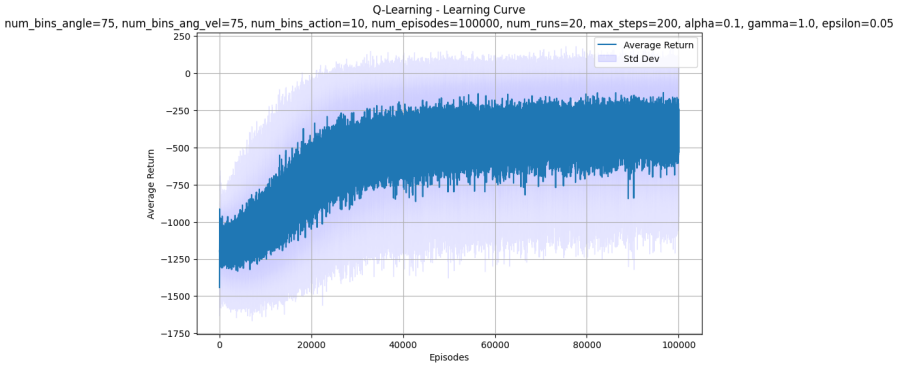

Figure 11: Learning Curve using Q-Learning algo for 20 runs

---

6. Further Extra Credit [16 Points, total]

(Question 6a. 8 Points) Answer question *(4b)* once again, but now using different discretization levels compared to the one you selected when solving Question 4. In particular, you should use the same hyperparameters as in Question 4: same value of α, q-function initialization strategy, exploration strategy, and method for controlling the exploration rate over time. Now, however, you will experiment with *different* discretization levels compared to the one you selected when solving Question 4. You should, in particular, run experiments with 25% fewer bins, 50% fewer bins, 25% more bins, and 50% more bins. Let us say, for example, that in Question 4 you decided to discretize the state features into N bins. In this extra-credit question, you should run the same experiments as in *(4b)*, but now using $0.5N$ bins, $0.75N$ bins, $1.25N$ bins, and $1.5N$ bins. Discuss the impact that each of these alternative discretization schemes had both on the performance of the policy learned by the algorithm, and on the (wall-clock) time needed to complete each of the four experiments, compared to the wall-clock time required to run the corresponding experiment in Question 4.

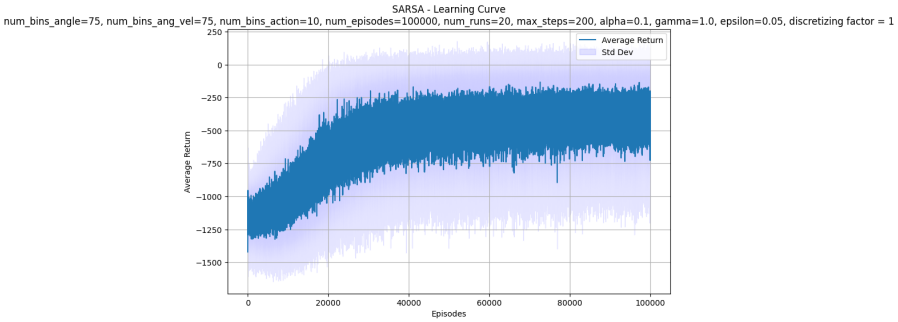

Figure 12: Learning Curve using SARSA algo for 20 runs when scaling factor is 1

```
Time Taken to run the algo (s) when discretization factor is 1: 65.93
```

Figure 13: Time taken to run SARSA algo for 20 runs when scaling factor is 1

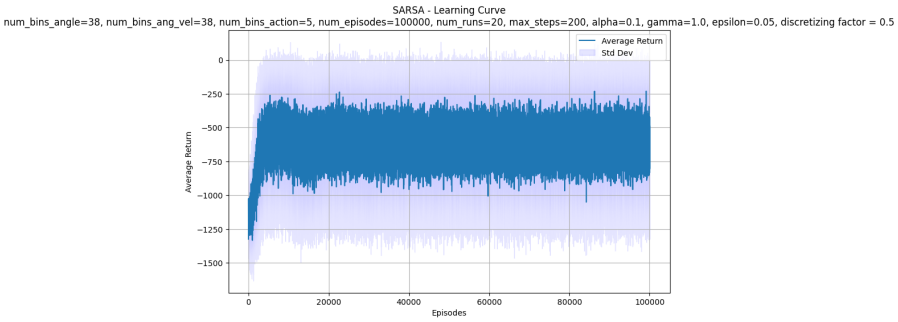

Figure 14: Learning Curve using SARSA algo for 20 runs when scaling factor is 0.5

```
Time Taken to run the algo (s) when discretization factor is 0.5: 58.42
```

Figure 15: Time taken to run SARSA algo for 20 runs when scaling factor is 0.5

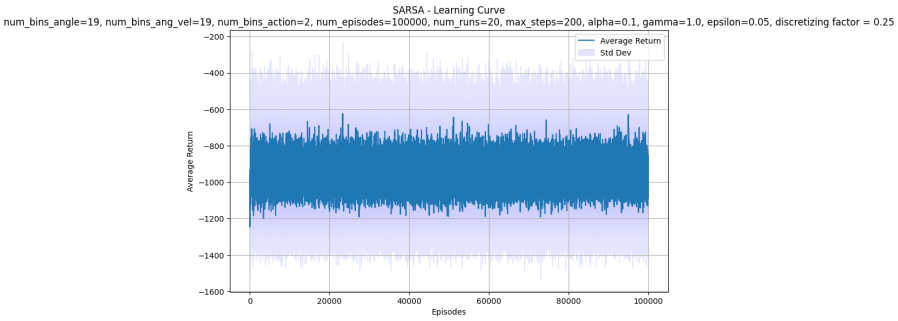

Figure 16: Learning Curve using SARSA algo for 20 runs when scaling factor is 0.25

```
Time Taken to run the algo (s) when discretization factor is 0.25: 52.24
```

Figure 17: Time taken to run SARSA algo for 20 runs when scaling factor is 0.25

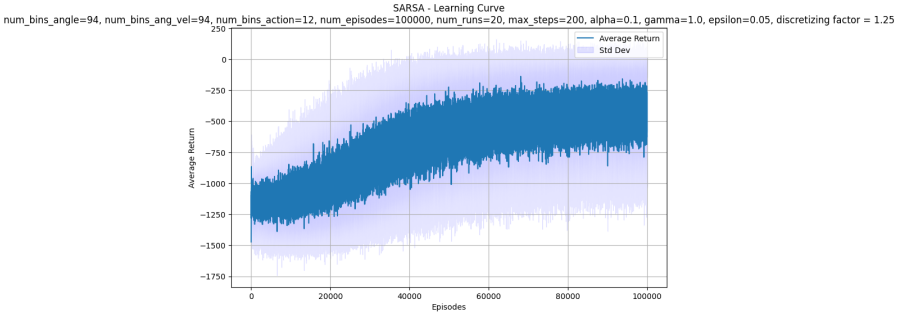

Figure 18: Learning Curve using SARSA algo for 20 runs when scaling factor is 1.25

```
Time Taken to run the algo (s) when discretization factor is 1.25: 86.26
```

Figure 19: Time taken to run SARSA algo for 20 runs when scaling factor is 1.25

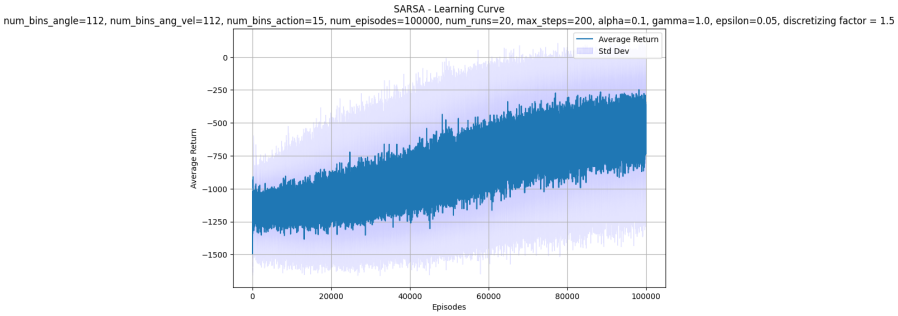

Figure 20: Learning Curve using SARSA algo for 20 runs when scaling factor is 1.5

```
Time Taken to run the algo (s) when discretization factor is 1.5: 88.46
```

Figure 21: Time taken to run SARSA algo for 20 runs when scaling factor is 1.5

From the above figures we see that as we decrease the scaling factor, the time taken to converge decreases and increases as we increase the scaling factor i.e. when number of bins are less, the code converges quickly compared to when the number of bins are more. In terms of the performance of the policy, we can see from the graph that as we decrease the number of bins, the policy performance slightly decreases probably due to the fact that coarse discretization might fail to capture subtle nuances in the pendulum's dynamics. When we increase the scaling factor though i.e. when number of bins increase, although it takes more time to converge but we get somewhat better results which provides a more detailed representation of the state-action space enhancing the potential for learning a more accurate and nuanced policy

(Question 6b. 8 Points) Answer question *(4b)* once again, but now using *two* different ways of initializing the q-function optimistically. You should use the same hyperparameters as in Question 4: same value of α

and discretization levels. Now, however, you will experiment with *different* ways of optimistically initializing the q-function. Assume that $M = -30$ is the maximum return achievable in the Inverted Pendulum Swing-Up domain. First, deterministically initialize all entries of the q-function with the optimistic value 0.0. Next, stochastically initialize all entries of the q-function with numbers drawn uniformly at random from the interval $\Big[0.5\,|M|,\ 1.5\,|M|\Big]$. In this extra-credit question, you should run the same experiments as in *(4b)*, but now using these alternative ways of optimistically initializing the q-function. **Importantly, when running these experiments you should set the exploration rate to zero (i.e., $\epsilon = 0$, always)**. Discuss the impact that these q-function initialization strategies had both on the performance of the policy learned by the algorithm, and on the (wall-clock) time needed to complete each of the four experiments, compared to the wall-clock time required to run the corresponding experiment in Question 4.

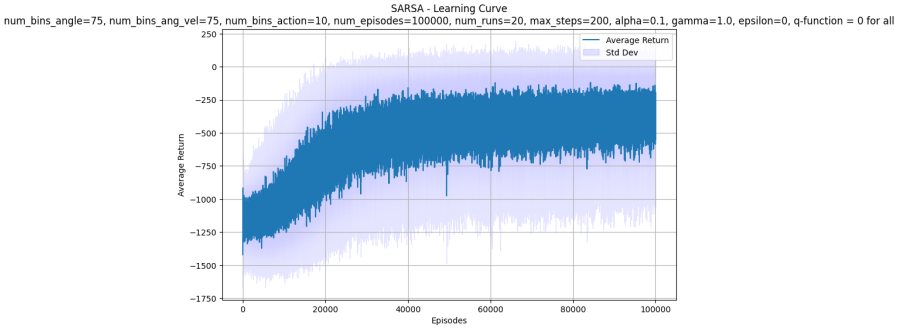

Figure 22: Learning Curve using SARSA algo for 20 runs when Q-Function is 0 initialized

```
Time Taken to run the algo (s) having q-function with the optimistic value 0.0 : 70.49
```

Figure 23: Time taken to run SARSA algo for 20 runs when Q-Function is 0 initialized

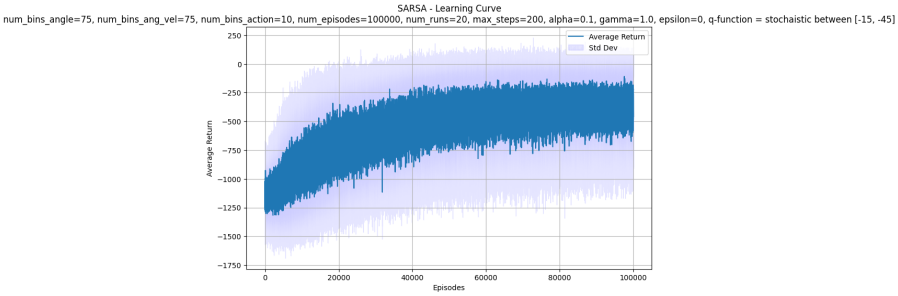

Figure 24: Learning Curve using SARSA algo for 20 runs when Q-Function is stochastically initialized between [-15, -45]

```
Time Taken to run the algo (s) having stochaistic Q-Function between [-15, -45]: 64.26
```

Figure 25: Time taken to run SARSA algo for 20 runs when Q-Function is stochastically initialized between [-15, -45]

Based from the figures we see that when q is 0 initialized with $\epsilon = 0$ it takes about 70s and when its stochastically initialized between $[-15, -45]$. While the stochastic initialization of the Q-function can speed up convergence by encouraging more aggressive exploration and learning, it may lead to slight biases in the learned policy. On the other hand, zero initialization provides a more neutral starting point but leads

to slower convergence. This is very slight difference not much though, on running it multiple times i found that more or less they take the same time only having no effect as such. From the graph it seems that the 0 initialized q-function and stochastically initialized q-function give more or less same performance/result.