Assignment 5 Manual

Reinforcement Learning

Faculty of Engineering

McMaster University

November 19, 2025

Assignment 5: Reinforcement Learning*

Due Date: December 6, 11:59 pm
Assessment: 5% of the course mark
Late submissions are not accepted.
If you submit an MSAF, the weight is transferred to the final exam.

1 Introduction

In this assignment, you will develop, train and compare 3 reinforcement learning (RL) agents to solve the Lunar Lander problem. Train your RL agents using Sarsa, Q-learning, and Expected Sarsa. You are expected to design, train, and fine-tune each RL agent to achieve optimal performance in the provided environment. Upon completion, you must submit the Python code used for training each agent along with a detailed report, on **Avenue to Learn**.

2 Environment Specifications and Setup Instructions

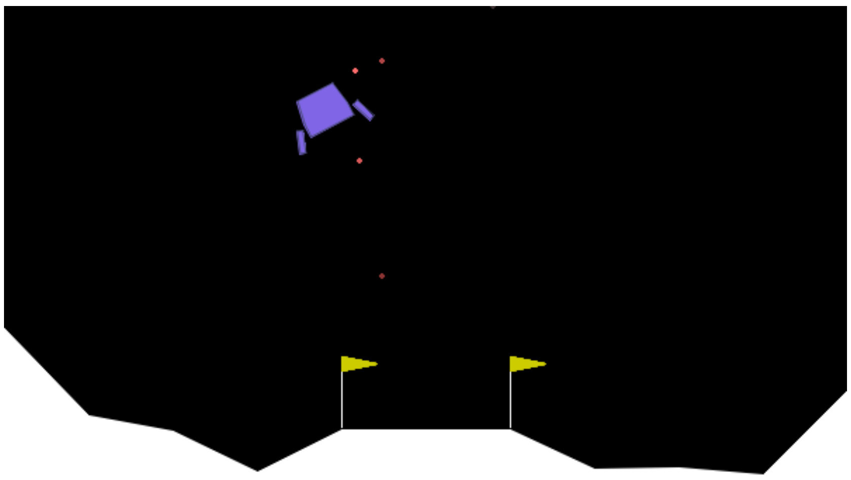

Figure 1: Visualization of the Lunar Lander environment [1].

2.1 Environment Specifications

The lunar lander environment simulates the physics of a spacecraft attempting to land on the Moon's surface. It is based on the `LunarLander-v3` environment from Gymnasium.

Further details about the environment can be found in the Gymnasium documentation [1].

Index	Variable	Range
0	**x_position**	$[-2.5, 2.5]$
1	**y_position**	$[-2.5, 2.5]$
2	**x_velocity**	$[-10.0, 10.0]$
3	**y_velocity**	$[-10.0, 10.0]$
4	**angle**	$[-2\pi, 2\pi]$
5	**angular_velocity**	$[-10.0, 10.0]$
6	**left_leg_contact**	$\{0, 1\}$
7	**right_leg_contact**	$\{0, 1\}$

Table 1: State variables and their updated ranges

2.1.1 State Space

The environment has a continuous state space represented by an 8-dimensional vector:
Example of a received state vector:

$$\begin{aligned}\text{state} = [&0.012, \quad 1.023, \quad -0.850, \quad -0.125, \\ &0.045, \quad 0.003, \quad 0, \quad 0]\end{aligned}$$

Each element corresponds to the variables listed above.

Note: You have to discretize the state space. You may use the provided `StateDiscretizer` class in `agent_template.py`.

2.1.2 Action Space

The action space is discrete with four possible actions:

Action	Description
0	Do nothing
1	Fire left orientation engine
2	Fire main engine
3	Fire right orientation engine

Table 2: Available actions

2.2 Resources

For an introduction to using Gym environments and how to use them in your RL training and testing loops, refer to this tutorial: Basic Gym Environment Usage.

*This assignment is an adaptation of Assignment 5 prepared by Hazem Taha in 2024.

3 Agent Template

The `agent_template.py` file provides a structured starting point for implementing a reinforcement learning agent to solve the Lunar Lander environment. Below is an overview of the main components in this template (check the provided script for the complete template)[1]:

```python
import gymnasium as gym
from state_discretizer import StateDiscretizer

class LunarLanderAgent:
    def __init__(self):
        pass

    def select_action(self, state, testing=True):
        pass

    def train(self, num_episodes):
        pass

    def update(self, state, action, reward, next_state, done):
        pass

    def test(self, num_episodes):
        pass

    def save_agent(self, file_name):
        pass

    def load_agent(self, file_name):
        pass

if __name__ == '__main__':
    agent = LunarLanderAgent()
    agent_model_file = 'model.pkl'
    num_training_episodes = 1000   # Choose an appropriate number according
     to your need
    agent.train(num_training_episodes)
    agent.save_model(agent_model_file)
```

Listing 1: Agent Template Overview

[1]This template was prepared by Hazem Taha in the Fall of 2024.

Key Components:

- `__init__()`: Initializes the environment, the agent's model (i.e., the Q-table), and the optional state discretizer. Add any necessary initialization for model parameters here.
  - We recommend implementing ϵ decay (exploration rate decay) by starting with a high value for ϵ and gradually decreasing it using a decay factor of your choice until reaching a minimum threshold (e.g., 0.01). This approach allows your agent to explore extensively in the early stages of training and gradually shift towards exploiting its learned knowledge.
  - If you use the provided `state_discretizer`, ensure that the learning rate, α, is scaled appropriately. A good practice is to set α to a value between 0 and $1/\text{num\_tilings}$, as shown below:

```
        self.alpha = alpha / self.state_discretizer.num_tilings
    # Learning rate per tiling

```

- `select_action(state, testing)`: Decides an action based on the current state, supporting both training and testing modes. In training mode, it uses an epsilon-greedy strategy to balance exploration and exploitation, while in testing mode, it relies on a purely greedy policy for deterministic action selection.

- `train(num_episodes)`: Contains the main training loop where the agent learns over multiple episodes. This function should also track performance (average of the previous 100 episodes cumulative rewards) and autosave the best-performing model.

- `update(state, action, reward, next_state, done)`: Updates the agent's knowledge, i.e., the Q-table, using the observed transitions.

- `test(num_episodes)`: Contains the testing loop where the trained agent interacts **greedily** with the environment over multiple episodes to collect rewards without learning. At the end of the testing phase, the function computes the average of the collected returns (cumulative rewards). You can use this function to compare the performance of your agents.

- `save_agent(file_name)`: Saves the Q-table of the agent to a file for later use. This is particularly useful for preserving the best-performing agent.
  - If you use the provided `state_discretizer`, make sure to also save its hash table `state_discretizer.iht.dictionary`. We have implemented this functionality in the template for you!
  - Within your training function, we recommend automatically calling `save_agent()` whenever your agent achieves better training performance (as measured by the average return over the last 100 episodes) and the exploration rate ϵ is sufficiently close to zero. The latter condition ensures that the detected performance accurately reflects the agent's learned behavior rather than being influenced by random exploratory actions. This approach not only helps to save the best-performing agent but also avoids saving models affected by performance fluctuations during training.

- `load_agent(file_name)`: Loads the previously saved Q-table, enabling evaluation or continued training from a saved state. If you used our `state_discretizer`, make sure to also load the saved hash table. We implemented it in the template for you.

The template supports Q-learning (with state discretization) and other function approximation methods.You may define other supporting functions in addition to the existing ones.

To enable state discretization, uncomment the `StateDiscretizer` relevant code snippets in the template. The `StateDiscretizer` employs **Tile Coding**—a sparse coding technique that efficiently transforms continuous input states into discrete indices. These indices allow the agent to map continuous states into discrete representations, making Q-learning viable for environments with continuous state spaces.
Once you obtain the indices from `StateDiscretizer`, use them to retrieve a vector of Q-values for each action from `self.q_table`. Sum these values to compute the Q-value for a given action, as shown below:

```
state_indices = self.state_discretizer.discretize(state)
Q_value = np.sum(self.q_table[action][state_indices])
```

4 Technical Requirements

- Train your agents until you achieve satisfactory performance. Then compare their performance on the same instance of the environement.
- To achieve the maximum mark, your trained agent should achieve an average return of at least 100 over 100 episodes (when tested).

5 Assignment Guidelines

5.1 Submissions

- **Code Files:** Submit your **best working code** along with any associated agent model files (e.g., Q-table). Ensure that your code is well-documented, properly organized, and includes any instructions necessary to run it.
- **Demo Video:** A demo video showcasing your code with explanations.
- **Report:** A report (a pdf file) that includes:
  - **Introduction:** Brief overview of the assignment objectives.
  - **Methodology:** Describe the reinforcement learning algorithm(s) used. Include a detailed explanation of each approach, including any modifications or enhancements made to the base algorithm.
  - **Results:** Present at least **three** learning curves for each agent that illustrate the performance of your agent over time under different configurations (e.g., varying hyperparameters or exploration strategies). Each learning curve should plot the average cumulative reward (return) over the past 100 episodes during training.
  - **Discussion:** Analyze and interpret your results. Discuss the impact of different hyperparameters or techniques used.

- **Conclusion:** Summary of findings and suggestions for potential improvements if more time was available.

6 Tips to Improve Your Agents' Performance

To develop better-performing agents, you may consider the following strategies:

6.1 Adjusting the Tile Coding Parameters

Given the continuous nature of the state space, simple discretization may not be sufficient. A **Tile Coding** discretization solution is provided within the *StateDiscretizer*. Tile Coding is a feature representation method that helps your agent generalize across similar states by representing the state space using overlapping tiles. You are encouraged to research Tile Coding further and adjust the parameters within `state_discretizer.py` to achieve better results.

6.2 Adjusting Learning Parameters

Carefully tuning your learning parameters is essential for achieving better agent performance. Consider adjusting the following:

- **Learning Rate** (α): Controls the size of the update steps during learning. A smaller learning rate can help stabilize updates, especially in complex environments.
- **Discount Factor** (γ): Determines the weight given to future rewards relative to immediate rewards. Balancing this parameter is key for aligning the agent's objectives with the desired behavior.
- **Exploration Rate** (ϵ): Governs the trade-off between exploration and exploitation. Ensuring ϵ decays appropriately over time is crucial.
- **Epsilon Decay**: Controls the rate at which the exploration rate ϵ decreases as training progresses. Choosing an optimal decay rate is important.

Adjusting these parameters to find an optimal balance can significantly impact your agent's learning stability and final performance.

6.3 Additional Tips

- Take advantage of online resources and tutorials to expand your knowledge and strengthen your understanding of advanced RL concepts.
- Experiment with different hyperparameters and **different algorithms** to find the optimal setup for your agent.
- Use tools like `matplotlib` to monitor training progress. Focus on tracking the average cumulative rewards over several recent episodes (100), rather than relying on single-episode performance. An easy way to do this is by using the `deque` data structure from Python's `collections` module.

- Try testing your agent in an environment with **rendering** enabled to visualize its performance! For example, you can use the following code snippet:

```
        env = gym.make("LunarLander-v3", render_mode="human")

```

References

[1] Gymnasium Project. Gymnasium lunar lander documentation. https://gymnasium.farama.org/environments/box2d/lunar_lander/, 2024. Accessed: 2024-11-12.