Assignment 5 Manual

4SL4: Reinforcement Learning Hackathon 2024

Faculty of Engineering

McMaster University

Teaching Assistant: Hazem Taha
Instructor: Sorina Dumitrescu

November 20, 2024

Assignment 5: 4SL4 RL Hackathon 2024

Due Date: December 1, 11:59 pm

1 Introduction

In this assignment, teams of 2 or 3 students will develop and train a reinforcement learning (RL) agent to tackle a specified challenge. Each team is expected to design, train, and fine-tune their RL agent to achieve optimal performance in the provided environment. Upon completion, teams must submit the Python code used for training their agent, along with a detailed report, on **Avenue to Learn**. Teams may also participate in the hackathon by submitting their trained agent to a designated server for evaluation, with multiple attempts allowed to improve performance.

This assignment carries a total weight of **10%**, divided into **6% Base Grade**—awarded for submitting a fully functioning RL agent that meets minimum performance standards, along with a comprehensive report—and up to **4% Bonus**, calculated based on the team's performance in the competition, with **$300** in Total Prizes!

2 Hackathon Theme

In 2040, McMaster University was chosen to lead an ambitious mission to the Moon: designing an autonomous lunar lander that can navigate and land safely with minimal human guidance. To bring this challenge to life on campus, we are hosting the **4SL4: Reinforcement Learning Hackathon 2024**, where engineering students will apply cutting-edge reinforcement learning techniques to solve a dynamic, real-world-inspired landing problem. This is an opportunity to push your boundaries in the field of RL and showcase your innovative solutions!

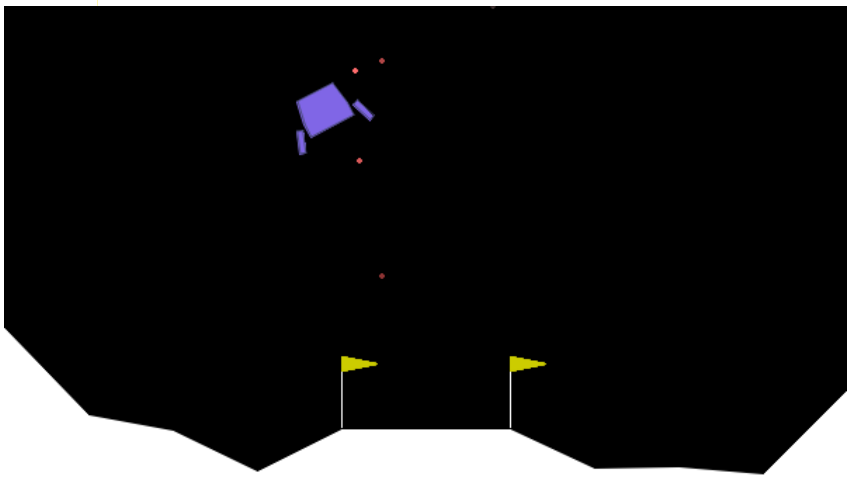


Figure 1: Visualization of the Lunar Lander environment [1].

3 Team Formation

- Teams must consist of **2 or 3 students**.
- Each team must select a unique **Team Name** and fill this form to declare registration. The registration deadline is Tuesday, November 19, 2024.
- Upon registration, each team will receive a **TEAM_SECRET_CODE** which is confidential and must be kept secure. This code is essential for submitting your solutions to the server.

4 Environment Specifications and Setup Instructions

4.1 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].

4.1.1 State Space

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

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

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: If you choose to implement a tabular value-based method, such as Q-learning, you must use the provided `StateDiscretizer` class in `agent_template.py`. However, if you opt for a function approximation technique, you may work directly in the continuous state space.

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

Table 2: Available actions

4.1.2 Action Space

The action space is discrete with four possible actions:

When submitting actions to the server, ensure the actions you are iteratively submitting are integers within the range $[0, 3]$, otherwise the server will reject the action.

4.1.3 Setting Up the Environment

To set up the necessary environment for this assignment, you need to set up your local development environment. This involves installing the necessary dependencies and ensuring that your setup is compatible with the evaluation server. We recommend using **Anaconda** for managing your Python environments.

4.2 Using Anaconda

1. **Create a Conda Environment:**

   ```
   conda create -n hackathon_env python=3.9.20
   ```

2. **Activate the Conda Environment:**

   ```
   conda activate hackathon_env
   ```

3. **Install Necessary Libraries:**

   Install the required dependencies using `pip`. Do it one at a time to make sure they are installed successfully:

   ```
   pip install numpy
   pip install swig               # For the environment
   pip install gymnasium[box2d]   # ..
   pip install torch  # If you're using PyTorch for your agent
   pip install python-socketio    # For server communications
   pip install python-socketio[client]
   ```

4.3 Importing the Provided Conda Environment

To simplify the setup process, we will provide you with an exported Conda environment file named `RL_Hackathon.yaml`. You can import this environment using the following command:

```
conda env create -f RL_Hackathon.yaml
```

After importing, activate the environment:

```
conda activate RL_Hackathon
```

4.4 Additional 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.

5 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:

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

class LunarLanderAgent:
    def __init__(self):
        self.env = gym.make('LunarLander-v3')
        # self.state_discretizer = StateDiscretizer(self.env)
        # self.q_table = [np.zeros(self.state_discretizer.iht_size) for _
    in range(self.num_actions)]

        # Set learning parameters
        # ...

    def select_action(self, state):
        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

Key Components:

- `__init__()`: Initializes the environment, the agent's model (e.g., Q-table or neural network), and the optional state discretizer for Q-learning. 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/num_tilings, as shown below:

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

```

- `select_action(state)`: Defines the action selection policy. Use this function to implement an action strategy, such as epsilon-greedy for Q-learning. The function should operate in training and testing modes, where in testing you will need to shut off epsilon-greedy 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 (e.g., Q-table or neural network) using observed transitions. For Q-learning, it updates the Q-table based on the transition.

- `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). This function is designed to help you evaluate your agent locally before submitting it to the server, giving you an estimate of the expected score.

- `save_agent(file_name)`: Saves the Q-table or the neural network of the agent to a file for later use and server submission. This is particularly useful for preserving the best-performing agent.
  - If you used our `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 or the neural network of the agent, 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 in the template for you!

The template supports Q-learning (with state discretization) and other function approximation methods.You may define other supporting functions but make sure to utilize the existing ones for their corresponding tasks.

To enable state discretization for Q-learning, 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:

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

6 Submission Script

The `submit_agent.py` script is provided to handle the submission of your agent to the evaluation server for participation in the hackathon. This script allows your trained agent to interact with the server without requiring you to manage server requests or responses manually.

Instructions

To prepare your submission:

1. **Update Team Credentials**:

   Open `submit_agent.py` and replace the placeholders with your team's credentials provided upon registration:

   ```python
   TEAM_NAME = 'Your Team Name'
   TEAM_SECRET_CODE = 'Your Secret Code'

   ```

2. **Specify the Server URL**:

   Ensure that the `SERVER_URL` is set to the correct server address provided by the hackathon organizers:

   ```python
   SERVER_URL = 'http://srv-cad.ece.mcmaster.ca:65000'

   ```

3. **Load Your Trained Agent**:

   Modify the code to load your trained model.

```
agent = YourAgentClass()
agent_model_file = 'path_to_your_saved_model.pth'  # Specify your
    trained agent's model file
agent.load_model(agent_model_file)

```

4. **Run the Submission Script**:

   Execute the script to submit your agent:

```
python submit_agent.py

```

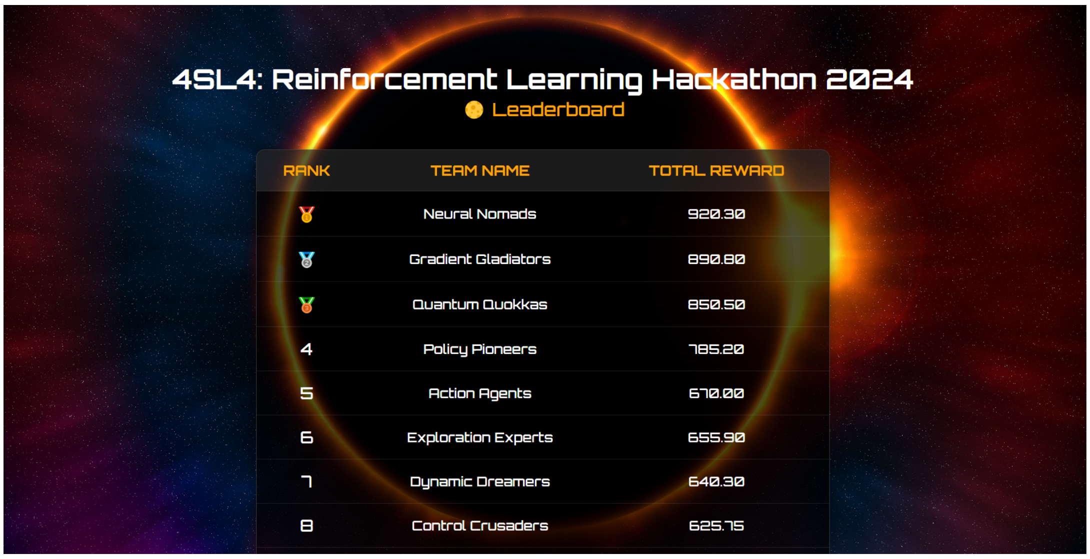
5. **View the Leaderboard**:

   After submission, the leaderboard should open automatically in your default web browser. You can also access it anytime through your browser at:

```
http://srv-cad.ece.mcmaster.ca:65000/leaderboard

```

   Figure 2 illustrates how the leaderboard displays each team's rank and score after a successful submission.



RANK	TEAM NAME	TOTAL REWARD
🥇	Neural Nomads	920.30
🥈	Gradient Gladiators	890.80
🥉	Quantum Quokkas	850.50
4	Policy Pioneers	785.20
5	Action Agents	670.00
6	Exploration Experts	655.90
7	Dynamic Dreamers	640.30
8	Control Crusaders	625.75

Figure 2: Leaderboard showing team rankings and scores after successful submissions.

Important Notes

- **Accessing the Server:** To access the server, you must either be connected to the university network or use a **VPN** provided by the university.
- **One Active Submission at a Time:** Teams are permitted to have only **one active submission** at any given time. While multiple submissions are allowed throughout the hackathon (ending on December 1), they must be sequential, not concurrent. Resubmitting the same solution is prohibited; teams must make changes before attempting a new submission.

- **Handling Interrupted Submissions:** If your submission is interrupted for any reason, please wait a few minutes before submitting again to allow your session to terminate and reset, enabling a new submission.
- **Do Not Modify Server Communication Code:** The server communication is fully implemented. You do **not** need to modify any code related to server interaction.
- **Agent Instantiation:** Upon submission, your `LunarLanderAgent` class will be instantiated, and the trained agent model file will be loaded. Ensure that the newly created agent operates as intended, relying solely on the saved model file and not on any external variables or hyperparameters from prior training.
- **Agent Testing:** During submission, your agent will be used solely for deciding actions by calling the `select_action` method. Make sure the function is set up to select valid actions based on the states provided by the server. Disable ϵ-greedy exploration outside of training to ensure deterministic behavior.
- **Action Space Compliance:** Your `select_action(state)` method must return a valid action integer between `0` and `3`, which corresponds to the available action space for the Lunar Lander environment.

7 Technical Requirements

- Train your agent **locally** on your own devices until you achieve satisfactory performance and are ready to submit your solution. Avoid submitting solely to test performance.
- Ensure that your agent is prepared for inference during submission by loading any required model files, and verify that the `select_action` method returns valid actions as required by the environment.
- You may choose any type of reinforcement learning algorithm to solve the environment, provided that you implement the algorithm manually. The use of pre-built reinforcement learning libraries is not permitted.

8 Assignment Guidelines

8.1 Submissions

- **Deadline:** All team members must submit their assignment by **December 1, 11:59 pm** on **Avenue to Learn** under the assignment titled **Assignment 5**.
- **Code Submission:** Each team must submit their **best working code** along with any associated agent model files (e.g., Q-table, neural network weights). Ensure that your code is well-documented, properly organized, and includes any instructions necessary to run it.
- **Individual Submission:** Although this is a team assignment, **each team member must submit** the assignment individually. Only one submission per team member is required.

- **Base Grade Requirement:** The code submitted on Avenue is essential for obtaining the **base grade** for the assignment. Participation in the hackathon server submissions is for the bonus competition and does not replace the requirement to submit your code on Avenue.

8.2 Report Writing

Your team should prepare a report that includes:

- **Introduction:** Brief overview of the assignment objectives.
- **Methodology:** Describe the reinforcement learning algorithm(s) you chose to implement (e.g., Q-learning, Deep Q-Networks). Include a detailed explanation of your approach, including any modifications or enhancements made to the base algorithm.
- **Results:** Present at least **four** learning curves that illustrate the performance of your agent over time. 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, algorithms, or techniques used.
- **Conclusion:** Summary of findings and suggestions for potential improvements if more time was available.

8.3 Competition Participation

- To participate in the completion, a team needs to submit their solution to the server at least once. Teams can make an **unlimited** number of submissions to the server throughout the hackathon duration to improve their ranking.
- Each team member will receive a calculated **bonus**, which will be added to their base assignment grade. Additionally, the top 3 teams will share a total cash prize of **$300**.

9 Assessment

- **Base Grade (6%):** To achieve full marks for the base grade, each team must submit:
  - A working code implementation that achieves an average cumulative reward (return) of at least 100 over 100 episodes.
  - The detailed report as specified in the guidelines.
- **Competition Bonus (up to 4%):** Teams can earn up to an additional 4% in bonus marks based on the performance of their submitted solution, calculated with the following formula:

$$Y = \max\left(\min(0.03X - 4.1, 4), 0\right)$$

  where Y represents the bonus percentage, and X is the average cumulative reward (return) over 100 episodes achieved by your agent.

- **Prizes:** In addition to bonus marks, teams will compete for **$300 in Total Prizes**, distributed as follows:
  - $150 for the 1st place team.
  - $100 for the 2nd place team.
  - $50 for the 3rd place team.

This assessment structure allows teams to achieve a maximum of 10% (6% base + 4% bonus), with the bonus incentivizing high-performing agents.

10 Tips to Improve Your Agent Performance

To develop a better-performing agent, consider the following strategies:

10.1 Enhancing Value-Based Methods

If you choose to implement a tabular Q-learning solution, you may consider the following tips to improve your agent's performance in the continuous state space of the Lunar Lander environment:

10.1.1 Adjust the Tile Coding

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 optimal results.

10.1.2 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.

10.2 Using Function Approximation - Deep Q-Networks (DQN)

To achieve better performance than the tabular Q-learning approach, consider implementing a **Deep Q-Network (DQN)**. Unlike Q-learning, DQN leverages neural networks to approximate the Q-value function [2], effectively handling continuous state spaces without requiring discretization. This approach is generally more powerful and can lead to significantly higher scores than traditional tabular Q-learning.

For improved stability and performance, you may also integrate enhancements such as **Experience Replay** and **Double DQN**:

- **Experience Replay**: Stores past experiences and samples them randomly during training, which helps break the correlations between sequential experiences and stabilizes learning.
- **Double DQN**: Addresses the issue of overestimation bias in Q-learning by decoupling action selection from action evaluation, leading to more accurate Q-value estimates.

We highly recommend using **PyTorch** to quickly and efficiently implement your DQN, as demonstrated in the *PyTorch Crash Course* session. PyTorch provides flexible tools and built-in support for GPU acceleration, which can significantly speed up training and make DQN implementation more manageable.

10.3 Exploring Policy-Based Methods

For those interested in delving deeper, consider exploring **Policy-Based Methods** and **Actor-Critic Methods**, such as Deep Deterministic Policy Gradient (DDPG) and Proximal Policy Optimization (PPO). These advanced methods are capable of yielding highly optimized solutions, although they come with additional complexity in implementation.

10.4 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 network architectures 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, rather than relying on single-episode performance.

  One way to do this efficiently is by using the `deque` data structure from Python's `collections` module. A `deque` is an optimized data structure for fast additions and removals from both ends, allowing you to maintain a fixed-size window of the latest n episode results and easily calculate the mean to evaluate current performance.
- Collaborate effectively with your team members to brainstorm ideas, troubleshoot issues, and refine your approach.

References

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

[2] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2nd edition, 2018.

We look forward to seeing your innovative solutions and wish you the best of luck in the Lunar Lander Hackathon!

Prepared by:
Hazem Taha
Teaching Assistant
Department of Electrical and Computer Engineering
McMaster University

Approved by:
Sorina Dumitrescu
Professor
Department of Electrical and Computer Engineering
McMaster University