

Lab 5: Reinforcement Learning

This lab’s objective is to use Q-learning to control a rocket in a continuous simulated environment. You will first implement a controller that keeps the rocket upright, and then extend it so that the rocket learns to hover. The lab uses table-based Q-learning, which requires you to represent continuous sensor values as a finite set of states.

Deliverable Demonstrate the angle controller and the hover controller to a teaching assistant. Also submit your completed `qlearning_controller.py` and `state_and_reward.py` files from the **exercise** folder, together with a short PDF report. The report should describe your state representations, rewards, and actions, and contain your answers to the questions in this document.

The final section on Deep Q-learning is optional. You do not need to complete it for the demonstration or the report.

Preparation

Read Chapter 17.1 in the 3rd and 4th editions of the course book (Chapter 16.1 in the Global Edition) about Markov Decision Processes. Also read Chapters 21.1–21.3 in the 3rd edition, Chapters 22.1–22.3 in the 4th edition, or Chapters 23.1–23.3 in the Global Edition about reinforcement learning. The last chapter in each of these ranges describes the Q-learning algorithm and is particularly important for this lab. Some versions of the book use different numbering. All references here are to the reinforcement learning chapter.

The Rocket Simulator

The simulator contains a rocket with three engines: left, middle, and right. Each engine can be switched on or off. There is no partial throttle. The rocket provides sensors for its position, velocity, and angle.

Setup

Install Python 3.10, 3.11, or 3.12 and `uv`. Installation instructions for `uv` are available at <https://docs.astral.sh/uv/getting-started/installation/>.

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

Open a new terminal and check that `uv` is installed with:

```
uv --version
```

Open a terminal in the repository and run:

```
cd lab5-ReinforcementLearning
uv sync
uv run python simulator.py
```

The first `uv sync` command creates a virtual environment and installs the required Python packages.

Controls

The simulator can be controlled with the following keys:

- **Arrow up:** Fire the middle engine.
- **Arrow left:** Fire the left engine.
- **Arrow right:** Fire the right engine.
- **Arrow down:** Reduce the velocity during manual testing.
- **Space:** Switch between manual control and the learning controller.
- **P:** Pause the learning controller.
- **O:** Start the learning controller.
- **E:** Switch exploration on or off.
- **R:** Reset the rocket.
- **G:** Switch graphics on or off. The simulator runs at maximum speed when graphics are off.
- **+ and -:** Speed up or slow down the simulation.

The buttons below the simulator provide the same controls.

Code Structure Overview

All student files are in the `exercise` folder. Work only in the following files:

QLearningController (`exercise/qlearning_controller.py`): This class reads the rocket sensors, selects an action, controls the engines, and stores the Q-table. The method `tick` is called once per simulation step.

StateAndReward (`exercise/state_and_reward.py`): This class contains the state and reward functions for the angle and hover controllers. It also contains a function for dividing a continuous interval into discrete values.

Deep Q-learning (`exercise/deep_q_learning.py`): This file is used only for the optional image-based exercise.

Simulator (exercise/simulator.py): This file contains the graphical interface and the rocket physics. You do not need to modify it.

The sensor objects provide a `getValue()` method. An engine is controlled with `engine.setBursting(True)` and `engine.setBursting(False)`. An engine remains on until the controller switches it off.

Part I: Tutorial

Start by finding where `QLearningController` reads the `x`, `y`, `vx`, `vy`, and `angle` sensors, and where it gets access to the three engines.

Tasks

1. Run the simulator and fly the rocket manually with the arrow keys.
2. Print the values of `angle`, `vx`, and `vy` from the controller loop.
3. Write a simple rule that switches the middle engine on when the vertical velocity passes a threshold.
4. Check that the engine remains on until your controller switches it off.

Part II: Q-learning Angle Controller

In this part, the task is to keep the rocket facing upward. The state only needs to contain the rocket angle. Since table-based Q-learning requires a finite set of states, divide the continuous angle into discrete intervals. The program stores each state as a string.

The Q-learning update is

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left(r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right).$$

The algorithm is described in Chapter 21.3, Figure 21.8 in the 3rd edition and Chapter 22.3, Figure 22.8 in the 4th edition of the course book.

Here, $Q(s, a)$ estimates the total future reward after taking action a in state s . The immediate reward is r , the discount factor is γ , and the learning rate is α .

For example, suppose that the angle is divided into intervals and the Q-table contains the following entries. The numbers are only examples.

Angle state	Action	Q-value
$0^\circ \leq \theta < 60^\circ$	No engine	7.0
$0^\circ \leq \theta < 60^\circ$	Middle engine	4.0
$0^\circ \leq \theta < 60^\circ$	Left engine	18.0
$0^\circ \leq \theta < 60^\circ$	Right engine	10.0

In this example, the greedy action is to fire the left engine because 18.0 is the largest Q-value for this state. The value 18.0 estimates the discounted future reward after selecting this action. It is not the immediate reward for firing the engine.

Assume that the controller observes a reward of 8.0, the largest Q-value in the next state is 20.0, $\alpha = 0.2$, and $\gamma = 0.95$. The update becomes

$$Q(s, a) \leftarrow 18.0 + 0.2 (8.0 + 0.95 \cdot 20.0 - 18.0) = 19.8.$$

The observation moves the old estimate towards the reward plus the discounted value of the best action in the next state.

The controller uses epsilon-greedy exploration. With probability epsilon, it selects a random action. Otherwise, it selects the action with the highest Q-value. Press E to switch exploration off when you evaluate the controller.

Exercise 1: State and reward

1. Implement the angle state and reward functions in `exercise/state_and_reward.py`. The corresponding locations are marked **Exercise 1a** and **Exercise 1b**.
2. Test the state function by controlling the rocket manually and printing the state. Check that different angles produce suitable states.

Exercise 2: Actions

1. Implement the action mapping in `perform_action`. A simple action set is no engine, middle engine, left engine, and right engine. The location is marked **Exercise 2** in `exercise/qlearning_controller.py`.

Exercise 3: Q-learning

1. Implement the Q-learning update in `tick`. The location is marked **Exercise 3** in `exercise/qlearning_controller.py`.
2. Press G to switch off graphics and let the controller learn.
3. Switch exploration off and check that the rocket remains upright.
4. Pause the controller, move the rocket into a bad position, and then start the controller again. Check whether it recovers.

Questions Answer the following questions in your report:

1. Describe your state representation, reward function, and actions.
2. Explain each part of the Q-learning update in your own words. What does a Q-value represent?
3. Switch exploration off before learning starts. What happens, and why?

Part III: Full Hover Controller

Extend the controller so that the rocket learns to hover. The state should include vertical velocity, horizontal velocity, and angle. The number of state-action pairs grows quickly when you add variables or use more discrete values. Keep the Q-table small enough to train in a reasonable time.

Start with a simple reward function. It is often useful to reward low vertical velocity, low horizontal velocity, and an upright angle separately, and then add the terms together. Keep the rewards positive while debugging.

Exercise 4: Hover control

1. Implement the hover state and reward functions in `exercise/state_and_reward.py`. The corresponding locations are marked **Exercise 4a** and **Exercise 4b**.
2. Change the controller so that it uses the hover state and reward. The corresponding location is marked **Exercise 4** in `exercise/qlearning_controller.py`.
3. Calculate the number of possible state-action pairs in your representation.
4. Train with graphics switched off. Watch the rewards and Q-values printed in the terminal while the controller learns.
5. Switch exploration off and demonstrate the hover controller to a teaching assistant.

Questions In your report, describe your hover state and reward functions. Explain the choices you made and how the size of the state space affected learning.

Part IV: Deep Q-learning from Images (Optional)

The controllers in the previous parts read velocity and angle sensors. In many reinforcement learning problems, these values are not available. A computer game, for example, may only provide an image of the current screen.

In the CNN lab, you used a convolutional neural network to classify images. Here, the network receives a 64 by 64 grayscale image and outputs one Q-value for each action. Get an image from the simulator with:

```
frame = rocket.get_cnn_frame()
```

The table-based controller stores one Q-value for every state and action pair. This requires discrete states. In Deep Q-learning, the image is passed directly to a neural network that approximates the Q-function:

$$Q_{\theta}(s, a).$$

In image classification, each output corresponds to a class. Here, each output estimates the future reward for an action. A common Deep Q-learning loss is

$$L(\theta) = \mathbb{E} \left[\left(r + \gamma \max_{a'} Q_{\theta^-}(s', a') - Q_{\theta}(s, a) \right)^2 \right],$$

where Q_{θ} is the network being trained and Q_{θ^-} is the target network. The target network is updated less often, which helps to keep training stable.

The image is the only state input in this exercise. The reward depends only on survival time. Each non-terminal step gives a reward of 1.0, and the final step gives a reward of 0.0.

Run the optional exercise with:

```
uv run python exercise/deep_q_learning.py
```

The program trains while the simulator window is open. Press G to switch off rendering and train at maximum speed. Press G again to watch the current policy.

Exercise 5: Deep Q-learning (Optional)

1. Inspect how the image is converted to a tensor.
2. Implement a CNN that maps the image to four Q-values.
3. Implement epsilon-greedy action selection.
4. Implement the Deep Q-learning loss using the replay buffer and target network provided in `exercise/deep_q_learning.py`. The three implementation locations are marked `Exercise 5a`, `Exercise 5b`, and `Exercise 5c`.
5. Train the network on the CPU and watch how the policy changes.

Questions

1. Why does the image-based state not need manual discretization?
2. How does the output of this network differ from the output of the image classifier used in the CNN lab?