Practice 1
Environments, Rewards and Policies

Introduction to Practice Assignments

The Practice Assignments will be presented in a Laboratory Class. You can work in class and finish it at home. You must submit the results at the end of the week where the Lab took place.

The practices consist of exercises that must be done in python. You can use Google COLAB, or your own environment. The exercises will not require a GPU (maybe except the group project that has higher computing requirements).

The exercises are based on the Gymnasium package and each one of them focuses on an algorithm that has been discussed in class.

Method	Model-Based / Model-Free	On-Policy / Off-Policy	Practice
Using Gymnasium - Russell's Grid	N/A	N/A	1
Dynamic Programming	Model-Based	N/A (requires a model)	2
Monte Carlo	Model-Free	On-Policy	2
Temporal Difference (TD)	Model-Free	On-Policy	3
SARSA and Q-Learning	Model-Free	On-Policy	3
DQN-Learning	Model-Free	Off-Policy	4

Table 1: Classification of RL Methods and Assignments/Practices

If you get stuck, ask for help. With each practice, I will provide some examples to see exactly what needs to be done. Please ask if you have any questions.

Recall the AI policy on this course. You can use AI, but you must disclose its use. Try to understand what the AI is writing for you, remember that each exercise has a critical line in the process (for instance the $Q(s, a)$ update-loop) and understanding this element is key on the whole exercise.

Learning Objectives

The objective of this Practice is to familiarise with the package Gymnasium, which was created as a sandbox to try RL algorithms.

With this assignment, you will work with existing environments and create a new one based on the classical textbook on artificial intelligence ([RN10]). This is the Russell's Grid Universe . We recommend you to use the book [SB18] as reference for the Reinforcement algorithms of the course. (This is the basic textbook for the course).

As we will be using the Gymnasium package, you can access the documentation at [Far24]. One note of advice: You will find that there are many solutions out there (in the Internet, tutorials, Github) with gym. The Gym package was the original Reinforcement Learning development that was later migrated to Gymnasium, as a result of this evolution there are some slight differences between both packages, Gymnasium is more elaborated and stable. You can use the examples from gym with some adaptation on how the calls are managed and some minor changes. if you are interested in using examples created with Gym but in the Gymnasium version you can use this documentation `https://gymnasium.farama.org/content/migration-guide/`.

1 Files for this Practice

```
00_RussellGrid_new_environment-BASELINE.ipynb
```

2 Activities

2.1 Implement and use Gymnasium environments - ACTIVITY 1

For the practices in this course you will require a Python development environment. We've planned all the practices in jupyter colab notebooks, so that you can either run them in Google COLAB or in your Laptop. We will discuss in class some hints on how to maintain a local jupyter notebook environment.

At the end of each jupyter notebook example you will find packages and versions used in the example. Review them to make sure your environment is fully aligned with the example notebook.

I have prepared a GitHub Repository with the examples used in the course. You can have access to it and use the examples freely. The github repository is open and open-sourced. it is

```
https://github.com/castorgit/RL_course
```

The course examples will be based in the **Gymnasium** package (see class material with the description of Gymnasium) [1]

Go to the Gymnasium documentation Basic Usage where you'll find all information regarding how environments behave. Each environment is created with an objective in mind and the methods may differ. For instance the description of the Taxi environment can be found in Taxi environment.

You can see in the course github repository several basic examples (they have the 00 prefix) like the

```
https://github.com/castorgit/RL_course/00_Cartpole_reender.ipynb
```

[1] Originally the Gymnasium package was called Gym, this is the reason you'll find many examples with Gym and not so many with gymnasium. To make code as homogeneous as possible, we import gymnasium as gym

This example shows how to call and render the Cartpole environment. You should do the same for other three environments, Taxi, Lunar Lander and Frozen Lake.

- Taxi environment ("Taxi-v3")
- Lunar landing environment ("LunarLander-v2")
- Frozen lake ("FrozenLake-v1")

Try to identify the characteristics of each environment and respond to the following questions for the 4 environments

- Which are the main elements of each environment? (describe the environment, the issues, the complexities.
- What is the observation space (S), The action space (A), and the reward function (R)?
- Is the environment continuous or discrete?
- What are the main differences between them?

Later on we will use them in some of the practice examples

2.2 Creating the Russell's Grid Environment - ACTIVITY 2

The challenge of this activity is to create a new environment from scratch. In this case we will create a well known universe defined by Russell in his textbook Artificial Intelligence [RN10].

Background on Rusell's Grid World

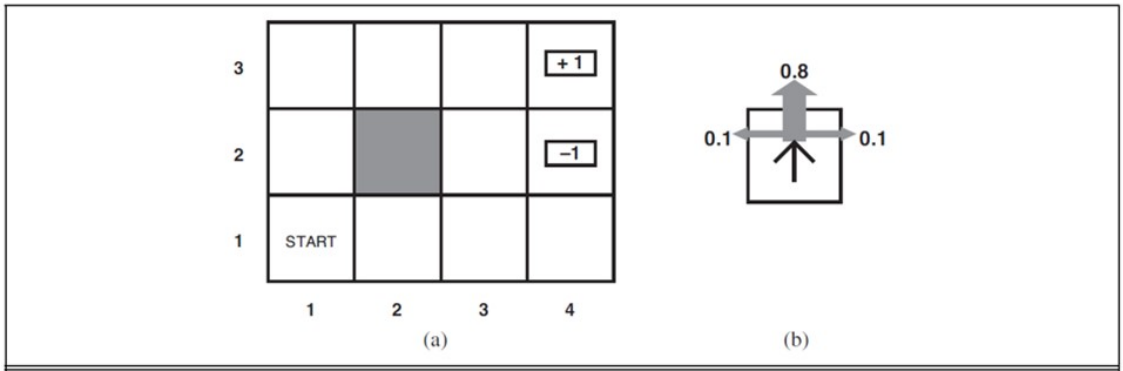

Figure 17.1 (a) A simple 4×3 environment that presents the agent with a sequential decision problem. (b) Illustration of the transition model of the environment: the "intended" outcome occurs with probability 0.8, but with probability 0.2 the agent moves at right angles to the intended direction. A collision with a wall results in no movement. The two terminal states have reward +1 and –1, respectively, and all other states have a reward of –0.04.

Figure 1: Image from the book [RN10], Chapter 17

Russell's World is a grid-based environment where an agent must navigate from a start position to a goal position while avoiding obstacles and possibly collecting rewards. The environment is defined by:

- **Observation space (S)**: The set of all possible positions in the grid.
- **Action space (A)**: The set of possible moves (e.g., left, right, up, down).
- **Rewards (R)**: Values received after moving from one state to another.
- **Transitions (T)**: The probabilities of moving from one state to another given an action.
- Agent starts in position (1,1)

This environment is based on a universe of a 3x4 board with a set of cells and two terminal states. Beginning in the start state, it must choose an action at each time step. The interaction terminates when the agent reaches one of the goal states ($+1$ or -1), one is a positive end the other negative.

The model is based in stochastic motion, this is the actions are unreliable. For example from the start square (1,1), the action has a probability 0.8 to go in the correct way or 0.1 to go other ways. For example, from the start square (1,1), the action Up moves the agent to (1,2) with probability 0.8, but with probability 0.1, it moves right to (2,1), and with probability 0.1. There is a black cell which can be considered a wall. If the action moves there the agent stays on its place.

What you have to do? You must create a class for the `Russell_Grid` environment. This class must have 5 methods:

1. `_init_()`: Is the initial state of the environment. You must define in this method the action space and the observation space. Assign the initial state to it.
2. `reset()`: The reset method must reset the environment to a possible start state and return the observation for that state
3. `step(action)`: Simulates the environment dynamics and the agent actions in one time step. Returns a a tuple (observation, reward, terminated, truncated, info) describing the transition. We don't use Truncated (can be False). info is a list with information for debugging we will leave it at by now.
4. `close()`: This method frees any handles, sockets, in this easy example just pass will work.

Use the environment in

```
$ 00_RussellGrid_new_environment.ipynb
```

This environment is complete, but write a Notebook where you include this environment. Remember, to generate an include you must create a .py file with the notebook and then, if you place it in the same folder as your program you can use import in your notebook.

The final code in your notebook should be something like this.

```
import MYRUSSELGRID
env = gym.make('RussellsGrid-v0', render_mode = 'human')
env.reset()
env.render()

done = False
while not done:
        action = env.action_space.sample() # Take a random action
        state, reward, done, truncated, info = env.step(action)
        env.render()
        print(f"State:{state},Reward:{reward},Done:{done}")
```

then you will see a rendering of the agent just following a random path.

2.3 Calculating Rewards, and policies using Russell_grid.env

You must complete Activity 1 before going into 2. Now load the notebook `Practice1_Activity2.ipynb` You can use jupyter on your laptop or Google Colab. Make sure to have all the files in the same folder.

In this activity we will create several strategies based on the RussellGrid Environment.

Remember that the environment coordinates work as see in 2.

0 [0,0]	1 [0,1]	2 [0,2]	3 [0,3]
4 [1,0]	5 [1,1]	6 [1,2]	7 [1,3]
8 [2,0]	9 [2,1]	10 [2,2]	11 [2,3]

Figure 2: Coordinates in RussellGrid Envionment

2.3.1 Activity 3.1 - One Random Episode

Define an episode in the environment using a random action. There is in the environment the method `space_sample()` that generates a random action.

The loop will be

```
one random episode

while not in final State
   reender or not reender
   sample action
   perform action
   calculate and append reward
   if final state then
      reender and reset
print rewards per step
```

Implement this loop in your notebook and execute and see if it works. You should be able to reender the environment

2.3.2 Activity 3.2 - Total Reward in an episode

Create a pyhton list `lr` which contains all the rewards in one episode.

Now we will set up a couple of strategies for the calculation of the total reward or long term reward of an episode by creating two functions called `long_term_reward1()` and `long_term_reward2()`that from the `lr` list calculate the reward of an episode.

`long_term_reward1()` just calculates the cumulative value of all rewards

```
long term reward

initialize TOTAL value to 0
for lenght LR list
   TOTAL = TOTAL + LR[i]
return TOTAL
```

What is the value of the total reward for your episode?

2.3.3 Ativity 3.3 - Total Reward in an episode with decay

Now we will calculate the global reward using a decay value, in this way we valuate stronger the values of the short term rewards vs the values of the long term rewards.

```
long term reward with decay

initialize TOTAL value to 0
initialize gamma decay value at 0.99
for lenght LR list
  TOTAL = TOTAL + LR[i] * GAMMA ** i
return TOTAL
```

Now calculate the long term reward using both strategies for 1000 or 10000 episode. What are your results?

You should get approximate results to these ones

2.3.4 Activity 3.4 - Total Reward in multiple episodes

Now create a function to calculate the reward for a number of episodes. Make sure you are not rendering the episode as it uses a lot of resources.

Calculate the reward for 10.000 episodes. What values do you get with and without decay?

2.3.5 Activity 3.5 - Policy

Now you will create your own policy for the environment.

Try to think in an optimal policy, but it can be any policy

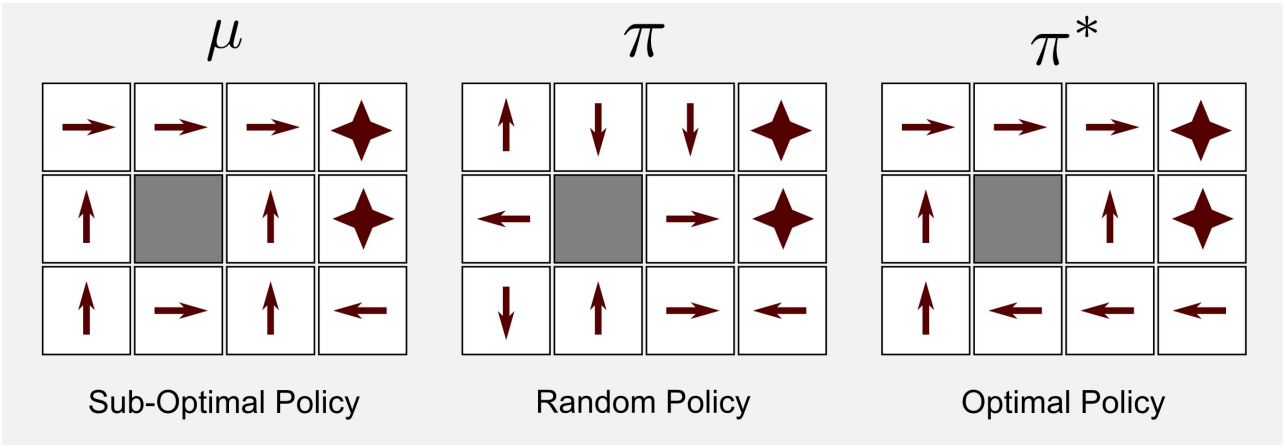

Figure 3: Optimal and random prolicies

You must implement the optimal policy in the array policy. Please, check the figure 2 to understand how to address each cell.

After modifying the policy array, create a function called `one_episode_my_policy()` replacing sample action with my policy

To create a policy, use a two dimensional array and load the optimal policy action. For instance, for the position [0,0] the action is 1 (right). Just create the policy by copying the optimal policy in the figure 3.

```
policy[0,0] = 1
policy[0,1] = 1
```

```
one_episode_my_policy()

while not in final State
  reender or not reender
  action from array policy
  perform action
  calculate and append reward
  if final state then
    reender and reset
```

2.3.6 Activity 3.6 - Visualizing the Policy

Include this code to visualize the policy. this kind of visualizations are useful to understand how the policy evolves.

```
def print_policy(env, policy):
    """Prints the policy in a grid format."""

    ncol = env.unwrapped.ncol
    nrow = env.unwrapped.nrow

    print("\nValue␣Optimal␣Policy␣Format␣(3x4):")
    print("-" * 50)
    actions = ['UP', 'RIGHT', 'DOWN', 'LEFT']
    for i in range(nrow):
        for j in range(ncol):
            s = i * ncol + j
            if (i, j) == (1,1):
                print('X', end='␣')
            elif (i,j) == (0,3):
                print('G', end='␣')
            elif (i,j) == (1,3):
                print('N', end='␣')
            else:
                a = int(policy[i,j])
                print(actions[a], end='␣')
        print()

env.reset()
```

Note: replace the arrowup, arrowright,... by the arrow signs to have a graphical description of the policy. Your image should be something like Figure 4.

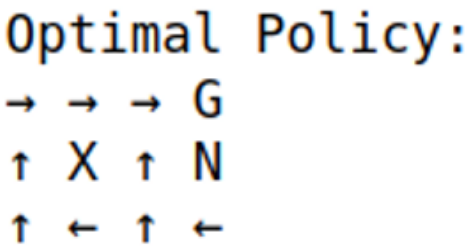

Figure 4: This should be the final output of your policy exercise

3 Deliverables and submission

Submit one Notebook where you have the Activities 3.1, 3.2, 3.3

Remember to name your assignment with your name creating a title of the notebook. Try to name it like this:

```
Assignment1_Peter_Rodrigues.ipynb
```

References

[RN10] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 4th ed. Prentice Hall, 2010. ISBN: 9780136042594.

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: `http://incompleteideas.net/book/the-book-2nd.html`.

[Far24] Farama. *Gymnasium package documentation*. `https://gymnasium.farama.org/`. Accessed: 2024-08-09. 2024.