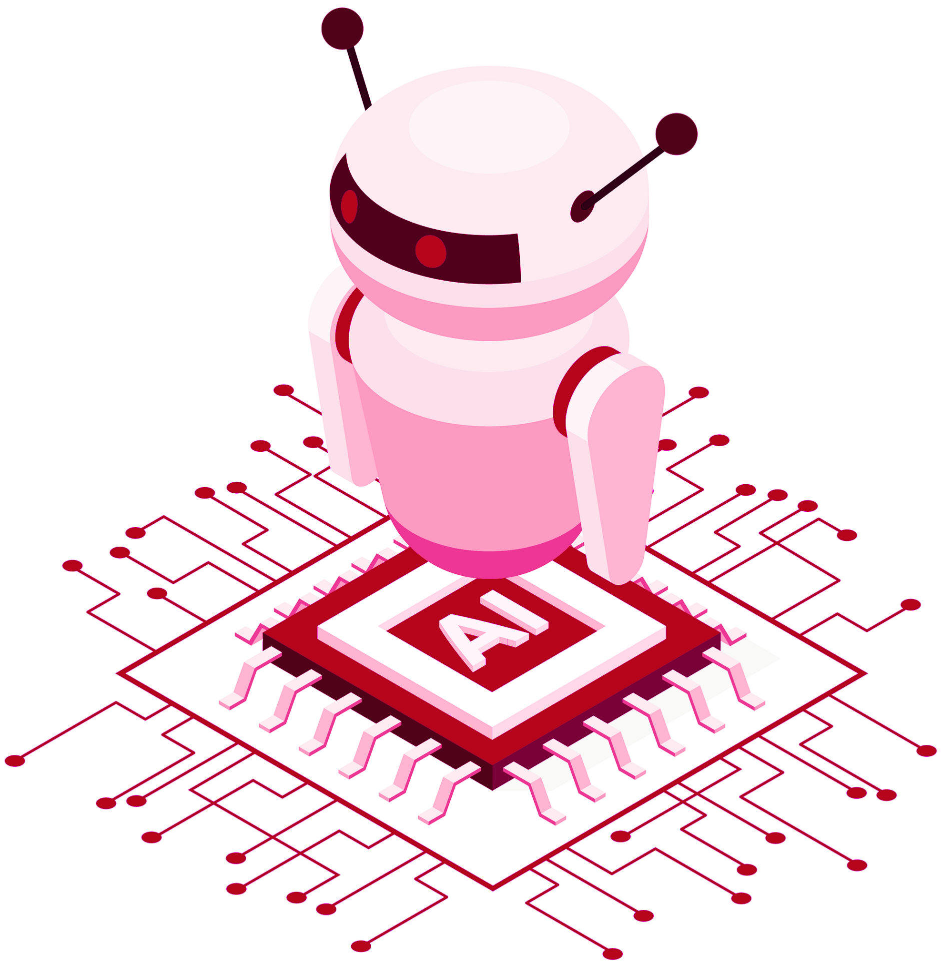


Deep Reinforcement Learning

Professor Mohammad Hossein Rohban

Homework 1:

Introduction to RL

By:

Amir Kooshan Fattah Hesari
401102191

Spring 2025

Contents

Grading

The grading will be based on the following criteria, with a total of 100 points:

Task	**Points**
Task 1: Solving Predefined Environments	45
Task 2: Creating Custom Environments	45
Clarity and Quality of Code	5
Clarity and Quality of Report	5
Bonus 1: Writing a wrapper for a known env	10
Bonus 2: Implementing pygame env	20
Bonus 3: Writing your report in Latex	10

Notes:

- Include well-commented code and relevant plots in your notebook.
- Clearly present all comparisons and analyses in your report.
- Ensure reproducibility by specifying all dependencies and configurations.

1 Task 1: Solving Predefined Environments [45-points]

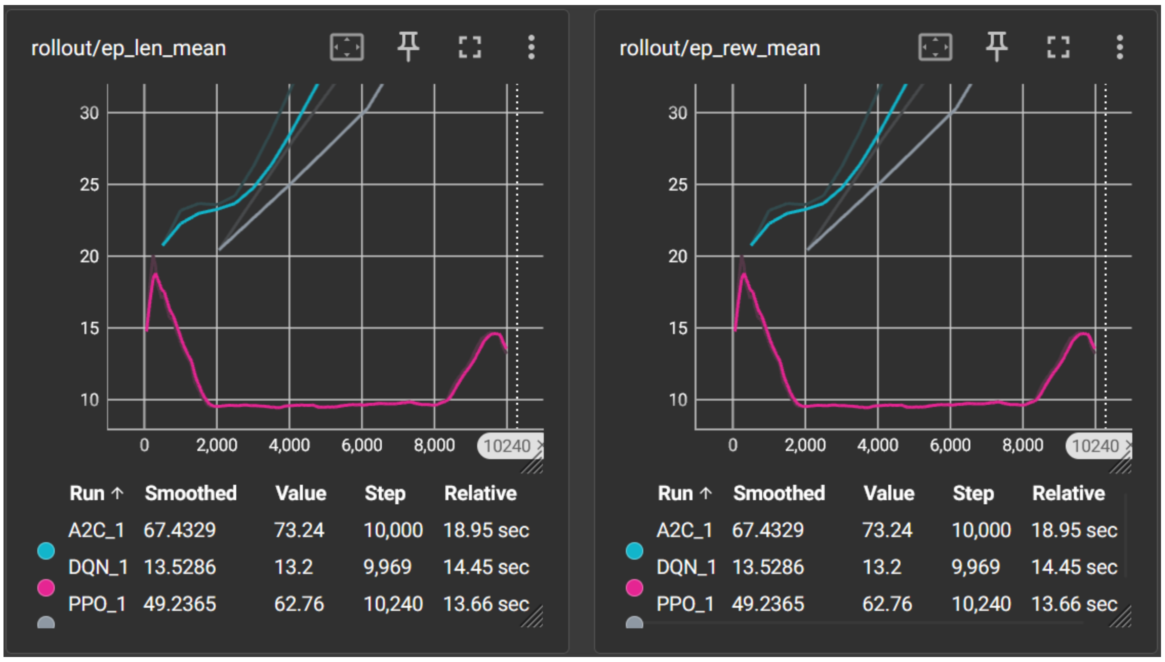
Important Note : Due to the large size, the chart in the notebook file have been commented out, and their images have been included in the report. If they were note commented , the final size of the notebook would be 100 Mb. The TAs can uncomment and check the correctness of the plots.

1.1 Learning curves

From the given environments, I chose the **Taxi** and **Cartpole** environment. First by using the gymnaisum pre-wrapped function `gymnasium.make()` i loaded the Cartpole environment to `env1`. It's action space has 2 values indicating the direction of the fixed force the cart is pushed with.

- 0 : Push cart to the left
- 1 : Push cart to the right

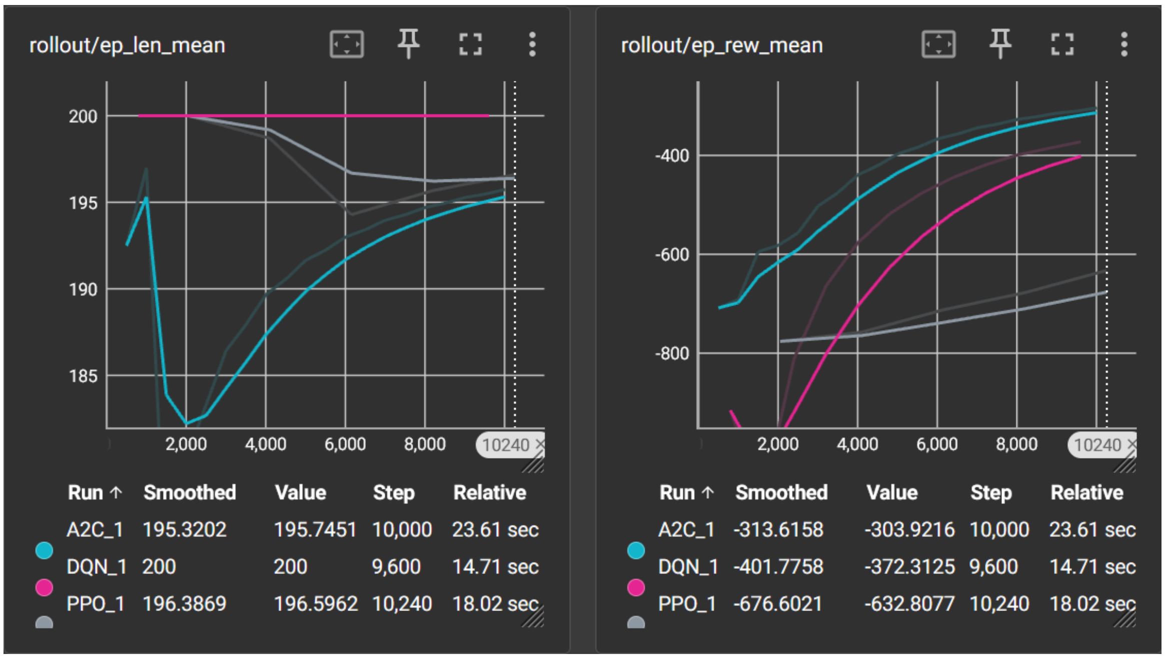
It's observation space (state space) is a `ndarray` with dimensions `(4,)`.
Then by loading the `PPO,A2C,DQN` algorithms from the `stable_baselines3` library , I traind three different models with these algorithms and saved the amount of reward in each episode by using the `tensorboard_log` option of the algorithm's functions. After training each model I used the `del model` to free up some storage and memory. The learning curves are as follows :



	Run ↑	Smoothed	Value	Step	Relative
●	A2C_1	67.4329	73.24	10,000	18.95 sec
●	DQN_1	13.5286	13.2	9,969	14.45 sec
●	PPO_1	49.2365	62.76	10,240	13.66 sec

	Run ↑	Smoothed	Value	Step	Relative
●	A2C_1	67.4329	73.24	10,000	18.95 sec
●	DQN_1	13.5286	13.2	9,969	14.45 sec
●	PPO_1	49.2365	62.76	10,240	13.66 sec

Figure 1: Learning curve of three different algorithms : PPO , A2C, and DQN for the CartPole-v1 environment



rollout/ep_len_mean

Run ↑	Smoothed	Value	Step	Relative
A2C_1	195.3202	195.7451	10,000	23.61 sec
DQN_1	200	200	9,600	14.71 sec
PPO_1	196.3869	196.5962	10,240	18.02 sec

rollout/ep_rew_mean

Run ↑	Smoothed	Value	Step	Relative
A2C_1	-313.6158	-303.9216	10,000	23.61 sec
DQN_1	-401.7758	-372.3125	9,600	14.71 sec
PPO_1	-676.6021	-632.8077	10,240	18.02 sec

Figure 2: Learning curve of three different algorithms : PPO , A2C, and DQN for the Taxi-v3 environment

We see that the reward in the **A2C** algorithm is better than the other two. By playing with the number of episodes we would also get the same results.

1.2 Playing with hyperparameters

we play with the hyperparameters and try to see how this will change the performance of the models. For the PPO algorithm we examin these hyperparameters :

- learning-rate: 4e-4, n-steps: 1536, batch-size: 96, gamma: 0.985, clip-range: 0.25
- learning-rate: 8e-4,n-steps: 768, batch-size: 48, gamma: 0.96, clip-range: 0.25
- learning-rate: 6e-4,n-steps: 3584, batch-size: 160,gamma: 0.94, clip-range: 0.15

and for the DQN algorithm we examin these hyperparameters :

- learning-rate: 1e-3, gradient-steps: 1, batch-size: 64, gamma: 0.99, tau: 1.0
- learning-rate: 5e-4, gradient-steps: 4, batch-size: 32, gamma: 0.95, tau: 0.01
- learning-rate: 2e-4, gradient-steps: 8, batch-size: 128, gamma: 0.97, tau: 0.05

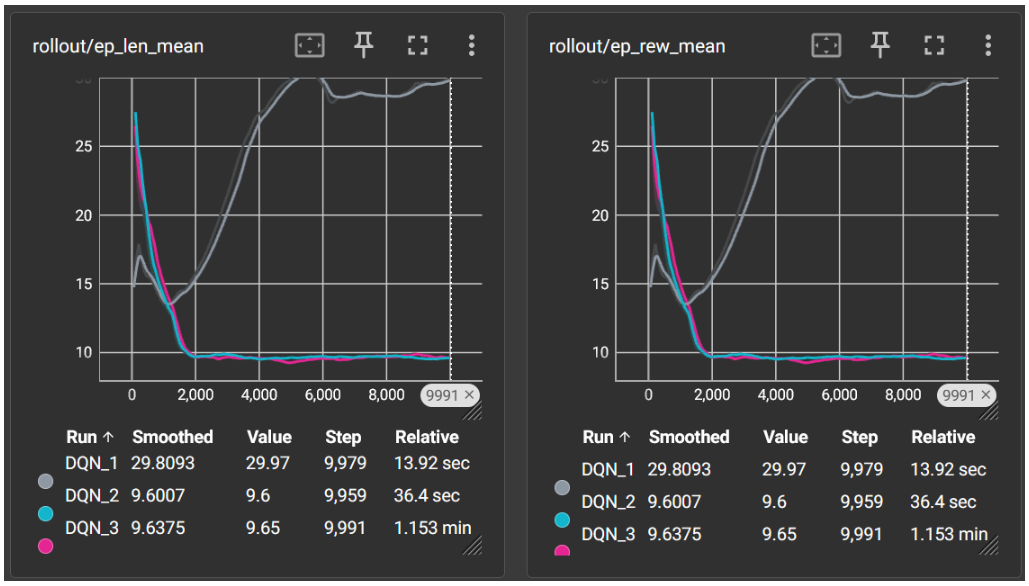
For the second environment we change the hyperparameters just a little bit as follows , for PPO :

- learning-rate: 3e-4, n-steps: 1436, batch-size: 86, gamma: 0.995, clip-range: 0.25
- learning-rate: 7e-4,n-steps: 568, batch-size: 38, gamma: 0.92, clip-range: 0.35
- learning-rate: 5e-4,n-steps: 2584, batch-size: 140,gamma: 0.91, clip-range: 0.15

and for DQN :

- learning-rate: 2e-3, gradient-steps: 1, batch-size: 64, gamma: 0.99, tau: 1.0

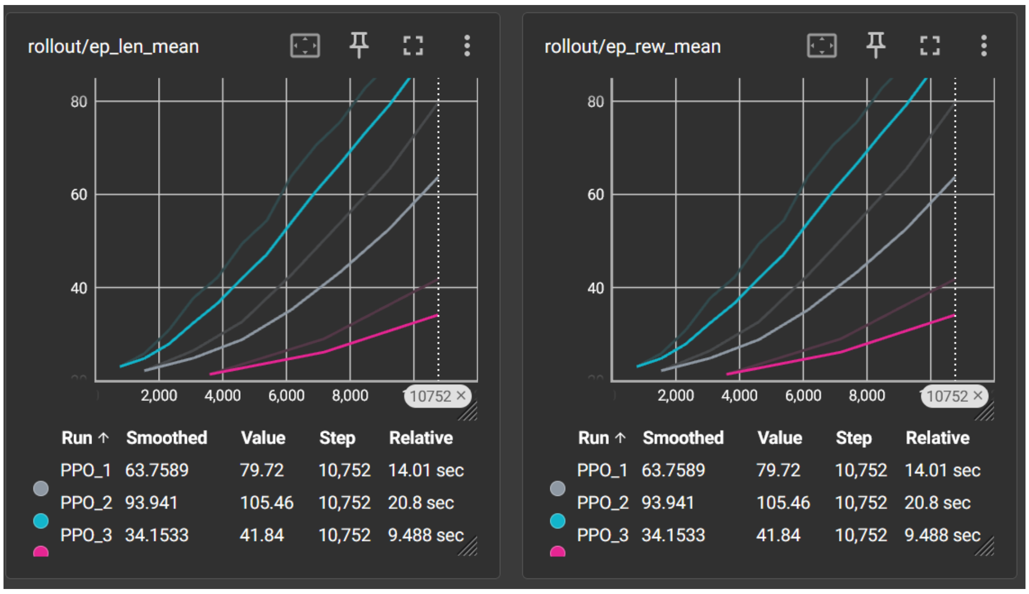
- learning-rate: 8e-4, gradient-steps: 3, batch-size: 32, gamma: 0.95, tau: 0.01
- learning-rate: 5e-4, gradient-steps: 6, batch-size: 128, gamma: 0.97, tau: 0.05



Run ↑	Smoothed	Value	Step	Relative
DQN_1	29.8093	29.97	9,979	13.92 sec
DQN_2	9.6007	9.6	9,959	36.4 sec
DQN_3	9.6375	9.65	9,991	1.153 min

Run ↑	Smoothed	Value	Step	Relative
DQN_1	29.8093	29.97	9,979	13.92 sec
DQN_2	9.6007	9.6	9,959	36.4 sec
DQN_3	9.6375	9.65	9,991	1.153 min

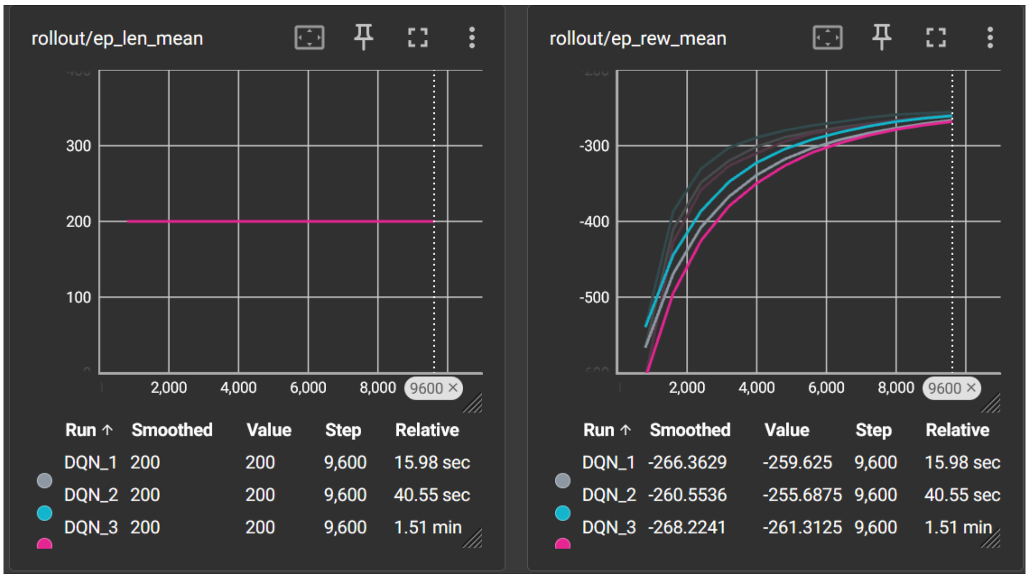
Figure 3: Learning curve of three different sets of hyperparameters for DQN algorithm for the CartPole-v1 environment



Run ↑	Smoothed	Value	Step	Relative
PPO_1	63.7589	79.72	10,752	14.01 sec
PPO_2	93.941	105.46	10,752	20.8 sec
PPO_3	34.1533	41.84	10,752	9.488 sec

Run ↑	Smoothed	Value	Step	Relative
PPO_1	63.7589	79.72	10,752	14.01 sec
PPO_2	93.941	105.46	10,752	20.8 sec
PPO_3	34.1533	41.84	10,752	9.488 sec

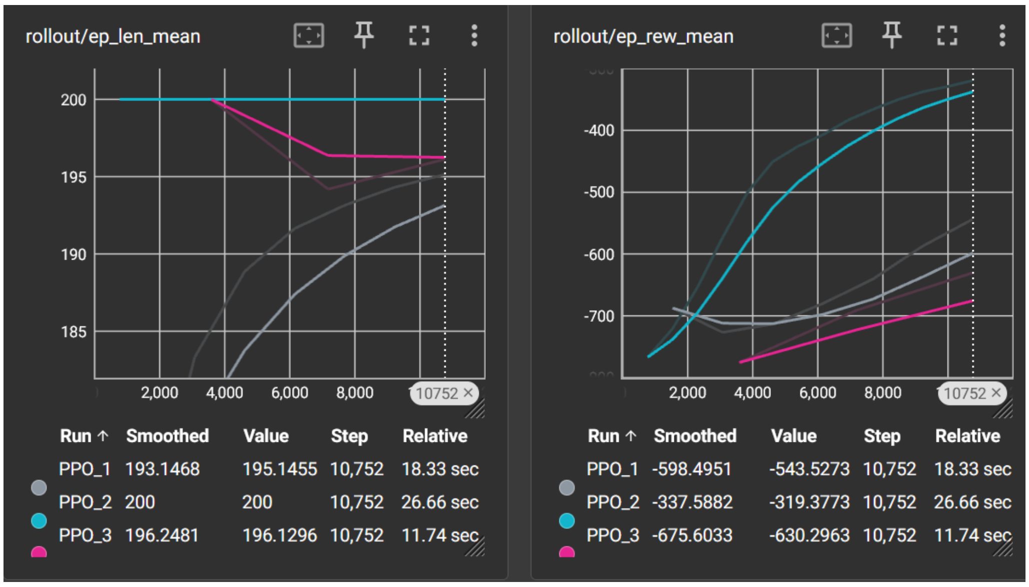
Figure 4: Learning curve of three different sets of hyperparameters for PPO algorithm for the CartPole-v1 environment



Run ↑	Smoothed	Value	Step	Relative
DQN_1	200	200	9,600	15.98 sec
DQN_2	200	200	9,600	40.55 sec
DQN_3	200	200	9,600	1.51 min

Run ↑	Smoothed	Value	Step	Relative
DQN_1	-266.3629	-259.625	9,600	15.98 sec
DQN_2	-260.5536	-255.6875	9,600	40.55 sec
DQN_3	-268.2241	-261.3125	9,600	1.51 min

Figure 5: Learning curve of three different sets of hyperparameters for DQN algorithm for the Taxi-v3 environment



Run ↑	Smoothed	Value	Step	Relative
PPO_1	193.1468	195.1455	10,752	18.33 sec
PPO_2	200	200	10,752	26.66 sec
PPO_3	196.2481	196.1296	10,752	11.74 sec

Run ↑	Smoothed	Value	Step	Relative
PPO_1	-598.4951	-543.5273	10,752	18.33 sec
PPO_2	-337.5882	-319.3773	10,752	26.66 sec
PPO_3	-675.6033	-630.2963	10,752	11.74 sec

Figure 6: Learning curve of three different sets of hyperparameters for PPO algorithm for the Taxi-v3 environment

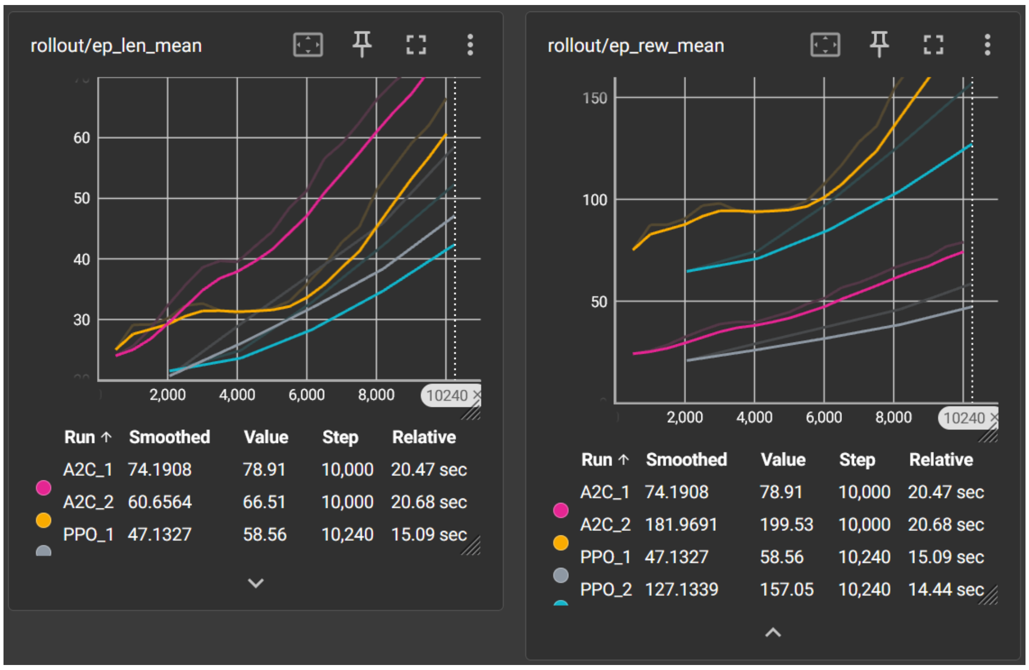
1.3 Reward Function Wrapper (Bonus)

I wrote the following wrapper for reward function :

```python
from gymnasium import RewardWrapper
class RewardFormula(RewardWrapper):
  def reward(self, reward):
        return reward * 2 + 1
EnvWrapped = make_vec_env(lambda: RewardFormula(gym.make("CartPole-v1",
    render_mode="rgb_array")), n_envs=1)
```

Listing 1: Reward Wrapper

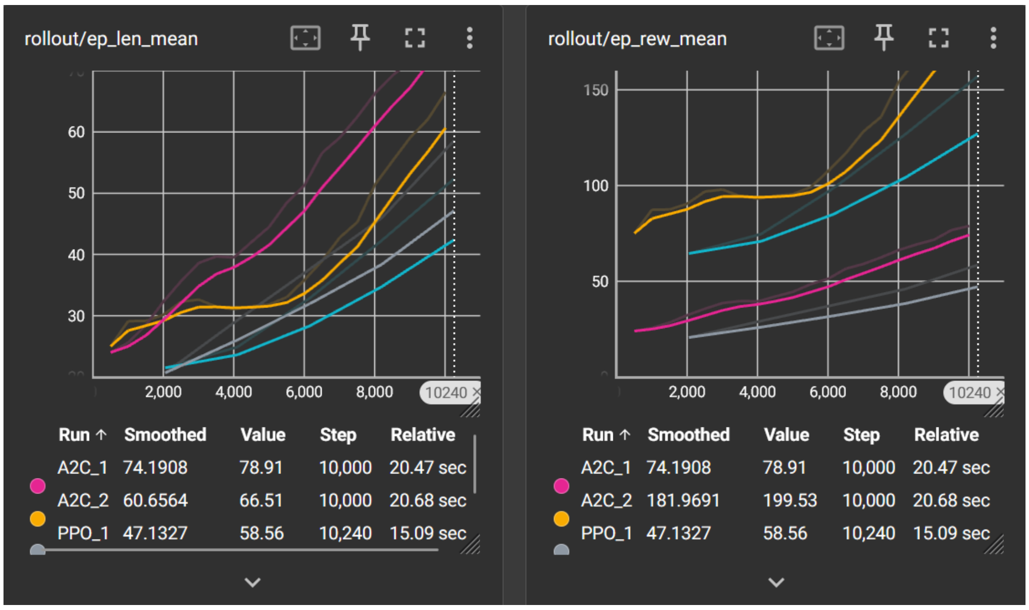
The use of `lambda` is due to the fact that the `RewardFormula` class expects an instance of the type `env` , not `VecEnv`, therefore we use this function to pass it to `make_vec_env` function.



	Run ↑	Smoothed	Value	Step	Relative
●	A2C_1	74.1908	78.91	10,000	20.47 sec
●	A2C_2	60.6564	66.51	10,000	20.68 sec
●	PPO_1	47.1327	58.56	10,240	15.09 sec

	Run ↑	Smoothed	Value	Step	Relative
●	A2C_1	74.1908	78.91	10,000	20.47 sec
●	A2C_2	181.9691	199.53	10,000	20.68 sec
●	PPO_1	47.1327	58.56	10,240	15.09 sec
●	PPO_2	127.1339	157.05	10,240	14.44 sec

Figure 7: Learning curve of the main PPO and A2C algorithms and their repsective wrapped one for the CartPole-v1 environment



	Run ↑	Smoothed	Value	Step	Relative
●	A2C_1	74.1908	78.91	10,000	20.47 sec
●	A2C_2	60.6564	66.51	10,000	20.68 sec
●	PPO_1	47.1327	58.56	10,240	15.09 sec

	Run ↑	Smoothed	Value	Step	Relative
●	A2C_1	74.1908	78.91	10,000	20.47 sec
●	A2C_2	181.9691	199.53	10,000	20.68 sec
●	PPO_1	47.1327	58.56	10,240	15.09 sec

Figure 8: Learning curve of the main PPO and A2C algorithms and their repsective wrapped one for the Taxi-v3 environment

We see that wrapping the reward function and returning `reward * 2 + 1` is evident in the plots that it makes the reward greater for both of the algorithms.

1.4 SL fitness discussion (Bonus)

In well-defined environments like CartPole and Taxi, SL can work well if you have a high-quality set of expert demonstrations and it can be an be very sample-efficient compared to RL, which learns through trial and error.
On the other hand some problems might rise. SL learns from a fixed dataset of expert demonstrations. When the learned policy makes a mistake, it may visit states that are not well-represented in the training data and distribution mismatch (covariate shift) leads to compounding errors and degrades the performance.
Another common problem is that if in the settings of the problem the reward is sparse or we have a long horizon , small deviations from expert behavior can lead the agent far from the rewarding states where SL might fail to provide the necessary corrections for these deviations, whereas RL methodsthrough iterative trial and errorcan learn to navigate these long-horizon challenges.

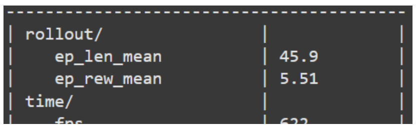
2 Task 2: Creating Custom Environments [45-points]

2.1 Discussion on implementation and modification

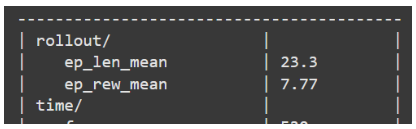
Based on the given tasks, first we implement a new `class` name **YourAwesomeEnvironment** which is a subclass of the `gym.Env` class. In the `__init__` method, we declare the basics of an environment; the position of tha agent , the position of the target , place of obstacles on the grid, the action space, and the observation space.
The action space contains the actions that the agent can do which are four ; going **up , down , right, and left**.
The observation space contains all the possible positions of the agent and target on the grid. In the `step(self,action)` function we define what happens to the position of the agent if a certain action is taken, e.g. if `action == 1` then the horizontal position of the agent increases by amount of one. If the new position is on an obstacle the position isn't updated and another action is taken.
Then if the agent's position and that of target's become the same, meaning they are at the same place, the agent is rewarded a value of 10 and the episode is over.
The `rest` function return the agent to its original place and sets everything else to default. Since we will need to compute observations both in `Env.reset()` and `Env.step()`, it is often convenient to have a method `_get_obs` that translates the environments state into an observation. The `render` function was implemented using **ChatGPT** because I didn't know what to render or even how to render using the `env.render` option, `it's extra` and for this simple environment we didn't have any need for a close method.
Also changing the hyperparameters in the used algorithms may lead to better and faster learning or chaning the reward given at different states.

2.2 Training results, learning curves, and observations

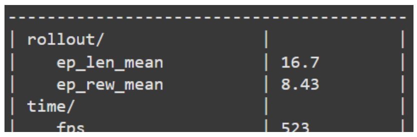
For trainig results using the PPO algorithm we have the following output : As it is clear in the pictures



```
-----------------------------------------
| rollout/                |             |
|    ep_len_mean          | 45.9        |
|    ep_rew_mean          | 5.51        |
| time/                   |             |
|    fps                  | 622         |
```



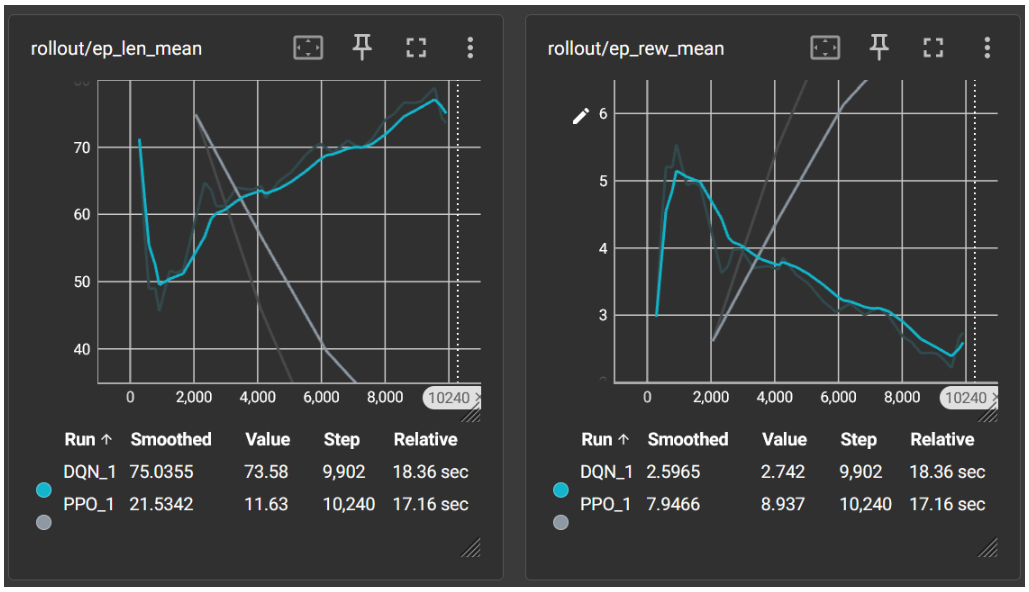
```
-----------------------------------------
| rollout/                |             |
|    ep_len_mean          | 23.3        |
|    ep_rew_mean          | 7.77        |
| time/                   |             |
```



```
-----------------------------------------
| rollout/                |             |
|    ep_len_mean          | 16.7        |
|    ep_rew_mean          | 8.43        |
| time/                   |             |
|    fps                  | 523         |
```

Figure 9: The training results of the custom environment on the PPO algorithm

, the average reward return per episode is increasing which means that the training is going well. Also comparing the PPO and the DQN algorithm yields the following results :



rollout/ep_len_mean

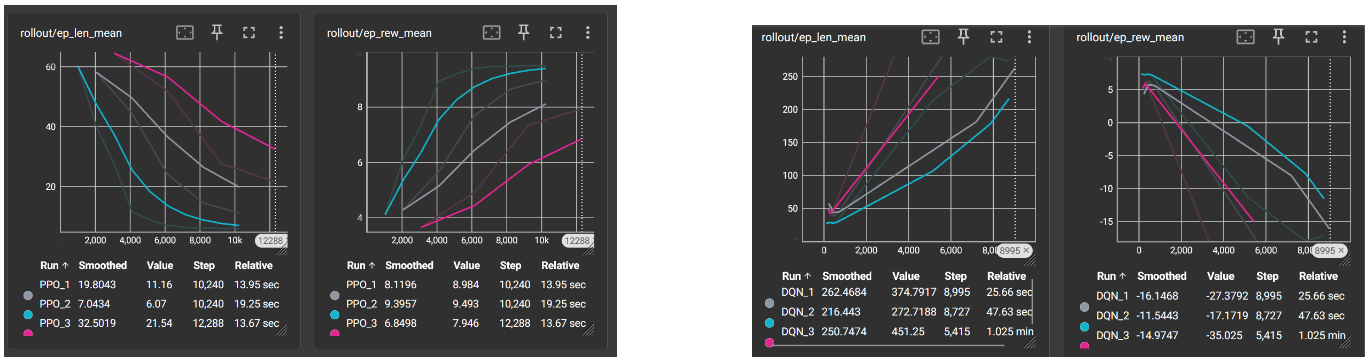
Run ↑	Smoothed	Value	Step	Relative
DQN_1	75.0355	73.58	9,902	18.36 sec
PPO_1	21.5342	11.63	10,240	17.16 sec

rollout/ep_rew_mean

Run ↑	Smoothed	Value	Step	Relative
DQN_1	2.5965	2.742	9,902	18.36 sec
PPO_1	7.9466	8.937	10,240	17.16 sec

Figure 10: Learning curves of the PPO and DQN algorithms in the custom environment

Also changing the hyperparameters and comparing the results with each other is as follows:



rollout/ep_len_mean

Run ↑	Smoothed	Value	Step	Relative
PPO_1	19.8043	11.16	10,240	13.95 sec
PPO_2	7.0434	6.07	10,240	19.25 sec
PPO_3	32.5019	21.54	12,288	13.67 sec

rollout/ep_rew_mean

Run ↑	Smoothed	Value	Step	Relative
PPO_1	8.1196	8.984	10,240	13.95 sec
PPO_2	9.3957	9.493	10,240	19.25 sec
PPO_3	6.8498	7.946	12,288	13.67 sec

rollout/ep_len_mean

Run ↑	Smoothed	Value	Step	Relative
DQN_1	262.4684	374.7917	8,995	25.66 sec
DQN_2	216.443	272.7188	8,727	47.63 sec
DQN_3	250.7474	451.25	5,415	1.025 min

rollout/ep_rew_mean

Run ↑	Smoothed	Value	Step	Relative
DQN_1	-16.1468	-27.3792	8,995	25.66 sec
DQN_2	-11.5443	-17.1719	8,727	47.63 sec
DQN_3	-14.9747	-35.025	5,415	1.025 min

Figure 11: The training results of the custom environment on the PPO algorithm

2.3 Code of the environment

The main components of the custom environment are:

```python
class YourAwesomeEnvironment(gym.Env):
    def __init__(self, size=4, obstacles):
        super().__init__()
        self.size = size
        self.agent_pos = [0, 0]
        self.target_pos = [size - 1, size - 1]
        obstacles = [(1, 1), (2, 2)]
        self.obstacles = set(obstacles)
        self.action_space = spaces.Discrete(4)
        self.observation_space = spaces.Dict({
            "agent": spaces.MultiDiscrete([size, size]),
            "target": spaces.MultiDiscrete([size, size]),
        })
```

Listing 2: Initialization

```python
    def step(self, action):
        x, y = self.agent_pos
        # Compute new position
        new_x, new_y = x, y
        if action == 0 and y > 0:   # Move Left
            new_y -= 1
        elif action == 1 and y < self.size - 1:  # Move Right
            new_y += 1
        elif action == 2 and x > 0:   # Move Up
            new_x -= 1
        elif action == 3 and x < self.size - 1:  # Move Down
            new_x += 1
        if (0 <= new_x < self.size and 0 <= new_y < self.size) and (new_x, new_y) not in self.obstacles:
            self.agent_pos = [new_x, new_y]  # Update position if it's a valid move
        reward = -0.1  # Small penalty per step
        done = False
        if self.agent_pos == self.target_pos:
            reward = 10  # Large reward for reaching the goal
            done = True  # Episode ends
        return self._get_obs(), reward, done, False, {}
```

Listing 3: step method

```python
    def reset(self, seed=None, options=None):
        self.agent_pos = [0, 0]
        return self._get_obs(), {}
```

Listing 4: reset method

A complete explanation of different parts is given in subsection 2.1.

3 Task 3: Pygame for RL environment [20-points]

3.1 Updated Pygame Environments

In the given notebook there was no pre-existing environment for this part, so creating one from scratch required a little of digging and searching.

```python
def __init__(self, grid_size=10, cell_size=40, render_mode="human"):
     self.grid_size = grid_size       # Number of cells per dimension.
     self.cell_size = cell_size       # Pixel size of each cell.
     self.width = self.grid_size * self.cell_size
     self.height = self.grid_size * self.cell_size
     self.render_mode = render_mode
     self.action_space = spaces.Discrete(4)
     self.observation_space = spaces.Box(low=0, high=self.grid_size - 1, shape
=(2,), dtype=np.int32)
     if self.render_mode == "human":
         pygame.init()
         self.screen = pygame.display.set_mode((self.width, self.height))
         pygame.display.set_caption("GridWorld Environment")
         self.clock = pygame.time.Clock()
     self.agent_color = (0, 255, 0)
     self.goal_color = (255, 0, 0)
     self.obstacle_color = (0, 0, 255)
     self.bg_color = (255, 255, 255)
     self.reset()
```

Listing 5: Initialization method

Just like the custom environment we create a subclass of the `gym.Env` class and initialize it in the same way, **except** that we specify a `render_mode` and initialize a pygame screen so that the game can be shown on it and give color to different components. Everything else like the agent position and obstacles and the size and spaces and ... are implemented in the same way as before.
The `step` and `reset` function explanations are like those given in Task 2.

```python
def render(self):
     if self.render_mode is None:
         return
     self.screen.fill(self.bg_color)
     for x in range(0, self.width, self.cell_size):
         for y in range(0, self.height, self.cell_size):
             rect = pygame.Rect(x, y, self.cell_size, self.cell_size)
             pygame.draw.rect(self.screen, (200, 200, 200), rect, 1)
     for obs in self.obstacles:
         rect = pygame.Rect(obs[0] * self.cell_size, obs[1] * self.cell_size,
self.cell_size, self.cell_size)
         pygame.draw.rect(self.screen, self.obstacle_color, rect)
     rect = pygame.Rect(self.goal_pos[0] * self.cell_size, self.goal_pos[1] *
self.cell_size, self.cell_size, self.cell_size)
     pygame.draw.rect(self.screen, self.goal_color, rect)
     rect = pygame.Rect(self.agent_pos[0] * self.cell_size, self.agent_pos[1]
* self.cell_size, self.cell_size, self.cell_size)
     pygame.draw.rect(self.screen, self.agent_color, rect)

     pygame.display.flip()
```

```
        self.clock.tick(30)
```

Listing 6: render method

Given the state of the episode , it draws the rectangles, the agent, and the target given their updater locations in a 400 * 400 window. For defining the render function **ChatGPT** has been helpful.

3.2 Screenshots of the modifications and explanations

The explanations are given in the first part and changing the positions and the size of the grid and reward may cause different learning rates with different algorithms, but here we examin the PPO algorithm. The results are in the notebook and a screenshot of the created environment is as follows :

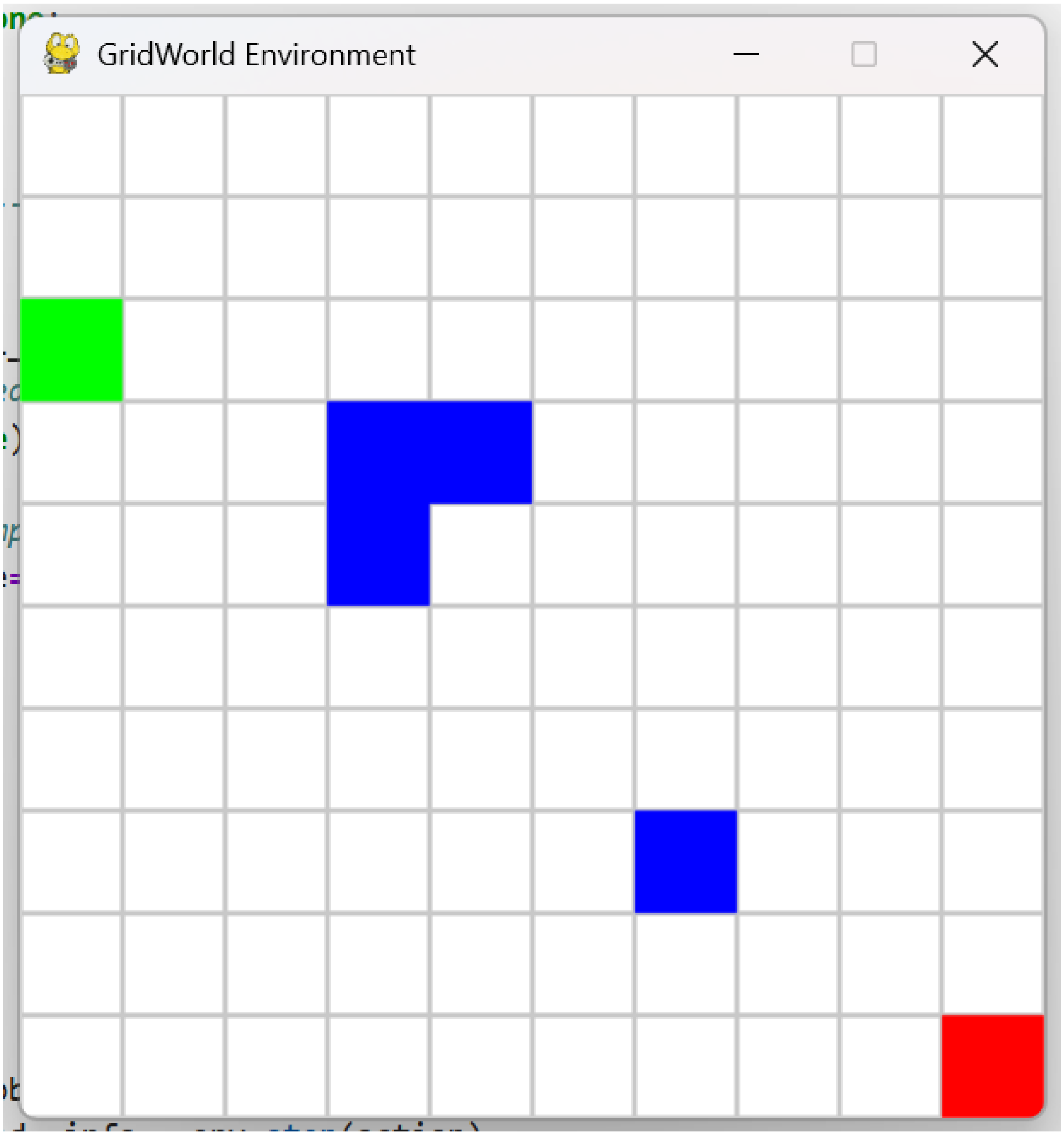


Figure 12: Pygame environment

References

[1] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*, 2nd Edition, 2020. Available online: http://incompleteideas.net/book/the-book-2nd.html

[2] A. Raffin et al., "Stable Baselines3: Reliable Reinforcement Learning Implementations," GitHub Repository, 2020. Available: https://github.com/DLR-RM/stable-baselines3.

[3] Gymnasium Documentation. Available: https://gymnasium.farama.org/.

[4] Pygame Documentation. Available: https://www.pygame.org/docs/.

[5] CS 285: Deep Reinforcement Learning, UC Berkeley, Pieter Abbeel. Course material available: http://rail.eecs.berkeley.edu/deeprlcourse/.

[6] Cover image designed by freepik