Creating a Custom Gymnasium Environment Tutorial

CS 780: Deep Reinforcement Learning

1 Introduction

This tutorial guides you through the process of creating a custom environment using the Gymnasium library. The code for this document is available at Google Colab.

1.1 Bandit Slippery Walk environment

In this Bandit Slippery Walk (BSW) environment, there are three states: 0, 1, and 2, and two possible actions at each state: a_0 and a_1.

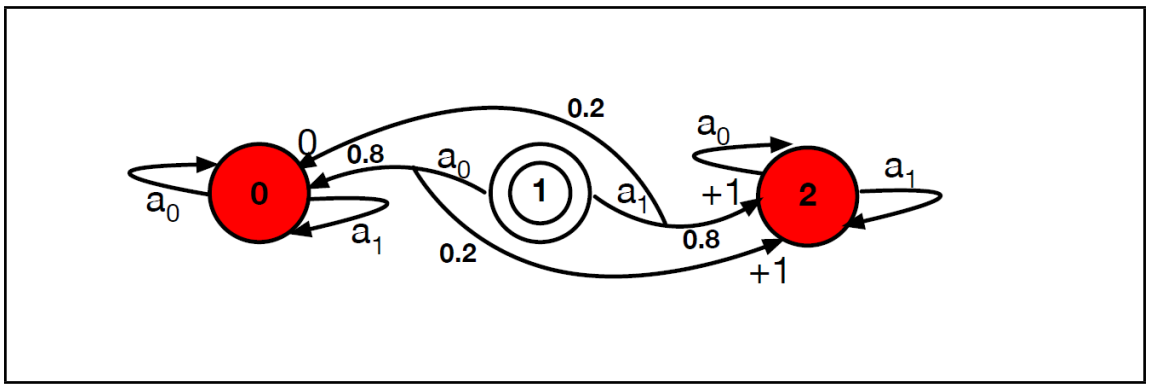


Figure 1: Bandit Slippery Walk environment

The agent starts at state 1 and can move to the right by taking action a_1 or to the left by taking action a_0. The transition from state 1 to state 0 has a probability of 0.8 under action a_0, but there is a 0.2 chance that the agent will slip and move to state 2. From state 1, the agent can move to state 2 with a probability of 0.8 using action a_1, or again, with a probability of 0.2, slip back to state 0. The goal is to reach state 2, which provides a reward of +1.

2 Overview of Gymnasium for Reinforcement Learning

2.1 What is Gymnasium?

Gymnasium is a continuation and enhancement of the original Gym library by OpenAI, designed for reinforcement learning (RL) projects. The Gymnasium interface is simple, pythonic, and capable of representing general RL problems, and has a compatibility wrapper for old Gym environments. Reference: Gymnasium API.

2.2 Key Parts of Gymnasium API

Here is a brief introduction to some of the components of Gymnasium API:

2.2.1 Env Class

The Gymnasium API is centered around the Env class, which encapsulates an environment with arbitrary dynamics. Key methods of this class are as follows:

- step(action): Advances the environment by one timestep using an action from the agent. It returns a tuple containing the next observation, reward, indicators if the episode has terminated or been truncated, and additional info.
- reset(): Resets the environment to an initial state, necessary at the start of a new episode. It returns the initial observation and additional information if needed.
- render(): Provides various modes for visualizing the environment,which can be helpful for understanding how your agent is interacting with the environment.
- close(): Ensures proper closure of the environment, especially important when external resources are utilized.

2.2.2 Spaces

The API provides mathematical sets that can be used to define action space and observation space attributes to specify the format of valid actions and observations within the environment.

2.2.3 Wrappers

It allows for modifications to environments in a modular fashion without altering the original environment code, facilitating customization and extension.

3 Making Your Own Custom Environment

3.1 Getting Started

To develop a custom environment in Gymnasium, it is recommended to operate within a Python virtual environment for isolated dependency management.

3.2 How to Create a Simple Environment

To illustrate the process of subclassing gymnasium.Env, we will implement a very simplistic environment for Slippery bandit walk that was discussed in the classroom to simulate a simple 1D grid world with a slippery surface.

In the given code, the environment consists of a fixed number of states. The agent can move left or right to change its position. There is some stochasticity in the environment which can change the effect of the action taken. The goal of the agent is to navigate to a target position in the environment.

- Observations provide the location of the agent.
- There are 2 actions in our environment, corresponding to the movements “right” and “left”.
- Rewards are binary and sparse, meaning that the immediate reward is always zero, unless the agent has reached the target, then it is 1.

```python
class BanditSlipperyWalkEnv(Env):
    metadata = {"render_modes": ["human", "rgb_array"], "render_fps": 4}

    def __init__(self, render_mode=None, slip_prob = 0.2):
        self.P = {
            0: {
                0: [(1.0, 0, 0.0, True)],
                1: [(1.0, 0, 0.0, True)]
            },
            1: {
                0: [(0.8, 0, 0.0, True), (0.2, 2, 1.0, True)],
                1: [(0.8, 2, 1.0, True), (0.2, 0, 0.0, True)]
            },
            2: {
                0: [(1.0, 2, 0.0, True)],
                1: [(1.0, 2, 0.0, True)]
            }
        }
        self.size = 3 # The size of the 1D grid
        self.window_size = 512  # The size of the PyGame window

        # We have 3 observations, corresponding to each position in the 1-D grid
        self.observation_space = spaces.Discrete(self.size)

        # We have 2 actions, corresponding to "left" & "right"
        self.action_space = spaces.Discrete(2)

        assert render_mode is None or render_mode in self.metadata["render_modes"]
        self.render_mode = render_mode
        """
        If human-rendering is used, `self.window` will be a reference
        to the window that we draw to. `self.clock` will be a clock that is used
        to ensure that the environment is rendered at the correct framerate in
        human-mode. They will remain `None` until human-mode is used for the
        first time.
        """
        self.window = None
        self.clock = None

        # The probability of the slip
        self.slip_prob = slip_prob
```

Figure 2: Subclassing Gymnasium.Env

3.3 Declaration and Initialization

Custom environments are derived from the 'gymnasium.Env' class. This step involves defining essential attributes and methods that encapsulate the environment's dynamics, observation and action spaces and other relevant attributes.

3.3.1 Initialization

The `__init__` method serves as the constructor for your environment, where you define its initial state and configuration. Parameters such as the size of the 1-D grid and the probability threshold for stochastic elements (e.g., slipperiness) are crucial for initializing the environment's dynamics. For rendering purposes, variables to manage the visual output, like window size, are set up here as well.

3.4 Constructing Observations From Environment States

Since we will need to compute observations both in reset and step, it is often convenient to have a private method `_get_obs` that translates the environment's state into an observation. However, this is not mandatory and you may as well compute observations in reset and step separately.

```
    def _get_obs(self):
        """
        A private function to return the locations
        """
        return {"agent": self._agent_location, "target": self._target_location}


    def _get_info(self):
        return {
            "distance": abs(self._agent_location - self._target_location)
        }
```

Figure 3: A private method for returning observations

3.5 Reset Method

The reset method is essential for initializing or reinitializing the environment at the start of an episode. It ensures that the environment is in a known state before beginning or after concluding an episode.

```
def reset(self, seed=None, options=None):
    super().reset(seed=seed)
    self._agent_location = 1
    self._target_location = self.size-1
    self._dead_state = 0


    observation = self._get_obs()
    info = self._get_info()

    if self.render_mode == "human":
        self._render_frame()

    return observation, info
```

Figure 4: Reset method

3.5.1 Seeding and Initialization

Seeding is the process of initializing the random number generator (RNG) with a specific seed to ensure deterministic behavior during experimentation. In Gymnasium environments, it's recommended to use the `self.np_random` RNG provided by the `gymnasium.Env` base class. To correctly seed this RNG, we call `super().reset(seed=seed)`, ensuring that any random operations are predictable and reproducible.

3.5.2 Return Values

Upon resetting, the method returns an initial observation and auxiliary information, which can be constructed using the `_get_obs` and `_get_info` methods defined earlier. These returned values provide the initial state of the environment and any relevant information that might be useful for the agent to begin the episode.

```python
def step(self, action):

    prev_location = self._agent_location
    transitions = self.P[prev_location][action]
    probabilities, next_states, rewards, terminals = zip(*transitions)

    # Randomly select a transition based on the probabilities
    index = random.choices(range(len(probabilities)), weights=probabilities, k=1)[0]
    self._agent_location, reward, terminated = next_states[index], rewards[index], terminals[index]

    truncated = False
    observation = self._get_obs()
    info = self._get_info()

    info["log"] = {"current_state": prev_location,
                   "action":action,
                    "next_state": self._agent_location}

    if self.render_mode == "human":
        self._render_frame()

    # Return the required 5-tuple
    return observation, reward, terminated, truncated, info
```

Figure 5: Step method

3.6 Step Method

The step method encapsulates the core logic of the environment. It processes the agent's action, updates the state of the environment accordingly, and returns a tuple containing the new observation, the reward obtained, indicators of whether the episode has terminated or been truncated, and any additional info.

The `step` function progresses the environment's state by taking an action, randomly choosing the next state based on predefined probabilities, and providing the resulting state, reward, and completion status. The `P` data structure holds the transition dynamics, including probabilities, potential next states, rewards, and whether a state is terminal, dictating how actions translate into outcomes within the environment.

3.7 Rendering

For visualization, we use PyGame, a popular library for writing video games in Python. PyGame provides a rich set of functionalities to create graphical applications, which can be leveraged to visually represent the state of the environment.

3.8 Close

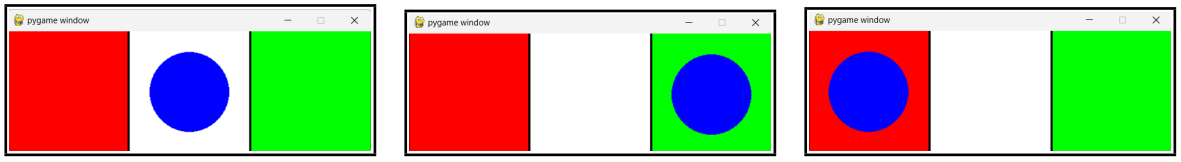
The close method in a Gymnasium environment is essential for resource management. It ensures that any resources, such as open windows or files used by the environment, are properly closed and cleaned up when the environment is no longer needed.

3.9 Registering Environments

To recognize and use custom environments, it must be registered. We placed the statement to register the environment directly in the python file where the custom environment is implemented. For more details, refer here.

3.10 Test your custom environment

Once an environment is registered, it can be instantiated and used within your projects. In our example code, within a loop of 10 iterations, we randomly select actions from the environment's action space to simulate an agent's decision-making process, applies these actions to the environment, and observes the outcomes, including the new state, received reward, and whether the episode has ended or been truncated. If an episode terminates, the environment is reset, and the simulation continues. As shown in Figure 6, the state transitions include start, target, and dead states. The start state (Figure 6a) shows the initial condition. The target state, as depicted in Figure 6b, represents the goal. Figure 6c illustrates the dead state, indicating a failure to reach the goal.



(a) Start state (b) Target state (c) Dead state

Figure 6: State transitions

4 Exercise

- Design and implement another Bandit slippery environment where an action can lead to one of the the following possibilities: the agent can stay in the same state, or move to the left or move to the right.
- The shared code implements a 1-D grid of size 3. Try to extend the environment where you specify the size or the number of cells when you create the environment.

5 Resources

1. Gymnasium Online documentation: `https://gymnasium.farama.org/`
2. Gymnasium: Tutorial for creating a custom environment: `https://gymnasium.farama.org/tutorials/gymnasium_basics/environment_creation/`
3. Pygame getting started: `https://www.pygame.org/wiki/GettingStarted`
4. Gymnasium Github: `https://github.com/Farama-Foundation/Gymnasium`