Reinforcement Learning

Gymnasium

Jordi Casas Roma

jordi.casas.roma@uab.cat

September 12, 2025

UAB
Universitat Autònoma
de Barcelona

Table of Contents

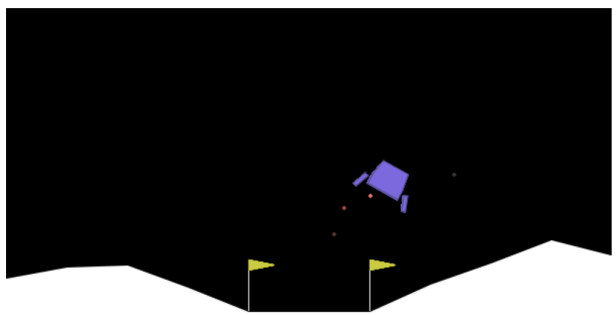
Introduction

Gymnasium

- An **API standard** for reinforcement learning with a diverse collection of reference environments.
- `https://gymnasium.farama.org/index.html`
- Version 1.2.0

Gymnasium

An API standard for reinforcement learning with a diverse collection of reference environments



Gymnasium

Table of Contents

Table of Contents

Introduction
Description

In a reinforcement learning environment there are **two main entities**:

- The **agent** is an entity that plays an active role in the environment, making decisions about the actions it will execute in the environment.
  - From the most pragmatic point of view, the agent is an application (code fragment) that implements a certain policy to execute certain actions in the environment.
  - Basically, this policy decides what action is required at any given time based on observations provided by the environment.
- The **environment** is a model external to the agent that has the responsibility of providing observations and rewards to the agent's actions.
  - The environment changes its state based on the agent's actions.

Introduction
Description

- We can see that for each action (A_t) that the agent performs at time instant t, the environment updates its state and returns:
  1. the reward associated with the action (R_{t+1}), and
  2. the observation associated with the new state (O_{t+1})

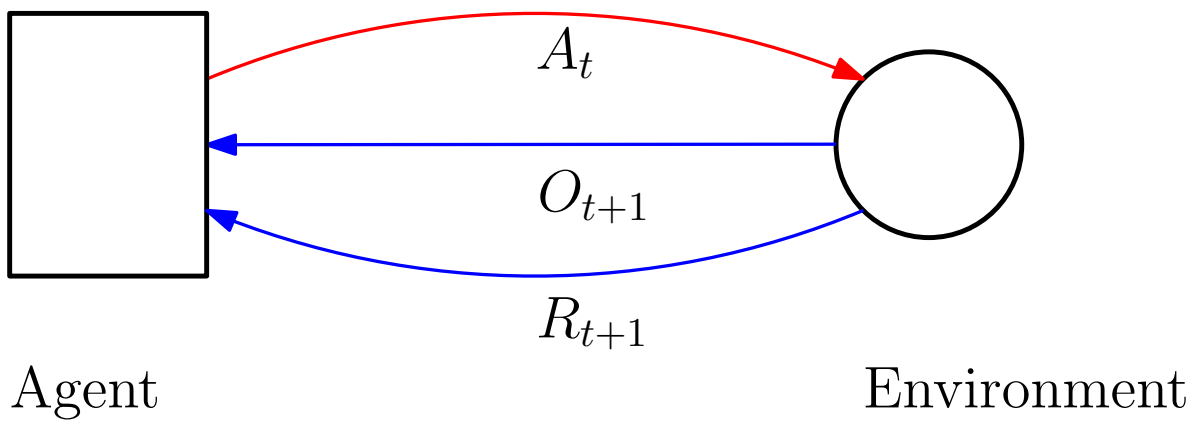


Table of Contents

Introduction

Basic operating diagram

Main code block:

- The classes corresponding to the environment (`Environment`) and the agent (`Agent`) are initialized.
- Once the classes are initialized, the agent iterates indefinitely using the `step()` method until the episode ends.

```
if __name__ == "__main__":
    env = Environment()
    agent = Agent()

    while not env.is_done():
        agent.step(env)

    print("Total reward got: {}".format(agent.total_reward))
```

Introduction

The Agent

The **agent** class:

1. The `__init__()` method: initializes the internal variables of the agent.
2. The `step()` method: execute the actions at time t.

```
class Agent:
    def __init__(self):
        # Agent init
        self.total_reward = 0.0

    def step(self, env: Environment):
        current_obs = env.get_observation()
        actions = env.get_actions()
        action = <SELECT desired action>
        reward = env.action(action)
        self.total_reward += reward
```

Introduction
The Agent

The **agent** class:

1. The `__init__()` method: initializes the internal variables of the agent.
2. The `step()` method: execute the actions at time t.

```
1  class Agent:
2      def __init__(self):
3          # Agent init
4          self.total_reward = 0.0
5
6      def step(self, env: Environment):
7          current_obs = env.get_observation()
8          actions = env.get_actions()
9          action = <SELECT desired action>
10         reward = env.action(action)
11         self.total_reward += reward
```

- What does the `step()` method do?

Introduction

Basic operating diagram

What does the `step()` method do?

1. Obtains the value of the observation of the current state, which is provided by the environment.
2. Obtains the values of possible actions, which is also provided by the environment.
3. Next, with the previous information, the agent must select the action that it wants to perform.
   - At this point is where the agent implements the selected method to choose the action to perform among the set of possible actions.
   - This set can be limited or unlimited, depending on the characteristics of the environment
4. Finally, it communicates the action to the environment, which returns the associated reward.

Introduction

The Environment

The **environment** class:

1. The environment must provide several methods to report current observations, possible actions, the status of the episode.
2. The `action()` method, which applies a certain action to the environment and updates it, providing the indicated reward.

```
class Environment:
    def __init__(self):
        # Environment initialization

    def get_observation(self):
        # Returns the observation values of the current state

    def get_actions(self):
        # Returns the list of possible actions

    def is_done(self):
        # Returns a boolean indicating whether the episode has ended

    def action(self, action):
        # Apply the indicated action in the environment
        # Returns the associated reward
```

Table of Contents

The Gymnasium API

Introduction

Introduction:

1. **Gymnasium** is a library for developing and comparing reinforcement learning algorithms.
2. Gymnasium makes no assumptions about the structure of the agent you want to implement, and is compatible with any numerical computing library, such as TensorFlow, Theano, or PyTorch.
3. Gymnasium offers a collection of test environments that can be used to train, evaluate and compare reinforcement learning algorithms.
4. These environments have a shared interface (API) that allows general algorithms to be developed.

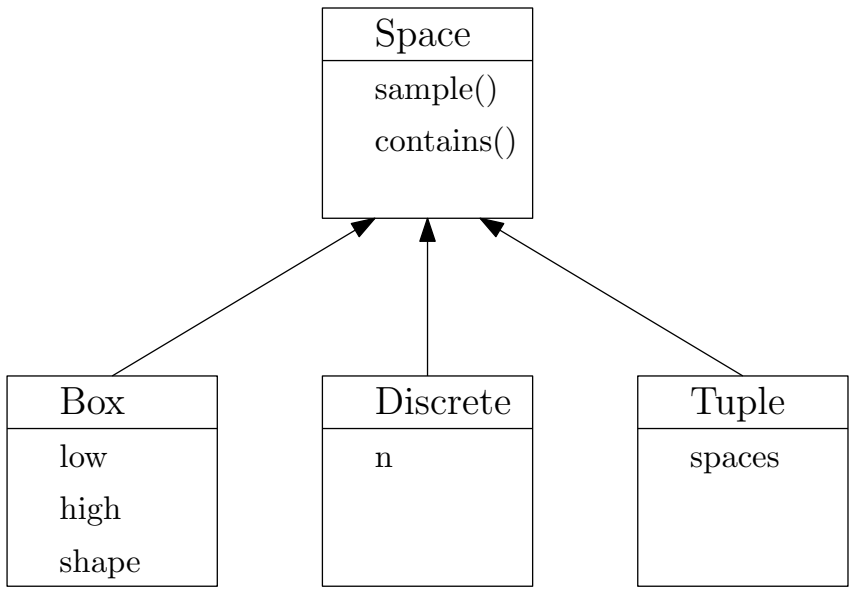
Table of Contents

The Gymnasium API

Space

Space:

1. The `Space` class allows you to define what the actions and states will be like, defining the type and range of allowed values.
2. There are multiple subclasses that implement the abstract class `Space`.
3. Any environment contains two members of type `Space`, called `action_space` and `observation_space`.



The Gymnasium API

Space

The methods of **Space** class:

- The superclass `Space` includes two relevant methods:
  - **sample()**, which returns a random sample of the space.
  - **contains(x)**, which checks whether the argument x belongs to the domain space.
- Both methods are abstract and are re-implemented in each of their subclasses.

The Gymnasium API

Space

The methods of **Discrete** subclass:

- The class `Discrete` represents a set of mutually exclusive elements, numbered from 0 to $n - 1$.
- Its main attribute (n) is a count of the elements it describes.

The Gymnasium API

Space

The methods of **Discrete** subclass:

- The class `Discrete` represents a set of mutually exclusive elements, numbered from 0 to $n - 1$.
- Its main attribute (n) is a count of the elements it describes.

Movement on a chess board

For example, `Discrete(n=4)` can be used to define an action space that represents movement in four directions, where the value 0 indicates move left, 1 moves right, 2 moves up, and 3 indicates moves down.

The Gymnasium API

Space

The methods of **Box** subclass:

- The class `Box` represents an *n*-dimensional tensor of rational numbers with the interval `[low, high]`.
- This class is widely used, since it allows representing a large amount and variety of information of a very different nature.

The Gymnasium API

Space

The methods of **Box** subclass:

- The class `Box` represents an *n*-dimensional tensor of rational numbers with the interval `[low, high]`.
- This class is widely used, since it allows representing a large amount and variety of information of a very different nature.

Acceleration pedal

For example, an accelerator pedal can be represented by a single value between 0.0 and 1.0, indicating the active percentage of the accelerator.

```
Box(low=0.0, high=1.0, shape=(1,), dtype=np.float32)
```

- The `shape` argument indicates that this is a tuple of length 1 with a single value.
- The range of this value is determined by the `low` and `high` values.
- The `dtype` parameter specifies the type of the variable, which in this particular case is a 32-bit NumPy float.

The Gymnasium API

Space

The methods of **Tuple** subclass:

- The `Tuple` class allows us to combine several instances of the `Space` class to create action and observation spaces of variable complexity and a high degree of adaptation to any environment.

The Gymnasium API

Space

The methods of **Tuple** subclass:

- The `Tuple` class allows us to combine several instances of the `Space` class to create action and observation spaces of variable complexity and a high degree of adaptation to any environment.

Driving a vehicle

The vehicle has several controls, including the angle of the steering wheel, the position of the brake pedal, and the position of the accelerator pedal. These three controls can be specified by **three float values** in a single `Box` instance.
However, the vehicle requires additional discrete controls: turn signal (which could be off, left or right) and the horn (on or off).
To combine all of this into an action space specification class:

```
Tuple(spaces=(Box(low=-1.0, high=1.0, shape=(3,),
dtype=np.float32), Discrete(n=3), Discrete(n=2)))
```

The Gymnasium API

Space

Atari Games example

Suppose we want to represent an observation of an **Atari game screen**, which is an image of size 210×160 in RGB (red, green and blue) using one byte to encode each colour, that is, 256 possible values.

How can we encode this information?

The Gymnasium API

Space

Atari Games example

Suppose we want to represent an observation of an **Atari game screen**, which is an image of size 210×160 in RGB (red, green and blue) using one byte to encode each colour, that is, 256 possible values.

How can we encode this information?

Atari game

We can represent this information using the following encoding:

```
Box(low=0, high=255, shape=(210, 160, 3), dtype=np.uint8)
```

Table of Contents

The Gymnasium API

Environment

Environment: The main goal of Gymnasium is to provide a large collection of environments with a common interface to allow different implementations to be evaluated and compared.

The environment is represented by the class `Env`, which has the following methods:

- `action_space`: Provides a specification for the actions allowed in the environment.
- `observation_space`: Specifies the observations provided by the environment.
- `reset()`: This method resets the environment to its initial state, returning the initial observation vector.
- `step()`: This method allows us to execute a certain action and returns information about its result, that is, the next observation, the reward and the end of the episode indicator.
- `render()`: This method allows us to obtain the observation in a human-friendly form.

The Gymnasium API

Environment

The **Env** class:

- The `step()` method is the central method in the environment's functionality. When this method is called, **several actions are performed**:
  1. Tell the environment what action we will execute in the next step.
  2. Get the new observation of the environment after this action.
  3. Get the reward generated by the action.
  4. Get a boolean indicating whether the episode has ended.
- The method has a single main parameter, which is the action that the agent wants to perform.

The Gymnasium API

Environment

The method `step()` of **Env** class:

The method returns a five-element tuple with the following types and meanings:

1. `observation`: NumPy vector or a matrix with the data of the next observation.
2. `reward`: Reward value, usually as a decimal (float) value.
3. `terminated`: Boolean flag, whether the agent reaches the terminal state.
4. `truncated`: Boolean flag, whether the truncation condition is satisfied (time limit, etc.).
5. `info`: additional information about the environment. In general, this value is usually ignored, although it can provide interesting information in certain cases.

The Gymnasium API

Environment

To instantiate the `Env` class, you must use the `make()` function of the Gym API.

1. This function has a single argument that must be indicated: the **name of the environment** to be created, which takes the form `EnvironmentName-vN`, where N is the number used to distinguish between different versions of the same environment (when, for example, For example, some bugs are fixed or other important changes are made).

The Gymnasium API

Environment

To instantiate the `Env` class, you must use the `make()` function of the Gym API.

1. This function has a single argument that must be indicated: the **name of the environment** to be created, which takes the form `EnvironmentName-vN`, where N is the number used to distinguish between different versions of the same environment (when, for example, For example, some bugs are fixed or other important changes are made).

For example, to create the environment "CartPole-v0" the following code must be executed:

```
1 import gym
2
3 env = gym.make('CartPole-v0')
```

Table of Contents

The Gymnasium API

Wrappers

Wrappers:

- Wrappers are a convenient way to modify an existing environment without having to alter the underlying code directly.
- Wrappers can also be chained to combine their effects.

Wrapper example

For example, when using image-based inputs, it is common to use a range of values [0,1] instead of [0,255], which are the standard values in RGB images.

The Gymnasium API

Wrappers

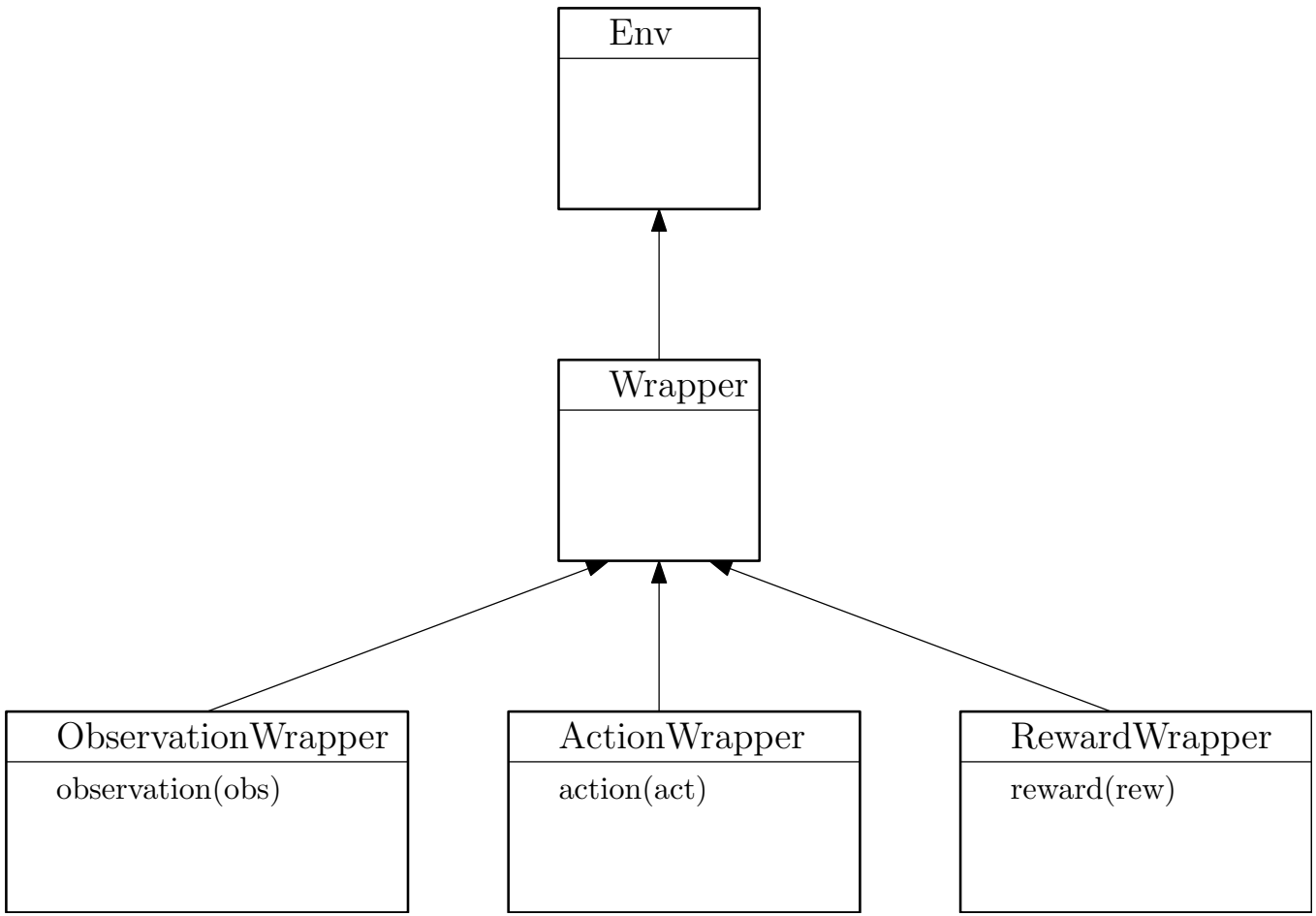
The **wrappers'** hierarchy:

- `ObservationWrapper` is used to modify the **observations** returned by the environment.
- `RewardWrapper` is used to modify the **rewards** returned by the environment.
- `ActionWrapper` allows you to modify the **actions** sent to the environment.

The Gymnasium API

Wrappers

The **wrappers'** hierarchy:



The Gymnasium API

Wrappers

In order to wrap an environment, we must:

1. First, initialize a base environment.
2. Then, we can pass this environment along with optional parameters to the wrapper's constructor.

```
import gymnasium as gym
from gymnasium.wrappers import RescaleAction

base_env = gym.make("Hopper-v4")
base_env.action_space

>> Box(-1.0, 1.0, (3,), float32)

wrapped_env = RescaleAction(base_env, min_action=0, max_action=1)
wrapped_env.action_space

>> Box(0.0, 1.0, (3,), float32)
```

The Gymnasium API

Wrappers

The implementation of the `Wrapper` class requires defining a `__init__` method that accepts the environment as a parameter.

```
class BasicWrapper(gym.Wrapper):
    def __init__(self, env):
        super().__init__(env)
        self.env = env

    def step(self, action):
        next_state, reward, terminated, truncated, info = self.env.
    step(action)
        # modifications
        return next_state, reward, terminated, truncated, info
```

And the new class `BasicWrapper` is instantiated as follows:

```
env = BasicWrapper(gym.make("EnvironmentName-vN"))
```

The Gymnasium API

Wrappers

Some **interesting** wrappers (among many others):

- `RecordVideo`: This wrapper records videos of rollouts.
- `RecordEpisodeStatistics`: This wrapper will keep track of cumulative rewards and episode lengths.
- Check at:
  - `https://gymnasium.farama.org/api/wrappers/misc_wrappers/`

Table of Contents

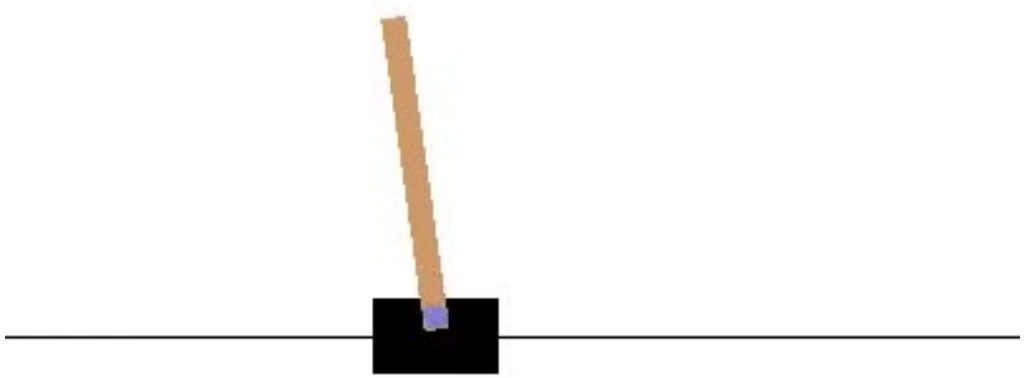
Table of Contents

Classic RL Examples

CartPole

The **CartPole** environment:

- A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track.
- The pendulum is placed upright on the cart and the goal is to balance the pole by applying forces in the left and right direction on the cart.



Gymnasium

Classic RL Examples

CartPole

Observation Space:

- The observation is a `ndarray` with shape (4,) with the values corresponding to the following positions and velocities:

Num	Observation	Min	Max
0	Cart Position	-2.4	2.4
1	Cart Velocity	$-\infty$	$+\infty$
2	Pole Angle	$\approx -24°$	$\approx +24°$
3	Pole Angular Velocity	$-\infty$	$+\infty$

Classic RL Examples

CartPole

Action Space:

- The action is a `ndarray` with shape (1,) which can take values {0, 1} indicating the direction of the fixed force the cart is pushed with:

Values	Action
0	Push cart to the left
1	Push cart to the right

- The velocity that is reduced or increased by the applied force is not fixed, and it depends on the angle the pole is pointing.
- The centre of gravity of the pole varies the amount of energy needed to move the cart underneath it.

Classic RL Examples

CartPole

Rewards:

- Since the goal is to keep the pole upright for as long as possible, a reward of $+1$ for every step taken, including the termination step, is allotted.
- The **threshold for rewards** is 500 for v1 and 200 for v0.

Classic RL Examples

CartPole

Starting State:

- All observations are assigned a uniformly random value in (-0.05, 0.05).

Classic RL Examples

CartPole

Episode End:

The episode ends if **any one of the following occurs**:

1. Pole Angle is greater than $\pm 12^{\circ}$
2. Cart Position is greater than ± 2.4 (center of the cart reaches the edge of the display)
3. Episode length is greater than 500 (200 for v0)

Classic RL Examples

CartPole

Relevant **questions** regarding this environment:

1. How is the **observation** space?
2. How is the **action** space?
3. What about the **rewards**?

Classic RL Examples

CartPole

Relevant **questions** regarding this environment:

1. How is the **observation** space?
2. How is the **action** space?
3. What about the **rewards**?

1. Continuos, `ndarray` (4,)
   - position $\in (-2.4, 2.4)$
   - velocity $\in (-\infty, \infty)$
   - angle $\in (-24, 24)$
   - angular velocity $\in (-\infty, \infty)$
2. Discrete, i.e. $\{0, 1\}$
3. Discrete, i.e. $+1$ for each step "alive"

Table of Contents

Classic RL Examples

FrozenLake

The **FrozenLake** environment:

- Frozen lake involves crossing a frozen lake from **start** to **goal** without falling into any holes by walking over the frozen lake.
- The player may not always move in the intended direction due to the slippery nature of the frozen lake.

S	F	F	F
F	H	F	H
F	F	F	H
H	F	F	G

Classic RL Examples

FrozenLake

Observation Space:

- The observation is an integer value representing the player's current position as:

  $$\text{current\_position} = \text{current\_row} \times \text{n\_rows} + \text{current\_col}$$

  where both the row and col start at 0.

Num	Observation	Min	Max
0	Current position	0	$n^2 - 1$

where n is the size of the grid ($n \times n$).

Classic RL Examples

FrozenLake

Action Space:

- The action shape is (1,) in the range {0, 3} indicating which direction to move the player.

Values	Action
0	Move left
1	Move down
2	Move right
3	Move up

Classic RL Examples

FrozenLake

Rewards:

- Reward schedule:

State	**Reward**
Reach goal	+1
Reach hole	0
Reach frozen	0

Classic RL Examples
FrozenLake

Starting State:

- The episode starts with the player in state `[0]`, i.e. location `[0, 0]`.

Classic RL Examples

FrozenLake

Episode End:

The episode ends if **any one of the following occurs**:

1. Termination:
   - The player moves into a hole.
   - The player reaches the goal.
2. Truncation (when using the `time_limit` wrapper):
   - The length of the episode is 100 for 4×4 environment,
   - or 200 for 8×8 environment.

Classic RL Examples

FrozenLake

Information:

1. `step()` and `reset()` return a `dict` with the following keys:
   - Transition probability for the state (p).
2. `is_slippery` attribute:
   - If **true** the player will move in intended direction with probability of $\frac{1}{3}$ else will move in either perpendicular direction with equal probability of $\frac{1}{3}$ in both directions.

Classic RL Examples

FrozenLake

Relevant **questions** regarding this environment:

1. How is the **observation** space?
2. How is the **action** space?
3. What about the **rewards**?

Classic RL Examples

FrozenLake

Relevant **questions** regarding this environment:

1. How is the **observation** space?
2. How is the **action** space?
3. What about the **rewards**?

1. Discrete, integer value $\in \{0, n^2 - 1\}$
   - where n is the size of the grid ($n \times n$).
2. Discrete, i.e. $\{0, 1, 2, 3\}$
3. Discrete, i.e. $+1$ for reaching the goal state

Table of Contents

Biliography
References

Some relevant references:

1. **Farama Foundation**, *Gymnasium Documentation*. Accessed on September 4, 2025. `https://gymnasium.farama.org/index.html`