Gymnasium Environments

Agenda

- This week:
  - Gymnasium custom environment: an upgrade for our Lab 1 CliffWalking environment class
  - Pygame for rendering
  - BlocksWorld-v0 environment (future Assignment 1)
  - stable-baselines3

What we've implemented so far

- Lab 1 Agent: <your_algonquin_id>_lab2_qlearning_agent.py
- Lab 1 Environment: <your_algonquin_id>_lab2_cliff_env.py (simple "homemade" gridworld and CliffWalking)

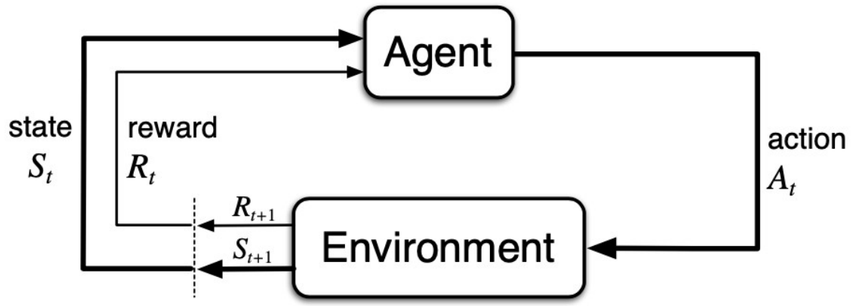


Figure 3.1: The agent–environment interaction in a Markov decision process.

- Lab 2: <your_algonquin_id>_lab2_cliff_env.py replaced with gymnasium environment
- Lab 2: <your_algonquin_id>_lab2_qlearning_agent.py updated to work with gymnasium environment

Where the implementation is going

- Lab 2: upgrade our cliffwalking environment to Gymnasium
- Turn the Prolog blocks world into a Gymnasium BlocksWorld-v0 environment (assignment 1)
- Use PyGame for rendering our environments
- Our <your_algonquin_id>_lab2_qlearning_agent.py implementation still plays the role of agent
- stable-baselines3 is a set of standard RL algorithms that work with Gymnasium environments
- We can use the stable-baselines3 algorithms: DQN, PPO, for new agents

Diagram comparison (Assignment 1)

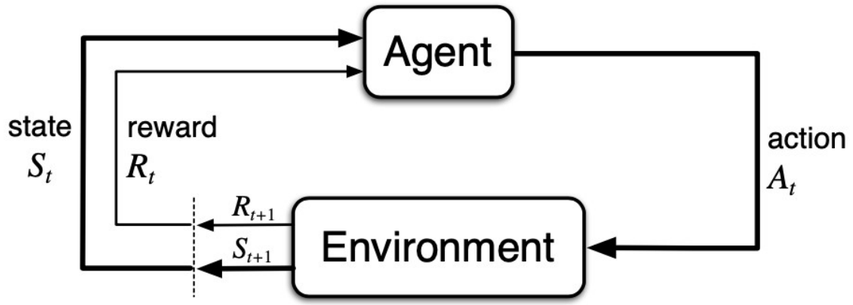
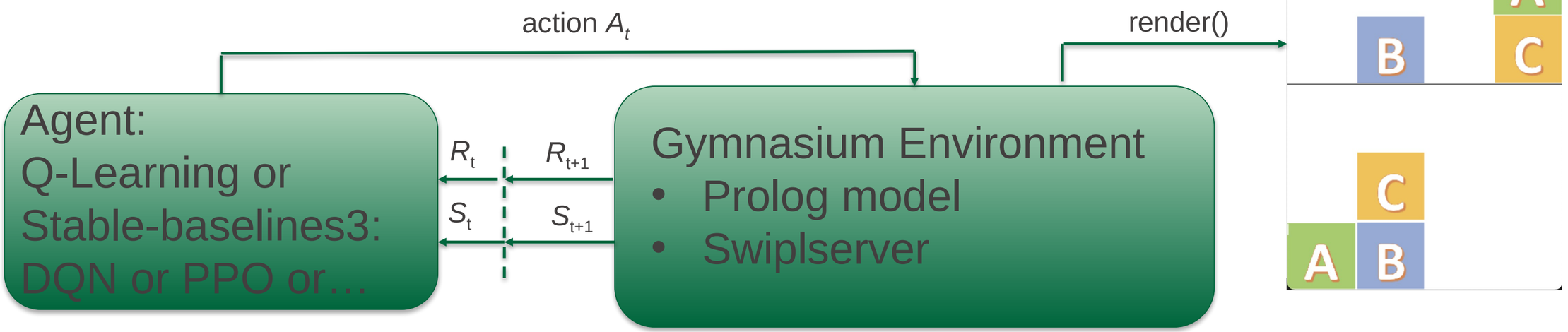


Figure 3.1: The agent–environment interaction in a Markov decision process.



Important Sites

- Basic Components from Lab1:

https://medium.com/data-science/math-of-q-learning-python-code-5dcbdc49b6f6

Custom environment for Gymnasium Lab 2:

https://gymnasium.farama.org/tutorials/gymnasium_basics/environment_creation/#

- PyGame is often used by Gymnasium Environments for rendering:

https://dr0id.bitbucket.io/legacy/pygame_tutorial00.html

https://dr0id.bitbucket.io/legacy/pygame_tutorial01.html

- Stable-baselines3:

https://stable-baselines3.readthedocs.io/en/master/guide/install.html

Gym to Gymnasium

- Gymnasium is common as the successor to Gym

```
import gymnasium as gym
```

- Reset method takes seed as parameter (for reproducing random sequences)

```
reset(self, seed=None, options=None)
```

- Step method returns truncated

```
return observation, reward, terminated, truncated, info
```

Creating a Gymnasium Environment

https://gymnasium.farama.org/tutorials/gymnasium_basics/environment_creation/#

Steps to creating a new Gymnasium environment:

- import gymnasium as gym
- from gymnasium import spaces # note: from gymnasium, not from gym
- Make the <somename>Env python class extend gymnasium.env
- Implement/override the gymnasium.env methods:
  - __init__(self, render_mode=None, size=5)
  - reset(self, seed=None, options=None)
  - step(self, action)->return observation, reward, terminated, False, info
  - render(self)
  - close(self)

Creating a Gymnasium Environment

- Define the state space and action space
- Example from gynmasium docs:

```
# Observations for the gridworld example are dictionaries with the agent's and the target's location.
# Each location is encoded as an element of {0, ..., `size`}^2, i.e. MultiDiscrete([size, size]).
self.observation_space = spaces.Dict(
    {
        "agent": spaces.Box(0, size - 1, shape=(2,), dtype=int),
        "target": spaces.Box(0, size - 1, shape=(2,), dtype=int),
    } )
 # We have 4 actions, corresponding to "right", "up", "left", "down"
 self.action_space = spaces.Discrete(4)
```

Creating a Gymnasium Environment

- Define the state space and action space
- In this case, the observation is a dictionary of 3-letter-state integers:

```
self.observation_space = spaces.Dict(
    {
        "agent": spaces.Discrete(len(self.states)),
        "target": spaces.Discrete(len(self.states)),
    }
)
self.action_space = spaces.Discrete(len(self.actions))
```

Creating a Gymnasium Environment

- Define the state space and action space
- In this case (per the coming Assignment 1 document), the observation is a single larger integer representing both the current state and target at the same time:

```
self.observation_space = spaces.Discrete(len(self.states))
self.action_space = spaces.Discrete(len(self.actions))
```

StableBaselines MlpPolicy vs MultiInputPolicy

- Dictionary observations:

```
self.observation_space = spaces.Dict(
    {
        "agent": spaces.Discrete(len(self.states)),
        "target": spaces.Discrete(len(self.states)),
    }
)
self.action_space = spaces.Discrete(len(self.actions))
```

- In the agent

```
# For spaces.Dict observations, use MultiInputPolicy
import gymnasium as gym
env = gym.make("aisd_examples/BlocksWorld-v0", render_mode="human")
model = DQN("MultiInputPolicy", env, verbose=1)
model.learn(total_timesteps=10000, log_interval=4)
```

StableBaselines MlpPolicy vs MultiInputPolicy

- Observation is a discrete number:

```
self.observation_space = spaces.Discrete(len(self.states))
self.action_space = spaces.Discrete(len(self.actions))
```

- In the agent

for spaces.Discrete observation_space, use MlpPolicy (Multi-Layer-Perceptron)

```
import gymnasium as gym
env = gym.make("aisd_examples/BlocksTarget-v0", render_mode="human")
model = DQN("MlpPolicy", env, verbose=1)
model.learn(total_timesteps=10000, log_interval=4)
```

Put our environment in a package to install

As outlined in the Gymnasium docs (they use the name gymnasium_env):

```
<algonquin_id>_blocksworld_env/
    pyproject.toml
    blocksworld_env/          # gymnasium_env/ would be at this level
        __init__.py
        envs/
            __init__.py
            blocks_world.py
```

Using our custom environment

Using our custom blocksworld environment with Discrete observations <your_algonquin_id>_lab2_qlearning_agent.py from Lab 2:

```
#from medium_qlearning_env import Env

import blocksworld_env

import gymnasium as gym

import matplotlib.pyplot as plt

import numpy as np

import time

import os

# create environment

#env = Env()

env = gym.make(”blocksworld_env/BlocksWorld-v0", render_mode="human")

# QTable : contains the Q-Values for every (state,action) pair

qtable = np.random.rand(env.observation_space.n, env.action_space.n).tolist()
```

Using our custom environment

Using our environment (Discrete observations) with stable-baselines3:

```
import gymnasium as gym

import blocksworld_env

from stable_baselines3 import DQN

env = gym.make("blocks_world/BlocksWorld-v0", render_mode="human")

model = DQN("MlpPolicy", env, verbose=1)         # for non-dictionary Discrete observation where state and target are combined

model.learn(total_timesteps=10000, log_interval=4)

model.save("dqn_blocks")

del model # remove to demonstrate saving and loading

model = DQN.load("dqn_blocks")

obs, info = env.reset()

while True:

    action, _states = model.predict(obs, deterministic=True)

    obs, reward, terminated, truncated, info = env.step(action)

    if terminated or truncated:

        obs, info = env.reset()
```

Prolog Blocks world environment (Assignment 1)

```
% situation calculus blocks world (just one fluent, clear is defined with on)
:- dynamic on/3.
% target means this is the goal (not used because Python picks random state)
target:-on(a,4,[]),on(b,c,[]),on(c,a,[]).
% reset asserts the initial state
reset:-
   retractall(on(_,_,[])),
   assert(on(a,1,[])),
   assert(on(b,3,[])),
   assert(on(c,a,[])).
```

Prolog Blocks world environment

```
state(S):-
   (block(A);place(A)),dif(A,a),
   (block(B);place(B)),dif(B,b),
   (block(C);place(C)),dif(C,c),
   dif(A,B),
   dif(B,C),
   dif(A,C),
   grounded(A,B,C),
   atomics_to_string([A,B,C],S).
```

```
grounded(A,B,C):-
   legal(A,B,C),
   (place(A);place(B);place(C)),!.


legal(A,B,C):-
   block(A),block(B),A=b,B=a,!,fail
   ;
   block(B),block(C),B=c,C=b,!,fail
   ;
   block(A),block(C),A=c,C=a,!,fail
   ;
   true.
```

Prolog Blocks world environment

```
% action(Act) means Act is concievable

action(Act):-
   Act = move(A,B,C),
   block(A),
   (block(B);place(B)),
   (block(C);place(C)),
   dif(A,B),
   dif(A,C),
   dif(B,C).
```

```
current_state(S):-
   on(a,A,[]),
   on(b,B,[]),
   on(c,C,[]),
   atomics_to_string([A,B,C],S).
step(Act):- poss([Act]),
   on(a,A,[Act]),
   on(b,B,[Act]),
   on(c,C,[Act]),
   retractall(on(_,_,[])),
   assert(on(a,A,[])),
   assert(on(b,B,[])),
   assert(on(c,C,[])).
```

PyGame for rendering

```
# import the pygame module, so you can use it

import pygame

class Display():

    def __init__(self):

        # initialize the pygame module

        pygame.init()

        # load and set the logo

        logo = pygame.image.load("logo32x32.png")

        pygame.display.set_icon(logo)

        pygame.display.set_caption("Blocks World")
```

```
# create a surface on screen that has the given size
# define a variable to control the main loop
self.running = True

IMAGE_SIZE_X = 100
IMAGE_SIZE_Y = 100
DEFAULT_IMAGE_SIZE = (IMAGE_SIZE_X, IMAGE_SIZE_Y)
self.screen = pygame.display.set_mode((4*IMAGE_SIZE_X,6*IMAGE_SIZE_
```

#see screen.py on brightspace