Gymnasium and the minigrid environment

Hugo Touchette

Department of Mathematical Sciences, Stellenbosch University, Stellenbosch, South Africa

- Last updated: 22 May 2025
- Based on material prepared by Prof. H Engelbrecht
- Updated from gym to gymnasium
- Python 3.10

1. Gymnasium

In [7]:
```
import numpy as np
import gymnasium as gym
from minigrid.wrappers import ImgObsWrapper
```

OpenAI created the RL package 'gym' but stopped maintaining it, presumably because they are too busy with ChatGPT. The Farama foundation took over the package, creating a fork of gym that they named 'gymnasium'. This is the package that we will be using:

https://farama.org

https://gymnasium.farama.org

https://github.com/Farama-Foundation/Gymnasium

This package is used extensively nowadays for teaching purposes and for benchmarking basic RL methods using a number of pre-programmed environments or RL problems, including navigation problems in grid worlds, control problems such as the cartpole and inverted pendulum, as well as robot-related tasks. The advantage of gymnasium is that we do not have to code these environments. Moreover, the interface for using them is fairly simple, consisting of basic functions or methods that implement the basic steps required in RL, namely

1- Define/load environment/problem (gym.make)

2- Initialise environment (gym.reset)

3- Evolution of the system/environment as the agent takes an action, receiving in the process an observation, a reward, and other information about the completion of the task (gym.step).

2. Minigrid environment

The problem or environment that we consider is called 'MiniGrid-Empty-8x8-v0', which was part of another minigrid package under gym, but is now part of gymnasium:

https://minigrid.farama.org

https://minigrid.farama.org/environments/minigrid/EmptyEnv/

In this section, we go over the details of this environment, covering

- The gridworld environment
- The action space of the agent
- The observation returned by the environment
- The reward signal

In [8]:
```
env = gym.make('MiniGrid-Empty-8x8-v0')
```

2.1. Gridworld environment

The gridworld that the agent lives in is an 8×8 grid of tiles or cells containing zero or one object, with walls around, so the agent actually moves in a 6×6 grid. Cells that do not contain an object have the value None.

2.2. Actions

The agent can perform one of 6 actions at each step:

- Turn left: 0
- Turn right: 1
- Move forward: 2
- Pick up an object: 3
- Drop the object being carried: 4
- Toggle (open doors, interact with objects): 5

For the assignment, we will only use the first three actions: 'Turn left', 'Turn right' and 'Move forward'.

To explore the environment, the agent needs to be able to randomly select an action to take (the basic of ε-greedy exploration). This can be done by

generating a random integer from 0 to $n-1$, n being the number of actions in the actions space (because of the numbering from 0):

In [9]:
```
a = np.random.randint(0, env.action_space.n-1)

print("Action a: %d was generated" % a)
```

```
Action a: 1 was generated
```

2.3. Observation returned by the environment

It is important to reset the environment after each episode has ended. This sets the number of steps the agent has taken to 0, and puts the agent back at the starting position. Resetting the environment is also used to obtain the first observation representing the current state S.

In [10]:
```
# Reset the environment to get first observation
obs, info = env.reset()
print(obs)
```

```
{'image': array([[[2, 5, 0],
        [2, 5, 0],
        [2, 5, 0],
        [2, 5, 0],
        [2, 5, 0],
        [2, 5, 0],
        [2, 5, 0]],

       [[2, 5, 0],
        [2, 5, 0],
        [2, 5, 0],
        [2, 5, 0],
        [2, 5, 0],
        [2, 5, 0],
        [2, 5, 0]],

       [[2, 5, 0],
        [2, 5, 0],
        [2, 5, 0],
        [2, 5, 0],
        [2, 5, 0],
        [2, 5, 0],
        [2, 5, 0]],

       [[2, 5, 0],
        [1, 0, 0],
        [1, 0, 0],
        [1, 0, 0],
        [1, 0, 0],
        [1, 0, 0],
        [1, 0, 0]],

       [[2, 5, 0],
        [1, 0, 0],
        [1, 0, 0],
        [1, 0, 0],
        [1, 0, 0],
        [1, 0, 0],
        [1, 0, 0]],

       [[2, 5, 0],
        [1, 0, 0],
        [1, 0, 0],
        [1, 0, 0],
        [1, 0, 0],
        [1, 0, 0],
        [1, 0, 0]],

       [[2, 5, 0],
        [1, 0, 0],
        [1, 0, 0],
        [1, 0, 0],
        [1, 0, 0],
        [1, 0, 0],
        [1, 0, 0]]], dtype=uint8), 'direction': 0, 'mission': 'get t
o the green goal square'}
```

The default observation returned contains information that is superfluous, such as the 'direction' and the 'mission'. We are only interested in the 'image' data, though the direction data could also be useful. We use a wrapper that is provided by the minigrid environment (see import at the top) to extract only the observation representing the partial view that the agents has of the environment.

In [11]:
```
env = ImgObsWrapper(env)
env.reset()
```

Out[11]:
```
(array([[[2, 5, 0],
         [2, 5, 0],
         [2, 5, 0],
         [2, 5, 0],
         [2, 5, 0],
         [2, 5, 0],
         [2, 5, 0]],

        [[2, 5, 0],
         [2, 5, 0],
         [2, 5, 0],
         [2, 5, 0],
         [2, 5, 0],
         [2, 5, 0],
         [2, 5, 0]],

        [[2, 5, 0],
         [2, 5, 0],
         [2, 5, 0],
         [2, 5, 0],
         [2, 5, 0],
         [2, 5, 0],
         [2, 5, 0]],

        [[2, 5, 0],
         [1, 0, 0],
         [1, 0, 0],
         [1, 0, 0],
         [1, 0, 0],
         [1, 0, 0],
         [1, 0, 0]],

        [[2, 5, 0],
         [1, 0, 0],
         [1, 0, 0],
         [1, 0, 0],
         [1, 0, 0],
         [1, 0, 0],
         [1, 0, 0]],

        [[2, 5, 0],
         [1, 0, 0],
         [1, 0, 0],
         [1, 0, 0],
```

```
          [1, 0, 0],
          [1, 0, 0],
          [1, 0, 0]],

         [[2, 5, 0],
          [1, 0, 0],
          [1, 0, 0],
          [1, 0, 0],
          [1, 0, 0],
          [1, 0, 0],
          [1, 0, 0]]], dtype=uint8),
 {})
```

This 'image' observation contains information about each tile around the agent. Each tile is encoded as a 3-dimensional tuple, (OBJECT_IDX, COLOR_IDX, STATE), where the OBJECT_TO_IDX and COLOR_TO_IDX mappings can be found in the 'minigrid/core/constants.py' file.

For the assignment, we will only use the information contained in the OBJECT_IDX and convert that information into a matrix using the following function:

In [12]:
```python
# Extract the object_idx information as a matrix
def extractObjectInformation(obs):
    (rows, cols, x) = obs.shape
    view = np.zeros((rows, cols))
    for r in range(rows):
        for c in range(cols):
            view[r,c] = obs[r,c,0]
    return view

# The following is a more efficient method of extracting the inform
def extractObjectInformation2(obs):
    (rows, cols, x) = obs.shape
    tmp = np.reshape(obs,[rows*cols*x,1], 'F')[0:rows*cols]
    return np.reshape(tmp, [rows,cols],'F')

obs, info = env.reset()
obs = extractObjectInformation2(obs)
print(obs)
```

```
[[2 2 2 2 2 2 2]
 [2 2 2 2 2 2 2]
 [2 2 2 2 2 2 2]
 [2 1 1 1 1 1 1]
 [2 1 1 1 1 1 1]
 [2 1 1 1 1 1 1]
 [2 1 1 1 1 1 1]]
```

The meaning of this 7×7 matrix in relation to the agent's (partial) observation is not so important for us. Essentially, the observation is encoded from the cell furthest from the direction the agent is looking on the left. Indexing is from leftmost row to rightmost row (row is defined as parallel to direction agent is looking). Within a row, the indexing is from furthers cell to

nearest cell. The agent will always be index [3][6]. The location right in front of the agent will always be at index [3][5].

2.4. Reward Signal

The minigrid environment has a sparse reward structure, as the agent only receives a non-zero reward when the goal is successfully achieved. The reward upon success is determined as follows:

$$r = 1 - 0.9 \times \frac{\text{number of steps}}{\text{maximum number of steps}}$$

The maximum number of steps here is 256 steps.

There are two instances when the episode is done:

- When the agent succesfully reached the goal within 256 steps, receiving a non-zero reward (done is True)
- When the agent has reached the maximum number of steps, receiving a zero reward (trunc is True).

We can check whether the agent has successfully reached the goal by verifying that the type of block right in front of the agent is of the type 'goal' (represented by 8). The minigrid environment has been written in such a way that an agent will successfully reach the goal if it moves forward when the block right in front of the agent is of the type 'goal'. This means that we need to check the current observation, and not the next observation, to determine if the agent has successfully reached the goal.

3. Basic training loop

The following gives an idea of a basic RL loop implemented in gymnasium using the gym.step() method. This program does not include any training or learning; this is for you to include by combining ε-greedy exploration with Q-learning.

In [ ]:
```
# Done and truncation flags for terminating task
done = False
trunc = False

# Create environment
env = gym.make('MiniGrid-Empty-8x8-v0')
env = ImgObsWrapper(env)

# Reset environment and get first observation
obs, info = env.reset()
```

```
# I don't want to see a while True loop!
while (done or trunc) == False:

    # Choose an action
    # Here we pick a random action
    a = np.random.randint(0, env.action_space.n-1)

    # Take action 'a', observe next state 'obs', receive reward 're
    obs, reward, done, trunc, info = env.step(a)

# Close environment
env.close()
```

4. Representation of the Q-table

For Q-learning, we need the value-function $Q(S, A)$ giving the value of each state-action pair (S, A). How do we store this table? One option is to use a Python dictionary, referencing the state S as a key to the dictionary, and to store a Numpy array of length 3 encoding the value of each of the 3 actions at that state/key.

This is a good representation, except that it has one problem: S is a 7×7 matrix representing the objects seen by the agent, and Python does not allow the use of a matrix as a key to a dictionary. As a result, we should convert that state (i.e., that matrix) into a valid key, either a numeral, a string or any combination thereof. For this, we can use a hash function, transforming each state S to a (hopefully) unique hash value, used as the key to store and retrieve the value-function.

For our purposes, many hash functions can be used. Python already has one that we can use by transforming the matrix observation into a string before hashing:

In [13]:
```
hash(str(obs))
```

Out[13]:
```
2662075026087883412
```

Alternatively, we can use one of the functions contained in the hashlib package:

In [14]:
```
import hashlib
```

In [15]:
```
def hashState(state):
    return hashlib.sha1(state.tobytes()).hexdigest()
```

In [16]:
```
hashState(obs)
```

Out[16]:
```
'94a787e79a7ac9179336a9ff525eaa024f66ea81'
```

In this case, the hash key is a string.

5. Loading and saving the Q-table

It is useful to be able to load and save the Q-table after training an agent, so the policy that is found can be used later. Assuming that we use a Python dictionary to store the value-function, the table can be saved and reloaded using Python's 'pickle' module, as illustrated below:

In [24]:
```
import pickle
from os.path import exists

# Declare value-function Q as Python dictionary
Q = {}

filename = 'qtable.pickle'

# Saving the value-function to file
with open(filename, 'wb') as handle:
    pickle.dump(Q, handle, protocol=pickle.HIGHEST_PROTOCOL)
    handle.close()

# Loading the value-function from file
if (exists(filename)):
    print('Loading existing Q values')
    # Load data (deserialize)
    with open(filename, 'rb') as handle:
        Q = pickle.load(handle)
        handle.close()
else:
    print('Filename %s does not exist, could not load data' % filen
```

```
Loading existing Q values
```