2 Coursework Part 2

Files

Take a look at the Python 3 files **random_environment.py**, **agent.py**, and **train_and_test.py**. The relevant parts of the code are well commented, so you may be able to understand these files by reading through them. Below is a short description of each file.

random_environment.py
This is similar to the **environment.py** file in Part 1 of the coursework. The difference now is that the environment it creates is more complex, and it creates a random environment every time the **Environment** class is instantiated.

agent.py
This is the only file which you should edit. There are currently five functions, which are used by **train_and_test.py** to allow the agent to interact with the environment. You should ensure that the names of these functions, and their argument lists, do not change. However, you are free to edit the code within each of these functions. And you are free to create new functions as you wish.

train_and_test.py
This is the script which will be used to train, and then test, your agent. It will create a new random environment, train your agent for 10 minutes in this environment, and then test your agent's greedy policy in this environment with an episode of 100 steps.

If you run **python3 train_and_test.py** from the command line, you should see the environment being displayed. The red circle is the agent's current state, and the blue circle is the goal. The light region is free space, which the agent can move through, and the dark region is an obstacle, which the agent cannot move through. If the agent tries to move into the obstacle, the agent will remain in its current state. The agent must navigate the "maze" and reach the goal as quickly as possible.

Implementation

Your task is to create your own deep DQN implementation in the file **agent.py**. Your implementation may be similar to the code you have written for Part 1 of the coursework, with some optimised parameters (e.g. episode length, mini-batch size, network architecture, reward function, learning rate, epsilon decay rate, etc.). But additionally, you may wish to implement any of the advanced methods introduced in Lecture 2 (e.g. Prioritised Experience Replay and Double Q Learning). Furthermore, you may wish to introduce some of your own ideas, although there is a limit on what is allowed (discussed below).

I suggest proceeding in the following order:

1. Take the code you have developed in Part 1, and rewrite it so that it fits into the structure of the new Agent class.

2. Study the effect of various parameters in your code, and find a set of parameters which work well for a range of different environments.
3. Introduce some of the Deep Q-learning extensions from Lecture 2, and see if they improve the agent's performance.
4. Introduce any other ideas you may have.

Remember that for this course, 50% of the grades are awarded for your courseworks. This is larger than most courses, and so you are expected to spend more time on coursework in this course, than in other typical courses. Part 2 of this coursework is advanced and is intended to challenge even the top students. If you are having difficulty, just focus on inserting in the basic DQN implementation from the tutorial, and you can still be awarded a decent grade.

Rules

Below are some rules which your solution must abide by:

- The magnitude of your action vector must not exceed 0.02. Otherwise, the agent will stay still. You can see this imposed in line 104 of **random_environment.py**.
- Your **agent.py** may import any Python 3 modules from the Python 3 standard library. However, it may only import the following additional modules: numpy and torch.
- Your implementation must be a genuine Deep Q-learning solution. You may not use tabular reinforcement learning. You may not create a hand-crafted controller (e.g. keep moving right, if the agent hits a wall, randomly move up or down until it can move right again, repeat ...). To check: your **Agent.get_greedy_action()** must return the action with the highest Q-value, based on the output of a Q-network. It cannot return an action based on a heuristic you have developed.
- You are not allowed any form of memory, other than by use of an experience replay buffer. For example, you cannot save a network if it is working well, and then load it later. You cannot create a variable which stores the highest reward so far. And you cannot create a variable which stores what the agent did in the previous timestep.
- Whilst you may use an experience replay buffer, this cannot be used for anything other than training the Q-network. For example, you cannot achieve any of the memory above by querying data in the replay buffer.
- You may not import anything from the **environment** module into the **agent** module. For example, you might be tempted to import the goal state, or the locations of the obstacles. This is not allowed; the agent must learn only from the **distance_to_goal** value which is returned from the environment. Importing the environment will not work anyway, since the **environment.py** file we test your code with is different to the one you have been provided.

- You may not copy and paste entire existing implementations you have found elsewhere. Any implementations must be your own, and we will be checking your code for plagiarism. Feel free to use other implementations for inspiration, but do not copy and paste.
- If you are not sure whether your idea is allowed, ask on Piazza!