MAS Project: Simulation of battle royal games with Python and implementing Q-Learning training algorithms

GUERIN Clément

guerin.clementpro@gmail.com,

5355484@studenti.unige.it

1: Introduction

This project has been made for my Multi Agents System class at the University of Genova. Its main goal is to implement a new scenario of agents fighting against each other. This work uses Python and the OpenAI gym library in order to simulate simple battle royal games (player moving and shooting at each other on a map, the winner is the last one standing).

This report will explain how the project works, how to use it, what it can do and what are its limitations.

The source code of this project is available on github:

https://github.com/Hvedrug/MAS_Project_Agents_fighting_on_a_map.git

2: Architecture and execution

2.1: Execution

In order to use this project you will need Python (version 3.7 at least), numpy and OpenAI gym. You can refer to OpenAI documentation[1] to install it.

Open a Terminal inside the project folder and execute the file masMain.py:

```
python masMain.py
```

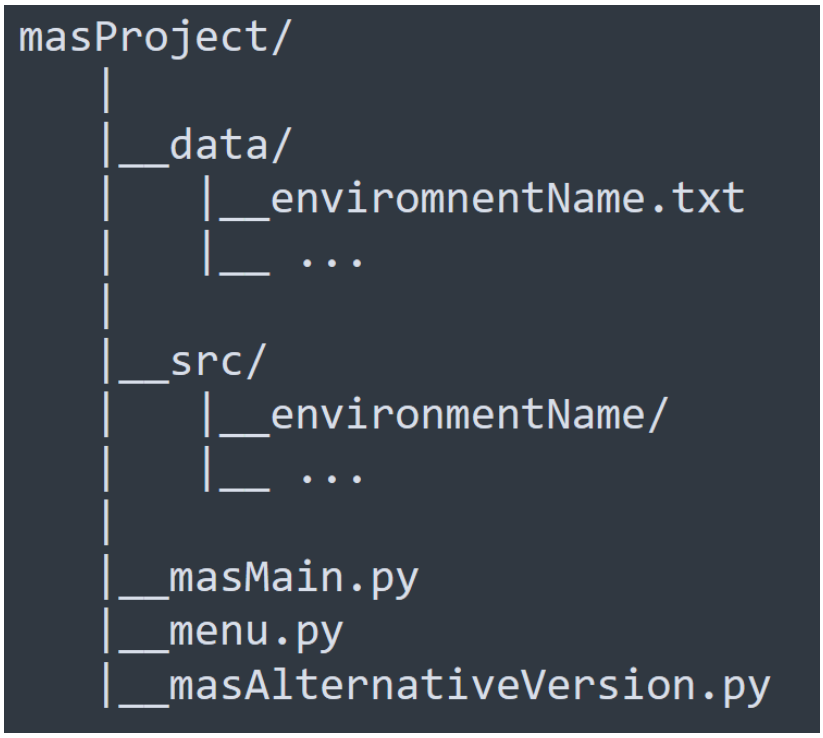
Some files are imported dynamically (depending on the environment you want to use, see under) and it is possible that your OS will not allows it. In that case you should use the file masAlternativeVersion.py:

```
python masAternativeVersion.py
```

With this file you need to manually select (by editing the file) the environment you want to use. To do so, from line 17, comment all “import” lines and uncomment the one corresponding to the chosen environment.

2.2: Project architecture

This project uses the following architecture:



- Data/ contains the file of the saved version of the agents of each environment. The user doesn't need to train the agents to use this project.
- Src/ contains the folders of each environment (detailed under), this allows the user to simply create a new working environment by duplicating an already existing one and modifying it.
- masMain.py is the only file that should be executed.
- menu.py contains the functions for the application menu.
- masAlternativeVersion.py should be executed only if masMain.py is not working (possible issues with dynamically importing the environments) or if the user wants to add a new environment.

2.3: Environment architecture

This is the architecture of an environment:

```
environmentName/ :
    |
    |__Launch.py :
    |    |__TrainingSaveAndTestOne()
    |    |__TestExistingAgent()
    |    |__TrainingAndTestOne()
    |
    |__importEnv.py :
    |__Environment.py :
    |    |__init() :
    |    |__encode() :
    |    |__decode() :
    |    |__step() :
    |    |__render() :
    |    |__reset() :
    |
    |__Policy.py :
    |    |__bestFirstPolicy() :
    |    |__randomPolicy() :
    |    |__randomMovingPolicy() :
    |    |__randomShootingPolicy() :
    |
    |__QLearning.py :
        |__test() :
        |__qLearningTraining_1agent() :
        |__getTrainedAgentFromFile()
        |__saveQTableToFile() :
```

- Launch.py is the file imported by the main function, it contains the functions to handle the main actions we want to do with our environment (a combination of training agents, saving results, testing agents).
- importEnv.py was used to import the other files, it should be merged with Launch.py, it is not present in the last environment.
- Environment.py implement a gym environment with its basic functions: init() and reset() to put the environment in an original state, encode() and decode() to obtain the name of the state and its parameters, step() to perform actions and render() to draw a representation of it.
- Policy.py contains the policies we want to use to choose an action to perform.
- QLearning.py contains the functions to train and test one or more agents. It also contains the functions to save in a text file an agent after its training and reuse it later.

3: How does it works

3.1: Agent training

To implement the Q-Learning algorithm I reused the code seen during the M.A.S Lab class and the slides of the lessons[3]. It uses a table (q_table) of size [number_states x number_actions] containing a value indicating if the action is good to perform in a chosen state. These values are computed during the training phase and then used during the test phase. During the test phase we always choose the action with the highest value in the table (argmax(q_table[current_state])).

We compute the value of each cell with:
q_table[state, action] = (1-alpha) * old_value + alpha * (reward + gamma * next_max)

alpha: learning rate (how much importance do we give to the future, between 0 and 1)
closer to one means we forget quicker what we have learned before.
gamma: discount factor (how important are the past or future rewards)
closer to zero means that we pay more attention to immediate rewards.
next_max: estimate optimal future reward.

3.2: The various agents types and actions

There are different types of agents:

- Dummy: it is not an agent, it doesn't do anything, it is used to train an agent to hit a target.
- Runner: often not a real agent, it is just moving randomly on the map.
- Killer: the agent we want to train, it can move and shoot, kill and getting killed.

The actions are divided in two categories:

- Moving (north, south, east, west)
- Fighting/shooting (shooting north, shooting south, shooting east, shooting west)

3.3: The various environment

To simplify the development process and the possible later improvement to this project, each Environment has its own version of each useful function. Some files and functions are similar and can create some unnecessary redundancy in the code but because we never import two different environments at the same time it will not be an issue at the execution. It also helps to keep these functions cleaner because we don't need to test what environment we are using them with.

These Environments have been created from the script of Taxi-v3 from the library gym[4]. Some similarities with the work of T. Dietterich[4] can be found in the structure of an environment.

We will now detail the various usable environments in the same order they have been made. These are only 2D environment, by default with a map of size 5x5 for the first ones.

- ***KillerVsDummy***: the map od the game is 5x5, there a mark on a cell, the killer need to shoot toward this cell. Each state represents the location of the agent and the dummy. Actions are movie or shooting in the four directions. The killer is trained by QLearning and will hit his target in 100% of the games.

- ***KillerVsRunner***: the map is still 5x5 and the dummy is now moved manually from outside of the environment. This is not a clean way of doing it, therefore the next Env. exists. The killer will hit his target 100% of the time.

- ***listKillerVsRunner***: This is the same environment as the previous one but it uses lists of actions, coordinates and reward that are processed all at the same time by the step() function. This is inspired by an answer from Jon Deaton on Stackoverflow[2]. The Runner uses only random moves and is not trained. The kill will hit his target 100% of the time.

- ***listKillerVsKiller***: This is still a 5x5 map environment like the previous one but this time with two killers trained at the same time. To do this I added a dimension to the QLearning table, instead of having q_table of dimension [num_states x num_actions] it is now [num_states x num_actions x num_agents] I could use the old q_table for both agents but there is a change that both agents end up doing symmetrical actions and end up stuck in loops. With this version it is also possible to get stuck in a loop (if best actions are shooting and missing for all agents). For this matter, if the list of actions is the same two times in a row, one of the agents move in a random direction.

- ***listThreeKillers***: This environment is the last 5x5 one, there is this time three killer agents on the map. It uses the same strategy as the previous one. It has been used to test if the system was easily scalable. It was useful to understand which part of the code takes the most time to execute (initiating the environment, training the agents, and saving the data). The save file was too big for github so this is the only environment coming without any trained agents.

- ***listWithOptions***: This is the version that ask the user the parameters of the environment at the beginning (size of the environment, and number of agents). I recommend augmenting gradually the size of the environment or you could quickly be in a situation where the number of states is exponential and therefor the memory needed became unreasonable.

- ***listWithOptionsOptimized***: This version tries to avoid memory overload by doing some optimization. This as a positive effect on the size of the saved data but it is not effective for the execution part. Bigger environments tend to require a important amount of space just to be initialized. (see warning under). This environment is able to simulate all the above (except the ones with dummy and runner)

Note that by training a new agent on an environment and saving your results you will lose the previously trained agent. That is why I recommend to only use already trained agents except for the last one, it is quicker and all of them are working. To be able to test the code, all agents can still be trained.
It would be interesting to implement a way to import an already trained agent and be able to keep training it and save the results. This has not been done because big environments tend to need a lot of time to be initialized at the beginning of the training.

WARNING: Please consider the fact that memory is limited on a computer and that using big numbers as parameters might result in memory overload. If you choose 10 rows, 10 cols and 5 agents with the simulation listWithOptions the RAM size required for training the agents all together is 80 Gb. 6x6 for 4 agents is more than 12 Gb and 5x5 with 4 agents is more than 2 Gb.

RAM size required > ((num_rows * num_cols) ** num_agents) * sizeof(Float)

Consider also that the execution time when the RAM is full augment exponentially even if the program is still functioning.

To be able to augment again the size of the simulation, some optimization has been tried. The first one is to train agents only 1vs1 and then for the test phase, split the state of the environment in state of an environment with the same size but with only two agents. Then get the best action against each other agent and choose the one with highest potential reward. This could lead to looping in the choice of the actions to perform. The second one is to not pass the big variable as arguments of functions but use global variables instead. And the third one is to reduce the size of some integers to 8 bit when more is not needed.

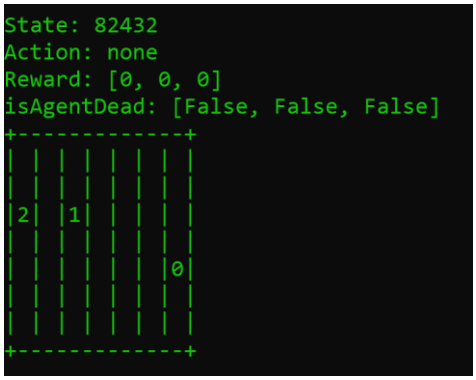
4: Rendering and testing

In this part we will see how the environment can be rendered and what result the user should expect to see on their screen. The all application run inside a Terminal and everything is displayed by it.
At the beginning you have a textual menu that ask you to answer several questions that later creates the parameters of the environment. This part can be skipped by the first question and the program will use the best Environment available and the best agents to run one test.

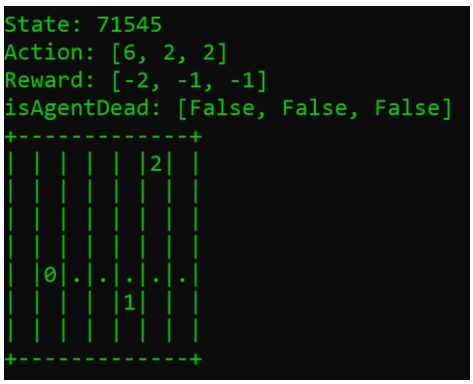
For each phase that can take a long time to execute, a counter will be displayed as an indicator of the progression. Up to 3 different counters can be triggered by the program depending on what the user wants to do.

example of counter

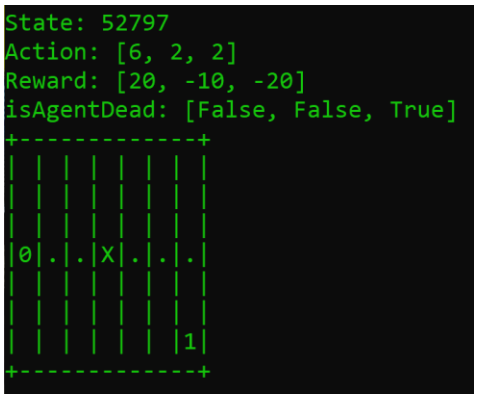
A test represents a game, the map is initialized, and the agents are placed on it:



When an agent shoots the user can see where the bullet is going, which actions have been performed, the rewards and which agents are dead.



If an agent is touched by a bullet, it is displayed as an X instead of its identifier. Two agents can touch each other in the same round and kill each other but we suppose that shooting is faster than moving so if an agent is shot at the beginning of the step it will not move. The dead agent will keep being displayed for the other rounds but some of the simulation ends after the first blood.



5: Limitation and possible improvements

We have already seen that memory can be an important bottleneck with the way the environments are implemented. Therefore, there is no 3D environment, it would have reduced a lot the maximum size of the environment and the number of agents that can be placed on it but the way it was supposed to work is the same as the one for 2D environments.

One solution to reduce memory usage would be to replace environment.P by a function. P contains the table of the rewards for each action. It has been implemented in Taxi-V3 and was not causing any issue because the map of the simulation was small.

Implementing a way to train an already trained agent and then specify which agent use which strategy could help showing what strategies are better. For example, we could compare an agent trained to kill the fastest possible but that would put his life in danger with an agent trained to escape first and then trained to kill a little. Maybe the second one would be more efficient.

6: Conclusion

This project gave me the opportunity to work with the gym library and to try implementing some simulation of game fighting. Compared to the initial goals of this project I did not got the time to implement 3D environment. Due to memory limitations the training part was to long to start and probably impossible to complete without rethinking the all project implementation. However, the 2D part is functional and results are usable. Agents can fight each other, and the games have an end. There is still some small rendering bugs and it would have been interesting to implement a way to choose which trained agent is on the map to compare different strategies.

References

[1] The OpenAI gym github repository: https://github.com/openai/gym

[2] Jon Deaton answer on how to make multi-agent games with gym: https://stackoverflow.com/questions/44369938/openai-gym-environment-for-multi-agent-games

[3] Angelo Ferrando slides about Reinforcement Learning: https://2021.aulaweb.unige.it/pluginfile.php/342264/mod_label/intro/RL.pdf

[4] Tom Dietterich work on Taxi-v3 environment for OpenAI gym: https://github.com/openai/gym/blob/master/gym/envs/toy_text/taxi.py