Reinforcement Learning Agents for a Battleship variant

TAU Group Motivational Test

Manh Hung Nguyen
CentraleSupélec
hung.nguyen@student-cs.fr

1 INTRODUCTION

Battleship is a strategy game for two players where one tries to destroy other player's fleet by guessing the location of the ships placed on the hidden map. In this project, we implemented reinforcement learning agents to play a variant of this game where the rules of the game are described as follows.

- Only one player (the RL agent) plays against a program generating the battle situation
- All ships take up only a single square
- Ships can touch each other and touch the board border eventually
- Ships are placed on a 10x10 game board
- Ships take several hits to sink (one take one hit, some take 2, some take 3)
- The agent wins when all ships are sunk. The goal is to sink all ships with as few hits as possible

0	0	0	0	0	0	0	0	0	**1**
0	0	0	**2**	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	**2**	0	0	0	**3**	0	0	0	0
0	0	0	0	0	**3**	0	0	0	0
0	0	0	0	0	0	0	0	**3**	0
0	0	0	0	0	0	0	**2**	0	0
0	0	**2**	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	**3**	0	0	0	0	**3**

Table 1: A 10x10 Battleship board with 10 ships and their health (24 hits in total to finish the game)

Table 1 shows an example of a 10x10 board with several ships placed. This example is used from now on in this report. Squares with "0" means there is no ship there, and other integer numbers represent the health of the ship placed at that square. The health value of a ship equals to the number of hit the agents have to make to destroy that ship.

In a normal Battleship game, players should be informed if they make a "hit", a "miss" or they just destroyed an opponent's ship so they can avoid the empty squares or the squares with sunken ships. However, to make the game harder, agents are allowed to shoot wherever it wants even the squares mentioned above. The agents are expected to learn how to avoid doing that by itself through the reward scheme to minimize the number of actions to finish the game.

TAU Group Motivational Test,
2020.

2 ENVIRONMENT

To be able to train a reinforcement learning agent, we first need to define the game environment including action space, state space and reward scheme for this Battleship variant. The game environment is implemented following the OpenAI Gym Environment interface.

2.1 Initial environment

The initial approach to model this game will be explained and discussed about why it failed to obtain an optimal result. The improved representation will be introduced in the next subsection.

In a Battleship game with a board size of n x n, a player can choose any square on the board to fire at any time step. Hence, actions can be represented by coordinates on the board:

$$\mathrm{A} = \{ (a_i, a_j) \mid 0 \leq a_i < n \text{ and } 0 \leq a_j < n \}$$

A state represents the position of the square that the agent has just fired into. Thus we have a similar representation for states:

$$\mathrm{S} = \{ (s_i, s_j) \mid 0 \leq s_i < n \text{ and } 0 \leq s_j < n \}$$

Each time the agent shoots at an empty square or a square where a ship has already sunk, it receive a reward of -1, otherwise the reward is 0.

An agent was trained by Q-Learning after 3000 episodes. Figure 1 shows how the trained agent plays in the last episode. It was able to detect exactly the locations of all 10 ships in the board. Then it tried to hit all of them 3 times which equals to the maximum health among the ships. The pattern of how the agent plays was repeated 3 times, which is illustrated in 3 dash line boxes in the figure 1. The number of actions to finish the game equals to the number of ships multiplied with the maximum health among those ships.

The agent was expected to stop hitting the same ship once the ship had been destroyed, however it failed to do that. The failure can be explained by the state representation that did not take into account the health status of a ship, which makes some of the states indistinguishable. The agent could not finish the game with an optimal number of actions.

2.2 Improved environment

It is clear that the agent trained in the previous environment did not have enough of information to learn when to stop hitting the same square. Therefore, the health status of a ship should be encoded in the state. The states are then represented as follows:

$$\mathrm{S} = \{ (s_i, s_j, s_h) \mid 0 \leq s_i < n, 0 \leq s_j < n \text{ and } 0 < s_h \leq h_{max} + 3 \}$$

with h_{max} is the maximum health among the ships. $h_{max} + 3$ represents other possible states apart from the health status.

The action representation and the reward scheme stays the same as in the previous environment. In this case, the size of the action space is n*n and the size of the state space is n*n*$(h_{max} + 3)$.

Figure 1: Episode: 3000, Cumulative Reward: -6, Number of Actions to finish: 30

```
Action: (3,5) Reward:  0
Action: (4,5) Reward:  0
Action: (0,9) Reward:  0
Action: (9,4) Reward:  0
Action: (6,7) Reward:  0
Action: (5,8) Reward:  0
Action: (3,1) Reward:  0
Action: (7,2) Reward:  0
Action: (1,3) Reward:  0
Action: (9,9) Reward:  0
Action: (3,5) Reward:  0
Action: (4,5) Reward:  0
Action: (0,9) Reward: -1
Action: (9,4) Reward:  0
Action: (6,7) Reward:  0
Action: (5,8) Reward:  0
Action: (3,1) Reward:  0
Action: (7,2) Reward:  0
Action: (1,3) Reward:  0
Action: (9,9) Reward:  0
Action: (3,5) Reward:  0
Action: (4,5) Reward:  0
Action: (0,9) Reward: -1
Action: (9,4) Reward:  0
Action: (6,7) Reward: -1
Action: (5,8) Reward:  0
Action: (3,1) Reward: -1
Action: (7,2) Reward: -1
Action: (1,3) Reward: -1
Action: (9,9) Reward:  0
```

3 AGENT TRAINING

3.1 Q-Learning vs SARSA

Q-Learning and SARSA were implemented to compare the results of off-policy learning and on policy learning, respectively. Both of the agents were running in 3000 episodes with the epsilon value decaying from 1.0 to 0. In this game variant, at any time step, the agent can choose any square in the board to shoot. That is why the future rewards should not have high weights compared to the immediate reward (the discount factor is set to 0.5)

Figure 2 and Figure 3 shows how many actions that the Q-Learning agent and the SARSA agent needed to finish a game during the 3000 episodes training. The red horizontal line represents the optimal number of actions to finish this game (24 hits which equals to the sum of all ships health). Both two agents were not stable at the beginning due to a high epsilon value. The SARSA agent was able to converge at episode 2000, and the Q-Learning agent needed nearly 400 episodes more to converge.

3.2 Deep Q-Learning

In this part, we want to take advantage of Neural Network to approximate the Q-value function. Two simple neural networks, each of them has 3 dense layers and LeakyReLU activations, were implemented to decouple the action selection from the target Q value generation. The two networks are periodically synchronized (every 200 actions) in terms of weights. They are trained on batches of samples obtained from the experience replay memory (buffer). The agent keeps pushing new experience to the memory while it is playing. The batch size is set to 256 to make the training more stable. Other hyper-parameters stay the same as for the Q-Learning agent and the SARSA agent.

Figure 2: Q-Learning Agent

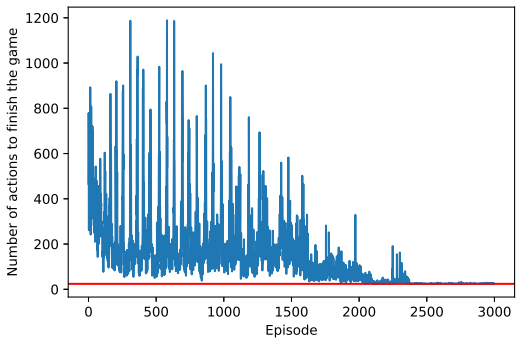


Figure 3: SARSA Agent

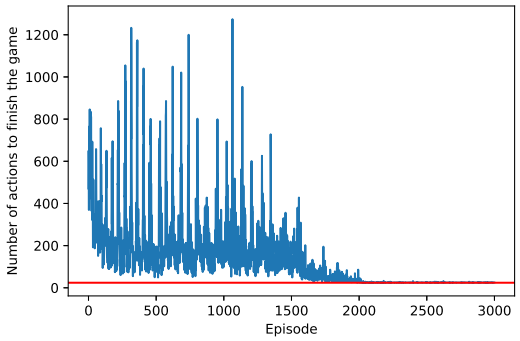
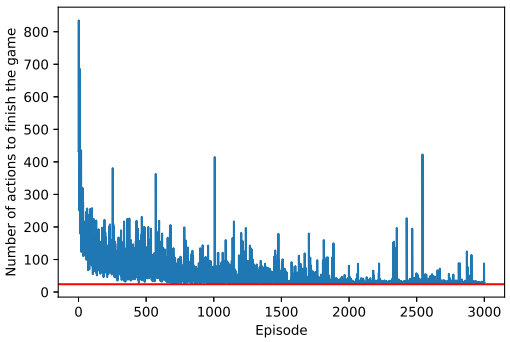


Figure 4 shows how many actions that the agent needed to finish a game during the 3000 episodes training. This agent seems more stable throughout 3000 episodes than the Q-Learning agent and the SARSA agent. However, it could not converge close to the optimal number of action (red horizontal line) during that amount of time.

Figure 4: Deep Q-Learning



4 CONCLUSION

After two weeks with this small project, I have learnt a complete approach to build a Reinforcement Learning Agent for a game:

- How to model a game by states, actions and reward scheme: The model should fully describe the game because it has a large influence on how the agents learn. The failure of my initial state representation and the improvements have been shown in Section 2.1 and Section 2.2, respectively.
- How to apply Q-Learning (off-policy) and SARSA (on-policy) in training agents and what is the difference between them.
- How to implement Deep Q-Learning with two neural networks to train agents. This took me a large amount of time to adjust the architecture and tune the hyper-parameters to get a decent results because it was very prone to diverging. Training the agent with DQN was much slower than the other two methods. I implemented two DQN agents, one with Tensorflow and the other with Pytorch. The one implemented with Pytorch was trained faster due to GPU advantages.

Complete source code of this project is available here. These are some useful resources that I used to get some ideas of how to create my environment for this game and train the agents: [5] [6] [1].

After spending some times playing with the agents, it seems to me that training the Q-Learning agent and the SARSA agent using the "lookup table" is memorizing the optimal moves for this specific game set up, which is not generalisable to other game set up (with different ships position and health). For me, using neural networks to approximate the Q function is more promising and it has more rooms for improving. Here are several techniques that could be applied to boost the DQN agent in this project.

- The state representation used in this project might not be a good one for training the DQN agent. Instead, we can feed the whole playing board into the network, which gives the agent a comprehensive view of the current state of the game. The network architecture and the hyper-parameters may need to be adjusted.
- Frame-skipping would be a promising technique that we can use to improve the exploration phase. It has been applied to train the agent to play games in [3]. The idea is the agent only selects actions every k time steps, its last action is repeated on skipped time steps. For this Battleship variant, if the agent hits the same squares several times, it can unlock new states faster because the states depend on the health status of the ship. We can implement this strategy for early episodes. This also reduces the computational cost for selecting new actions every time step.
- We can implement Double DQN [7] which is more robust and able to reduce overestimations compared to DQN. We may want to tell the agent where it should focus to learn by introducing Prioritized Experience Replay [4]. There are many other improvements for DQN have been proposed. Some of them have been combined together and obtained significant results [2]. These techniques would definitely boost the agents in this Battleship game.

REFERENCES

[1] Markel Sanz Ausin. 2020. Q-Learning with Neural Networks, Algorithm DQN. Medium
[2] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. 2017. Rainbow: Combining Improvements in Deep Reinforcement Learning. arXiv:cs.AI/1710.02298
[3] V. Mnih, K. Kavukcuoglu, D. Silver, Andrei A. Rusu, J. Veness, Marc G. Bellemare, A. Graves, Martin A. Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, S. Petersen, C. Beattie, A. Sadik, Ioannis Antonoglou, H. King, D. Kumaran, Daan Wierstra, S. Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nature* 518 (2015), 529–533.
[4] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2016. Prioritized Experience Replay. arXiv:cs.LG/1511.05952
[5] Michele Sebag and Herilalaina Rakotoarison. 2020. Reinforcement Learning labs. https://gitlab.inria.fr/hrakotoa/m2-rl-upsaclay
[6] Alessio Tamburro. 2020. An Artificial Intelligence Learns to Play Battleship. Medium
[7] Hado van Hasselt, Arthur Guez, and David Silver. 2015. Deep Reinforcement Learning with Double Q-learning. arXiv:cs.LG/1509.06461