CS9670 Reinforcement Learning
Project Report

Adversarial Battleship

Jialin Sun

251241006

Introduction

In this project report, a classic board game: Battleship is to be discussed. The goal of this project is to find a solution to the problem with deep reinforcement learning approach. The github page for this project: https://github.com/JialinSun1/Battleship_RL/

About the game

To give a basic idea about the game rule of Battleship:

The game starts with a 10x10 grid map of the ocean, where ships occupy various numbers of grids of the map. There are in total 5 ships, and their size is 1x5, 1x4, 1x3, 1x3 and 1x2 respectively. The setup of the ships could be random or deliberately decided by another player. We call this player a 'defender'. Figure 1 is an example of a ship setup.

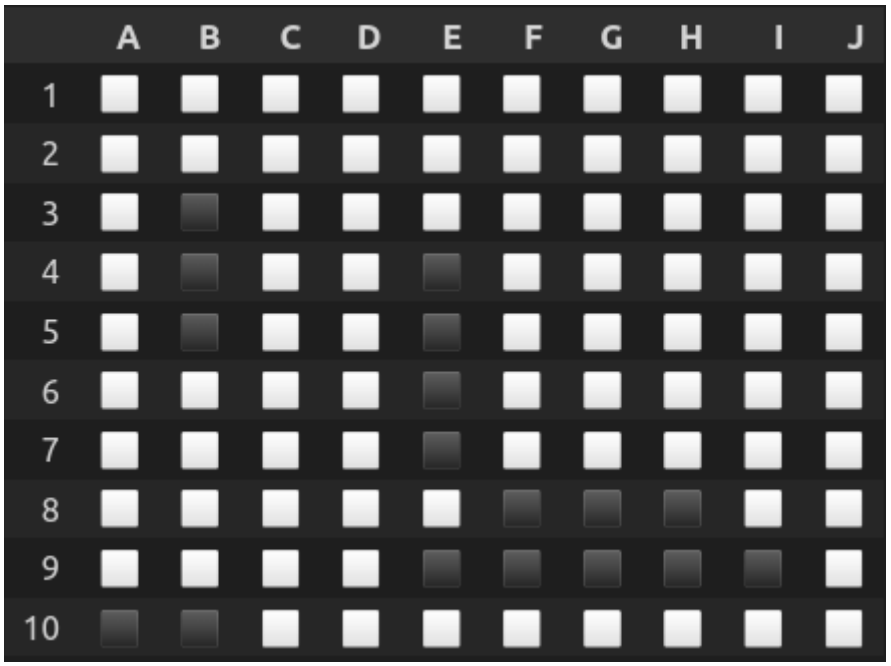

Figure 1

Standing on the opposite side of the game, another player takes a turn and bomb a grid. This player is called an 'attacker'. A bombed grid cannot be bombed again.

For the attacker side, the goal is to take as less turn as possible before all the ship grids are bombed.

For the defender side, the goal is to set up ships properly to delay the win turn of the attacker.

When all the ship grids are destroyed, the game ends, like Figure 2.

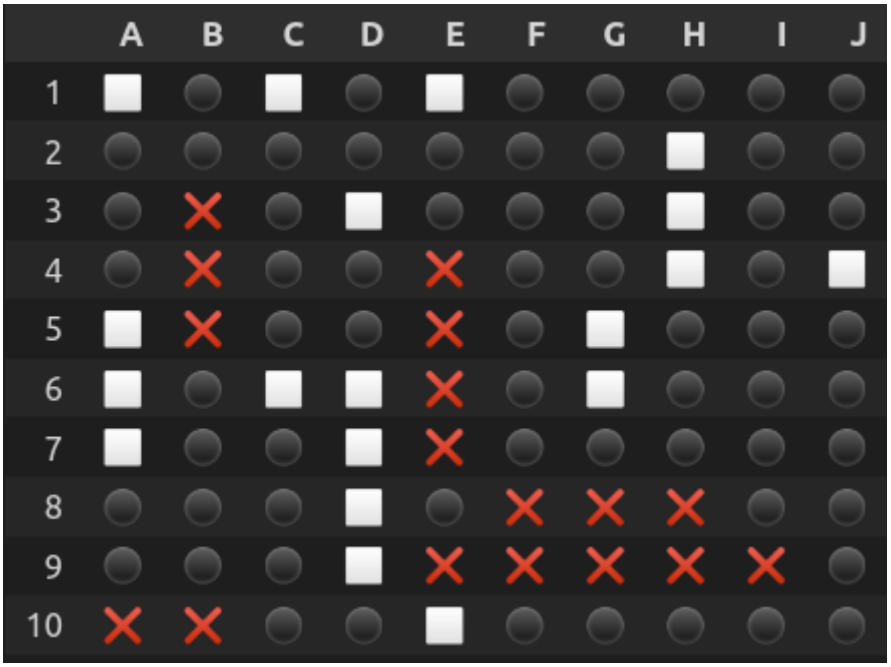

Figure 2. Game ends when all ships destroyed

There is a basic game environment based on OpenAI gym implemented by Thomashirtz. The link is provided here: https://github.com/thomashirtz/gym-battleship

Idea on reinforcement learning

To start solving the game, we firstly focus on the attacker side. Battleship is a highly stochastic game due to the randomness of the ship setup. Every gameplay situation might be different from the others. Each turn, any possible grid might be the bombing target. This makes the state space so big that it would not be handled by tabular method. For action space, it seems to be discrete and being limited in the 10x10 grid. However, according to the gamerule, a bombed grid cannot be bombed again, meaning that the action space shrinks as the game goes on. Previously selected actions are not able to be selected again, making the problem complicated. Function approximation method based on neural network is applied. Here I took both Deep-Q network and actor-critic model as comparison of the performance.(For the actor-critic code part, I mainly referred to assignment5 content) Reward setting is rather intuitive, as we could simply set missed bomb as negative, and hit bomb as positive. We will talk more about this in the later part.

After the attacker agent is tested, reinforcement learning is also deployed on the defender side. The objective of the defender agent is to give a strategy for deciding ship setup. Starting from the biggest ship, the defender takes an action of setting a ship (the coordinate of its stern or bow, and its orientation), and continues until all five ships are set properly in the map. The situation for state space and action space is as complicated as the attacker. The state is the current sea map about which grids are already occupied, and what ships have already set. Action space also varies due to the sea map since it is not allowed to put a ship on occupied grids. Once ships are all set, the attacker starts bombing. After the attacker is done, we record the number of turns taken before the attacker ends the game and take the number

as the reward for the defender. This reflects back to the defender as the reward of the ship setup. I used the Deep-Q network for the defender part.

With RL strategies for both sides, we could construct a generative adversarial algorithm (GAN) approach on this game and see how they perform.

Methodology

We start from the Attacker agent. To provide a clear idea about the state, action and reward, we define:

Each state consists of two 10x10 matrix $m1$ and $m2$ made up of 0 and 1s, representing the current situation of the map. m1 implies which grids are not bombed or missed. m2 implies which grids are not bombed or hit. For example, in the Figure 2 situation, m1 and m2 look like the following:

```
[0. 1. 0. 1. 0. 1. 1. 1. 1. 1.][0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[1. 1. 1. 1. 1. 1. 1. 0. 1. 1.][0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[1. 0. 1. 0. 1. 1. 1. 0. 1. 1.][0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
[1. 0. 1. 1. 0. 1. 1. 0. 1. 0.][0. 1. 0. 0. 1. 0. 0. 0. 0. 0.]
[0. 0. 1. 1. 0. 1. 0. 1. 1. 1.][0. 1. 0. 0. 1. 0. 0. 0. 0. 0.]
[0. 1. 0. 0. 0. 1. 0. 1. 1. 1.][0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
[0. 1. 1. 0. 0. 1. 1. 1. 1. 1.][0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
[1. 1. 1. 0. 1. 0. 0. 0. 1. 1.][0. 0. 0. 0. 0. 1. 1. 1. 0. 0.]
[1. 1. 1. 0. 0. 0. 0. 0. 0. 1.][0. 0. 0. 0. 1. 1. 1. 1. 1. 0.]
[0. 0. 1. 1. 0. 1. 1. 1. 1. 1.][1. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
```

Each action is a discrete integer $a \in [0, 100]$, each number representing the coordinate of a grid. However, once an integer has been chosen before, it is not to be chosen again.

For the reward, I defined that if the action results as a miss, a -10 penalty is applied. If the action hits a ship, a +5 reward is applied. If the agent consecutively hits a ship, the reward accumulates. For example, suppose previous action is a hit, providing a +5 reward. If the following action is also a hit, a reward is +5+5=10. If the third action also hits, a reward of +10+5=15 is given. This encourages the agent to positively chase the hit. Once there is a miss, the accumulation resets to 0.

The first problem comes to my mind is to find a way to solve the dynamic action space problem. The original environment allows the agent to take repeated actions. However, this makes the learning slow, taking more steps to complete a full episode. So I applied an action mask is before the policy takes an action, so that the previously chosen actions would not be chosen again in an episode. This makes the maximum step of an episode to be 100, because there are only 100 grids in the map. In this case, the agent bombs all the grids in the maps, and all the ships should eventually being detroyed.

As for the state, according to the game environment, two 10x10 matrices are returned. To handle this type of state representation, convolutional neural network are introduced to be able to receive 2 channels. However, with some more further experiments, setting the kernel

to be a greater size is not going to help the agent learn better, so I set the size to be 1, which makes it similar to a linear method.

Both Deep-Q network and actor-critic methods are tested. For the Deep-Q network, a neural network with 3 layers with reLU as activation function and Adam as backpropagation method are introduced for function approximation, using mean square loss function. The behavior policy I used is epsilon greedy with ε=0.2. It is applied with mini-batch feature with 100 capacity for each batch, and 1 million capacity of experience replay buffer. This helps the agent learn from the previous episodes.

The design of both actor and critic network for actor-acritic approach are similar to the Deep-Q network.The major difference is that the critic network returns a state value. For actor-critic, the batch size is per episode. However, experience replay is not being used, because the training of both actor and critic network requires the action log probability distribution, and this could be much more complicated compared to Deep-Q network which only needs to store the best action for each experience.

For the defender agent, to give a clear view on state, action and reward:

The state consists of a 10x10 matrix m1, and a 1x10 array. m1 represents the current ship setup on the map. If a grid is empty, a 0 is given; If a grid is occupied already, a 1 is given. The 1x10 array represents the next ship size it is going to be put onto the map. There is an alternative designing strategy that we could actually put this as an action part. However, the implementation of the game decides the structure of the state, so it will be in the state part.

The action consists of two integers a1 and a2. a1 stands for the grid of the stern or bow of a ship, ranging from 0 to 100. a2 stands for the orientation of the ship being either 0 or 1. 0 stands for the ship being horizontal, 1 meaning the ship being vertical. However, due to the masking issue, the problem would be far more complicated than it looks like. I will give further details in the later paragraph.

The reward depends on the performance of the attacker agent. The attacker takes more turns to solve the game, the defender receives a higher reward, and so on so forth. To give a definition of reward, r = t - 70, where t is the number of turns taken by the attacker before the game ends. For example, if the game ends at the 95th turn, the defender receives a reward of 25.

For the defender part, the action masking is a time-consuming job. According to the structure of an action, we need to decide coordinate and orientation. Before we choose to put a ship at a specific grid with a specific orientation, we must know if the ship is overlapping with any existing ship already in the map, or if the ship is going out of the border of the map. So for each grid, we need to know if the ship could be put in horizontally or vertically respectively. This means there will be two masks for each grid. Then we have an action mask of 2 10x10 matrices, or a 200 array, each representing a grid with an orientation. Every time before an action is decided, we calculate for each grid if it is available according to the state received from the previous action. To give an example, the following scenario is given.

Suppose we start a new game, we need to put the biggest ship with size 1x5 onto the map first. The 1s in the following two mask matrices m1 and m2 is not available, Where m1 represents all grids for vertical availability, m2 represents horizontal.

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.] [0. 0. 0. 0. 0. 0. 1. 1. 1. 1.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.] [0. 0. 0. 0. 0. 0. 1. 1. 1. 1.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.] [0. 0. 0. 0. 0. 0. 1. 1. 1. 1.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.] [0. 0. 0. 0. 0. 0. 1. 1. 1. 1.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.] [0. 0. 0. 0. 0. 0. 1. 1. 1. 1.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.] [0. 0. 0. 0. 0. 0. 1. 1. 1. 1.]
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.] [0. 0. 0. 0. 0. 0. 1. 1. 1. 1.]
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.] [0. 0. 0. 0. 0. 0. 1. 1. 1. 1.]
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.] [0. 0. 0. 0. 0. 0. 1. 1. 1. 1.]
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.] [0. 0. 0. 0. 0. 0. 1. 1. 1. 1.]
```

Assume that we put the ship at (1,7) with orientation vertical. For the next ship with size 1x4, the mask matrices m1 and m2 becomes:

```
[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.] [0. 0. 0. 0. 0. 0. 0. 1. 1. 1.]
[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.] [0. 0. 0. 0. 1. 1. 1. 1. 1. 1.]
[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.] [0. 0. 0. 0. 1. 1. 1. 1. 1. 1.]
[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.] [0. 0. 0. 0. 1. 1. 1. 1. 1. 1.]
[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.] [0. 0. 0. 0. 1. 1. 1. 1. 1. 1.]
[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.] [0. 0. 0. 0. 1. 1. 1. 1. 1. 1.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.] [0. 0. 0. 0. 0. 0. 0. 1. 1. 1.]
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.] [0. 0. 0. 0. 0. 0. 0. 1. 1. 1.]
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.] [0. 0. 0. 0. 0. 0. 0. 1. 1. 1.]
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.] [0. 0. 0. 0. 0. 0. 0. 1. 1. 1.]
```

Notice that in m2, the ship could not be put horizontally in grid (1,5), however , according to m1, it could be put in vertically. grid (0,7) does not work for neither, because vertical makes the ship overlap with the previously set ship, and horizontal makes the ship out of the border.

The reason of doing the masking is that after some random tests on this pre-implemented environment, I found that if a invalid setup is executed in the environment, the ship would simply disappear from the map. It is necessary to block invalid setups in case that the defender might take advantage through this trick which makes ships disappear to avoid ships being destroyed.

Defender agent is constructed by Deep-Q network with 2 layers of neural network with reLU and Adam is used. The neural network receives the state made up of a 10x10 matrix and a 10-length array. The replay size is 1 million, and batch size is 5, which is the number of steps taken pre episode (because there are 5 ships that need to be set in total).

Experiment result

As I pre-train the attacker agent before we go further to adversarial gameplay, I found that Deep-Q network starts to fit random ship setups after 2,000 episodes, resulting in an average 76 steps taken per gameplay. However, what surprises me is that the actor-critic performs poorly in the task. It does not fit into the problem, resulting in an average 95 steps taken per gameplay. From my perspective, this might be because this is a highly stochastic game, every

game might be completely different. With the help of experience replay buffer, Deep-Q network could sample a previous experience from the replay buffer and learn as a batch. Actor-critic lacks such an advantage, making it vulnerable to randomness and stochasticity due to the game mechanism. If there is more time, a method named Actor-critic with Experience replay (ACER) could be carried out, making it a fair game. Figure 3 is the performance comparison between DQN and actor-critic based on the reward and steps taken as more episodes are learnt.

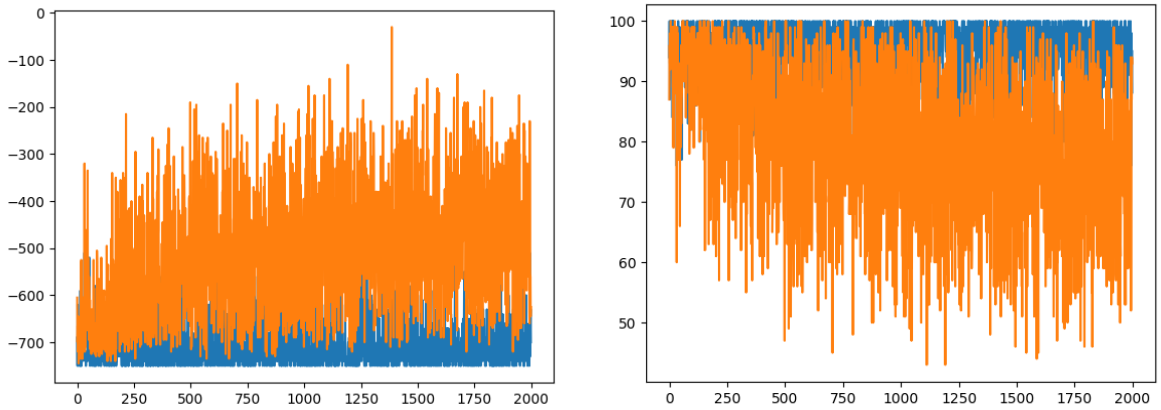

Figure 3. Blue: Actor-critic; Orange: Deep-Q network

As for the adversarial game experiment. I use a pre-trained DQN attacker agent and see how the adversary goes with the defender agent.
The first test is to see how the defender learns ship setup strategy against the attacker, while the attacker learns from the defender ‘s ship setup strategy and improve the bombing performance. Figure 4 represents the average step taken by the attacker, reflecting the adversary between the attacker and the defender. In the first 150 episodes, the attacker is performing better, and the defender is getting into struggle handling the attacks from the attacker. However, after 150 episodes, the defender is catching up, and getting a stable performance, controlling the step count at 90 to 95.

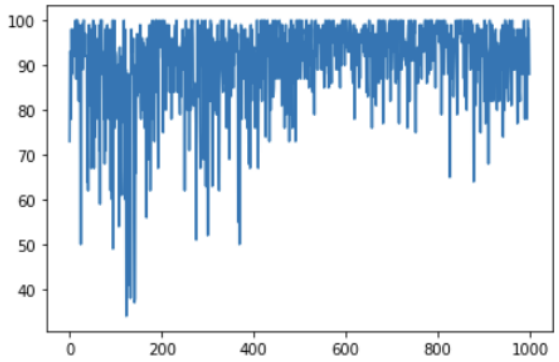

Figure 4

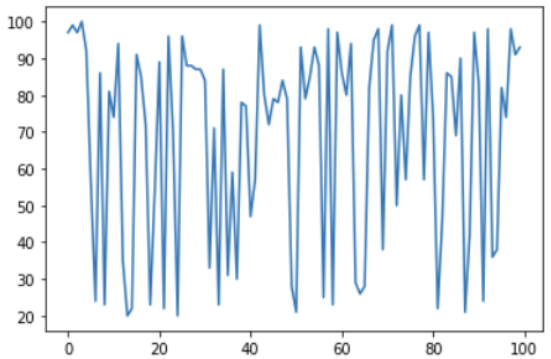

Figure 5

Then we save the trained network for both sides and run the next experiment, that is to let the attacker continue to learn, while the defender stops learning. As result in Figure 5, the attacker is easily getting a relatively low step count averaging at 68.6, meaning a better attacker performance.

Then we change the position, to let the defender learn, and prevent the attacker from learning. As plot shows in Figure 6, the average step count becomes 93.5, meaning that the ship setup strategy is working better against the attack.

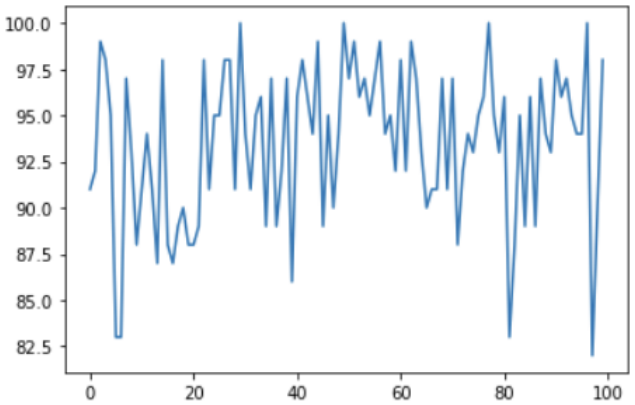

Figure 6

Conclusion

To sum up, in this project, various deep reinforcement learning strategies are being applied and implemented to solve the classic board game Battleship. For both attacker and defender, deep reinforcement learning-based methods are considered and designed. Not only to solve a game with a better insight on fine-tuning, but also I got a touch on generative adversarial algorithms, making agents fight against each other. By comparing the performance between DQN and Actor-critic, for a stochastic problem like Battleship, I found that a great number of experience replay is necessary and important. I also get hands-on experience in the deep learning area, knowing how to use tools like pytorch, which I will definitely benefit from in my future career.