Reinforcement Learning Project

Auguste Crabeil, Rémi Pommé, Alexandre ver Hulst

***Abstract*—Our aim was to fully solve the naval battle problem using the tools provided by reinforcement learning. This was achieved in several stages:**

- **Implementation of an environment adapted to handling the tools (in particular with the .step(), .reset() functions, etc.).**
- **Training an agent to optimally search for randomly placed boats**
- **Training an agent to optimally place boats in front of an agent applying a given strategy.**
- **Combine the two agents in the best possible way.**

To do this, we used the tools we had seen in class, and we mainly used Deep Q-Learning, combining it with our environment.

INTRODUCTION

Our project is to create an autonomous agent which will learn how to play the battleship game. In this project, we specifically focused on a model who was using Deep Q-Learning (DQL) with experience replay. DQL is a form of reinforcement learning that combines Q-learning with deep neural networks to approximate the Q-value function, which represents the expected future rewards of taking a particular action in a given state.

This aspect is particularly interesting and relevant to reinforcement learning because it addresses the challenge of making sequential decisions in a partially observable environment. Indeed, in the battleship game and our implementation, the agent only knows the size of the grid and the further information is acquired through experience and actions. This difficulty to access information about the opponent's board is one of the challenges of the game. Another challenge is the creation of an accurate and functional environment, as the values of the rewards are primordial.

Some work has been done on the battleship game, including the creation of the environment and the agents[3][4]. However, we found it very difficult and we wanted to simplify it and to adapt it to the classes we followed. Other papers exist about this game but they only use deterministic algorithms.

We decided to split the problem into several sub-problems, as it was easier to spot the accuracy of our play agents if we divided the initial project, as shown in [3]:

- Implement the environment, which will basically be a squared grid
- Create our agent that will search optimally the boats that we preliminary placed randomly. We then compare the results of our agent with deterministic agents that we created
- Implement a placing technique for our agents and make two agents confront to find the optimal placement of boats and search.

The task was globally succesfull, but is limited to certain grid sizes to limit the size of the state space. It is to be noted that the number of episodes for our first training agent to improve its performance is very large (50 000 episodes approximately) but was really efficient in the end. The large number of episodes needed to train the agent implies also a large period of time to train.
We then tried to optimize and enhance our model to incorporate more complex methods (such as a replay buffer, a target q-network, CNN for the model and hyperparameters optimization) in order to expand the number of ships and the size of the grid. Secondly, we implemented a simple adversarial method to find the optimal strategy for placing the boats. The agent can be train on these adversarial method to learn a new strategy. Finally, we try to pit two agents against each other to train them.

Links for the code :

V1 : Simple reward function and algorithms from the Labs with very few adaptation. This implementation give no result.
https://colab.research.google.com/drive/18aPqEvJ8x6tHZjmFV9fiCABhg

V2 : More complex reward function and simple DQN to learn from thousands of episode (very good results on a simple problem). We also have here all the part about adversarial agent.
$https://colab.research.google.com/drive/1O3rz4UEBZ8nkwWy$
$sharing$

V3 : Attempt to add complex methods but impossible to increase the complexity of the problem.
$https://colab.research.google.com/drive/1D1MFhYCWGekAQ$
$sharing$

I. Background

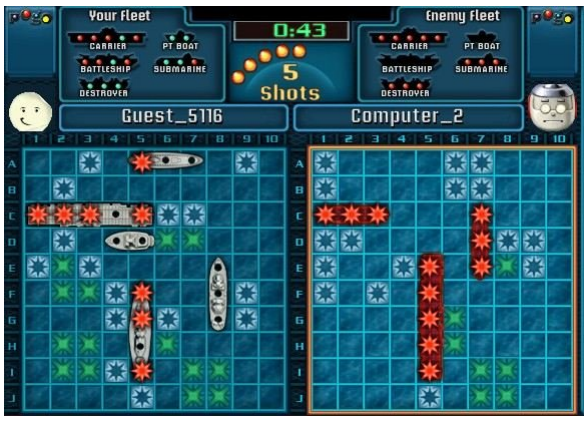

Fig. 1. Battleship game

Firstly, we will quickly redefine the rules and the objectives of the game. A battleship game is between two players, each one of them has a certain number of boats, each of a particular length (ex. 1 boat of length 2, 2 of length 3, ...). Here are the steps of a game:

- each player places each of the boats he has wherever he wants on the grid, as shows in Figure 1, by the grid on the left. It is important to note that each player does not know the location of the other's boats.
- Each player in turn indicates a square on the grid that they want to bomb to their opponent, who then tells them whether they have hit a boat, sunk a boat or shot into the void.
- The game ends when one player has destroyed all the other's boats.

The game involves elements of hidden information and sequential decision-making, making it an interesting domain for Reinforcement Learning research.

For the design of our environment, we decided to model the board as an $n \times n$ array (n being the length of the board). Each value of the array represents the state of the corresponding case on the board: 0 for a case that has not been touched, -1 for a "miss" case, case targeted by the player but where there was no boat and 1 for a "touched" boat by a move from the player.
The states of our environment (S) are all the possible arrays (and is thus of size $3^{n \times n}$). The different actions are all the cases that the players can hit, so there are $n \times n$ actions. An action is "legal" if it has not been played previously in the game. An "illegal" action is an action which has already been played in the game.
Finally, the rewards (r) are different for a new case missed, a new case touched and a case that was already targeted.

II. Methodology and approach

First, we had to build our environment modeling the battleship game. The argument taken by the environment is the size of our board. Then, we created different methods to place the ships, update their status (in order to determine if a ship is completely sunk) and to check if all the ships are sunk (and if the game is won). The most important method is the "step" method, which determines the reward and the new state after playing an action at a state s.
As we had to determine the value of the different rewards, our first idea was to give high rewards to the hit which touched the ships ($r = 1$), a very high reward for the hits which completely sunk the ships ($r = 5$), quite low rewards for the missed hits ($r = -0.1$) and very low rewards for the hits that had already been targeted previously in the game ($r = -5$). A similar approach has been developed in [5].

However, after our first evolved models, we saw that our environment was not adapted for deep learning, as all our models were not learning after being trained. We searched for documentation on this problem and we found that our reward values were not adapted. The advised values were : $r = 0$ for any legal action. If our model tries to play an illegal action, the reward of our next legal action is reduced by 1, for each illegal move made before the legal move. Finally, if our action hits a ship, our reward increases : $r = r + R \times 0.5^M$, where M is the number of missed after the preceding hit and R the number of hits so far for the given ship [3].

We then wanted to create our first models to play the game. As the battleship is composed of two distinct phases, we decided to not consider the phase where the players have to place their ships in the first time. In order to easily train our models, we adapted our environment to create each time random boards.

First, we implemented the random policy, where we chose a random action in all possible actions and we play it. This policy was made to check our environment.
Secondly, we created a deterministic policy called "search_and_hunt", which had two phases. In the first phase called "search", we search for ships using a random policy, but only on the legal actions. When an action hits a ship, the model turns into the "hunt" phase, where it hunts for the other parts of the ship around the first hit. This policy was implemented in order to test the efficiency of our agents.

We have therefore tried to implement the Deep Q-Learning algorithms studied in Lab06, starting with the linear neural network for a more simplistic initial approach. This model receives in argument the visible board of the game (i.e. the state s) and returns the Q-table of the game. For the training of our agent, we update our Q-table according to the formula seen in class[1] : $Q(S_t, a_t) += \alpha(R_t + \gamma \times max(Q(S_{t+1}, a)) - Q(S_t, a_t))$.
The results are very good within the framework of a 5x5 grid, but the results remain disappointing as soon as you

move away from the very defined framework of the 5x5 grid and a boat. This is why we decided to develop our model in several ways:

- A high α would have enabled our agent to adapt more quickly to opposing optimal strategies, but the risk remains instability. This is why we concluded that the value of $\alpha = 0.001$. We also tried to adapt the value to the size of the model. A problem with a larger grid / more boats required a smaller α to give the agent time to adapt.ù
- We have also adapted the value of γ to match the size of the problem. The larger the γ, the more the agent will encourage long-term strategies, which are more important if the problem is larger.
- For more complex problems, we have also encouraged exploration in the EpsilonGreedy method by decreasing the value of epsilon_decay so that there is time to explore more different strategies.
- We have also implemented the replay buffer to ensure the right exploration of all different situations.
- we have also tried to add a target q-network to make the model more stable.
- Obviously, we had to modify the neural network. We then thought of introducing a convolutional layer made up of multiple conv2Ds to take into account the two-dimensional aspect of our problem. We have seen the benefits of this operation in terms of the reliability of our results.

All these ideas have been tested to increase the difficulty of the problem as training progresses (these tests can be found in file V2).

Finally, our agent is very well trained on 5×5 grids, but it's still very complicated to have an optimal strategy on larger grids.

We then tried to think of making a complete agent who places boats.

The teacher recommended us to interest about self-play [2]. The main idea of self-play is that our agent needs to learn against an opponent which has the same level and experience at the battleship game than him. Indeed, if the opponent is too strong, our agent won't learn anything. On the other hand, if the opponent is too weak, our agent will overfit on strategies which can be irrelevant when playing against a stronger opponent. That is why a good solution is to put a copy of our actual agent as the opponent, and we update the copy as our agent acquire experience on the game. A good way to quantify the experience and skills acquired to the game is the "ELO" rating system. This is a score which begins at a specific score (usually 1200) and which increases as the player win matches and decreases with losses. This ELO score could permit us to quantify the difference between our actual agent and our copy to know when it is useful to change the copy (see end of V2 file).

The implementation of the "ELO" system would work like this :

- We would make a new model, then we would make a copy of this model
- we train the first model on the opponent's strategy, until we have reached an optimal strategy against the opponent (we wanted to model this using a special calculation system, but the idea is only to have a sufficiently large number of games won compared with those lost)
- As soon as our model wins sufficiently often, we create a copy of the trained model, and retrain the ancient model on the copy, and so on, ...

We started to use a deterministic method to determine the optimal placement strategy. Based on the trained model, we determined the average number of strokes the model takes to destroy it for each starting position of a boat. In this way, we were able to use a softmax method to obtain an agent that places the boats in a probabilistic manner, based on the mean number of hits that the pretrained model takes to destroy the boat.

In this way, we have the optimal placement that rivals any attack strategy. We then trained the model to counter this opponent. We can then create a new opponent and start again. Our last attempt (see end of file V2) was to implement a complete agent (which places and searches for boats) and not simply to separate the two problems. This raises complex reward issues because we have reward for finding boat and for wining the game.

III. Results and discussions

As we have many different results, we will present them in the order of the methodology. The implementation of our first environment and our first reward function (see V1 file) works very well. We then simply took the TD6 algorithms and immediately noticed that the problem was non-trivial. With this simple reward function and algorithms not adapted to the problem, we were not getting any results.

Our complex reward function, based on a better model (allowing us to run a large number of episodes), has enabled us to obtain our first results. To do this, we had to adapt the problem to a size of 5*5 and with a single boat. But after 50,000 episodes we obtained the optimal boat placement method for this problem.

First we saw a real improvement in the obtained reward through the training :

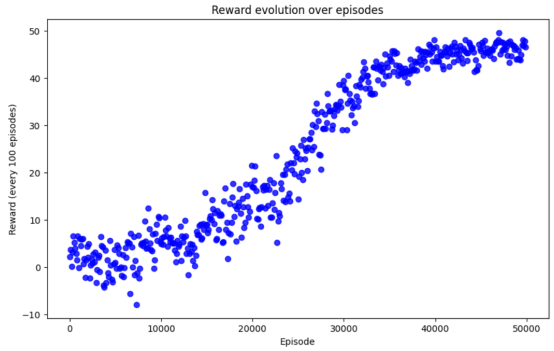

Fig. 2. Reward of our training agent over the episodes

This improvement shows that the training of our agent is useful and that it gains experience through the episodes. We also see that at the end of its training, the rewards seems to stabilize around 50 000.

Another way to check the efficiency of our model is to monitor the number of moves and to compare it to the search and hunt model.

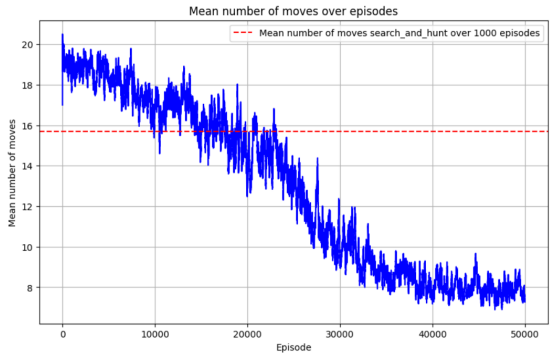

Fig. 3. Number of moves of our agent and the search and hunt method over the episodes

This graph shows that at the beginning, our agent had worst results than the search and hunt method. However, with training, our agent gets quickly better results (after 20 000 episodes approximately).

When we display a game we can see that the agent's choices seem optimal. This is why it is no longer able to learn.

Unfortunately, as soon as you increase the difficulty (even 6*6), you get no more results.

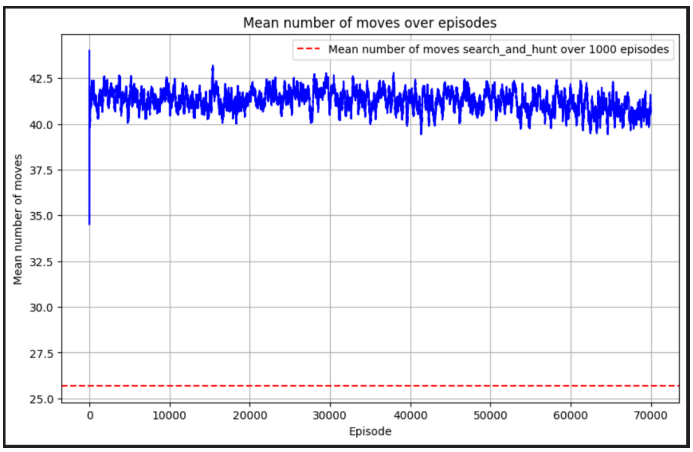

Fig. 4. Number of moves of our agent on a 7*7 grid with one ship and the search and hunt method over the episodes

As we can see from this graph, the model no longer learns anything.

A first simple idea to improve our performances is to complexify our model. As we said earlier in the report, we can complexify our Deep Learning model, for instance putting a CNN model instead of the Linear one. Indeed the game is a 2D problem, so we add 2D convolutional layers.

The results of these changes are clearly visible when learning the 5×5 grid. Learning is much faster than with the linear model.

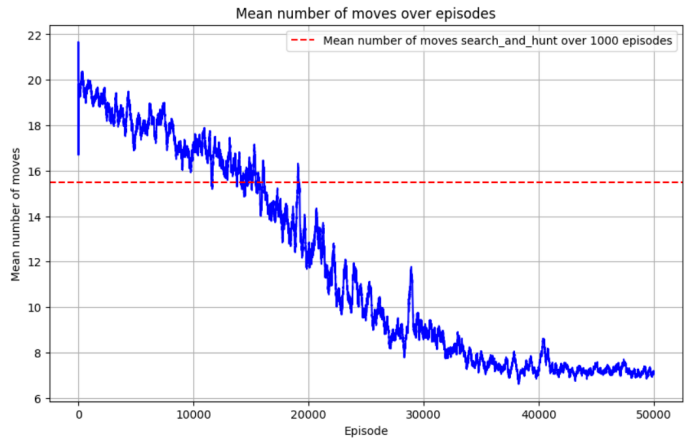

Fig. 5. Number of moves of our convolutional agent on a 5×5 grid with one ship and the search and hunt method over the episodes

We can even see some learning on a 6×6 grid. But these

training are really time-consuming.

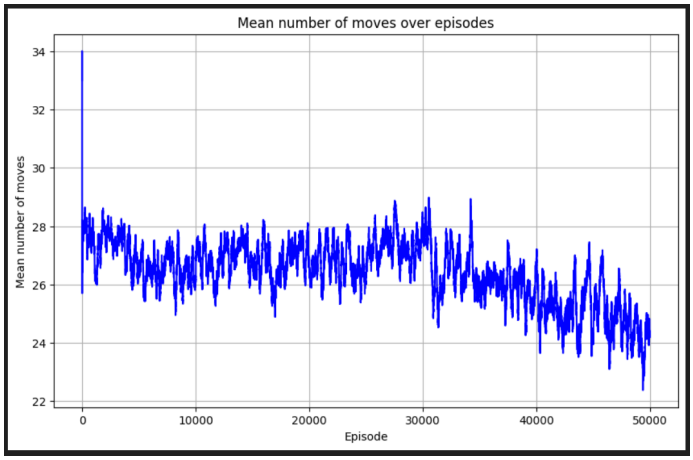

Fig. 6. Number of moves of our convolutional agent on a 6*6 grid with one ship and the search and hunt method over the episodes

As explained above, we tried many complex methods (Replay Buffer, Target Q-Network,...) but unfortunately none of these methods improved the learning of our agent on more complex games.

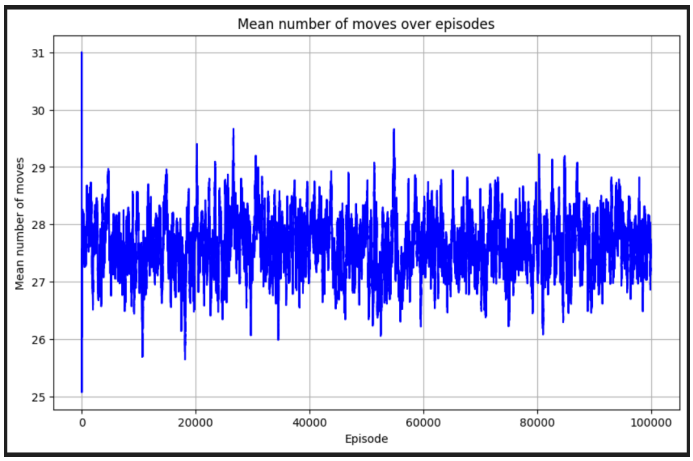

Fig. 7. Number of moves of our complexe methods agent on a 6*6 grid with one ship and the search and hunt method over the episodes

The training sessions are really long and we've tried a lot of different variations, trying to think about the different hyperparameters for each one. After many unsuccessful attempts we decided to move on to the self-play part.

Our first idea is to say that the opponent's placement will be to choose as a priority the shots for which the model is bad. We can then draw a list of possible placements weighted by the number of moves our agent makes from that placement.

We then have an opponent who greatly increases the number of shots our agent can take (12 instead of 8 on average).

But we can then re-train our agent against this opponent and we can see that very quickly our agent becomes extremely good against this opponent (see end of file V2).

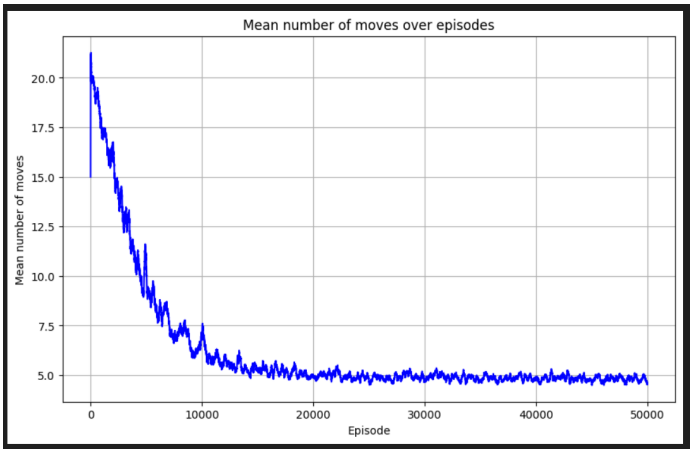

Fig. 8. Number of moves of our re-train agent against his opponent

We can then repeat this operation. So we've learnt how to create an opponent and train against him. We then tried to combine everything into one agent (see end of file V2). There are many problems here, because we have a gain linked to finding boats and a gain linked to winning games. Unfortunately, we've never managed to stabilise all this and we haven't found any paperwork linked to it. We think it is possible to link these two rewards so that the agent tries to find winning placements and at the same time adapts to the adversary.

Conclusion

Finally, this project enabled us to implement a complete environment and to understand the difficulties associated with this (such as choosing the reward function). We also understood that trying to simply apply the methods from the course would not work for a reinforcement learning problem. We are happy with the results we obtained. We have succeeded in creating an agent that learns! We would have liked to be able to complicate the problem and be able to play directly against our full agent but this just shows that we still have a lot to learn!

References

[1] Course6:ReinforcementLearningIII

[2] https://huggingface.co/learn/deep-rl-course/en/unit7/self-play

[3] https://towardsdatascience.com/an-artificial-intelligence-learns-to-play-battleship-

[4] https://is.muni.cz/th/oupp1/Reinforcement_Learning_for_the_Game_of_Battleship_Archive.pdf

[5] https://link.springer.com/chapter/10.1007/978-3-319-00542-3_20

[6] https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=8bf029478280cf4a53067b26dc8f15e9e28efb69