Deep Reinforcement Learning to Train an AI to Play Battleship Salvo

by. Amrit Ramesh

Introduction

I wanted to design an AI that would be able to play a version of Battleship called *Salvo.* In this version, the number of shots you have per turn is dependent on the number of ships you have. So in a game with 3 ships, you get 3 shots to hit your opponent every turn until your opponent sinks one of your ships after which you have 2 shots per turn. I initially wanted to incorporate all elements of the Salvo edition into the training of the AI, but after some thought I realized that I would need to have another player in order to make the number of shots mechanism work. More specifically, if both opponents have three ships I would have needed to simulate an opponent such that when the opponent sinks a ship the AI gets one less shot. Now I could have done this using a random shot creator, but I believed, that since a person's decision making in battleship is largely independent of the moves an opponent makes (you shoot at coordinates based on what you have done in the past not what your opponent has done) that simulating an opponent in order to incorporate the shot decrementation would add minimal value in training the behaviours and strategies of the AI. Therefore, I instead decided to play the Salvo variation where regardless of what moves the opponent makes the AI will always have a consistent number of shots >1 that is never decrementing. This also has the added side benefit of reducing the complexity of training the AI. The variation where at each turn the AI has n (>1) moves, already increases the complexity of the state actions space at an exponential rate (based on n). So if I were to include the mechanic where there would be a decrementing shot amount I would have the added complexity of needing to train the AI to consider that it may have fewer shots in the future (not to mention the added complexity of needing a random opponent to play in every game simulation). I thought this required significantly more work and computation than what it was worth in the value add it would create for the AI.

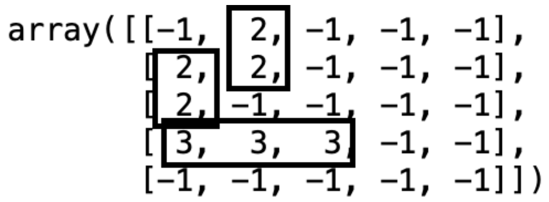
One thing to note with this variation is that the state and action space increases exponentially based on the number of shots we have per turn. More specifically, in regular battleship (where n = 1) an action space would simply be the number of coordinates on the board (each coordinate is a place we can possibly shoot at). But in the Salvo variation, where n > 1, the number of actions would be (Size of Board) choose (n). This is because for any given move we can choose two coordinates out of the number of coordinates on the board. So for a 5x5 board with 2 moves per turn, we'd have an action space of 300 actions, vs. the regular Battleship version where we'd have an action space of 25 actions. This quickly growing action space makes it almost impossible for me to generalize a policy using a Q table. Therefore I looked at two deep reinforcement learning alternatives, where I could leverage the architecture of a NN to store intricate information about a policy efficiently.

I initially looked at an action-value focused Deep Q Network, where the network would learn the most optimal Q-function to make consistent discrete actions given a state space. The problem with this is that Battleship is a game of incomplete knowledge (where we don't know the state of the entire board - just the places we shot at). This makes training a Q-function, that aims to find discrete actions with more complete information, harder. Therefore, I instead opted to use another popular Reinforcement Learning methodology, called Policy Gradient (implemented using a NN architecture) that aims to learn the policy directly by rewarding particular actions (iteratively and probabilistically) if they lead to good outcomes. This means the output of this methodology is a probability distribution across all the possible actions allowing me to be stochastic with my policy decision making, helping deal with the incomplete knowledge we have.

One article that dives deep into how to use a Policy Gradient NN to train a Battleship AI is "Deep Reinforcement Learning, Battleship" by Jonathan Landy. I leverage this code to do two main things: to build out the architecture of the NN using tensorflow, and to build out the mechanisms for training that NN including the calculating the rewards function, calculating the gradient of that rewards function, and using that output to backpropagate and optimize weights across the network. At the same time, Landy focuses on a variation of Battleship that made it impossible to simply appropriate his code and strategy. Specifically, he trained his AI to work on a 1-dimensional board with only one ship. Furthermore, he also trained his AI to play the normal variation of Battleship. Therefore, in order to be able to use his NN architecture and his mechanisms for training that network, I had to develop a series of functions that would be able to deal with a different state space (because we are training the AI on a 2-D board) and the different actions space (with the Salvo variation).

Methodology

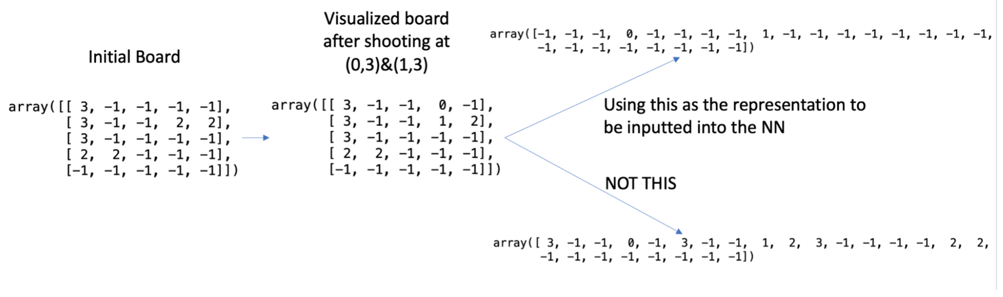
In order to make everything work dynamically, so we can test out different board sizes and number of moves/turn, I implemented a series of functions. First I created a function called “create_random_board” that would take in a board size n and a list of ship sizes and output a random nxn matrix with the ships represented by their size. Here is an example of an output with a board size of 5 and a ship list of [3,2,2]:



```
array([[-1,  2, -1, -1, -1],
       [ 2,  2, -1, -1, -1],
       [ 2, -1, -1, -1, -1],
       [ 3,  3,  3, -1, -1],
       [-1, -1, -1, -1, -1]])
```

As you can see the ships are represented by their ship sizes and all the empty spaces are represented by -1’s (this will be important when it comes to representing the input for the NN).

The next step of the process was to be able to play moves on the board. This required me to not only build out the mechanics of the game play given some input move but also have a consistent strategy of representing the resulting moves. I opted to use the same strategy as the one discussed by Landy in his article. More specifically, for any given shot at a coordinate, if that coordinate has a ship on it I will change that coordinate to a 1 (representing a hit shot). If that coordinate didn’t have a ship and instead had water I will change that coordinate to a 0 (represented a missed coordinate). One important thing to note is that when passing this board state into the NN all unknown spots will be treated as -1’s. So any spot that has not been shot at, whether it has a ship on it or not, will be treated as the same. This is because the AI should not know that there is a ship at a coordinate or not until it has shot at a given coordinate. So, the states going into the NN should not have information pertaining to where the ships are unless the coordinates that have been shot at have been labeled as hits. Here’s an example:


Initial Board
Visualized board after shooting at (0,3)&(1,3)
Using this as the representation to be inputted into the NN
NOT THIS

```
array([[ 3, -1, -1, -1, -1],
       [ 3, -1, -1,  2,  2],
       [ 3, -1, -1, -1, -1],
       [ 2,  2, -1, -1, -1],
       [-1, -1, -1, -1, -1]])
```

```
array([[ 3, -1, -1,  0, -1],
       [ 3, -1, -1,  1,  2],
       [ 3, -1, -1, -1, -1],
       [ 2,  2, -1, -1, -1],
       [-1, -1, -1, -1, -1]])
```

```
array([-1, -1, -1,  0, -1, -1, -1, -1,  1, -1, -1, -1, -1, -1, -1, -1, -1,
       -1, -1, -1, -1, -1, -1, -1, -1])
```

```
array([ 3, -1, -1,  0, -1,  3, -1, -1,  1,  2,  3, -1, -1, -1, -1,  2,  2,
       -1, -1, -1, -1, -1, -1, -1, -1])
```

The point of this is to be able to easily visualize the game board and hits/misses using the second still frame, while being able to accurately represent the different states of a specific coordinate (a hit, miss, or unknown) consistently for the NN.

With those two pieces I was then able to simulate a full game. What I did was first initialize the network with a given architecture (the same as what Landy uses) and some set of random weights. I also initialized logs to keep track of the states of the board, the NN version of those board states, the action taken to get to that state, and the number of hits for each action played within a game. Finally, I also randomly generated a board. In order to get the first action, that board was converted into its NN representation and passed through the network in order to output a softmax probability distribution across all possible actions. In order to account for coordinates that have already been shot at, I zeroed out the respective probability indices and then renormalize so everything adds up to 1 again (such that we can only shoot at coordinates we haven't shot at yet). Then based on that new distribution I take a stochastic approach to pick the action. In the case where we have multiple moves per turn that action will actually be an action set. So if we have a 5x5 board where we shoot twice in each turn we will have 5 choose 2 = 300 possible action pairs where each pair is a set of two coordinates we will shoot at. The NN in this case will have an output layer corresponding to each of the 300 possible action sets. After choosing an action or action set, we use the play game mechanic to shoot at the chosen coordinate(s) and append the resulting board, NN board representation, action, and # of hits to their respective logs. We can then repeat the process for the new game board. Without consideration to some additions in order to deal with special cases, where we have an uneven number of coordinates and an even number of actions within an action set (so there may be a point where there will be no possible action sets to deal with the last coordinate), this is how the game play simulation is set up. The idea is that after every game play I will use the actions taken and the hits resulting from those actions to calculate a reward and then backpropagate/optimize the weights. After many game play simulations and the resulting reoptimization we should have hopefully taught the AI the strategies needed to play on a random game board.

Therefore, after being able to play a full game the next step was to figure out the rewards function that would be used in order to update the weights in the NN. I ended up using the same one as the one used by Landy:

$$r(a; t_0) = \sum_{t \geq t_0} \left(h(t) - \overline{h(t)} \right) (0.5)^{t-t0}$$

First, what I do is take the history of hits for a specific game played. This means a game which took five moves to finish, where there is 1 ship of size 3 on the board, will have a hit log that looks something like this (1,0,0,0,2). The hits at each given action will be calculated every time the play move function is used in the game. Given the log I will then calculate the discounted weighted sum of a given hits index and all future hits. So if we hit a coordinate with a ship in the 1st action and then hit two coordinates with ships on the 5th action we'll have a weighted average reward value for the first action of 1*.5^0 + 0 + 0 + 0 + 2*.5^3 where the future rewards that took a lot of moves are discounted (to incentivize making hits earlier). Similarly, the weight average reward will be 0*.5^0 + 2*.5^1 for the 4th action and 2*.5^0 for the 5th action. By accounting for potential rewards for future actions for a given action taken in the rewards calculation I am incentivizing the AI to care about successive actions that will lead to or enable successive future hits and therefore are also incentivizing efficient board exploration(i.e. if we hit a ship, reward the AI more if it then looks in the vicinity for other ship coordinates rather than exploring somewhere else on the board). Furthermore, by discounting those future actions in the weighted sum calculation I am also making the current move the most important factor in determining the reward.

At the same time, in addition to just using the hits log as a way to get the weighted average, I also subtract the number of hits for a given action by the expected number of hits (h(t) hat) we would have had if we had random game play. This expected number of hits for random game play is calculated by dividing the total number of ship coordinates left by the total number coordinates left to be shot at. So using the example above the new weighted average for the 4th action would be (0-ev(h(t)))*.5^0 + (2-ev(h(t)))*.5^1 (where the expected hit value for a random player is represented by ev(h(t)). This addition allows me to only reward the AI if it is outperforming the expected value of its random counterpart.

The ultimate output of this system will be a list of weighted rewards for each given action (so in the example above it would be a list of 5) that will then be used, in conjunction with the state resulting from the action, to train the NN after the game is simulated.

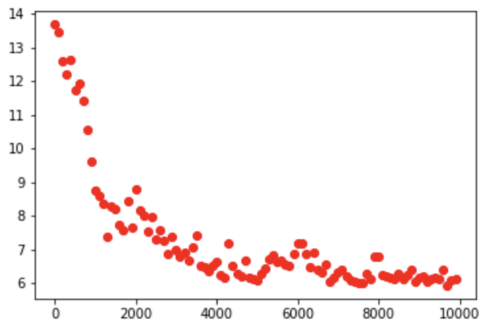
While this rewards function is meant to be used for the normal version of Battleship, its main ideas are transferable to the Salvo version. More specifically, if we have n shots per turn and manage to hit multiple ships with those n shots, we will simply be rewarding the AI more for that success by having that number of hits in the hit log (if we hit 3 ship coordinates within a turn append 3 to the hit log). This makes sure to incentivize the AI to hit as many ship coordinates within a turn as possible (aim for coordinates that are next to each other rather than far away from each other). That being said, all other ideas of this rewards function stay the same: I reward the AI more for getting successive hits, I ensure that future hits are not valued as much as current hits, and I reward the AI only if its doing better than random game play.

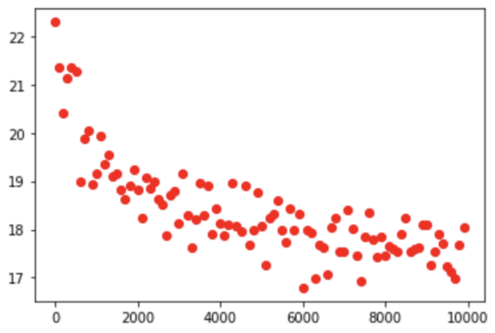
Experimental Results

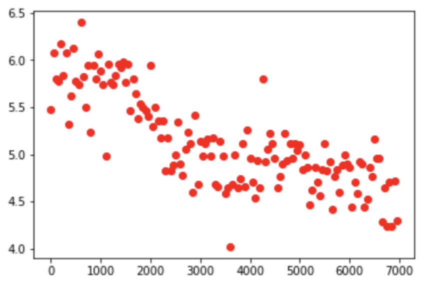
In order to understand the relative effectiveness of my AI, I found the average number of steps it took to solve a specific case of game board (based on the size of the board, the number/size of the ships, and the number of shots per turn) as compared to a random solver. The idea is that my AI should hypothetically be smarter at taking shots than a random AI and therefore should take fewer moves to sink all the ships. Furthermore, I also looked at several cases of board size, number/size of ships, and number of shots per turn to see if the AI was better at handling certain cases than others. Here is the list of cases I used:

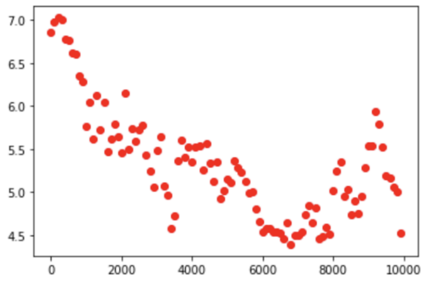
1. Board size 4; ship sizes 4; number of moves per turn 1
2. Board size 5; ship sizes 4,2; number of moves per turn 1
3. Board size 4; ship sizes 2; number of moves per turn 2
4. Board size 4; ship sizes 4; number of moves per turn 2
5. Board size 4; ship sizes 4,2; number of moves per turn 2

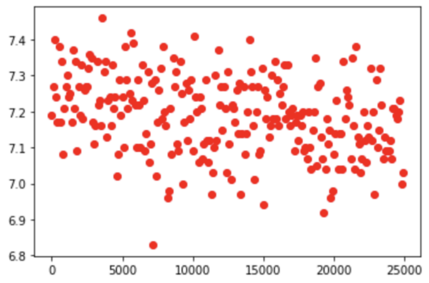
Ultimately, I decided to go with relatively small board sizes since larger board sizes had a significantly larger state and action spaces and it would take far too long to train the NN to an optimal state. I also decided to limit the number of moves per turn (for the Salvo edition) to 2 for that same reason (25 choose 3 is a lot more action sets than 25 choose 2). For each case, I looked at the average number of moves it took to solve the boards for each 100 iteration set across a 10000 iterations (games played) of training. After training I also looked at the average number of moves the AI took to solve 500 game boards of a specific case and compared it against a random player. Here are those results:


1st case: Exploring the effectiveness of our AI at a very simple base case where we are playing normal Battleship (1 shot/turn). Average # of moves for 500 games played: AI - 6.3, Random - 13.7


2nd case: Exploring the effectiveness of our AI at a more complex case where we are playing normal Battleship (1 shot/turn). Average # of moves for 500 games played: AI - 17.9, Random - 22.1


3rd case: Exploring the effectiveness of our AI at a simple base case where we are playing Salvo Battleship (2 shots/turn). Average # of moves for 500 games played: AI - 4.6, Random - 6.2


4th case: Checking the same conditions as last time but seeing if a larger ship affects performance. Average # of moves for 500 games played: AI - 4.9 , Random - 6.7


5th case: Exploring the effectiveness of our AI at a complex case where we are playing Salvo Battleship (2 shots/turn) with multiple ships on the board. Average # of moves for 500 games played: AI - 7.1 , Random - 7.4

Discussion

Ultimately, as you can see through the results, my AI is fairly good at dealing with cases where we are playing normal Battleship (1 shot per turn) regardless if the board set up is complex or not (decreased the number of moves by 54% and 19% respectively for cases 1 and 2). This is likely because of the relatively simple state and action spaces. It is also fairly good at dealing with cases where we are playing the Salvo version and we only have 1 ship on the board regardless of size (decreased the number of moves by 26% and 27% respectively for cases 3 and 4). This is likely because it has been trained to shoot both shots in the same area and then follow on in the same area if it experiences success. But the problem arises when we are playing the Salvo version on a board with multiple ships. Here the AI does marginally better than the random player, and the average number of moves curve shows that as well.

I believe there are a couple of possible reasons for that. Since we are shooting multiple shots per turn and simply yielding the number of hits from those shots in the hit log the AI has no understanding of which coordinate to follow onto if we only hit 1 of 2 shots. One way I could correct for that is by somehow emphasizing the first coordinate hit into the state space more. This may increase the size of state space, but it may end up being worth it if it solves that problem. Similarly, if the reason the AI doesn't do well with multiple ships (for the Salvo version) is because it doesn't know when it has sunk a ship and its time to move on to the next one, I can incorporate the ship sizes as well as whether either one has been sunk into the state space. In that light in order to reduce complexity I can also push the coordinates through a convolutional layer to further drive the AI to look within a specific area. Finally, I could be thinking about the rewards function wrong. Since the idea is that I want to incentivize shooting multiple shots around the same area, it may be worth rethinking the rewards function to be able to accommodate that strategy (the exact function that could do that is something I do not know).

Appendix

Instructions on how to use the notebook:

1. Import libraries
2. Run all functions from the top up to rewards_calculator
    a. If you want you can test out the create_random_board function or the play_move function based on the output of that board to see how it works
3. Change the board case that you want to train the AI on (list of ship sizes, the board size, and the number of shots/turn)
    a. WARNING: please limit the number of shots/turn to 2 - the action space becomes far too large otherwise
    b. WARNING: If you are doing 2 shots/turn please limit the board size to <5 -> at a board size of 5x5 it took around 30 mins to train the AI over 10000 iterations of the game
4. Initialize the NN architecture in the cell below
    a. WARNING: If you might get an error message: “module 'tensorflow' has no attribute 'placeholder'” if you are running this in Google Collab
        i. Try this website if you get that error: https://github.com/theislab/scgen/issues/14
5. Run everything until the cell where we are training the AI over 10000 games
    a. WARNING: If you get an error about the probs being NaN run the cell referenced in step 4 again – for some reason if you run it again it works
6. If you want to compare the effectiveness of the AI compared to a random player the next two cells get the average number of moves taken to finish a game over 500 games for both the trained AI and the random player
7. The rest of the cells are graphs of the progression of the average number of moves needed to finish a game across the 10000 games played
    a. WARNING: If you want to see that graph for your specific board case then please copy the code from any of the cells and paste + uncomment it in a new cell. If you run any of the existing cells you will erase the existing graph there corresponding to some board case I tested (board case specified in the comment at the top)

References

“Deep Reinforcement Learning, Battleship” by Jonathan Landy: https://www.efavdb.com/battleship

Understanding the differences of Deep Q Networks vs. Policy Gradients: https://medium.com/deep-math-machine-learning-ai/ch-13-deep-reinforcement-learning-deep-q-learning-and-policy-gradients-towards-agi-a2a0b611617e