# Battleship Buster Project Update

Jack Michaels
*Stanford University*
jackfm@stanford.edu

Yusuf Zahurullah
*Stanford University*
yusufz01@stanford.edu

Jack Xiao
*Stanford University*
jackxiao@stanford.edu

***Abstract*—This project aims to build a reliable Battleship player using deep function approximation. Classic Battleship was not considered due to its simplicity and a mutated version was used in its place. Function approximation was chosen because of the high dimensional state space explored since classic value iteration algorithms must iterate through the entire state space. To reconcile with the fact that we're dealing with two agents (our agent and an opponent agent), we assume the opponent agent pursues random policies.**

***Index Terms*—deep function approximation, bomb, line shot**

## I. Introduction

### A. Our Version of Battleship

The standard version of Battleship involves two players taking turns firing shots at the opposing player's 'fleet of ships'. The game begins with a pre-game phase where both players place a set 'fleet of ships' on an $n \times n$ grid, which represents water. Inspect Figure 1 for a reference of each ship used.

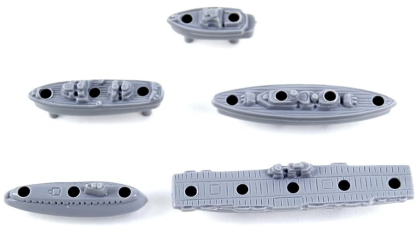

Fig. 1. Fleet of ships used in Battleship. There are always 5 ships of 4 sizes: (1, 2), 2 x (1, 3), (1, 4), (1, 5)

For this project, we will assume the fleet detailed in Figure 1 and will always assume a valid random placement of those ships on the $n \times n$ for both players.

Once ships have been placed, the game begins. Each player takes turns firing shots on the opposing player's grid, hoping to hit and sink all of the opponent's ships. This project assumes we always take the first shot. In traditional Battleship, the action of firing a shot is limited to only hitting a 1 by 1 grid element. If we assume our opponent places ships randomly, our agent can learn no better policy than random choice. Thus, we must alter classic Battleship to make the problem more interesting. Our altered version of Battleship will impact the actions that each agent can take, in other words the type of shot fired. In our version of Battleship, the standard 1 by 1 shot will remain, but the agent will also have the option to fire a bomb, which has a 3 by 3 impact, and a line shot, which has a 10 by 1 impact, in the case of a 10 by 10 grid. Each agent will only have 3 of those shot types available.

Finally, it is worth noting that it must be announced to the other player once a ship has been sunk.

### B. State Space

The state space must include the state of the opponent's board as we see it. Thus, for every grid element on the opponent's $n \times n$ board, we need to keep track of: whether we've explored it, if we shot and missed at the location, or if we shot and hit. Thus, we can store the opponent's board as an $n \times n$ matrix where each state is an integer $\in \{0, 1, 2\}$ where $0$ represents we haven't explored it, $1$ represents we attempted a shot but missed, and $2$ represents a hit.

To inform our algorithm about it's own current board, we can provide it with another entire $n \times n$ matrix representing its own board. While this may work, it is likely overkill. Instead, we can provide our algorithm with information on how many ship 'points' it has left (total number of available slots remaining on ships).

Finally, to deal with game mechanics, we can include integer information on how many shots are left of each type (bomb and line shots) and how many opponent's ships have been sunk. Thus, overall, we'll have a 4 dimensional vector representing each state where the first entry is an $n \times n$ matrix, the second entry is an integer, the third is a 2 dimensional vector, and the fourth is another integer.

### C. Rewards/Successors

Formulating an accurate reward function is vital to the performance of our algorithm since it must be able to determine which actions are 'good'. Theoretically, the best reward function may assess how close our shot is to a particular opponent's ship where 'close' is determined using a distance metric. Since our states don't include information on the opponent's ship placement (since that would be cheating), we can only go off of our view of the opponent's board. Our reward function is thus '+1' if we hit an opponent's ship and '-0.5' otherwise (since we're 'running out of time' to hit the opponent's ship when we miss). Note that for multi-shot capabilities, hitting an opponent twice would thus correspond to a reward of '+2'. This is rather naive, so if unproductive agents are produced we may consider extending to a heuristic

where the distance from previously 'hit' grid elements is added to the reward. This may involve extending the state space.

Related to rewards are how we determine successor states. Fundamentally this involves implementing the logic for our Battleship variant. There is one small yet ever important detail though; the opponent's action. Classic algorithms like minimax may assume an optimally performing opponent. However, due to the random nature of Battleship we can approximate a high performing opponent as a random opponent. Thus, successor states are determined by taking the current agent action we are considering and then assuming a random opponent action.

### *D. Deep Function Approximation*

Function approximation is well suited for the creation of an effective Battleship player because of the high dimensional state space explained above. During testing, it's entirely possible that we encounter a state not seen during training. Our predictor function needs to be able to accurately score all potential actions so that we can produce an effective policy in novel situations. To support this, we can use deep function approximation which transforms our predictor function into a deep network instead of a linear layer.

Predictor function updates are implemented using the Q-learning update rule described in Equation 2. Note that in this linear classifier example predictions are made using $w \cdot \phi(s,a) = Q_{\text{opt}}(s,a;w)$, thus $\nabla_w Q_{\text{opt}}(s,a;w) = \phi(s,a)$.

$$w \leftarrow w - \eta[Q_{\text{opt}}(s,a;w) - (r + \gamma V_{opt}(s'))]\nabla_w Q_{\text{opt}}(s,a;w) \tag{1}$$

$$w \leftarrow w - \eta[Q_{\text{opt}}(s,a;w) - (r + \gamma V_{opt}(s'))]\phi(s,a) \tag{2}$$

Modifying this for use in multi-layer contexts, we can use back propagation to generate weight updates for each layer and continually train. Computation requirements will increase with architectural complexity, though it is hypothesized that agent ability will increase correspondingly.

It is worth noting that the accuracy of our model is well determined by the objective function Equation 3. This objective function aims to make the value prediction for the current state equal to the reward plus the value prediction for the future state given the assumed action. In other words, our predictor needs to be able to predict current and future value precisely given the state.

$$(Q_{\text{opt}}(s,a;w) - (r + \gamma V_{opt}(s'))^2 \tag{3}$$

## II. Remaining Work

The majority of the remaining work involves implementing the Battleship variant itself and the deep function approximation algorithm. We have a working implementation of naive function approximation where future values are predicted with one linear layer. The majority of this code can be recycled and the predictor can be transformed into a deep network to better support accurate predictions. It is worth noting that the implementation of the Battleship variant need not be complex if the goal is to just create a smart Battleship player; the only requirement is that we develop a consistent successor function which takes in a given state, action pair and produces a valid next state.