AI Learning Agent for the Game of Battleship

Progress Report (CS 221 Fall 2016)

Jordan Ebel (jebel), Kai Wan (kaiw)

Task Definition

The game of Battleship is played on a square grid (traditionally 10-by-10). A number of ships, each 1 square wide and between 1 to 5 squares long, are placed on the grid either horizontally or vertically with no overlap. The ships are hidden from the player, who each turn makes a guess and selects a square to strike. The player is then informed whether the strike resulted in a hit or a miss. The game continues until a player has successfully hit all the squares occupied by ships (i.e. sunk all the ships).

In this project, we are implementing a Battleship-playing agent that will learn to become proficient at playing our modified version of Battleship. For the consistent evaluation and comparison of this agent with other approaches, we are making some specific modifications to the basic game rules as outlined below. In addition, we plan to introduce greater complexity to the game in subsequent phases of the project.

- The game is played by a single player (AI agent), who selects one square of the board to strike each turn.
- The game board will be of size m-by-m. For the initial phase, we are working with the traditional board size of 10-by-10. In subsequent phases, we plan to increase and decrease the board dimensions to evaluate the performance of game agents under those conditions.
- The number and lengths of the ships can customized from traditional arrangement (5 ships of lengths 5, 4, 3, 3, 2).
- The player (AI) has no prior knowledge of the number and sizes of the ships. The agent is required to make the optimal decision based solely on the layout of the game board.
- In subsequent phases of the project, we plan to extend our project to include a separate algorithm for placing ships. With this extension, we can support full two participant gameplay.

For the first phase of the project, we will be running our game agents on the “classic” game rules within the above definitions: On a 10-by-10 board, the player receives an unlimited number of torpedoes, and one ship of length 5, one ship of length 4, two ships of length 3, and one ship of length 2.

The main metric for game-playing proficiency will be the player’s hit-rate, i.e. the ratio of strikes taken that resulted in a hit. Equivalently, a player should attempt to sink all ships on the board in as few strikes as possible.

Infrastructure

We have formalized the game of Battleship into a state space model. This model consists of a start state, the set of legal actions that can be taken from the current state, a generation of the successor state from a given action, and an evaluation for whether the current state represents the game’s end.

A single *state* consists of a game board and game-related statistics (score, number of moves taken, ships remaining). On a game board of m x m squares, each square may be in one of the following statuses: Unexplored, hit, or missed. To represent the game board efficiently, the model only needs to track the board’s dimensions (e.g. 10-by-10), and maintain lists of squares that have been either hit or missed (with the remaining squares assumed to be unexplored).

Separately, the state maintains a list of ships and the squares that they occupy. This information can be represented by as a position, length and orientation (vertical or horizontal). This information is accessible only by the state object and not exposed to game-playing agent.

The initial state the game involves resetting all squares of the grid to unexplored status. The game randomly selects the positions and orientations of the ships, while ensuring ships lie within board boundaries and there are no overlaps (**Figure 2**).

An *action* is the selection of a square on the game board to strike, denoted by a pair of coordinates. At any given state, a legal action is a square that has not been previously struck, that is, an unexplored square not marked as either hit or missed. A successor state is generated from this action by comparing the action against the state's game board and ship locations. The statistics and the square selected by the action are both updated accordingly.

The game ends when all the ships of the current state has been sunk. Equivalently, this means all the squares occupied by ships, as maintained by the state, are included in the list of hit squares. To simplify this evaluation, we maintain a separate variable that tracks the number of ships (and their corresponding squares) remaining.

The gameplay itself is modeled as the interaction between a game controller module and game agents (**Figure 1**). The controller solicits actions from agents, generates the successor state and provides the successor state to the next agent. (At this first phase of the project, we plan to use only one game agent.) As part of the gameplay infrastructure (**Figure A1**), additional modules are implemented for displaying the game board, tracking game history, and to allow expansion of the project scope in subsequent phases.

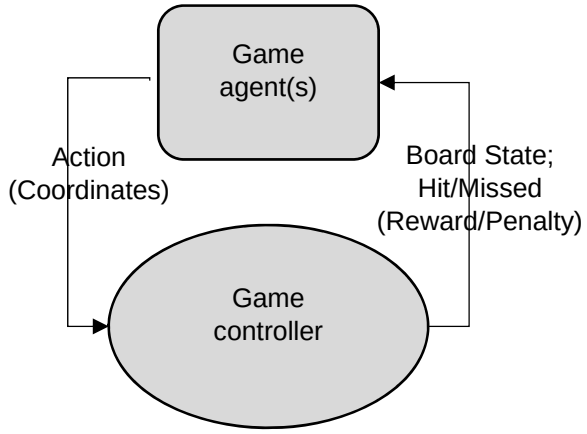

Figure 1: Gameplay modeled as the interaction between controller module

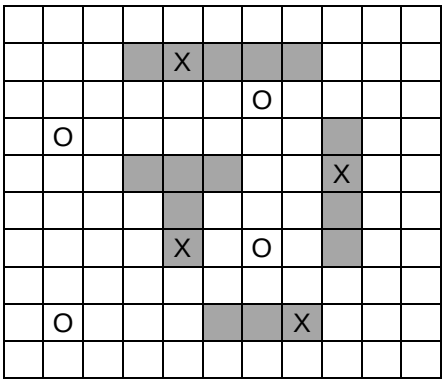

Figure 2: Example game board representing one possible state. Shaded squares contain parts of hidden ships. Squares with 'X' represents hits; those with 'O' represent misses

Approach

To implement an AI game agent for the game of Battleship, we are approaching the game fundamentally as an optimization problem. The task then becomes searching the state space to find the optimal policy for selecting actions, i.e. coordinates of the square to strike.

At a high level, our agent employs a reinforcement learning technique for Markov decision process. This approach reflects the nature of the game of Battleship, where the location of the ships are hidden from the player. With each action (square selected to strike), there is an unknown probability of transitioning to each possible successor state (either a miss or a hit), and the corresponding rewards are therefore unknown as well. But from studying the layout of the game board, it is possible for a learning agent to learn, over repeated iterations, the probability that a specific square contains a ship.

Specifically, our algorithm is based on the Q-learning algorithm to find the optimal policy. Each time we select a square on the game board to strike (action *a*), the algorithm performs value iteration to update the expected utility of selecting that square based on the result (V(s')). Through this process, we eventually converge to the true expected utility and obtain the action that yields the highest expected value at each state.

$$Q(s,a) = (1-\eta)Q(s,a) + \eta[r + \gamma V(s')]$$

For the step size η , we scale it based on the inverse of the square of the number of iterations to gradually reduce the step size as we converge on the optimal policy, that is:

```
def getStepSize(self):
        return 1.0 / math.sqrt(self.numIters)
```

To ensure that we are learning the optimal policy while the state space is properly explored, our agent uses the epsilon-greedy algorithm to produce the actual action. Currently, we assign epsilon to 0.1. Therefore we will generally choose the action based on policy, but for 10 percent of the time will take a random action that takes us into a new state not yet covered by the existing policy.

```
def getAction(self, state):
        self.numIters += 1
        if random.random() < self.epsilon:
            return random.choice(self.actions(state))
        else:
            return max((self.getQ(state, action), action) for action in
self.actions(state))[1]
```

A learning algorithm relies on the proper assignment of rewards to derive the expected utility of a policy. We use a simple scoring mechanism that is based on weighted portion of ships sunk discounted by the total number of moves executed:

$$\text{Score} = \left[\sum_{\text{ships}} \frac{\text{Squares hit}}{\text{Total squares occupied}} (\text{Ship's value})\right] - \text{NumberMovesTaken}$$

Each ship is assigned a value based on its size. A ship of length 5 is assigned a value of 100, a ship of length 4 has value 80, a ship of length 3 has value 60, and so forth. In essence, the algorithm can use this system to learn to sink as many ships in as few moves as possible.

A major challenge for our agent is exploring the large state-space, which increases exponentially with the game board size. Currently, our algorithm does not employ any pruning techniques to reduce the state space to be searched. Going forward, we expect to incorporate pruning to our algorithm to search the state space more quickly and efficiently.

Baselines and Oracles

We are using several non-learning approaches as baselines and oracles to compare with our learning-based AI agent:

- Baseline 1: A random algorithm can simply select squares at random to strike each turn. It is simple but not very interesting. We can use this to represent the absolute lower bound on effectiveness.
- Baseline 2: A hunt-and-target algorithm is a simple implementation that improves on the random algorithm. At first, it also selects squares at random to strike. But after a strike results in a hit, it will target the squares adjacent to the hit square during subsequent turns.

- Oracle: In general, a human can play the game of Battleship reasonably well. Most humans employ a strategy similar to the hunt-and-target approach but will subconsciously incorporate addition intuition to determine the likelihood that a square contains a ship. For example, an unexplored square that is mostly surrounded by missed strikes is not very likely to contain a ship, since a ship would likely have occupied several of the adjacent squares that have already been exposed. As such, a human would likely prioritize other squares as the target.

It is also useful to note that the theoretical upper bound to effectiveness would be 100% hit rate, i.e. all strikes result in a hit and the game is completed in the minimum possible number of turns.

Preliminary Experiment Results

Preliminary results for game agents playing Battleship on a 10 by 10 game board with classic rules is shown in **Table 1** below.

Table 1: Classic Battleship Preliminary Results

Agent	Average Num. Moves	Average Score
Random	95	244.59
Hunt and Target	64	275.65
Human	56	284.00
Q Learning	90	249.88

The Random agent requires the most moves to complete the game, and receives the lowest score. The Random agent's poor performance can be attributed to its complete lack of any game-specific knowledge. The Random agent will always pick a new board square to attack, even if the Random agent scored a hit on a previous attack.

The Hunt and Target agent performs much better than the Random agent, requiring about one third fewer moves to complete the game. The Hunt and Target agent also selects its targets at random, but the Hunt and Target agent includes the game-specific knowledge that ships are oriented in straight lines on the game board, and the object of the game is to sink the opponent's ships. Using this knowledge, the Hunt and Target agent will attack squares surrounding a square that was a hit previously. The Hunt and Target agent greedily attempts to sink ships it knows about, and otherwise randomly explores the game board.

A Human agent performs the best of all agents tested. The Human agent uses 12.5% fewer moves to complete the game than the Hunt and Target agent. The Human agent acts similarly to the Hunt and Target agent when attempting to sink a known ship. However, when the Human is searching for an unknown ship, the Human is capable of visually scanning unexplored game squares and selecting the minimum amount of shots required to explore regions of the game board. The Human knows the amount and length ships on the game board, so the Human can improve on randomly searching unexplored spaces.

The Q Learning agent performs in between the Random and Hunt and Target agents. The Q Learning agent is a preliminary implementation of the Q Learning algorithm with a simple identity feature extraction algorithm that memorizes states and actions. The feature extraction algorithm does not generalize to never-before-seen states and actions at all, and does not include any game-specific features to guide the Q Learning agent. However, the Q Learning agent is learning during each game, so in some games when a state has been seen before, the learning agent is capable of performance near the level of the Hunt and Target agent. In most games, the severely limited feature extraction of the Q Learning agent limits performance to near the Random agent.

Appendix

Figure A1: Overview of game infrastructure including game controller, agents, data structures and back-end modules for display and maintenance.