Mastering Halite with Reinforcement Learning

Jack Buttimer
Undergraduate
University of Pennsylvania

Luigi Mangione
Undergraduate
University of Pennsylvania

Saran Mumick
Undergraduate
University of Pennsylvania

Abstract

In this paper, we experiment with applying machine learning to create computerized bots for Halite III, a resource-management game. We focus on a single-player, single-ship game, and develop two classifiers: one using Support Vector Machine with Supervised Learning, and one using a Deep Neural Network with Reinforcement Learning. We evaluate the performance of each of these bots against two benchmark bots, one simple rule-based bot and one genetically-tuned rule-based bot. We find that the machine learning bots successfully learn gameplay, and performing at a level between the two benchmarks.

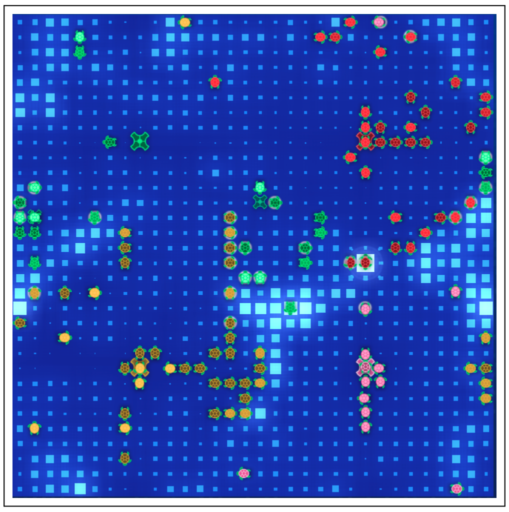

Figure 1: **Halite Game Board**

1 Introduction

Halite is an annual open source artificial intelligence challenge, created by Two Sigma. This years competition, Halite III, is a resource management game in which players must program a bot that builds and commands ships to explore the game map and collect halite, an energy resource that can be found scattered across the map. Ships use halite as an energy resource, and the player with the most halite at the end of the game is the winner.

1.1 Game Rules

Each player begins a game of Halite III with a shipyard, where ships spawn and halite is stored. Players spend halite to build a ship, move a ship, and to convert a ship to a dropoff. Players can interact by seeking inspiring competition, or by colliding ships to send both to the bottom of the sea.

Players each start the game with 5,000 stored halite, a shipyard, and knowledge of the game map. Each game is played in groups of two or four on a two dimensional map with a unique symmetric pattern of halite.

Ships can make one action per turn: they can move one unit in any cardinal direction, collect halite from the sea in their current position, or convert into dropoffs. When a ship is over a friendly shipyard or dropoff, it automatically deposits its halite cargo increasing the player's collected halite.

Each turn, the game engine sends the players the positions of all ships and dropoffs, and an updated game map. Players have up to two seconds to issue their commands for the turn. The game engine will parse and execute the commands, calculating each players resulting halite score and resolving all movement. The game continues for 400 to 500 turns, depending on the game map size. The winner of the game is the player who has the most collected halite at the end.

An example of a particular instance of a game state for Halite is shown in Figure 1. For more information about the rules and recordings of games of Halite III, visit the website, halite.io.

1.2 Importance of this topic

Since the mid-twentieth century, we have been programming machines to play games. From checkers and tic-tac-toe in the 1950s, to the more complicated games of chess and Go that computers focus on today, machines have not been spared. Computers playing games has always been fascinating to people. When IBM's Deep Blue, a chess playing computer, defeated the then chess world champion, Garry Kasparov, in 1996 for the first time, the news garnered global attention. Since then though, more and more progress has been made. Nowadays, the chess application on your phone can compete with wold champions.

This rapid growth in how machines learn to play games was in no small part due to the discovery and use of new methods and algorithms along the way. One such algorithm is called minimax which works to minimize the maximum potential chances of loss. This algorithm is used in chess engines and can be applied to general decision making when there exists an element of uncertainty. The development of algorithms like this one have enhanced our ability to solve problems (Ontanon, 2012). This is why learning games is important.

2 Problem and Approach

2.1 Problem Statement

Our objective is to create a machine learning bot for Halite III that optimizes single-player resource gathering on a set map size. The decisions this bot will make per turn of the game will be with regards to ship navigation. Afterward tackling this problem, we can see how the bot generalizes in a multiplayer setting or on differently sized maps; however, this is not the focus of this project. Evenso, it is worth noting that the approach we take can potentially be generalized the the multiplayer setting by simply having multi single-player classifiers, one for each player.

Our simplification of the problem was achieved by creating a new engine locally and setting up an environment such that only one player can play at a time all the while maintaining a 32x32 map size.

Another simplification made was that the decision of when to create a ship will be left as a rule-based decision. In order to correctly make this decision a lot of data would have to be generated since for the majority of moves in a game, the player doesn't have enough . Also, this problem would have to be learned separately from the problem of ship navigation meaning that another classifier would have to be used. The use of another classifier would cause confounding in the results making it hard to draw conclusions. For this reason, the decision was made to make ship generation rule-based.

The final simplification made was that training and learning will be done with a single ship. This will make training easier because if there are multiple ships, then the environment from turn to turn would be non-deterministic. This would mean that we cannot model our ship as a complete agent, the ”random” aspect of the map state would need to be simulated, and, in general, would require a very different and more complex model than was used.

2.2 SVM Bot

The first machine learning approach to solving this problem that we considered was a supervised approach by creating data from replays. On the website for Halite, there exits a repository of all of the replays of each game that each player has played. Thus, the data we would use to train our bot came from 50 of the top ranked player's games.

The idea here is that the behavior of the ships of the best player in a competitive game will generalize to maximizing resource collection in the single player setting. This is admittedly a bit of a jump though since because we are trying to learn one problem from a different, more complex, one. As a result, we decided just to take data from the first 150 moves of each of the 50 games, as this would be the period in the game where the least amount of interactions with other players would occur. This gives in total 750 moves to learn from. However, for each move, data from multiple ships is generated.

For each ship on each move, data is parsed. Information regarding the amount of halite the ship is carrying, the location of the shipyard, and the location and amount of nearby available halite is encoded. This represents our feature space. The label is the navigational decision that the ”good” player's bot made. Training becomes the simple process of showing the model this labeled data.

The model we used is an SVM with an RBF kernel weighted classifier. This was chosen because it is a well-understood kernel widely used in many practical settings and ensures data is linearly separable. It is available in many standard libraries.

One potential problem of this supervised method is that it only takes data from one player. As a result, it performs perfectly when it performs exactly like that player, which means that it will never surpass the performance of the bot it is learning from. For this reason, it may be helpful to train on multiple different boys; however, this runs into another problem: if those bots have different strategies then there is no guarantee that the SVM bot we train will learn something substantive.

Another potential problem with this supervised approach that may limit the SVM bot's its ability to continuously improve is that the SVM bot makes moves that mimic the data it trained on without taking into account the strategy behind those moves.

2.3 Deep-Q Learning Bot

A second goal involves attempting to build a more complex, robust algorithm to improve upon the performance of the bot. To do this, we consider training a neural network that takes into consideration the current game state and is able to make a prediction of which move to take.

While this is a reasonable model to consider, one key challenge that arises is how to determine whether a particular move on a particular game state is considered a "good move", since the only metric we have to evaluate performance is simply to see how well the bot did only after the game has ended. Therefore, a natural approach to this issue involves using reinforcement learning. At a high level, we want to make a move that gives the highest chance of optimizing the expected reward in the long-term. To train, we retroactively give each move in a game a reward based on the final outcome of the game.

The data we will use for this approach from which our features will be drawn will be taken from games the bot itself plays. The features will be layers in a neural network, each layer, of which there will be three, will be a 33x33 grid centered around the ship. The first layer will contain all of the available halite in each position around the ship. The second layer will contain information about nearby ships. It will be encoded by placing in the grid the amount of halite each ship is holding in the position on the grid of the ship. In this way, all positions without ships will be encoded as zero, and the central position of the graph, the position of the ship that we train, will always contain the amount of halite that ship is currently holding. Note that due to simplifications made, this central position will be the only potentially non-zero position in this layer. The final layer, another grid, will contain a single one where the shipyard is located and zeros elsewhere. This ensures that the ship will have knowledge about where to go in order to deposit its collected halite.

To be more specific, we consider Deep-Q learning as a model to train our classifier. Q-learning is a specific implementation of reinforcement learning, and is useful to use in our specific model because it uses a discrete action space and a continuous state space. While in theory our state space is discrete, there are such a large number of possible states that considering the space to be discrete would be impractical for a learning algorithm to train on. Deep-Q learning has been experimented in similar applications to ours and proven successful, including applying it to the Atari Games, which are computerized game bots that face similar issues as ours, including a high-dimensional visual input (Mnih, 2013).

To create our Reinforcement Learning bot, we utilized the DQN baseline developed by OpenAI. Given the process through which halite games are instantiated by the halite engine, simultaneously playing games and applying Q-learning requires significant modification of the game engine. Thus, we opted to build a simulation of the single-ship scenario through an OpenAI gym environment, and then learn using this entirely new environment. Note that while we trained this bot in a simulated environment, the learning problem remained identical. In terms of selecting adequate parameters, we primarily based our selections on the models used to train DQN on Atari games.

We created two reward function for our DQN: immediate and delayed reward. Immediate reward rewards ships fro depositing halite as they do it while delayed reward sets the reward after a game to be the amount of halite collected within the game. We will want to determine which reward function is viable and in the case that they are both viable which one performs better.

Two key components to the way this model trains in the Q-learning is via an exploration and exploitation phase. At a particular iteration, the model either guesses ("explores") or leverages the data in the Q-table ("exploits") in order to determine the move to make next. The model chooses which of these two to perform based on the epsilon-greedy strategy, which dictates to explore with probability $1 - \epsilon$ and to exploit with probability ϵ. Initially ϵ starts off small, and as the number of iterations in the game increases, this value increases. Based on the results of the chosen parameter at a particular iteration, the values in the Q-table get updated accordingly (Mnih, 2013).

The algorithm used also leverages the dueling enhancements, which is an approach in which multiple neural nets compete with each other in an attempt to have better learning, similar to human imagination (Wang, 2016).

2.4 Expectations

We expect our second approach using reinforcement learning to perform better than our supervised learning approach for a number of reasons. As explained above, the supervised learning approach takes data from a more complicated problem and tends to learn to mimic, while the reinforcement learning approach is tailored for this problem where we have a discrete action space to choose from. We anticipate that it may be difficult to tune the parameters correctly for our DQN approach but regardless we predict that it will outperform the SVM. Overall, we predict that the genetic benchmark and reinforcement learning bots will achieve the highest performance, as these bots are specifically tuned for the single-ship, 300-turn scenario.

3 Results and Evaluation

To evaluate the performance of our DQN and SVM bots, we compared against two benchmarks: a simple rule-based bot and a genetically-tuned rule-based bot.

Specifically, we compared performance by running each bot over 500 games on a 32x32 map with 300 turns per game. We tested each of the four bots on the same 500 map seeds.

3.1 Rule-Based Bot

For our first benchmark, we created a simple rule-based bot that operates using a greedy choice. The ship harvests from its current cell until the cell is below 100 halite, at which point it moves to the adjacent cell with the most halite. When its cargo reaches the maximum capacity of 1000, it travels to the shipyard and deposits.

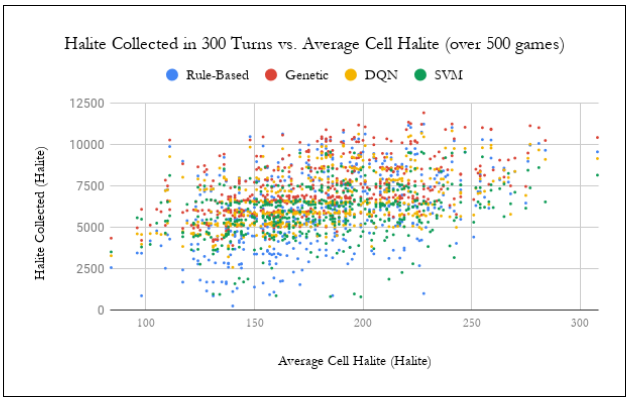

Figure 2

3.2 Genetic Bot

For our second benchmark, we developed a high-performance rule-based bot using a custom genetic algorithm to tune various parameters. The ship scans surrounding cells for the closest cell above a certain threshold. The scan distance, and threshold amount are both tuned genetically. When harvesting, the ship harvests its current cell until it is considered "scarce." This notion of scarcity is dependent on a multiplier of the average surrounding halite and an absolute minimum, both of which are tuned. Finally, the cargo threshold at which the ship returns to the shipyard is tuned.

Our genetic algorithm is initialized with 20 individuals, each with random "genes" for each parameter. Each generation, each individual plays five games. Using a fitness function proportional to the cumulative percentage of halite harvested over the five games, fittest individuals are selected using tournament selection. The parameters of the offspring are determined by crossing over and mutation.

We ran this custom algorithm for 200 generations with 20 individuals per generation, at which point performance stabilized and we had developed a formidable benchmark.

3.3 Results

For each of the 500 games played by the four bots, we record the total halite deposited and the average halite per map cell. Figure 2 depicts a scatter plot of each of the 2000 data points, with average cell halite plotted on the X-axis and total halite collected on the Y-axis. Figure 3 depicts the trailing average with period 16 of this same data. We chose this metric, as dense halite maps naturally result in more collected halite, so it is

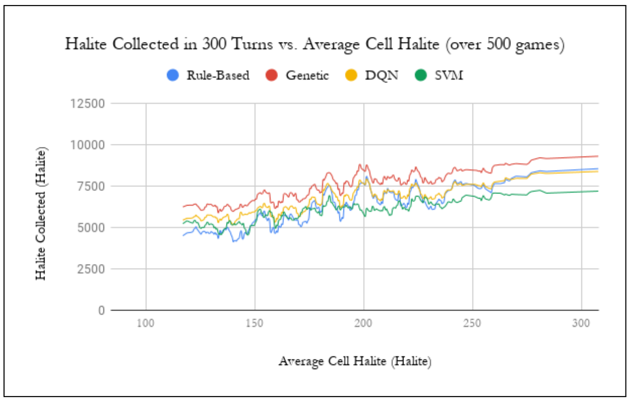

Figure 3

insufficient to simply compare average collection amounts over the 500 games.

Our results indicate that both machine learning bots adequately learn halite collection behavior, performing on a similar level to the two benchmarks. We find that the genetically-tuned bot outperforms all other bots across all map densities. Our DQN bot outperforms the rule-based bot for low halite maps, but achieves similar performance for medium and high density maps. Finally, our SVM bot performs on a comparable level as the rule-based bot for low halite games, while it underperforms all bots in medium and high density scenarios.

3.4 Discussion

As predicted, we find that the Reinforcement Learning bot and genetically-tuned bot achieve the highest performance. These results were expected, as the DQN and genetically-tuned bot are specifically trained and evolved for the single-ship, 300 turn scenario. Meanwhile, the SVM is trained on a slightly different dataset and the rule-based bot trains on nothing.

We observe that the DQN was able to learn gameplay simply given map data and rewards immediately upon depositing halite. In our experiments, however, we were unable to achieve the same performance using the delayed reward function. Using this function, the DQN bot only receives an award proportional to the total amount of halite deposited at the end of the 300 turn game. Given that the DQN receives no reward for 99.66 percent of turns using this reward function, it is understandable that inadequate performance was achieved. Regardless, we find that using the immediate reward function, the DQN is able to learn the difficult problem of harvesting halite and returning to the shipyard. We note that this is a difficult problem, because the DQN cannot simply learn to return to the shipyard once its halite cargo reaches a certain threshold, as moving between cells costs halite and will bring ships cargo below this threshold. Thus, this would result in a never-ending cycle. Instead, a more complex function for behavior must be learned.

We find that the SVM, while achieving slightly worse performance than all other bots, achieves a relatively comparable level of play. Given that this SVM trains on the more complex, multiple-player multiple-ship scenario, we find that it still generalizes well to the modified single-ship game.

4 Future Work

Through the experiments conducted and the results obtained, we have shown that applying machine learning to creating bot algorithms is an interesting and useful application. However, there is much more exploration to be done in this context.

Notably, the experiments conducted in this context were of simplified versions of the Halite game; we reduced the number of ships to one, fixated the map size, fixated the number of iterations, and stuck only to single-player optimization goal. In the real game, the parameters are more complex and there are more variables and states to consider at each iteration (most interestingly, the other players' states).

Adapting the trained bots to multiplayer mode in particular raises some more interesting questions and explorations, including possible game-theoretic considerations in which a player might try to optimize his move by guessing what his opponents will do in order to maximize their chance of winning.

In addition, there are much more types of machine learning bots that would be interesting to test and consider their performance, along with variations in the parameters, and also variations in the map size used for the game. One final interesting area to experiment with is to also relate these approaches to other games, and see how well they translate to them.

5 Related Work

Incorporating machine learning tactics into creating game algorithms has been heavily experimented with. Different types of classifiers and topics, commonly neural nets and reinforcement learning, have been used to create computer bots for several games. One such application, done by DeepMind, involves applying reinforcement learning to creating a highly performing AI for many games, including Chess, Shogi, Go, and others (Silver, 2017). In particular, using a type of reinforcement learning called *tabula rasa* reinforcement learning, the authors were able to train an algorithm that, given no information about the game at all other than the rules of the game, outperformed the best known previous algorithms in each game, which even translated well to computationally harder games such as Shogi.

While many bots have been created using machine learning for multiple games, however, we note that the game-specific problem of applying this tactic to Halite is somewhat innovative. There are many possible ways to successfully apply machine learning principles to create a bot, and the distinct feature representations used in our algorithms along with the tactics applied to our specific simplified version of Halite have, to the best of our knowledge, not been specifically experimented with previously.

6 Conclusion

In summary, we trained two bots with machine learning; one supervised bot via an SVM classifier, and one based off of a neural network with reinforcement learning. After evaluating the performance of the classifiers to our benchmark bots, we see that the neural network bot performed considerably better than the SVM bot, yet overall still performed significantly worse than the genetically-trained rule-based bot. Nevertheless, we know that both machine learning bots performed at a level above complete randomness because they had a similar level of performance to the rule-based benchmark. Hence, we can say that both machine learning bots managed to make meaningful decisions.

Although our neural net bot performed fairly well as compared the benchmarks, it did not perform as well as we would have initially hoped. One way to potentially improve this could be to introduce a reward function that gives a reward after a constant number of moves equal to the amount of halite that was collected during those moves. In other words, rewards would be spaced out among intervals instead of either being delivered instantly when a deposit is made or at the end of the entire game. This would allow the bot to learn that a pattern of behavior is good as opposed to specific actions being good. This would potentially allow the bot to learn startegies more easily.

References

Mnih. 2013. *Playing Atari with Deep Reinforcement Learning*, volume 1. University of Toronto, Toronto, Canada.

Santiago Ontanon. 2012. *Experiments with Game Tree Search in Real-Time Strategy Games*, volume 1. arXiv.

Silver. 2017. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*, volume 1. DeepMind, London, UK.

Wang. 2016. *Dueling Network Architectures for Deep Reinforcement Learning*, volume 1. Google, London, UK.