Building a Convolutional Neural Network to Play Halite II

Michael Hong

December 12, 2019

Abstract

Halite is an open-source artificial intelligence programming challenge created by Two Sigma. Halite II is its second rendition, where bots play on a two-dimensional virtual board in a simultaneous turn-based game, almost akin to a slow real-time strategy game. The virtual board randomly places planets symmetrically on the board. Players spawn with three ships which can then dock on planets and mine for Halite, which will spawn more ships to further mine Halite or attack enemy ships and planets.

This project attempts to use a machine learning approach to playing the game—according to Two Sigma’s post-mortem, not many bots used a machine learning approach when the challenge was live. Supervised learning, and in particular a CNN, was chosen for a few reasons. First, an immense replay database is available for training. Second, the game board lends itself well to encoding it as an image for the CNN. Finally, the memoryless nature of the game means that any given turn or frame can be looked at individually, which further simplifies the game and therefore the network as well.

For the implementation, I modified Two Sigma’s machine learning starter bot to use a Convolutional Neural Network (CNN) to attempt to improve on their bot, which uses a standard feedforward neural network. The input encodes various data from the game state into an image, with each layer representing different data. The output then determines the proportion of ships to allocate to each planet. Ultimately, a lot of information is lost from Two Sigma’s default implementation, but our model still performs well despite not using the heavy machinery as in the default network.
Contents

1 Introduction 3
1.1 Background and Motivation .............................. 3
1.2 Setup ........................................ 3
1.3 Gameplay ....................................... 3

2 Methodology 5
2.1 Input .................................. 5
2.2 Output ........................................ 5
2.3 The Neural Network .......................... 6

3 Results 8
3.1 Loss Minimization .................................. 8
3.2 Performance against other Bots ......................... 9
    3.2.1 Settler .................................. 9
    3.2.2 Random Bot ............................ 10
    3.2.3 Starter Bot ............................ 11
    3.2.4 Summary ............................... 12
3.3 Time ........................................ 12
3.4 Summary ..................................... 12

4 Conclusion 14
4.1 Challenges ..................................... 14
4.2 Future Work ..................................... 14
4.3 Summary ..................................... 15
1 Introduction

1.1 Background and Motivation

One of the greatest breakthroughs in the field of artificial intelligence is AlphaGo, which beat top players in Go, a game with immense complexity using supervised learning and reinforcement learning. This project is motivated by the first portion of the supervised learning part of AlphaGo. With the restriction that this project is an undertaking by an undergraduate with limited resources, a full reinforcement learning agent to defeat the top bots is likely impossible. However, a supervised learning agent that can defeat an average bot is certainly possible, and useful as a setup for a potential reinforcement learning agent in the same way AlphaGo is set up.

1.2 Setup

The game is played with two or four players—in the implementation of this agent, we will assume the two-player game. The board is rectangular, with a 3:2 aspect ratio. The board size can vary from 240 x 160 units to 384 x 256 units. Planets are randomly placed, with a cluster of four guaranteed to be in the center. Other planets are placed symmetrically and can be larger than those found in the central cluster. Players start with three ships and no owned planets, arranged symmetrically across from the other team.

1.3 Gameplay

The game concludes when a bot wins (all enemy ships dead or all planets owned) or the turn limit is reached. Every turn, bots receive the game state and have two seconds to issue commands for every ship. Turns are calculated in the following order:

1. The status of any docked ships is updated.

2. Player commands are processed. For instance, a new thrust command will instantaneously update the ship velocity.

3. Movement, collisions, and attacks are resolved simultaneously.

4. Damage from combat is applied.

5. Planet resources/ship creation is updated.

Ships have 255 integral health points, zero to seven integral velocity in any direction, and a weapon that deals 64 damage per turn with a range of five units. Ships move with a thrust command, with an angle in integral degrees and an integral velocity. A command must be given every turn to move the ship.
Planets have a radius and health associated with it. They have unlimited resources, but a maximum number of ships can dock on it depending on its radius. Only one team can dock on a ship. A ship can dock on a planet once it is within 4 units of a planet, and it will fully dock in five turns. Once docked, the ship begins to mine the planet. Each ship docked produces six units per turn, and 72 units are required to produce a ship, which spawns two units from the planet. The ship fails to spawn if there is not enough space near the planet. Finally, undocking takes five turns and returns control of the ship to the owner. A ship that is docking, docked, or undocking cannot fire its weapon.

Ships can damage other planets and ships in a few ways.

First, a ship can collide into a planet, dealing damage to the planet equal to the amount of health the ship had. A planet has 255 health for every unit of radius it has—a full-health ship has 255 health, so a planet can withstand one collision with a full-health ship for every unit of radius it has. A ship can also collide into another ship, and both will die.

Ships can fight other ships once they come within the five-unit weapon radius, dealing 64 units of damage per turn. If multiple enemy ships are within range, the damage is split evenly among them. Ships may only fire its weapon once, which applies when there are multiple enemy ships that come within range within one turn. A ship that moves toward multiple enemy ships fires at the first one it encounters within the turn, and does not fire at others. If it encounters multiple at the same time, then it splits the damage evenly.
2 Methodology

First, the bot must train a model from a replay database. For our CNN, it turns out that the training is arguably limited by time, rather than the number of replays. I have an extensive replay database provided by Two Sigma, with thousands upon thousands of replays, which is more than enough for our model to train on.

I load a certain number of replays to train on and find the player who won the most games. I then train on this player’s games, trying to emulate their playstyle as well as possible. I will now describe the structure of my network.

2.1 Input

First, I scale down the precision by a scalar factor, thus simplifying the game. Essentially, I turn the game into a grid of squares with side lengths of my scalar factor. I have chosen this scalar factor to be 8, but it is roughly arbitrary, chosen to minimize run-time while maintaining accuracy.

I then have the following four input image layers.

1. The first layer is an image that represents the planets, with each neutral planet assigned the value of 1, the target player’s planets assigned the value of 2, and enemy planets assigned the value of 3. Planets can have different sizes, so this layer attempts to also encode the size of the planet by assigning the ownership value to all grid blocks that cover the planet.

2. The second layer is an image that represents the number of the player’s ships in a given grid block.

3. The third layer is an image that represents the number of enemy ships in a given grid block.

4. The fourth layer is an image that represents the number of ships that can still dock onto a planet. This layer uses the same method in the first layer, where the value is assigned to all the grid blocks that cover the planet to stay consistent with the encoding of planet sizes.

From a high-level perspective, we have encoded the location and size of the planets, the location of the player’s ships, the location of the enemy’s ships, and the remaining spots for docking for every planet.

2.2 Output

For our output, we must be able to command our ships to act in a certain way. Unfortunately, the number of ships is variable and technically uncapped, and there are a number of possible commands that can be given in a single turn (thrust, turning, docking, etc.) and therefore directly
training on executed ship commands is nearly impossible. We must use a heuristic to simplify the commands to every ship that works no matter how many ships we have. Ultimately, thinking from a high-level perspective, we will send ships to a destination in mind. After all, it usually does not make sense to have ships float around without intent.

The solution that Two Sigma’s starter bot uses is to determine what proportion of ships are allocated to each planet, and use that as the output layer. As Two Sigma explains it, their model uses a neural network to try to answer the question: Given the current situation in the game, what I used this heuristic as well.

Therefore, our output is a vector of length 28, which is the maximum number of planets that can be on a map. Each element in the output represents the percentage of ships that should be sent to each corresponding planet.

2.3 The Neural Network

The neural network itself has a very standard structure for a CNN:

1. Input layer, described earlier
2. 2D Convolution + Rectified Linear Unit (ReLU) Layer
3. Pooling Layer
4. 2D Convolution + ReLU Layer
5. Pooling Layer
6. 2D Convolution + ReLU Layer
7. Flattening Layer
8. Fully Connected Layer
9. Dropout Layer
10. Fully Connected Layer
11. Softmax Layer

The convolution and ReLU layer extracts features from the image using a convolutional filter over the image and running it through a ReLU activation function to eliminate negative values and add non-linearity to our model.
We then add a pooling layer, which downsamples by taking the most important information. In this case, I have chosen to take the max of every 2x2 square. This helps with overfitting, as it reduces some of the noise and makes our model more general.

Repeat these layers a few more times, and then we flatten the output so we can connect it to a dense network to train the high level features extracted from our convolutional layers to the correct output. We insert a dropout layer, which randomly sets some of our nodes to zero. This obviously makes training harder, but it helps with overfitting [1], which was a significant problem throughout this project.

Finally, we connect to a softmax layer to normalize our predictions. We then train to reduce cross-entropy loss from our predictions. We use cross-entropy loss because it uses log-loss, which is used for models predicting outputs with probabilities between 0 and 1.
3 Results

Once we have successfully trained our neural network, we want to know whether it is any good. We have a few ways of looking at this, and we will analyze our performance on each of these metrics.

1. How much did we minimize loss?
2. How does our bot perform against the extremely simple greedy algorithm, Settler?
3. How does our bot perform against the same bot but with random output?
4. How does our bot perform against the provided ML starter bot?
5. How much time does it take to train our model?

3.1 Loss Minimization

First, we can compare our training loss and cross-validation loss with that of the original ML (non-CNN) starter bot that was provided. Evidently, as can be seen in the below figure, the starter bot achieves a lower loss, converging near 2.3.

Figure 1: Loss for ML Starter Bot (1000 samples, 5000 steps)

![Figure 1: Loss for ML Starter Bot (1000 samples, 5000 steps)](image)

However, our loss for our CNN does not achieve quite the same success. As seen from the figure
below, there is a bit of overfitting, as the cross-validation loss diverges from the training loss. The model converges at around 2.6.

![Loss for CNN (500 samples, 5000 steps)](image)

Evidently, in terms of loss, our bot does not perform as well as the original starter bot, as it overfits and also has more loss. However, it remains to be seen exactly how important this difference is.

### 3.2 Performance against other Bots

To test our performance against other bots, we will run our bot against the other bot 100 times. Our analysis will stem from these results, as well as a few sampled replays.

#### 3.2.1 Settler

We begin by comparing our bot against certain benchmark bots. The first will be the non-ML starter bot, Settler, which uses a fairly simple greedy algorithm to allocate bots. We anticipate that our bot should achieve success against this bot, partially because the greedy algorithm in our bot is already more powerful than Settler.

Indeed, we achieve quite remarkable results:

\[
\text{Win ratio [Settler.py (P1) to MyBot.py (P2)]} = 1:99
\]
Obviously, this is a resounding success for our bot. However, that single loss is concerning. A look into the replay reveals that two of our ships crashed into each other and disintegrated, leaving us with a 1:3 ship deficit, as can be seen in the figure below:

Figure 3: CNN (magenta) self-collision against Settler (cyan)

The reason this occurs is fairly evident. There is no safeguard in either the neural net or the greedy algorithm to prevent two of our own ships from crashing into each other. The preprocessing removes the fine granularity that would tell us that ships are about to collide, since we consider the game in 8x8 grid blocks. Luckily, this disaster occurs infrequently enough that our bot can still achieve significant success against Settler, and thus should not significantly affect our performance in general.

3.2.2 Random Bot

Clearly, we can do very well against a simple non-ML bot. However, how much of our success against Settler can be attributed to the heavy machinery used in the greedy algorithm, or the general weakness of the Settler bot? We want to see how good our model is, and we can do this by comparing our bot against itself, except with random inputs/allocations to the greedy algorithm, with no neural network. We anticipate that we will beat this random bot, or else all our work would be for naught.

\[
\text{Win ratio [RandomBot.py (P1) to MyBot.py (P2)]} = 26:74
\]

We indeed defeated the random bot, by a considerable margin as well. Clearly, we see that
the greedy algorithm used is quite powerful, as the random bot achieves a whopping 25 more wins compared to Settler. However, we are still pleased to see that our bot defeats the random bot a significant amount of the time.

3.2.3 Starter Bot

Finally, we want to see how our model fares against the original starter bot that we modified. From the analysis of the loss, it seems inevitable that our bot will lose against the starter bot.

\[ \text{Win ratio [StarterBot.py (P1) to MyBot.py (P2)]} = 94:6 \]

Unfortunately, our bot gets obliterated. Perhaps this was to be expected, but the margin of defeat is immense. Upon analysis of the replays, the starter bot is more consistent in its strategy. The CNN bot sometimes sends ships to turn around seemingly without aim, whereas the starter bot tends to avoid this behavior.

The figure below shows an example where the starter bot benefits from this behavior. In the below example, my ships and starter bot’s ships spawn roughly in between the six planets on the left and right, respectively. The starter bot immediately beelines for the nearest available planets, whereas my CNN bot roams around the middle for a bit before finally committing to the middle planets.

Figure 4: Starter bot (cyan) gets a head start over CNN (magenta) by turn 9

![Figure 4: Starter bot (cyan) gets a head start over CNN (magenta) by turn 9](image)
3.2.4 Summary

In the table below, I have included the win rates of the four different bots I have tested.

<table>
<thead>
<tr>
<th></th>
<th>Settler</th>
<th>Random</th>
<th>Starter</th>
<th>CNN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Settler</td>
<td>X</td>
<td>7:93</td>
<td>0:100</td>
<td>1:99</td>
</tr>
<tr>
<td>Random</td>
<td>93:7</td>
<td>X</td>
<td>0:100</td>
<td>26:74</td>
</tr>
<tr>
<td>Starter</td>
<td>100:0</td>
<td>100:0</td>
<td>X</td>
<td>94:6</td>
</tr>
<tr>
<td>CNN</td>
<td>99:1</td>
<td>74:26</td>
<td>6:94</td>
<td>X</td>
</tr>
</tbody>
</table>

Evidently, the starter bot achieves dominant success against the other two, while my CNN manages to knock off 6 games out of 100. The CNN is not as good as the starter bot, but there is a lot of progress that was made.

In terms of high level behavior, it seems that the Settler bot generally plods slowly to the nearest planets, and wastes a lot of time thrusting slowly.

The random bot will move without purpose, generally going fairly random places, but always in the direction of some planet.

My CNN will typically send ships to the nearest planets, but it often gets confused when a ship is between planets, often opting to fly between them rather than to either one.

Finally, the starter bot avoids this problem and generally flies its ships toward planets with meaning.

3.3 Time

Time is a significant cost for training the model, especially because there is such a massive replay database.

The following tables are the output from the shell script `time` for training starter bot and the CNN. On the left is the CNN with preprocessing, the middle is CNN after preprocessing, and the right is the starter bot.

<table>
<thead>
<tr>
<th></th>
<th>real</th>
<th>user</th>
<th>sys</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNN</td>
<td>56m39.229s</td>
<td>29m7.915s</td>
<td>23m14.737s</td>
</tr>
<tr>
<td>CNN</td>
<td>12m58.773s</td>
<td>25m44.753s</td>
<td>6m7.082s</td>
</tr>
<tr>
<td>starter bot</td>
<td>9m32.738s</td>
<td>4m30.985s</td>
<td>2m46.864s</td>
</tr>
</tbody>
</table>

Clearly, even after preprocessing, the CNN is slower to train than the starter bot. With preprocessing, the cost is many times more, to the point where it is almost prohibitive to train the thousands upon thousands of replays available.

3.4 Summary

We achieved success against the non-ML bot Settler, against the random agent RandomBot, and managed to take a few wins off of the powerful StarterBot. The results against StarterBot were
rather lackluster, with our CNN winning only 6 of 100 games. However, this is actually somewhat promising, given that the game itself does not lend itself particularly well to being converted into an image. Our preprocessing was prohibitively expensive, as our input is not given to us in the form of an image, but instead as a collection of data of the current ships and planets. Therefore, our CNN must translate this information into an image.

We lose both detail and speed, as the data is extremely precise—the coordinates are given as floats, for instance. Though the map is at best 384x256, converting this image into an array with all the exact locations is infeasible, as we would need a much larger array to account for all the possible floats.

Furthermore, there is information that we can only attempt implicitly encode in the image that is explicitly calculated in the StarterBot, such as a planet feature named "gravity" (the sum of all ship healths divided by the squared distance from the planet). My implementation allows for most of the features to be encoded, but even then there were already issues with overfitting.
4 Conclusion

4.1 Challenges

The biggest challenge in this problem is translating the game into an appropriate input for our CNN. Processing the game map into an image with exact granularity would have been impossible. In fact, a surprising amount of detail had to be sacrificed for reasonable running time. We managed to save some of this time by preprocessing the game states before throwing it into our neural network, but even then, it was incredibly inefficient.

This preprocessing process of translating the game state into an acceptable input for our CNN was both an issue in terms of time, but also information. A lot of information is conveyed through the data that is not easily conveyed in the image, especially with the scaling factor that we used. For instance, ship health is completely ignored in the CNN, as many ships can occupy a single grid block. Also, planets are represented by the blocks that cover it, but many planet radii are approximately 6-10 units, and our scaling factor is 8. Consequently, planets end up being represented by 4 grid blocks in a square, and the actual radius is obfuscated, as the network cannot extract any notable features since every planet more or less looks the same.

Another significant challenge is overcoming overfitting. In the end, our model still overfit the data by a bit, but it was a challenge to train well on the training set, but also test well on the testing set. Multiple techniques were used to mitigate this problem—the dropout layer, making the model smaller, preprocessing[2].

I believe that these problem can be overcome, though the results I have are likely near the best one can do. The cross-validation loss for the starter bot settles above 2.3, while my bot settles at 2.6. With the imprecision that results from the first issue I presented, I hypothesize that it is impossible to achieve or beat the starter bot. Certainly, a loss of 2.6 is still a bit away from 2.3, but it is close enough that any improvements would be marginal.

4.2 Future Work

For future work, I would apply the methods used in this project for another game, especially one that suits a CNN better. For instance, traditional CNN on an old-school game with limited pixels could be successful. Pong, for example, seems like a textbook example of where a CNN could be useful.

In terms of improving on the results from this project, I would see how I can fix the issue of overfitting and also improve the cross-validation loss. I would be interested to see if it would be possible to improve the CNN to the point of defeating the starter bot, which I previously hypothesized to be an impossible feat, especially with similar resource constraints. This achievement would
be an exceptional breakthrough within the scope of this project.

4.3 Summary

In the end, this project took something that worked very well, and produced something that did not work quite as well. However, the results are actually quite encouraging. Given the complicated nature of the game, as well as the inconvenient form of the input, it is quite encouraging that we can achieve good results against the random bot and reasonable success against the starter bot. I believe that the method of using a CNN to train a supervised learning agent to play a game could achieve better success than a typical dense network if the environment is right.

References
