Halite Agent Optimization via GPU-Accelerated DQN

Brendan Caywood[1], Alexander Fountain[2], Thomas Bailey3, and Jarred Parr[4]

***Abstract*— Actor Critic Agents provide a robust, reliable way to product highly efficient bots that can self learn as a result of their training.**

I. INTRODUCTION

The Halite AI competition is commonly seen as a way to build AI best practices, try some new strategies, and also just learn something new. In most cases, people use a minimal amount of the myriad of machine learning based tools available. This is typically done in favor of more simple, and streamlined approaches. The top scoring agents in the leader board are currently mostly hard-coded with complex logic handling situations like collisions, collection, and offensive strategies. Genetic algorithms were fairly common to tune the games parameters and make bots that used their limited senses efficiently. All of the aforementioned strategies were attempted with mediocre results. Whether it was lack of time, ingenuity, or otherwise, our team decided that something else needed to be done. After looking for ways to improve computation times on our genetic algorithm, the idea of reinforcement learning to self tune the entire model was brought up and considered as a viable approach to solve the issues imposed upon the bots.

II. BACKGROUND

Reinforcement learning is a perfect blend between decisions, and the delayed rewards those decisions can produce. In most cases, reinforcement learning acts as a result of states, and the actions in which it can take based on that state. The goal is for the algorithm to maximize reward without making mistakes that would incur a harsh punishment. As a result, this appeared to be the perfect use case for the halite agents. In this case, the goal was to find the moves that maximized the hailte at the end state of the game. Through the actions it takes, the goal was to prioritize the location of a global maximum in which the agents would have robust enough movements to not only perform well, but also act in a way of self preservation in the event of enemy collision. There were a number of things that had to be considered to make this happen, which will be discussed below. This is how this problem can be solved with a standard reinforcement approach which, in this case, would be just a Q-learning algorithm. To make things a bit more complex, and also to facilitate more automated tuning, we utilized a neural network (commonly referred to as $deepreinforcementlearning$, to optimize the mapping of state-action pairs to a given reward. This is done via our policy agent system, which can be seen in $policy\_net.py$. A policy agent does exactly what was just discussed, it takes the state action pairs, and their outcome and, over time, it begins to be able to infer the best possible outcome given the pair. Our neural network is able to learn in this way. Since we do not need to worry about the generated data, this gives us the freedom to allow the agent to self train, and we really only need to tune the γ, which is our discount factor to prevent taking the local best, and a few other hyperparameters for our optimization algorithm, which uses the $AdamOptimizer$. The Adam Optimizer is a better choice than gradient descent in some places due to it being able to converge more quickly and with less tuning, however, it comes at the cost of computation time. In this context, we were able to optimize past the potential bottlenecks introduced as a result of this to great avail.

III. CONSIDERATIONS

Many processes are in place to optimize well known algorithms such as this, and the team made an effort to apply both known and holistic guess-and-check systems. For example, we were able to observe that the use of a deeply layered neural network caused the granularity to be a big boost at the cost of run time performance. While it was still able to perform well, we had to do a lot of tuning on the number of layers to avoid large bottlenecks within our architecture. This is primarily because, as mentioned previously, with greater granularity comes an extremely large amount of computational overhead as you get into 6+ layers. Unfortunately, our data was far from linearly separable which prevented us from utilizing a more simple architecture, and when embedded in the

Q network, it took some time to get things from timing out as a result of the jumps between different pieces of software. Another particularly challenging system was optimizing the system to run well on a CPU. The test environment for the final system will be run on a MacBook Pro, so reliance on GPUs in the final model will not be a viable choice, which required a strong rewrite of some of our core modules. Outside of that, the only other main considerations were that of the typical machine learning process of inference, testing, and tuning of hyperparameters. This was most of the process for converging on a final model which will be used in the presentation.

IV. EXPERIMENTS

A vast majority of experiments took place when selecting the model. Initially, the process of selecting a model was daunting. The team tried everything from a multi-class support vector machine to a genetic algorithm approach. Unfortunately, these were all bottle necked by the movement algorithm that we had been working on. Something needed to be done as it would have simply taken too much effort to try and get all of these systems talking effectively in the little time we had remaining.

A. The Cartpole Experiment

Cartpole is a popular reinforcement learning exercise available within the OpenAI gym simulator. It basically is a game where the cart learns to balance a pole sticking straight upward. It's pretty standard, and most decent algorithms can get things working pretty quickly. However, upon coding this up and trying for ourselves (via the PyTorch example series), we were able to see that much more was available with this type of process. We found that, when applied to the context of the halite game, we were able to achieve a great degree of control over how the game operated. We were able to accomplish the task of removing collisions and also were able to maximize the results of our bot far beyond what was originally provided via a more brute-force approach.

V. GROWTH AREAS

As a team, Jarred and Thomas had the most experience with reinforcement learning. Through teaching Brendan and Alex they not only were able to strengthen their own knowledge, but also could create a more stable and robust model as a result of the increased experimentation on the testing environment. Overall, everyone grew in knowledge of common libraries, python programming, and the general steps of the machine learning process.

VI. CONCLUSIONS

Overall, this project was a very steep learning curve the entire time. Halite had an API that was very minimally documented and sometimes unclear. An extensive amount of reading of the source within the provided code was pivotal in the design of our system. Debugging and other development systems were able to be streamlined by the existence of the ability to see just how everything was working "under the hood" so to speak. The challenge itself was fun and engaging, however, it is a shame there wasn't more time available to work on the task and really perfect the model we were creating.

REFERENCES

[1] Skymind AI, A Beginners Guide to Reinforcement Learning, 2019