Reinforcement Learning Hearthstone Battlegrounds

1st Noah Lattari
noah.lattari@ryerson.ca
500760404

2nd Darryn Roopnarine
darryn.roopnarine@ryerson.ca
500876974

3rd Nichalus Tofflemire-Varela
ntofflem@ryerson.ca
500744672

4th Rony Verch
rverch@ryerson.ca
500761324

***Index Terms*—Reinforcement-Learning, Video-Games, Agent, RL, Hearthstone**

I. Introduction

GITHUB: https://github.com/noahlattari/RL-Hearthstone

Artificial Intelligence (or more namely Reinforcement-Learning) has a long lasting relationship with video-games. Many researchers believe video-games to be a perfect fit for training an agent to achieve a human-level performance [1].

"Hearthstone" is a free-to-play digital card game developed by Blizzard Entertainment that builds upon the lore of their previously popular video-game "World of Warcraft". The original Hearthstone is a complex turn-based card-combat simulator between real life opponents that requires many skills that would be found in other traditional card games.

"Hearthstone Battlegrounds" is similar in the fact that it is still a strategic turn-based card game, but you are facing off versus 7 other random opponents to attempt to become the last player standing. The complexity, strategy, and variability of each game of Hearthstone Battlegrounds sounded like the perfect test for a Reinforcement Learning agent to crack.

Part of our motivation for such a project comes from reading about Google Research's "Alpha Go" defeat Lee Sedol, the former 18-time world champion at the Chinese table-top game of "Go" [2]. This match was extremely significant due to the fact that the general public widely believed that Go was always too complex and variable of a game for a Reinforcement Learning agent to conquer.

As previously mentioned, one of the biggest challenges when training a Reinforcement Learning agent to play such game is the amount of variability and possible outcomes that could happen in each turn. With over 50 different cards (some with unique abilities) the game can become very complex very quickly.

Due to this, we have developed a text-based simulator using Python to recreate Hearthstone Battlegrounds rounds within the command-line. This command-line simulation approach is much more efficient compared to teaching an agent to navigate the actual game embedded with an executable file. It allowed us to focus more on the learning environment, and less on the game implementation.

The following report will outline the problem statement environment, the model used to train the agents and the results of the training.

II. Problem Statement and Environment

We built the Python simulator that recreates a text-based Hearthstone Battlegrounds game. With that program, we generate a "roll" (an example state of a current board), based on the roll, the agent has the ability to purchase new minions (cards), sell minions they have, or play minions.

For our reward, we came up with an equation to reward the agent who performed well during the combat phase (played a better hand than their opponent). This equation is scaled to have a bigger reward (or penalty) the longer you are into the game. The more late into the game you "survive", the more damage your character takes on a loss (and the more damage it deals to other players on a win), meaning we should penalize or reward the agent harder the later the rounds go. The following is our equation on how we calculate positive rewards within the combat phase.

$$weight = 1 - 1 \div (curr\_round + 1) \tag{1}$$

$$(win\_chance \times damage) - (loss\_chance \times damage) \times weight \tag{2}$$

The states are the observations the agent gets to use. The following are our available states: roll, board, hand, tavern cost, gold available, health, current round, enemy health, enemy tier. Enemy health and tier are based on who the agent is fighting next. For the roll, board, and hand, since the minions are text based minions (they have a name), these minions have to be vectorized in order to have a numerical value to represent them.

The actions are the steps the agent is able to take. The following are our available actions: upgrade tavern, reroll, freeze tavern, buy, sell, play, and end the turn (recruitment phase).

III. Methods and Models

For extraction methods, we did not need to do too much extra work. We built the Python simulator (as previously mentioned) to emulate a proper Hearthstone Battlegrounds game which produced access to everything we'd need in environment. With that, we can get all relevant information (current roll, board, health etc.) to pass to the Reinforcement Learning Proximal Policy Agent to produce our policy.

With that policy we were able to pass in observations and gather actions. These actions (a list of 28 values) are probabilities which became our weights. The choice of which action

to take would be influenced by the weight but would still have some entropy to it. The reason we wanted randomness is because the action with the highest probability won't always be the action that should be used. If it always chooses the "top" action it will only learn to take the best actions for the current turn, not incorporating future rewards. We add some entropy to the decision so it can learn more based on diverse scenarios and actions. Afterwords, we iterate through all actions until the end of the turn is reached.

Originally our reward was based off of two different scenarios. The first being for a negative action. If the agent attempted an action that was impossible (buying a minion without the required gold available) that would yield a negative reward. The following is our equation on how we were calculating negative rewards. We assume an average round length of 15 based on our knowledge of Hearthstone Battlegrounds.

$$avg\_round\_length = 15 \tag{3}$$

$$-1 \times (curr\_round \div avg\_round\_length) \tag{4}$$

We had another reward for combat, but this ended up over complicating the algorithm with too many rewards. Too many rewards ended up confusing the algorithm because it did not understand the difference between a negative reward from a combat loss (losing a round) versus an action loss (doing an impossible task), resulting in a poorly trained agent.

We considered two methods to solve this problem. A proximal Policy Agent (PPO Agent) and a DQN (Deep-Q Network). We decided on a Proximal Policy Agent because it is generally simple and easy to estimate, very general, adaptive to different scenarios, and usually out performs other policy gradient methods [3]. In terms of why we chose a policy gradient solution vs the traditional Deep-Q Network approach, most researchers believe that Policy Gradients are able to be applied to a wide range of problems. Also, when the reward function is too complex a DQN can fail "miserably" [4]. Policy Gradient algorithms also tend to converge faster than Deep-Q Network algorithms [4].

The only reward we use is the previously mentioned combat based reward. Doing impossible actions is bad, but with the negative reward it was learning that certain actions were wrong and ruling them out entirely. Just because actions are possible (rerolling all your gold and not buying minions) it doesn't translate to good rewards within the combat phase. Isolating the rewards only to combat rewards yielded longer games and better trained agents that become more cautious towards the end of the game. The end of the game (round 10-15) were more important as users take more damage and have an easier chance to die.

For performance metrics, we considered the total reward for various agents, loss for the different agents, and game length. In regards to game length (rounds per game), we wanted to conclude if there was a pattern between agents learning and how long or short the games go for. Do better trained models average shorter or longer games? This was the main question we wanted a solution for.

IV. Results and Discussions

The following results are based off of training the agents for 700 Epochs (700 simulated Hearthstone Battleground games).

For game length based on the graph (Fig. 1) we were able to conclude that the amount of games/rounds we trained for yielded no specific correlation between the agents and the rounds of the game. We initially assumed that there would be a correlation, but based off of our training we did not find one. Our original assumption was that the games would either be consistently longer or shorter. If the games were longer, it could mean that the agents were more evenly matched and are able to last for a longer period of time. On the other hand, if the games were shorter, it could mean that certain agents are training better then others (which was expected) and would have an advantage by knowing to buy minions that could defeat other opponents. With further training, we believe a conclusion or pattern could appear.

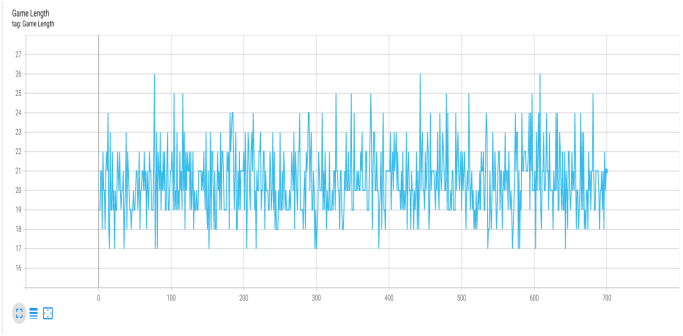

Fig. 1. Game Length by Epochs

In regards to the total reward different agents received throughout the training (Fig. 2, Fig. 3, Fig. 4) we could see agents improving over time. The total rewards for agent 7 shows that the agent started off playing well, defeating the rest of the agents consistently and continued throughout many epochs. On the other hand, agent 2 showed that it began poorly, loosing a lot of combat scenarios and continued to lose throughout the training epochs. This shows us that agent 7 had the advantage by winning combats early on in the training cycle, and continuing to win due to other agents not being able to train quickly enough. Agent 2 most likely started off being one of the first agents to die every game, and was not given a good opportunity to learn as they were part of less episodes. Finally, agent 1 is a good example of an agent learning linearly over time. The agent began with negative rewards which continued for approximately the first 150 Epochs, but was able to learn and begin to consistently get positive rewards (win rounds).

In addition, we compiled graphs for training loss for each training iteration, but the graphs did not show any correlation to the agents improving so we decided not to include them.

In order to attempt to mitigate the issue of not enough training being completed (number of games simulated), we used the win/loss probabilities based off of if the combat was repeated 1000 times. As the combat was automatically simulated in a real Hearthstone Battlegrounds game and have randomness as a big aspect to them (what minion is attacked and when), this resulted in less randomness for our rewards after each combat. This did mitigate the issue of not enough training epochs to a certain extent, as visible in agent 1 who was shown to have learned, but further training would continue on improving the models.

Lastly, if we were to continue the project one thing we would do is repeatedly update the agents to be a copy of the best one after a set of epochs. With agent 2, it was clear that since the agent was loosing combats early on, it continued loosing all throughout. If after a set of iterations we set all the agents to be a copy of our agent 7 (the best performing agent), all the agents would be of the same skill, and would diverge from there, with some better then others. As well, with the current training the best performing agents may have only done so well due to the worst performing agents, giving them easy combat wins. Updating the agents to be copies of the best one, we believe would significantly improve the agent self playing.

V. Implementation and Code

Our final project is mainly broken up into two parts. The first part is the actual Python Hearthstone Battleground command-line simulator which everything but the combat phase was coded from scratch. The reason we did not code the combat phase (simulation of the battles) from scratch is due to the fact we found a handy C++ GitHub repository that was able to simulate it for us. This application takes advantage of Python's Object-Oriented Programming features in order to organize the project into different classes (Player, Tavern, Minion etc). We have also coded all minions (cards) and their specific (and sometimes unique) effects.

For the combat phase simulator, we pass in a text representation of the board-state from our Python tavern simulation. The combat phase will then return a statistically probability of an agent winning that round based on the board given. An example input and output scenario is shown below:

For the Reinforcement Learning portion, we decided to use the popular Machine Learning package Tensorflow alongside the tf_agent class. To monitor real-time analytics and performance we decided to setup TensorBoard. To code this we used the Tensorflow documentation heavily for everything Reinforcement Learning related. For a baseline of our project we referenced the "Dota 2 with Large Scale Deep Reinforcement Learning" paper by OpenAI [5] since it seemed like a pretty similar (albeit expanded) version of our project.

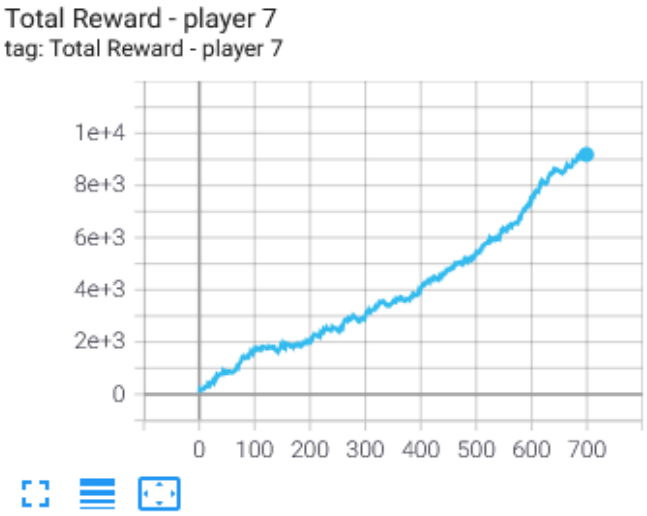

Fig. 2. Total Rewards of Agent 7

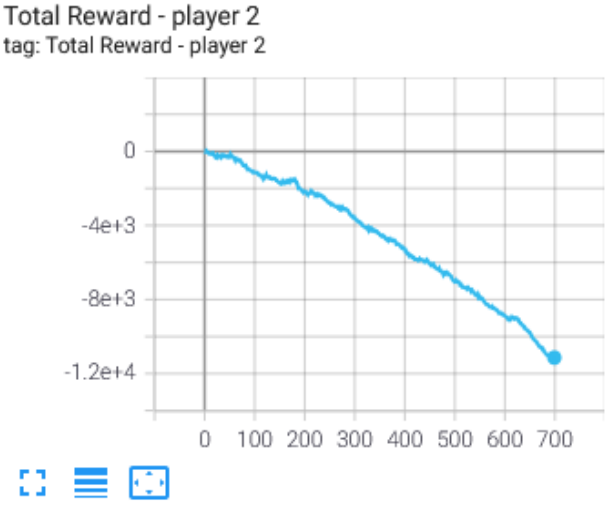

Fig. 3. Total Rewards of Agent 2

Fig. 4. Total Rewards of Agent 1

```
Turn 8
* 4/6 Cave Hydra
* 8/2 Kaboom Bot
* 10/4 Kaboom Bot
* 11/6 Nightmare Amalgam
* 2/2 Lightfang Enforcer
* 4/6 Imp Gang Boss
VS
* 2/2 Kaboom Bot
* 2/2 Kaboom Bot
* 6/3 Cobalt Guardian, divine shield
* 2/6 Security Rover
* 4/2 Micro Machine
* 1/5 Junkbot
* 5/6 Psych-o-Tron, taunt, divine shield
--------------------------------
win: 2%, tie: 3%, lose: 94%
mean score: -6.573, median score: -7
percentiles: -14 -11 -9 -8 -8 -7 -6 -5 -4 -3 12
actual outcome: -9, is at the 20-th percentile
```

Fig. 5. Example input and output.

ACKNOWLEDGMENT

We would like to acknowledge Github user ”twanvl” for their C++ implementation of the Hearthstone Battlegrounds combat phase which helped our project tremendously.

REFERENCES

[1] K. Shao, Z. Tang, Y. Zhu, N. Li, and D. Zhao, “A Survey of Deep Reinforcement Learning in Video Games,” arXiv.org, 26-Dec-2019. [Online]. Available: https://arxiv.org/abs/1912.10944. [Accessed: 18-Apr-2021].

[2] C. Moyer, “How Google’s AlphaGo Beat a Go World Champion,” The Atlantic, 08-Apr-2016. [Online]. Available: https://www.theatlantic.com/technology/archive/2016/03/the-invisible-opponent/475611/. [Accessed: 18-Apr-2021].

[3] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal Policy Optimization Algorithms,” arXiv.org, 28-Aug-2017. [Online]. Available: https://arxiv.org/abs/1707.06347. [Accessed: 19-Apr-2021].

[4] “Felix Yu,” Deep Q Network vs Policy Gradients - An Experiment on VizDoom with Keras — Felix Yu. [Online]. Available: https://flyyufelix.github.io/2017/10/12/dqn-vs-pg.html. [Accessed: 19-Apr-2021].

[5] OpenAI, : C. Berner, G. Brockman, B. Chan, V. Cheung, P. Debiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, H. P. d. O. Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang, “Dota 2 with Large Scale Deep Reinforcement Learning,” arXiv.org, 13-Dec-2019. [Online]. Available: https://arxiv.org/abs/1912.06680. [Accessed: 20-Apr-2021].