**Exercise in Reinforcement Learning Using *StarCraft: Brood War***

ECET 49100 Senior Design Phase II – DRAFT Project Report

Ben Harruff

Submitted

April 27th, 2020

Table of Contents

[Executive Summary iv](#_Toc38923749)

[1 Introduction 1](#_Toc38923750)

[1.1 Objective 1](#_Toc38923751)

[1.2 Motivation 2](#_Toc38923752)

[2 Project Description and Goals 3](#_Toc38923753)

[3 Technical Specification 5](#_Toc38923754)

[3.1 Agents and Machine Learning 5](#_Toc38923755)

[3.2 The Markov Decision Process 6](#_Toc38923756)

[3.3 Reinforcement Learning 7](#_Toc38923757)

[3.4 Neural Networks 10](#_Toc38923758)

[3.5 Policy-based and Value-based Learning 12](#_Toc38923759)

[3.6 Actor-Critic 13](#_Toc38923760)

[3.7 Proximal Policy Optimization (PPO) 13](#_Toc38923761)

[3.8 Definition of Actions and Consequences 15](#_Toc38923762)

[3.9 Software Function 17](#_Toc38923763)

[3.10 Map Scenario 18](#_Toc38923764)

[4 Design Approach and Details 20](#_Toc38923765)

[4.1 Design Details 20](#_Toc38923766)

[4.2 Codes and Standards 21](#_Toc38923767)

[5 Schedule, Tasks, and Milestones 22](#_Toc38923768)

[6 Results and Acceptance Testing 24](#_Toc38923769)

[7 Budget and Cost Analysis 26](#_Toc38923770)

[8 Conclusions and Future Work 27](#_Toc38923771)

[9 References 30](#_Toc38923772)

[10 Bibliography 32](#_Toc38923773)

[Appendix A – Code 33](#_Toc38923774)

[avoid\_reavers\_PPO\_Ben\_v4.py 33](#_Toc38923775)

[PPO.py 39](#_Toc38923776)

[Appendix B – Tables 44](#_Toc38923777)

[Appendix C – Training Data 45](#_Toc38923778)

[Appendix D – Final Results 47](#_Toc38923779)

# Executive Summary

In this exercise, a reinforcement learning agent using Proximal Policy Optimization (PPO) and Actor-Critic techniques was created to interact with a simple scenario in the video game *StarCraft: Brood War*. It was coded in Python, and used a framework released by the Samsung Research Team for the express purpose of training *StarCraft* agents. Within the framework, there is a connection between the agent and its environment, which allows it to collect data about the scenario. This data is processed by the agent using two parallel neural networks. After development was finished, the finalized agent took just under 28 hours to train. After training the agent, it was able to achieve a 99.363% deviation from the theoretically perfect score, which is enough to meet the 95% deviation standard set forth initially. It served as a learning opportunity and challenge for someone interested in machine learning, and it continues to stand as an opportunity for others to learn as well. While the agent was well within the standards for success, there is much that could be improved on. The knowledge gained from this experience will likely be extrapolated to work for a much more ambitious agent in the future.

**Exercise in Reinforcement Learning Using *StarCraft: Brood War***

# Introduction

The team spent 300 hours over 14 weeks to develop an agent to interact with tailored scenarios within the *StarCraft: Brood War* video game environment. This required no monetary expense.

## Objective

Essentially, the purpose of the project is to produce an AI (artificial intelligence), or agent, that can complete a scenario created in the video game *StarCraft: Brood War* (often shortened to *StarCraft*, Brood War, or SC:BW). The agent was developed in Python 3.6.8 using Visual Studio Code. The framework for the agent was developed by the Samsung Research and Development Team (SAIDA), which uses a specialized API (Application Programming Interface) library called BWAPI (Brood War API) that acts as an input/output (I/O) bridge between the agent and the game environment. Additionally, there are several libraries that were necessary to implement reinforcement learning for the agent, such as Tensorflow and Keras (with Tensorflow as a backend). This allows the agent to slowly improve between episodes. This means that the agent starts with no information regarding actions to take, but it understands actions it could take and the result of the action when it has been performed. This means that after every episode, a new and improved iteration of the agent is generated in a process referred to as training. Ultimately, the goal of the fully trained agent is to perform well in the scenario presented to it.

## Motivation

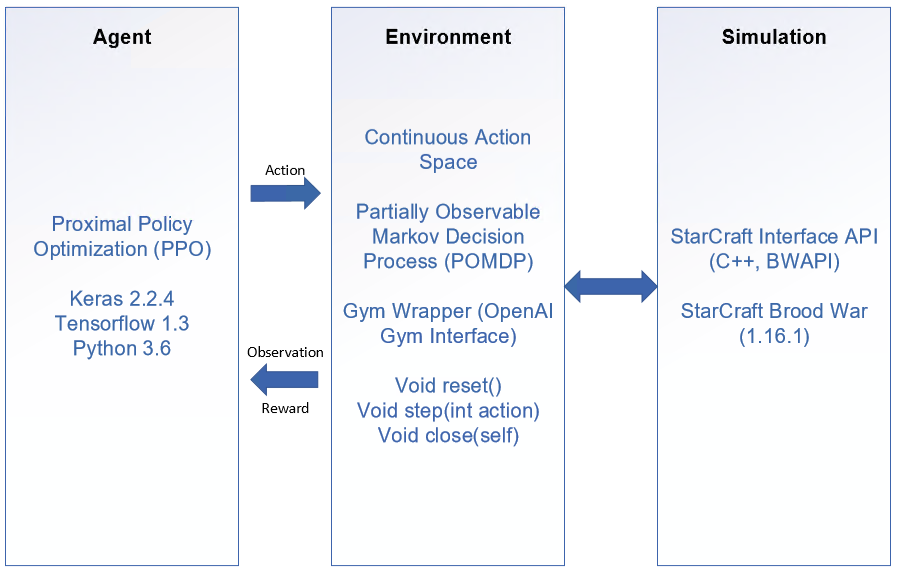
The justification for the project is to further the general understanding of reinforcement learning using a unique and fun development environment. I have personally been working on these *StarCraft* scripts for over two years, it is something that I am both very familiar with and very passionate about developing and sharing with others. Scripts that I have developed have competed in an ongoing online tournament known as the *Student StarCraft AI Tournament* (SSCAIT) [1] [2] [3], organized by the Games and Simulations Research Group of Czech Technical University in Prague. In fact, watching a livestream of the tournament is what sparked my passion for developing AI in *StarCraft*, a game I have played since I could use a computer. Because I am interested in reinforcement learning, and because many of the most competitive scripts developed for the tournament make extensive use of it, I thought that this project would be a perfect opportunity to combine both of my passions.

In addition to these reasons, machining learning has great potential to impact several established industries. Companies from Samsung to Facebook have put together research teams to explore how machine learning could be used to benefit them. Both, in fact, have created agents that have competed in the non-student category of the SSCAIT. Facebook’s CherryPi bot won the tournament in 2017, and Samsung’s SAIDA bot placed the best on the competitive ladder in the 2018-19 season [2]. The time and resources poured forth from both massive companies goes to show how *StarCraft* is the current frontier for research in machine learning and the diversity between the two companies, particularly Facebook, exemplifies the range of industries advances in machine learning can impact.

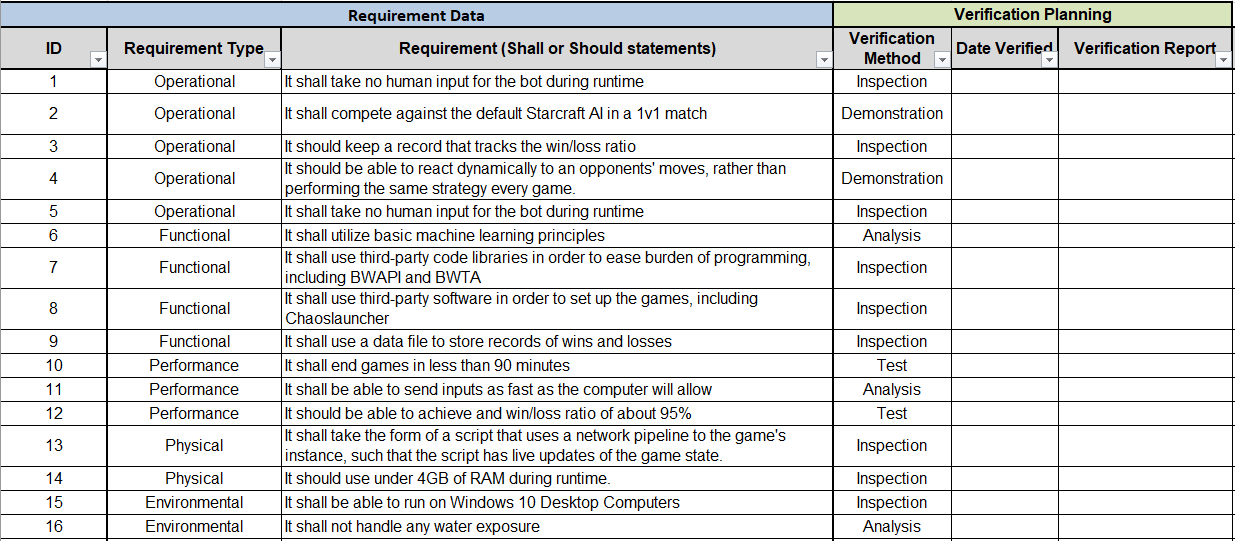
# Project Description and Goals

The agent was developed in Python 3.6.8 using Visual Studio Code. The framework for the agent was developed by the Samsung Research and Development Team (SAIDA), which uses a specialized API (Application Programming Interface) library called BWAPI (Brood War API) that acts as an input/output (I/O) bridge between the agent and the game environment. Additionally, there are several libraries that were necessary to implement reinforcement learning for the agent, such as Tensorflow and Keras (with Tensorflow as a backend). This allows the agent to slowly improve between episodes. This means that the agent starts with no information regarding actions to take, but it understands actions it could take and the result of the action when it has been performed. After every episode, a new and improved iteration of the agent is generated in a process referred to as training.

The environment that the agent interacts with is a custom-made *StarCraft BW* scenario, one of many provided by the suite of tools made by SAIDA. It is a simple scenario with a complicating twist: the agent must move from point A to point B while avoiding 3 randomly moving obstacles. This creates an interesting decision-making process in a stochastic environment for the agent to converge on an optimal solution for.



Compared to the original project requirements I set, due to drastic scope changes, most of the requirements were made either irrelevant. Here are the original system requirements:



Here are two requirements that were changed or cannot be met:

* Requirement 2 – The original requirement was to defeat the in-game default AI in a normal 1v1 match, now it is to complete the scenario.
* Requirement 12 – The original requirement was to achieve 95% win rate, but now it is to achieve a successful agent that is within 95% episode score as compared to near-perfect agent.

Additionally, the original place was to use framework of my own design from previous work, but as a result of research, it now uses a framework designed by the SAIDA team specially made to perform the tasks necessary to complete the project [4].

# Technical Specification

## Agents and Machine Learning

According to Oxford, an *agent* is defined as “a person who acts on behalf of another person or group.” In terms regarding machine learning, an agent interacts with an environment. The agent has a range of actions that can be taken every step of a process, which directly impact the environment’s state. Based on the state of the environment resulting from the action, the agent learns which action to take or not to take, either immediately or over the course of several similar environment states.

In terms of the environment in question being *StarCraft,* the agent acts in place of the human Player while the environment is how the in-game environment is received by the agent. This is an important distinction, as the actual in-game environment is not what the agent perceives as is the case with the OpenAI team’s work with the Atari domain [5]. Rather, a there is third-party software called an API that bridges the gap between the agent and the environment. This will be gone into much more detail in Section 3.8, but for now, that is all that is important to know about .

## The Markov Decision Process

A Markov Decision Process (MDP) is used to describe an agent in an at least partially stochastic environment. Based on state *s* of the environment, the agent must choose an action *a*. Actions change the environment, leading to a new state. Based on the difference between the new state and the old state, a reward *r* may be given to the agent. The agent forms a *policy* for how to choose actions based on the rewards it gets from state transitions. The entirety of the states, actions, and policies crafted by the agent in a stochastic environment is the MDP as it is implemented in this exercise. A sequence of states, actions, and any possible sets of resulting rewards and states is referred to as a *trajectory* τ, and is shown as such:

τ = s0, a0, r1, s1, a1, r2, s2, … sn−1,an−1,rn,sn

In an episode of the Markov process, there is an amount of rewards that is equal to the amount of states minus the terminal state. The sum of these rewards is called the *total reward*.

R = r1 + r2 + r3 + … + rn

The *total future reward* relative to time *t* is represented as:

Rt = rt + rt+1 + rt+2 + … + rn

Due to the fact that the environment is stochastic, every state past *st* becomes less predictable, meaning that we must put less weight into the potential rewards from the future states. This allows the agent to make decisions that will lead to the most cumulative reward. Thusly, using the last equation, one could represent *discounted future reward* as:

Rt = rt + γ(rt+1 + γ(rt+2 + γ(…))) = rt + γRt+1

where the discount is represented as gamma *γ*. The higher the value of gamma, the more weight we put into future rewards, and vice versa. Gamma values tend to be between the values 0 to 1. A value of 0 means that any reward past the next immediate reward is ignored while a value of 1 means that every foreseeable reward is as valuable as the immediate one. In essence, discounted future reward allows one to place a value on predictions, thus giving the agent the ability to foresee which sequence of actions is most optimal. Since future rewards are imperative to account for in complex problems such as the one in this exercise, the value of gamma is typically 0.9900, but can otherwise range from 0.8000 to 0.9999 for most problems.

Now that we have an understanding of these concepts, we have laid the foundation for reinforcement learning as it pertains to the agent in this exercise.

## Reinforcement Learning

For the agent to behave optimally in any situation presented to it by the game, the agent will use a principle called reinforcement learning (RL). RL employs some principle(s) of machine learning to build a sequential decision-making process to optimally solve a problem. Essentially, RL is an MDP where the states and actions are clearly defined and observable, but the probabilities and rewards are unknown and must be learned. In layman’s terms, RL is concerned with learning how to solve a problem via a system of defined actions, the consequences of those actions, and whether the consequences produced a good or bad result. For example, in a well-programmed agent, if the result produced from the agent is good, then the agent is more likely to take that action in a similar situation in the future. Conversely, if the result of the action is bad or less good than another action, the agent is discouraged from taking that action again in that situation. To that end, the goal for the agent is to maximize the amount of good actions it takes. Each action and state together represent a step, and many steps occur over each episode, and many episodes occur over the entire training session. More good actions taken over bad actions should theoretically get the agent closer to maximizing its success.

In order to come close to perfection, the agent must rule out sub-optimal policies. In doing so, the agent runs into the exploration vs. exploitation dilemma. The agent must choose either to explore, meaning that it attempts to rule out sub-optimal policies, or to exploit, which means to take what the agent knows to be the optimal route. Heavy exploitation leads to the agent choosing what it thinks to be optimal, but may not choose what is actually the most optimal. Heavy exploration leads to the agent exhausting all of its options, which makes the training process take far longer due to how action sequence possibilities compound on themselves with every timestep. Thus, it is important to strike a balance. The goal is to explore *enough* options to make an assumption on what the optimal policy must be to exploit.

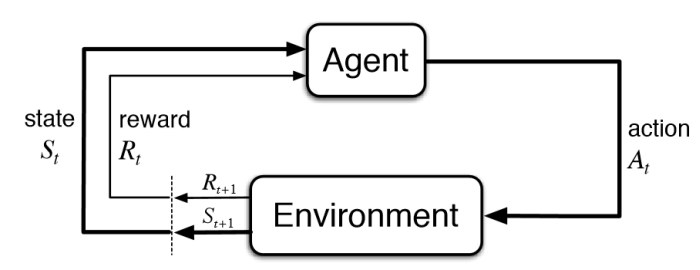


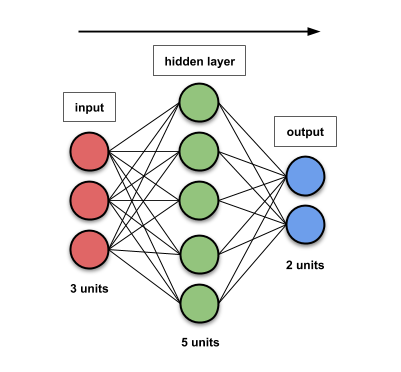
Figure 2.1: Basic diagram depicting reinforcement learning. Source: [towardsdatascience](https://towardsdatascience.com/applications-of-reinforcement-learning-in-real-world-1a94955bcd12)

In order to come close to perfection, the agent must rule out sub-optimal policies. In doing so, the agent runs into the exploration vs. exploitation dilemma. The agent must choose either to explore, meaning that it attempts to rule out sub-optimal policies, or to exploit, which means to take what the agent knows to be the optimal route. Heavy exploitation leads to the agent choosing what it thinks to be optimal but may not choose what is actually the most optimal. Heavy exploration leads to the agent exhausting all its options, which makes the training process take far longer due to how action sequence possibilities compound on themselves with every timestep. Thus, it is important to strike a balance. The goal is to explore *enough* options to assume what the optimal policy must be to exploit.

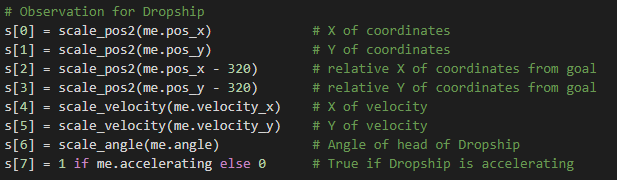
There are a couple ways to curb this problem. One is to employ an entropy coefficient, which is used to determine how quickly to converge on an optimal action given a state. The smaller the coefficient, the slower the rate of randomness reduction for the action decision-making process. The typical range is 0 to 0.01, 0.01 of which is used as the coefficient for this exercise simply because that is what is most common [6]. Another way to curb the dilemma is addressed with Proximal Policy Optimization, which is covered in Section 3.8.

## Neural Networks

A neural network, named for its similarity to how the brain functions, is a system of layers of web made up of many simple computation nodes tightly woven together [7]. In this exercise, the neural network flows in one direction, from input to output. This means that a layer has nodes that are tied to every node in the previous layer and the following layer, but not its own layer. These nodes represent weights of values of input data, where, as data flows through the neural network, they are multiplied by the weights of the nodes it passes through. Before data continues from one node to the next, it must meet a threshold in order to pass. Training a neural network, where every weight and threshold starts at random values, means that these values are tweaked until all inputs reach a similar output (preferably a good output). Below is an example of a simple Feed-Forward Neural Network (FFNN) that consists of an input layer of size 3 units, a hidden layer of size 5, and an output layer of size 2.



In similar exercises, such as the OpenAI team’s work with the Atari domain [5], typically screencaps of the environment are taken and processed by the agent in order to determine states. This exercise is unique in that BWAPI allows us to parameterize properties of the units in order to use them as input. Once the values are normalized, they can be used in a neural network.



These parameters keep track of the position of the dropship (0-1), the position of the dropship relative to the goal (2-3), the velocity vector of the dropship (4-5), the angle of the dropship (6), and whether or not the dropship is accelerating (7). This gives 8 parameters to keep track of for the dropship as input, as well as the same 8 parameters for the 3 obstacles, giving a total of 32 parameters. This means that the input layer must be 32 in size. Similar to the input layer, the output layer size must be as big as there are outputs. In this scenario, the agent has the choice between going in any one of 20 directions a constant distance or to stop. This means that the output layer size should be 21.

The layers in between the input and output layers are known as hidden layers, as their values are not seen by the agent. This is where the complex computations occur, and because the number of hidden layers as well as their size can be whatever we want, it can basically be decided how robust the decision-making process is. The values used in the exercise will be covered more directly in Section 4. For the size of the hidden layers, it can range from as low as 32 for straightforward problems, and as high as 512 for the most complex problems [8]. For the number of layers, this usually ranges from 1 to 3, where more layers are necessary for more complex problems.

## Policy-based and Value-based Learning

Policy-based methods are best suited to stochastic environments, which are in large part used in this exercise because of this reason. Simply put, a policy as it refers to RL is a probability distribution for actions that can be taken in an observed state. The objective of policy based RL is to learn optimal polices for a given set of parameters. Policies are denoted by pi *π*, often with subscript theta *θ* denoting a set of parameters. Thus, πθ(*a*|*s*) represents a policy for specific values of a set of parameters at for action *a* at state *s*. All policies are tuned such that we maximize the objective function.

Value-based methods, rather than understanding the value of actions to form policies, strive to understand the value of being in a specific state. By understanding the optimal states that are required to maximize the objective function, the policies formed from policy-based methods can be molded to choose actions that the value-based methods deem to be most valuable. This forms the basis for actor-critic methods, which is discussed in the next subsection.

## Actor-Critic

Actor-Critic methods attempt to combine the benefits of policy-based and value-based learning. The actor is the policy-based part of the agent while the critic is the value-based part. The critic deals with determining the value of actions taken by the actor in order to influence the policy in a meaningful way (i.e. policy evaluation). In short, the actor makes policies and the critic assigns values to the policies. Both the actor and critic start with copies of the same neural network, and then proceed to populate them individually.

Practically speaking, actor-critic share parameters between their neural networks, but this means nothing unless we use values from both in a meaningful way. Thus, both the surrogate function and

## Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) is a cutting-edge method for policy gradients in RL developed by OpenAI [6]. A traditional stochastic policy gradient ascent updates once every time-step, which has shown to be very effective for optimizing neural networks [9]. However, PPO furthers this idea by introducing epochs, which refers to the number of times that the same time-step is iterated upon. Additionally, PPO employs surrogate functions, which serves to simplify the objective function, decreasing the time it takes to train. This is done by taking a finite sample of data points and finding the set of actions that produce the maximum reward. The surrogate function can be further improved through use of clipping, meaning that any values outside of a certain range are ignored. The clipping-loss coefficient is represented at epsilon ε. Clipping the surrogate function cuts down the objective function dramatically, allowing for must quicker policy updates than what is normally possible [6]. The naive objective function would thusly be represented as:

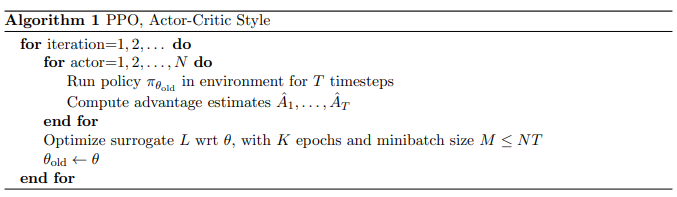
,

where *LCLIP* represents the clipped surrogate function, Ê*t* represents the expectation, *rt(θ)* represents the surrogate objective, and *Â* represents the advantage function. In summary, the surrogate function is clipped as per ε, and then the minimum value between the surrogate function and the unclipped surrogate function is found.

The actual algorithm for the objective function in PPO, actor-critic style is represented as:

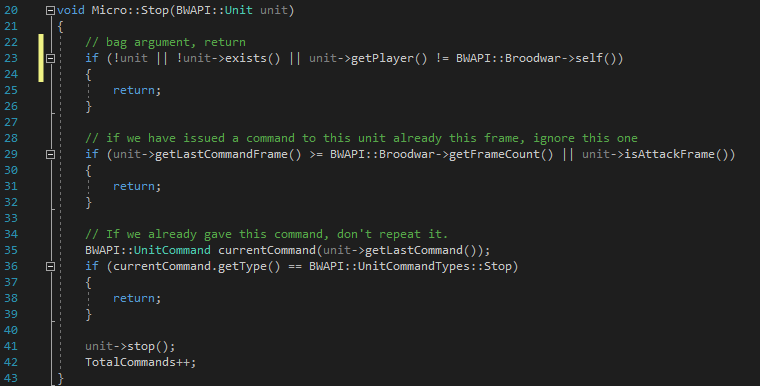


where . An estimate is made from the clipped surrogate function, modified by a squared error loss and by an entropy term, respectively.



## Definition of Actions and Consequences

The agent will need to have extensive definitions for actions (output). Take a Marine for example, which is a Unit that can be controlled by the Player (Player being the agent). Marines, like most Units, have a few commands that can be issued to them by default: Move, Stop, Patrol, Attack, and Hold Position. On the surface, one might say that this means that there are exactly 5 actions that the Marine could take, those being 1 of the 5 actions previously listed. However, it’s not so simple. For example, if the Move command is issued to the Marine, the Player must select a Position for the Marine to go. Since Positions are defined down to the precision of pixels, there are typically several thousands of possibilities regarding which Position to move the Marine. The same goes for the Hold Position and Attack commands. Stop doesn’t require a Position, but Patrol requires two Positions. This is one of the simpler cases. Some units can fly, allowing them to reach terrain that can’t be reached by ground units, meaning that there’s more Position possibilities. Some units have additional commands, like Burrow, Cloak, Morph, etc. An example of an action coded out in BWAPI is in the following figure. In summary, all these additional variables presented by these commands need to be thoroughly outlined and heavily considered when determining the optimal action for a Unit to take.

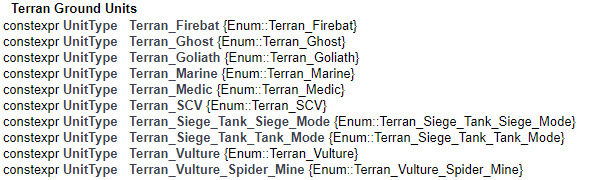


The agent also needs to have extensive definitions for consequences as a result of actions (input). Consequences, meaning the result of an action, are defined by how it affects the progression towards the goal. Ultimately, the goal is victory over the opponent, but to win in Brood War, all the opponent’s buildings need to be destroyed. This would be simple enough if the Player knew where the buildings are, but the Player can only see portions of the map revealed by their own Units. Thus, even though revealing portions of the map technically doesn’t bring the Player closer to victory, it is a means to the end. The more the Player reveals the map, the more the Player either knows where the opponent’s buildings are or, in most cases, where they aren’t. Either result is good, and so revealing the map is a high priority for the Player, and thus is a high priority for the agent to strive for. Another example would be the Player losing Units, typically by means of combat with the opponent’s Units. Losing units hampers a Player’s ability to reveal portions of the map and ability to destroy enemy buildings. Thus, even though the Player’s buildings aren’t being destroyed directly, the Player losing Units makes defeat more likely, and should be avoided at all costs.

To summarize, consider an encounter between a Marine and another Unit called the Zergling. The agent controls the Marine and the opponent (the default AI) controls the Zergling. Through trial and error, the agent has learned that it wins most fights with a single Marine against a single Zergling. With that information, the agent would decide to Attack the Zergling with the Marine because the agent thinks that attacking the Zergling indirectly brings it closer to achieving victory. One result would be that the Marine wins the fight. Because the Marine killed the opponent’s Zergling, the agent’s idea that that action was good is reinforced, and thus is even more likely to perform that action given the circumstances than it was before. The other result would be that the opponent’s Zergling was able to pick off the agent’s Marine. The agent’s action brought about bad consequences, and thus is discouraged from taking that fight in that situation again. Perhaps next time, in a similar situation, the agent would decide to retreat the Marine back to base (use the Move command) until reinforcements arrive, where victory would surely be more likely. The point is that defining the system that the reinforcement learning interfaces with is essential to producing an agent that will perform in an ideal fashion, and in turn is essential to the success of this project.

## Software Function

The agent utilizes an application programming interface (API) built expressly to form a connection between the agent and Brood War, aptly named Brood War API (shortened to BWAPI). BWAPI provides a library of commands, utilities, and functions for the agent to use for output, while also providing a myriad of variables regarding the game for the agent to use for input (Figure 2.3). In other words, BWAPI can interpret code from the agent as actions to take in the game while concurrently relaying every facet of the game state back to the agent. Because this read/write acts on every frame of the game as it unfolds, not only does this allow for extremely responsive I/O, but additionally this allows for thousands of actions per second.



Example of BWAPI documentation.

This is a small section of the BWAPI::UnitTypes namespace, which is how Units in the game are distinguished from one another in the agent. In other words, Units like Marines and Medics can be told apart in the agent by using this. While it may be obvious to someone observing the game, this is necessary for the agent to “read” the game.

## Map Scenario

The scenario consists of a board of tiles, 10 high and 10 wide. The top-left flag represents the beginning episode position, the bottom-right square represents the goal, and the 3 yellow guys (Reavers, as they are called in the game) are the obstacles. The Reavers are programmed to move to a random location in the blue area repeatedly, thus creating a stochastic environment. At the beginning of the episode, the dropship spawns on top of the flag, then the agent takes over and controls the dropship from there. The episode ends when the dropship reaches the bottom-left. The following graphic displays the scenario as it was created in the *StarCraft* Map editor software (ignore the red and white circles).



# Design Approach and Details

## Design Details

The detail of the design lies in the code in Appendix A, and other information in Appendix B. I’ve omitted several pages of code from the SAIDA [4] and Keras/Tensorflow libraries [10] both for the sake of brevity and because I had no hand in modifying it. I’ve included the PPO algorithm because of the implementation of the algorithm, and I’ve also included the controller of the agent that was created. For quick reference, here’s the values that were used to create the agent in tabular form:

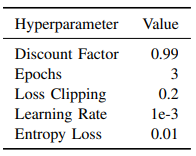


Table 1: PPO Hyperparameters used to train agent.

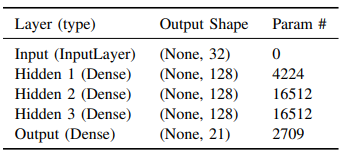


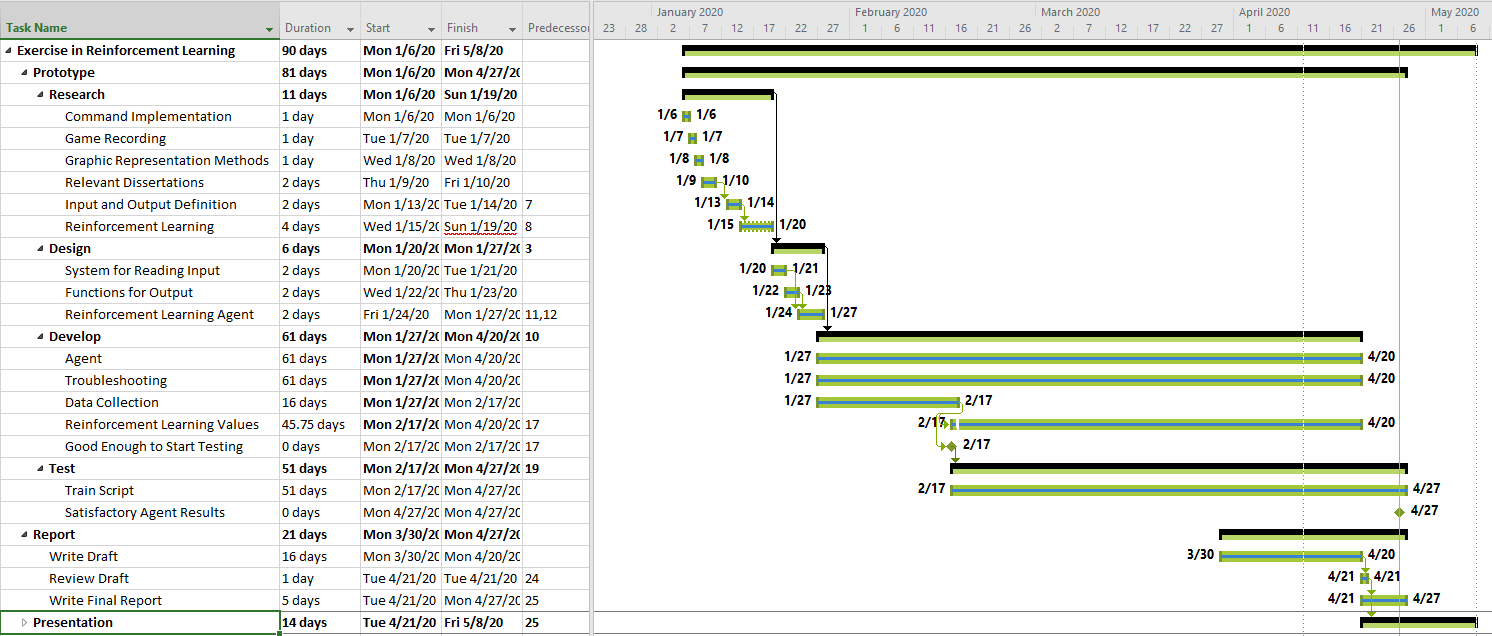
Table 2: Feed-Forward Neural Network information.

## Codes and Standards

As far as I am aware, there for no codes or standards that apply to this project.

# Schedule, Tasks, and Milestones

Below is a Gantt chart describing the timeline of the exercise:



The biggest milestones were as follows:

* Completing Research, which allows for development to begin.
* Completing Development, which allowed for testing to begin.

# Results and Acceptance Testing

The agent was able to successfully train over the course of 4,000,000 timesteps, which comes out to just under 28 hours of total training assuming that each timestep only takes 25ms to complete. The following graph represents the process of training for start to finish, which came out to form a nice logarithmic trendline:

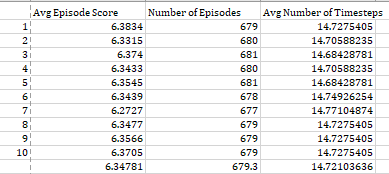
In order for me to find the results, and thus the training, to be satisfactory, I’d like to see the agent achieve a total reward that is within 95% of the near-perfect episode score. Theoretically speaking, the agent should be able to achieve an average episode score of 6.3771 at best according to the following computations.

(10000 steps/ *n* episodes) = *x* steps per episode.

Optimally, the agent advance its position every step, resulting in +0.1 score for each of those steps. The agent’s last timestep is it reaching the goal, resulting in +5.0 reward. The agent should not hit any obstacle or move away from the goal, as this would result in -5.0 or -0.1. By these rules, the best score the agent can achieve is over the course of 10,000 timesteps is:

*total episode reward = (x – 1)(0.1) + (1)(5.0)*

After demonstrating the agent’s improvement up until the submission of this report, these are the results of 100000 timesteps, which ended up being 6793 episodes total. It should be noted that the agent is not training during this demonstration, it is simply using the knowledge it has gathered and learned from up until that point. For frame of reference, statistics from previous benchmarks in training will be provided.



According to the results, the total average score per episode was 6.3478 (rounding down). Now it needs to be compared to the highest possible score. Using the equation derived earlier, one can derive the highest possible score where x is avg number of timesteps:

*highest possible avg. episode reward = (14.72103636 – 1) \* (0.1) + (1) \* (5.0)*

*highest possible avg. episode reward = (13.72103636)\*(0.1) + 5.0*

*highest possible avg. episode reward = 6.37103636*

*highest possible avg. episode reward = 6.3710*

Now with the highest possible score and the tested score of the agent, one can determine the percent deviation from the “true value.”

***Deviation = 99.636%***

Thusly, according to the calculations, the agent is well within being 95% of being perfect, coming in at 99.636%. That means that the results were satisfactory. No additional testing is planned as of submitting this report.

# Budget and Cost Analysis

This project required no monetary expense on the part of the team or a sponsor.

# Conclusions and Future Work

After downsizing the scope of the initial project considerably, I’d say that the exercise in reinforcement learning has been a success. I’ve learned a great deal about reinforcement learning, which after all was the whole purpose of the exercise in the first place. Actor-Critic with PPO was certainly the best option available to me from the beginning, due to the complexity of the problem and the expedience that was required to finish the project by the deadline. In hindsight, I feel like the way I went about constructing the agent was the best way to do it. Consequently, I cannot in good confidence state what I would do differently for this exercise, or what I’d suggest to someone willing to pick up where I left off. To me, it has served its purpose; I would simply use my knowledge gained from this to expand the scope of the next exercise relative to this one. However, while I do believe that the agent I’ve trained is effective enough to meet my standards for success, it is far from perfect. In other words, there are several potential ways for the agent to be improved in this scenario:

* Continued training – While I wasn’t exactly pressed for time with finding satisfactory results, I do believe that continued training would lead to an improved agent. However, due to the level of optimization that the agent has already and due to the nature of converging, it will continue to take longer to train the agent to achieve a noticeable change in behavior. In other words, it would improve regardless, but for the time invested, one would experience diminishing returns in improvement.
* Hyperparameter tweaking - The learning rate can be lowered considerably. This would allow for training that converges to a more optimal solution but takes considerably more time to do so. Additionally, greater buffer sizes, batch sizes, and epoch quantities could lead to more optimal solutions [6].
* Complete agent reconstruction - A different agent could be developed from the ground up for this scenario. There is a plethora of knowledge regarding different methods for reinforcement learning. While I did find Actor-Critic with PPO to be most effective, beating out Q-Learning and Deep SARSA in my findings, there are several other methods that I did little to no testing on. Moreover, machine learning research is burgeoning, to the point that it always seems like there is a new, more effective method with its own body of research to back it up.
* Change optimizers - What would be simpler than rebuilding the agent would be to toy around with different policy optimizers. The one used in this agent is called Adam, which is a commonly used method of policy optimization specifically for stochastic environments. I found it to be effective enough, but there is a lot behind the algorithm that frankly I know nothing about. To that end, I’m not confident that Adam is the best optimizer for the job. In some cases, standard stochastic gradient descent (SGD) is found to be more effective than Adam [11], so changing the optimizer to that would be a good starting point.

My very first initial goal was to create an entire AI that would play the game as a player would through means of reinforcement learning techniques. It was not soon after beginning research that I realized that this was a little too ambitious, specifically after learning that both Facebook and Google had to dedicate a research team of (presumably) very intelligent individuals to make one. However, after my experience with this exercise, I’ve learned a great amount about how agents are constructed, and I may very well continue my efforts to create better agents for *StarCraft* in the future.

# References

|  |  |
| --- | --- |
| [1] | M. Čertický and D. Churchill, "The Current State of StarCraft AI Competitions and Bots," *In Proceedings of the AIIDE 2017 Workshop on Artificial Intelligence for Strategy Games,* 2017. |
| [2] | M. Čertický, D. Churchill, K.-J. Kim, M. Čertický and R. Kelly, "StarCraft AI Competitions, Bots and Tournament Manager Software," *IEEE Transactions on Games (ToG),* vol. 1, no. 13, p. 1, 2018. |
| [3] | D. Churchill, M. Preuss, F. Richoux, G. Synnaeve, A. Uriarte, S. Ontanón and M. Čertický, "Starcraft Bots and Competitions," in *Encyclopedia of Computer Graphics and Games (ECGG)*, Springer International Publishing, 2016. |
| [4] | TeamSAIDA, "SAIDA\_RL," *GitHub repository,* 2019. |
| [5] | V. Mnih et al., "Playing Atari with Deep Reinforcement Learning," *DeepMind Technologies.* |
| [6] | J. Schulman, F. Wolski, P. Dhariwal, A. Radford and O. Klimov, "Proximal Policy Optimization Algorithms," *CoRR,* vol. abs/1707.06347, 2017. |
| [7] | L. Hardesty, "Explained: Neural Networks," *MIT News Office,* 2017. |
| [8] | AurelianTactics, "PPO Hyperparameters and Ranges," 25 July 2018. [Online]. Available: https://medium.com/aureliantactics/ppo-hyperparameters-and-ranges-6fc2d29bccbe. [Accessed 6 April 2020]. |
| [9] | S. John et al., "Trust Region Policy Optimization," *University of California, Berkeley, Department of Electrical Engineering and Computer Sciences,* no. https://arxiv.org/pdf/1502.05477.pdf, 2017. |
| [10] | Keras Team, "Keras: The Python Deep Learning Library," *GitHub repository,* 2019. |
| [11] | N. S. Keskar and R. Socher, "Improving Generalization Performance by Switching from Adam to SGD," *Salesforce Research,* no. https://arxiv.org/pdf/1712.07628.pdf, 2017. |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# Bibliography

|  |
| --- |
| P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. R. A. Plappert, J. Schulman, S. Sidor, Y. Wu and P. Zhokhov, "OpenAI Baselines," *GitHub repository,* 2017. |
| U. Jo, Writer, *SAIDA RL: An Open-source Reinforcement Learning Platform.* [Performance]. Samsung Developers, 2019. |
| A. Juliani, "Simple Reinforcement Learning with Tensorflow Part 0: Q-Learning with Tables and Neural Networks," 25 August 2016. [Online]. Available: https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-0-q-learning-with-tables-and-neural-networks-d195264329d0. [Accessed 3 April 2020]. |
| A. Juliani, "Simple Reinforcement Learning with Tensorflow Part 4: Deep Q-Networks and Beyond," 2 September 2016. [Online]. Available: https://medium.com/@awjuliani/simple-reinforcement-learning-with-tensorflow-part-4-deep-q-networks-and-beyond-8438a3e2b8df. [Accessed 6 April 2020]. |
| T. Matiisen, "Demystifying Deep Reinforcement Learning," 19 December 2015. [Online]. Available: https://neuro.cs.ut.ee/demystifying-deep-reinforcement-learning/. [Accessed 06 April 2020]. |
| V. Mnih, B. Adria Puigdomenech, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver and K. Kavukcuoglu, "Asynchronous Methods for Deep Reinforcement Learning," *CoRR,* vol. abs/1602.01783, 2016. |
| D. Seita, "Actor-Critic Methods: A3C and A2C," 28 June 2018. [Online]. Available: https://danieltakeshi.github.io/2018/06/28/a2c-a3c/. [Accessed 06 April 2020]. |
| O. Vinyals et al., "Grandmaster level in StarCraft II using multi-agent reinforcement learning," *Nature,* vol. 575, pp. 350-354, 2019. |
| C. Yoon, "Understanding Actor Critic Methods and A2C," 5 February 2019. [Online]. Available: https://towardsdatascience.com/understanding-actor-critic-methods-931b97b6df3f. [Accessed 6 April 2020]. |

# Appendix A – Code

## avoid\_reavers\_PPO\_Ben\_v4.py

This was the controller for the agent.

# Updated work from:

# Ben Harruff

# harrbs02@pfw.edu

# 04/27/2020

#

# Original work from:

# Copyright (C) 2019 SAMSUNG SDS <Team.SAIDA@gmail.com>

#

# This code is distribued under the terms and conditions from the MIT

#   License (MIT).

#

# Authors : Uk Jo, Iljoo Yoon, Hyunjae Lee, Daehun Jun

#

# Initial framework taken from

# https://github.com/jaara/AI-blog/blob/master/CartPole-A3C.py

import numpy as np

import os

from datetime import datetime

import math

import argparse

from keras.models import Model

from keras.layers import Input, Dense

from keras.optimizers import Adam

from core.algorithm.PPO import PPOAgent

from core.common.processor import Processor

from saida\_gym.starcraft.avoidReavers import AvoidReavers

from core.callbacks import DrawTrainMovingAvgPlotCallback

import saida\_gym.envs.conn.connection\_env as Config

from core.common.util import \*

# Hyper parameter definitions

# Move angle out of 360 degrees

# For example, if set to 90, then the agent can move

# in 4 directions (one at 0, 90, 180, and 270 relative

# to orgin)

MOVE\_ANG        = 18

# Move distance in terms of tile length

# Each tile is 32 x 32 pixels, so when move distance

# is set to 2, then the agent would move 64 pixels away

# from the origin.

MOVE\_DIST       = 2

# Determine whether to render the environment or not.

# True results in very messy but quicker training

NO\_GUI          = False

# Discount Factor (Usually .9900, range 0.8000 - 0.9997)

GAMMA           = .99

# Number of Epochs (K), aka minibatch updates.

# Policy updates are done with a number of gradient ascents

# known as epochs. This allows more expedient and robust

# training while still being easy enough to implement

# Value depends on Batch size primarily, where larger

# batch sizes warrant more epochs. In atari experiments,

# batch size is 512 and num epochs is 3, so that is reflected

# in this experiment. However,the typical range is 3 - 10.

EPOCHS          = 3

# Total number of steps for training session

NB\_STEPS        = 200000

# Maximum number of steps allowed before episode termination

# The environment MUST end the episode because the training

# will hang for it and won't continue otherwise.

# NB\_MAX\_EPISODE\_STEPS = 100

# Total episodes for training session

NB\_EPISODES     = 1

# Size of state for network

# Value of state size explained later\*\*\*

STATE\_SIZE      = 8 + 3 \* 8

# Only implemented clipping for the surrogate loss

# Paper said it was best

# Commonly denoted as epsilon

LOSS\_CLIPPING   = 0.2

# Learning Rate (3e-3 to 5e-6)

# Determines how much of an impact on learning each gradient

# step has. Greater learning rates mean faster, but possibly

# less robust agents. The more complex the problem, the lower

# the learning rate should probably be.

LR              = 1e-4

# Exploration noise

# \*\*\* Not used in Discrete - value doesn't matter

NOISE           = 0.1

# Entropy Coefficient

# Used to determine how quickly to converge on an optimal

# action given a situation. The smaller the coefficient, the

# slower the rate of randomness reduction for the action

# decision-making process

# Typical range is (0 to 1e-2)

ENTROPY\_LOSS    = 1e-2

# How many steps to perform and collect data from before

# updating the model. Should be a multiple of batch size.

# Larger values lead to slower but more stable updates.

# Typical range is 2,048 to 409,600 for discrete

#BUFFER\_SIZE     = 256

BUFFER\_SIZE     = 51200

# Number of steps before next iteration of gradient descent

# Should always be multiple of buffer size.

# Typical range is 32 to 512 for discrete

#BATCH\_SIZE      = 64

BATCH\_SIZE      = 512

# Size of hidden layers in the neural network.

# More complex problems demand larger hidden layer sizes

# Typical range is 32 for a straightforward problem and

# as high as 512 for very complex problems.

#HIDDEN\_SIZE     = 80

HIDDEN\_SIZE     = 128

# Number of hidden layers present after observation input

# More layers are necessary for more complex problems

# Typical range is 1 to 3

NUM\_LAYERS      = 3

# Argument Parser

# Hyper Parameter definitions

parser = argparse.ArgumentParser(description='PPO Configuration for Avoid\_Reaver')

parser.add\_argument(OPS.N\_STEPS.value,      help='nbsteps',  default=NB\_STEPS, type=int)

parser.add\_argument(OPS.LEARNING\_RATE.value,help='lr',  default=LR, type=float)

parser.add\_argument(OPS.GAMMA.value,        help='gam', default=GAMMA, type=float)

parser.add\_argument(OPS.EPOCHS.value,       help='epo', default=EPOCHS, type=int)

parser.add\_argument(OPS.MOVE\_ANG.value,     help='ang', default=MOVE\_ANG, type=int)

parser.add\_argument(OPS.MOVE\_DIST.value,    help='dst', default=MOVE\_DIST, type=int)

args = parser.parse\_args()

# Populate post\_fix with hyper-parameter definitions for file names when saving models and graphs.

# Used to diffentiate results from one another based on the parameters used.

# Example: 20200424220350\_

dict\_args = vars(args)

post\_fix = '\_' + yyyymmdd24hhmmss()

for k in dict\_args.keys():

    post\_fix += '\_' + k + '\_' + str(dict\_args[k])

def scale\_velocity(v):

    return v

def scale\_angle(angle):

    return (angle - math.pi) / math.pi

# Scale position

def scale\_pos(pos):

    return pos / 16

def scale\_pos2(pos):

    return pos / 8

def exponential\_average(old, new, b1):

    return old \* b1 + (1-b1) \* new

# Reshape the reward in a way you want

def reward\_reshape(reward):

    """ Reshape the reward

        Starcraft Env returns the reward according to following conditions.

        1. Invalid action : -0.1

        2. get hit : -1

        3. goal : 1

        4. farther : 0

        5. closer : 2

    # Argument

        reward (float): The observed reward after executing the action

    # Returns

        reshaped reward

        1. Invalid action : -1

        2. get hit : -5

        3. goal : 5

        4. farther : -0.1

        5. closer : 0.1

    """

    if math.fabs(reward + 0.1) < 0.01:

        reward = -1

    elif reward == -1:

        reward = -5

    elif reward == 1:

        reward = 5

    elif reward == 0:

        reward = -0.1

    elif reward == 2:

        reward = 0.1

    return reward

# Define's the agent's Processor

class ReaverProcessor(Processor):

    def \_\_init\_\_(self):

        self.last\_action = None

        self.success\_cnt = 0

        self.cumulate\_reward = 0

    def process\_action(self, action):

        self.last\_action = action

        return action

    def process\_step(self, observation, reward, done, info):

        state\_array = self.process\_observation(observation)

        reward = reward\_reshape(reward)

        self.cumulate\_reward += reward

        if reward == 10:

            if self.cumulate\_reward > 0:

                self.success\_cnt += 1

            self.cumulate\_reward = 0

            print("success\_cnt = ", self.success\_cnt)

        return state\_array, reward, done, info

    def process\_observation(self, observation, \*\*kwargs):

        """ Pre-process observation

        # Argument

            observation (object): The current observation from the environment.

        # Returns

            processed observation

        """

        if len(observation.my\_unit) > 0:

            s = np.zeros(STATE\_SIZE)

            me = observation.my\_unit[0]

            # Observation for Dropship

            s[0] = scale\_pos2(me.pos\_x)             # X of coordinates

            s[1] = scale\_pos2(me.pos\_y)             # Y of coordinates

            s[2] = scale\_pos2(me.pos\_x - 320)       # relative X of coordinates from goal

            s[3] = scale\_pos2(me.pos\_y - 320)       # relative Y of coordinates from goal

            s[4] = scale\_velocity(me.velocity\_x)    # X of velocity

            s[5] = scale\_velocity(me.velocity\_y)    # Y of velocity

            s[6] = scale\_angle(me.angle)            # Angle of head of Dropship

            s[7] = 1 if me.accelerating else 0      # True if Dropship is accelerating

            # Observation for Reavers

            for ind, ob in enumerate(observation.en\_unit):

                s[ind \* 8 + 8] = scale\_pos2(ob.pos\_x - me.pos\_x)    # X of coordinates relative to dropship

                s[ind \* 8 + 9] = scale\_pos2(ob.pos\_y - me.pos\_y)    # Y of coordinates relative to dropship

                s[ind \* 8 + 10] = scale\_pos2(ob.pos\_x - 320)        # X of relative coordinates from goal

                s[ind \* 8 + 11] = scale\_pos2(ob.pos\_y - 320)        # Y of relative coordinates from goal

                s[ind \* 8 + 12] = scale\_velocity(ob.velocity\_x)     # X of velocity

                s[ind \* 8 + 13] = scale\_velocity(ob.velocity\_y)     # Y of velocity

                s[ind \* 8 + 14] = scale\_angle(ob.angle)             # Angle of head of Reavers

                s[ind \* 8 + 15] = 1 if ob.accelerating else 0       # True if Reaver is accelerating

        return s

def build\_actor(state\_size, action\_size, advantage, old\_prediction):

    state\_input = Input(shape=(state\_size,))

    x = Dense(HIDDEN\_SIZE, activation='tanh')(state\_input)

    for \_ in range(NUM\_LAYERS - 1):

        x = Dense(HIDDEN\_SIZE, activation='tanh')(x)

    out\_actions = Dense(action\_size, activation='softmax', name='output')(x)

    model = Model(inputs=[state\_input, advantage, old\_prediction], outputs=[out\_actions])

    return model

# Not used for discrete environments\*\*\*

def build\_actor\_continuous(state\_size, action\_size, advantage, old\_prediction):

    state\_input = Input(shape=(state\_size,))

    x = Dense(HIDDEN\_SIZE, activation='tanh')(state\_input)

    for \_ in range(NUM\_LAYERS - 1):

        x = Dense(HIDDEN\_SIZE, activation='tanh')(x)

    out\_actions = Dense(action\_size, name='output', activation='tanh')(x)

    model = Model(inputs=[state\_input, advantage, old\_prediction], outputs=[out\_actions])

    return model

def build\_critic(state\_size):

    state\_input = Input(shape=(state\_size,))

    x = Dense(HIDDEN\_SIZE, activation='tanh')(state\_input)

    for \_ in range(NUM\_LAYERS - 1):

        x = Dense(HIDDEN\_SIZE, activation='tanh')(x)

    out\_value = Dense(1)(x)

    model = Model(inputs=[state\_input], outputs=[out\_value])

    return model

# Main method

if \_\_name\_\_ == '\_\_main\_\_':

    training\_mode = True

    load\_model = True

    FILE\_NAME = os.path.basename(\_\_file\_\_).split('.')[0] + "-" + datetime.now().strftime("%m%d%H%M%S")

    action\_type = 0

    env = AvoidReavers(move\_angle=dict\_args[OPS.MOVE\_ANG()], move\_dist=dict\_args[OPS.MOVE\_DIST()], frames\_per\_step=12

                       , verbose=0, action\_type=action\_type, no\_gui=NO\_GUI)

    ACTION\_SIZE = env.action\_space.n

    continuous = False if action\_type == 0 else True

    # Build models

    actor = None

    ADVANTAGE = Input(shape=(1,))

    OLD\_PREDICTION = Input(shape=(ACTION\_SIZE,))

    if continuous:

        actor = build\_actor\_continuous(STATE\_SIZE, ACTION\_SIZE, ADVANTAGE, OLD\_PREDICTION)

    else:

        actor = build\_actor(STATE\_SIZE, ACTION\_SIZE, ADVANTAGE, OLD\_PREDICTION)

    critic = build\_critic(STATE\_SIZE)

    agent = PPOAgent(STATE\_SIZE, ACTION\_SIZE, continuous, actor, critic, GAMMA, LOSS\_CLIPPING, EPOCHS, NOISE, ENTROPY\_LOSS,

                     BUFFER\_SIZE,BATCH\_SIZE, processor=ReaverProcessor())

    # Compile the agent's actor and critic with their own Adam optimizer and metrics

    agent.compile(optimizer=[Adam(lr=LR), Adam(lr=LR)], metrics=[ADVANTAGE, OLD\_PREDICTION])

    if load\_model == True:

        agent.load\_weights(os.path.realpath("C:/Senior\_Project\_Repository/python/saida\_agent\_example/avoidReaver/save\_model/"))

    cb\_plot = DrawTrainMovingAvgPlotCallback(os.path.realpath("C:/Senior\_Project\_Repository/python/saida\_agent\_example/avoidReaver/save\_graph/" + FILE\_NAME + '\_'+post\_fix + '.png'), 50, 5, l\_label=['episode\_reward'])

    agent.run(env, NB\_STEPS, train\_mode=training\_mode, verbose=1, callbacks=[cb\_plot], action\_repetition=1, nb\_episodes=NB\_EPISODES)

    if training\_mode == True:

        agent.save\_weights(os.path.realpath("C:/Senior\_Project\_Repository/python/saida\_agent\_example/avoidReaver/save\_model/"),"avoid\_Reavers\_PPO"+post\_fix)

    env.close()

## PPO.py

This is the algorithm working behind the scenes to train the agent.

# Additional documentation and some edits from:

# Ben Harruff

# harrbs02@pfw.edu

# 04/27/2020

#

# Copyright (C) 2019 SAMSUNG SDS <Team.SAIDA@gmail.com>

#

# This code is distribued under the terms and conditions from the MIT License (MIT).

#

# Authors : Uk Jo, Iljoo Yoon, Hyunjae Lee, Daehun Jun

from core.common.agent import Agent

from core.common.util import \*

import math

class PPOAgent(Agent):

    def \_\_init\_\_(self, state\_size, action\_size, continuous, actor, critic,

                 discount\_factor=0.99, loss\_clipping=0.2, epochs=10, noise=1.0, entropy\_loss=1e-3,

                 buffer\_size=256, batch\_size=64, \*\*kwargs):

        """

            Constructor for PPO with clipped loss

         #Arguments

            state\_size(integer): Number of state size

            action\_size(integer): Number of action space

            continuous(bool): True if action space is continuous type

            actor(Keras Model): network for actor

            critic(Keras Model): network for critic

            discount\_factor(float): discount reward factor

            loss\_clipping(float): hyper parameter for loss clipping, in PPO paper, 0.2 is recommended.

            epochs(integer): hyper parameter

            noise(float): hyper parameter

            entropy\_loss : hyper parameter

            buffer\_size : hyper parameter

            batch\_size : hyper parameter

        #Return

            None

        """

        super(PPOAgent, self).\_\_init\_\_(\*\*kwargs)

        self.action\_size = action\_size

        self.state\_szie = state\_size

        self.continuous = continuous

        self.critic = critic

        self.actor = actor

        self.episode = 0

        self.discount\_factor = discount\_factor

        self.loss\_clipping = loss\_clipping  # Only implemented clipping for the surrogate loss, paper said it was best

        self.epochs = epochs

        self.noise = noise  # Exploration noise

        self.entropy\_loss = entropy\_loss

        self.buffer\_size = buffer\_size

        self.batch\_size = batch\_size

        self.dummy\_action,self.dummy\_value = np.zeros((1, action\_size)), np.zeros((1, 1))

        self.observation = None

        self.reward = []

        self.reward\_over\_time = []

        self.gradient\_steps = 0

        self.batch = [[], [], [], []]

        self.tmp\_batch = [[], [], []]

    def reset\_env(self):

        self.reward = []

    def discounted\_reward(self):

        for j in range(len(self.reward) - 2, -1, -1):

            self.reward[j] += self.reward[j + 1] \* self.discount\_factor

    def forward(self, observation):

        self.observation = observation

        self.tmp\_batch[0].append(observation)

        if self.continuous:

            p = self.actor.predict([self.observation.reshape(1, self.state\_szie), self.dummy\_value, self.dummy\_action])

            action = action\_matrix = p[0] + np.random.normal(loc=0, scale=self.noise, size=p[0].shape)

            self.tmp\_batch[1].append(action\_matrix)

            self.tmp\_batch[2].append(p)

            return action, action\_matrix, p

        else:

            state = np.reshape(self.observation, [1, self.state\_szie])

            p = self.actor.predict([state, self.dummy\_value, self.dummy\_action])

            action = np.random.choice(self.action\_size, p=np.nan\_to\_num(p[0]))

            action\_matrix = np.zeros(self.action\_size)

            action\_matrix[action] = 1

            self.tmp\_batch[1].append(action\_matrix)

            self.tmp\_batch[2].append(p)

            return [action]

    def backward(self, reward, terminal):

        if self.train\_mode is False:

            return

        self.reward.append(reward)

        if terminal:

            self.discounted\_reward()

            for i in range(len(self.tmp\_batch[0])):

                obs, action, pred = self.tmp\_batch[0][i], self.tmp\_batch[1][i], self.tmp\_batch[2][i]

                r = self.reward[i]

                self.batch[0].append(obs)

                self.batch[1].append(action)

                self.batch[2].append(pred)

                self.batch[3].append(r)

            self.tmp\_batch = [[], [], []]

            self.reset\_env()

        # When we've completed the last batch of the experience buffer, then...

        if len(self.batch[0]) >= self.buffer\_size:

            obs, action, pred, reward = np.array(self.batch[0]), np.array(self.batch[1]), np.array(self.batch[2]), np.reshape(

                np.array(self.batch[3]), (len(self.batch[3]), 1))

            self.batch = [[], [], [], []]

            pred = np.reshape(pred, (pred.shape[0], pred.shape[2]))

            obs, action, pred, reward = obs[:self.buffer\_size], action[:self.buffer\_size], \

                                        pred[:self.buffer\_size], reward[:self.buffer\_size]

            # Determine what the critic thinks the actor should do given the observation

            pred\_values = self.critic.predict(obs)

            # Update the stochastic gradient

            self.gradient\_steps += 1

            # Calculate Advantage

            advantage = reward - pred\_values

            # Make the current prediction the old prediction

            old\_prediction = pred

            advantage = (advantage - advantage.mean()) / advantage.std()

            # Calculate Actor loss

            # Actor determines what action to take using the observations, the advantage, and the old predictions

            actor\_loss = self.actor.fit([obs, advantage, old\_prediction], [action], batch\_size=self.batch\_size,

                                        shuffle=True, epochs=self.epochs, verbose=False)

            # Calculate Critic loss

            #

            # Critic determines reward of actor's action

            critic\_loss = self.critic.fit([obs], [reward], batch\_size=self.batch\_size, shuffle=True, epochs=self.epochs,

                                          verbose=False)

    def compile(self, optimizer, metrics=[]):

        """

        # Argument

            optimizer (object) : [0] = actor optimizer, [1] = critic optimizer

            metrics (Tensor) :  [0] = Keras Tensor as an advantage , [1] = Keras Tensor as an old\_prediction

        # Return

            None

        """

        # Compile actor model

        advantage = metrics[0]

        old\_prediction = metrics[1]

        if self.continuous:

            self.actor.compile(optimizer=optimizer[0],

                               loss=[self.proximal\_policy\_optimization\_loss\_continuous(

                                   advantage=advantage, old\_prediction=old\_prediction)])

        else:

            self.actor.compile(optimizer=optimizer[0],

                               loss=[self.proximal\_policy\_optimization\_loss(

                                   advantage=advantage, old\_prediction=old\_prediction)])

        self.actor.summary()

        #Compile critic model

        self.critic.compile(optimizer=optimizer[1], loss='mse')

        self.compiled=True

    def proximal\_policy\_optimization\_loss(self, advantage, old\_prediction):

        def loss(y\_true, y\_pred):

            prob = y\_true \* y\_pred

            old\_prob = y\_true \* old\_prediction

            r = prob / (old\_prob + 1e-10)

            # Equation 7 from PPO paper with the addition of entropy bonus to ensure proper exploration

            return -K.mean(K.minimum(r \* advantage, K.clip(r, min\_value=1 - self.loss\_clipping,

                                                           max\_value=1 + self.loss\_clipping) \* advantage) + self.entropy\_loss \* (

                                   prob \* K.log(prob + 1e-10)))

        return loss

    def proximal\_policy\_optimization\_loss\_continuous(self, advantage, old\_prediction):

        def loss(y\_true, y\_pred):

            var = K.square(self.noise)

            pi = math.pi

            denom = K.sqrt(2 \* pi \* var)

            prob\_num = K.exp(- K.square(y\_true - y\_pred) / (2 \* var))

            old\_prob\_num = K.exp(- K.square(y\_true - old\_prediction) / (2 \* var))

            prob = prob\_num / denom

            old\_prob = old\_prob\_num / denom

            r = prob / (old\_prob + 1e-10)

            return -K.mean(K.minimum(r \* advantage,

                                     K.clip(r, min\_value=1 - self.loss\_clipping,

                                            max\_value=1 + self.loss\_clipping) \* advantage))

        return loss

    def load\_weights(self, filepath):

        algorithm\_critic\_folder = 'critic'

        algorithm\_actor\_folder = 'actor'

        critic\_filepath = filepath + os.path.sep + algorithm\_critic\_folder + os.path.sep

            + 'avoid\_Reavers\_PPO\_20200427184511\_nsteps\_200000\_lr\_0.0001\_gamma\_0.99\_epo\_3\_move\_a\_18\_move\_d\_2\_critic\_20200427200020.h5f'

        actor\_filepath = filepath + os.path.sep + algorithm\_actor\_folder + os.path.sep

            + 'avoid\_Reavers\_PPO\_20200427184511\_nsteps\_200000\_lr\_0.0001\_gamma\_0.99\_epo\_3\_move\_a\_18\_move\_d\_2\_actor\_20200427200020.h5f'

        self.actor.load\_weights(actor\_filepath)

        self.critic.load\_weights(critic\_filepath)

    def save\_weights(self, filepath, filename=None, overwrite=False):

        algorithm\_critic\_name = '\_critic\_'

        algorithm\_actor\_name = '\_actor\_'

        algorithm\_critic\_folder = 'critic'

        algorithm\_actor\_folder = 'actor'

        # 03/05/2020 - added 'critic' and 'actor' to seperate .h5f files

        critic\_filepath = filepath + os.path.sep + algorithm\_critic\_folder

            + os.path.sep + filename + algorithm\_critic\_name + yyyymmdd24hhmmss() + '.' + 'h5f'

        actor\_filepath = filepath + os.path.sep + algorithm\_actor\_folder

            + os.path.sep + filename + algorithm\_actor\_name + yyyymmdd24hhmmss() + '.' + 'h5f'

        self.critic.save\_weights(critic\_filepath, overwrite)

        print('{} file is saved as a critic model for evaluation'.format(critic\_filepath))

        self.actor.save\_weights(actor\_filepath, overwrite)

        print('{} file is saved as a actor model for evaluation'.format(actor\_filepath))

        return [actor\_filepath, critic\_filepath]

# Appendix B – Tables

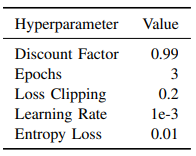


Table 1: PPO Hyperparameters used to train agent.

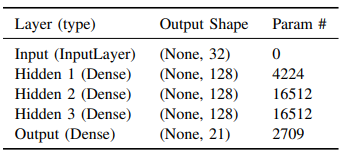


Table 2: Feed-Forward Neural Network information.

# Appendix C – Training Data

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | -0.7322 |  | 51 | 1.8038 |  | 101 | 4.4083 |  | 151 | 5.5671 |
| 2 | -0.7442 |  | 52 | 1.9511 |  | 102 | 4.3994 |  | 152 | 5.6018 |
| 3 | -0.8130 |  | 53 | 1.8663 |  | 103 | 4.3983 |  | 153 | 5.6427 |
| 4 | -0.7774 |  | 54 | 1.9847 |  | 104 | 4.4686 |  | 154 | 5.6320 |
| 5 | -0.7470 |  | 55 | 1.9648 |  | 105 | 4.4259 |  | 155 | 5.6837 |
| 6 | -0.6922 |  | 56 | 2.0284 |  | 106 | 4.6409 |  | 156 | 5.6590 |
| 7 | -0.6171 |  | 57 | 2.4039 |  | 107 | 4.6729 |  | 157 | 5.6391 |
| 8 | -0.5830 |  | 58 | 2.4683 |  | 108 | 4.7650 |  | 158 | 5.6979 |
| 9 | -0.6980 |  | 59 | 2.4060 |  | 109 | 4.6887 |  | 159 | 5.6921 |
| 10 | -0.6568 |  | 60 | 2.4184 |  | 110 | 4.7347 |  | 160 | 5.6988 |
| 11 | -0.6977 |  | 61 | 2.4775 |  | 111 | 4.9104 |  | 161 | 5.7192 |
| 12 | -0.5357 |  | 62 | 2.6411 |  | 112 | 4.8480 |  | 162 | 5.6863 |
| 13 | -0.5774 |  | 63 | 2.7257 |  | 113 | 4.8270 |  | 163 | 5.7481 |
| 14 | -0.4846 |  | 64 | 2.7329 |  | 114 | 4.8667 |  | 164 | 5.7180 |
| 15 | -0.5500 |  | 65 | 2.8212 |  | 115 | 4.8993 |  | 165 | 5.8026 |
| 16 | -0.5440 |  | 66 | 2.7665 |  | 116 | 5.0068 |  | 166 | 5.7121 |
| 17 | -0.5547 |  | 67 | 3.1987 |  | 117 | 5.0315 |  | 167 | 5.7106 |
| 18 | -0.5591 |  | 68 | 3.1629 |  | 118 | 5.0070 |  | 168 | 5.6985 |
| 19 | -0.4697 |  | 69 | 3.3268 |  | 119 | 4.9886 |  | 169 | 5.7812 |
| 20 | -0.4791 |  | 70 | 3.1576 |  | 120 | 5.0270 |  | 170 | 5.8199 |
| 21 | -0.3669 |  | 71 | 3.2752 |  | 121 | 5.1461 |  | 171 | 5.8094 |
| 22 | -0.4318 |  | 72 | 3.4585 |  | 122 | 5.1457 |  | 172 | 5.7701 |
| 23 | -0.3594 |  | 73 | 3.4995 |  | 123 | 5.1442 |  | 173 | 5.8038 |
| 24 | -0.3605 |  | 74 | 3.4224 |  | 124 | 5.1345 |  | 174 | 5.8162 |
| 25 | -0.3833 |  | 75 | 3.4968 |  | 125 | 5.1787 |  | 175 | 5.7588 |
| 26 | -0.1898 |  | 76 | 3.5430 |  | 126 | 5.2312 |  | 176 | 5.7631 |
| 27 | -0.0383 |  | 77 | 3.7123 |  | 127 | 5.1930 |  | 177 | 5.8191 |
| 28 | -0.0909 |  | 78 | 3.7362 |  | 128 | 5.1969 |  | 178 | 5.8374 |
| 29 | -0.0168 |  | 79 | 3.7346 |  | 129 | 5.2951 |  | 179 | 5.8351 |
| 30 | 0.0139 |  | 80 | 3.7692 |  | 130 | 5.3496 |  | 180 | 5.8408 |
| 31 | 0.0667 |  | 81 | 3.6987 |  | 131 | 5.3272 |  | 181 | 5.8479 |
| 32 | 0.4196 |  | 82 | 3.9067 |  | 132 | 5.3226 |  | 182 | 5.8398 |
| 33 | 0.5902 |  | 83 | 3.8678 |  | 133 | 5.3259 |  | 183 | 5.8551 |
| 34 | 0.3926 |  | 84 | 3.8334 |  | 134 | 5.3118 |  | 184 | 5.9027 |
| 35 | 0.4560 |  | 85 | 3.8483 |  | 135 | 5.4004 |  | 185 | 5.8701 |
| 36 | 0.5723 |  | 86 | 3.9106 |  | 136 | 5.4104 |  | 186 | 5.9005 |
| 37 | 0.7332 |  | 87 | 4.1274 |  | 137 | 5.3592 |  | 187 | 5.8531 |
| 38 | 0.7522 |  | 88 | 4.0026 |  | 138 | 5.4351 |  | 188 | 5.8907 |
| 39 | 0.8585 |  | 89 | 4.0797 |  | 139 | 5.4098 |  | 189 | 5.9376 |
| 40 | 0.7911 |  | 90 | 4.1808 |  | 140 | 5.4704 |  | 190 | 5.9504 |
| 41 | 0.8665 |  | 91 | 4.1760 |  | 141 | 5.5319 |  | 191 | 5.9620 |
| 42 | 1.2980 |  | 92 | 4.2983 |  | 142 | 5.5369 |  | 192 | 5.9384 |
| 43 | 1.3955 |  | 93 | 4.2404 |  | 143 | 5.5423 |  | 193 | 5.9062 |
| 44 | 1.4565 |  | 94 | 4.2915 |  | 144 | 5.5305 |  | 194 | 5.9371 |
| 45 | 1.2891 |  | 95 | 4.2186 |  | 145 | 5.5743 |  | 195 | 5.9465 |
| 46 | 1.3524 |  | 96 | 4.3129 |  | 146 | 5.5891 |  | 196 | 5.9715 |
| 47 | 1.7651 |  | 97 | 4.4308 |  | 147 | 5.5574 |  | 197 | 5.9737 |
| 48 | 1.5509 |  | 98 | 4.4078 |  | 148 | 5.5475 |  | 198 | 5.9525 |
| 49 | 1.5772 |  | 99 | 4.4258 |  | 149 | 5.5764 |  | 199 | 5.9774 |
| 50 | 1.7331 |  | 100 | 4.3688 |  | 150 | 5.6299 |  | 200 | 5.9128 |

Average Episode Score of Training Intervals 201-400 (Timesteps 2,010,000 – 4,000,000)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 201 | 5.9264 |  | 251 | 6.0735 |  | 301 | 6.2028 |  | 351 | 6.3013 |
| 202 | 5.9492 |  | 252 | 6.1179 |  | 302 | 6.2209 |  | 352 | 6.2873 |
| 203 | 5.9423 |  | 253 | 6.1144 |  | 303 | 6.1797 |  | 353 | 6.3005 |
| 204 | 5.9550 |  | 254 | 6.0967 |  | 304 | 6.2223 |  | 354 | 6.2909 |
| 205 | 5.9355 |  | 255 | 6.1103 |  | 305 | 6.1841 |  | 355 | 6.3092 |
| 206 | 5.9884 |  | 256 | 6.0974 |  | 306 | 6.1883 |  | 356 | 6.3301 |
| 207 | 5.9542 |  | 257 | 6.1451 |  | 307 | 6.1985 |  | 357 | 6.2847 |
| 208 | 5.9766 |  | 258 | 6.1382 |  | 308 | 6.2173 |  | 358 | 6.3074 |
| 209 | 5.9603 |  | 259 | 6.0765 |  | 309 | 6.2325 |  | 359 | 6.3237 |
| 210 | 6.0148 |  | 260 | 6.1254 |  | 310 | 6.1531 |  | 360 | 6.2804 |
| 211 | 6.0132 |  | 261 | 6.1329 |  | 311 | 6.2172 |  | 361 | 6.3226 |
| 212 | 6.0122 |  | 262 | 6.1322 |  | 312 | 6.2460 |  | 362 | 6.2955 |
| 213 | 5.9648 |  | 263 | 6.1716 |  | 313 | 6.2139 |  | 363 | 6.3117 |
| 214 | 6.0174 |  | 264 | 6.0879 |  | 314 | 6.2156 |  | 364 | 6.3175 |
| 215 | 5.9958 |  | 265 | 6.1187 |  | 315 | 6.1887 |  | 365 | 6.2675 |
| 216 | 6.0363 |  | 266 | 6.1292 |  | 316 | 6.1592 |  | 366 | 6.3154 |
| 217 | 6.0113 |  | 267 | 6.0883 |  | 317 | 6.2130 |  | 367 | 6.3401 |
| 218 | 6.0543 |  | 268 | 6.0948 |  | 318 | 6.2257 |  | 368 | 6.2528 |
| 219 | 6.0549 |  | 269 | 6.1345 |  | 319 | 6.2278 |  | 369 | 6.3005 |
| 220 | 5.9968 |  | 270 | 6.1258 |  | 320 | 6.2105 |  | 370 | 6.2833 |
| 221 | 6.0178 |  | 271 | 6.0947 |  | 321 | 6.1987 |  | 371 | 6.2836 |
| 222 | 6.0535 |  | 272 | 6.1214 |  | 322 | 6.2075 |  | 372 | 6.2655 |
| 223 | 6.0155 |  | 273 | 6.0957 |  | 323 | 6.2358 |  | 373 | 6.3181 |
| 224 | 6.0493 |  | 274 | 6.1652 |  | 324 | 6.1929 |  | 374 | 6.2592 |
| 225 | 6.0587 |  | 275 | 6.1672 |  | 325 | 6.2085 |  | 375 | 6.3080 |
| 226 | 6.0369 |  | 276 | 6.1293 |  | 326 | 6.2545 |  | 376 | 6.3118 |
| 227 | 6.0802 |  | 277 | 6.1308 |  | 327 | 6.2606 |  | 377 | 6.3150 |
| 228 | 6.0862 |  | 278 | 6.1387 |  | 328 | 6.2658 |  | 378 | 6.3245 |
| 229 | 6.0901 |  | 279 | 6.1067 |  | 329 | 6.2245 |  | 379 | 6.2948 |
| 230 | 6.0549 |  | 280 | 6.1374 |  | 330 | 6.2979 |  | 380 | 6.3055 |
| 231 | 6.0847 |  | 281 | 6.1312 |  | 331 | 6.2622 |  | 381 | 6.2865 |
| 232 | 6.0622 |  | 282 | 6.1088 |  | 332 | 6.2927 |  | 382 | 6.3177 |
| 233 | 6.0618 |  | 283 | 6.1768 |  | 333 | 6.2296 |  | 383 | 6.3660 |
| 234 | 6.0326 |  | 284 | 6.1261 |  | 334 | 6.3148 |  | 384 | 6.3144 |
| 235 | 6.0675 |  | 285 | 6.1591 |  | 335 | 6.2513 |  | 385 | 6.3391 |
| 236 | 6.0579 |  | 286 | 6.1222 |  | 336 | 6.2735 |  | 386 | 6.3439 |
| 237 | 6.0172 |  | 287 | 6.0889 |  | 337 | 6.2813 |  | 387 | 6.3994 |
| 238 | 6.0488 |  | 288 | 6.0975 |  | 338 | 6.2525 |  | 388 | 6.3378 |
| 239 | 6.0561 |  | 289 | 6.1527 |  | 339 | 6.2672 |  | 389 | 6.3448 |
| 240 | 6.0390 |  | 290 | 6.1660 |  | 340 | 6.2840 |  | 390 | 6.3263 |
| 241 | 6.0890 |  | 291 | 6.1521 |  | 341 | 6.2804 |  | 391 | 6.3217 |
| 242 | 6.0584 |  | 292 | 6.1602 |  | 342 | 6.2792 |  | 392 | 6.3511 |
| 243 | 6.0907 |  | 293 | 6.1622 |  | 343 | 6.2516 |  | 393 | 6.2989 |
| 244 | 6.0938 |  | 294 | 6.1845 |  | 344 | 6.2105 |  | 394 | 6.3060 |
| 245 | 6.0676 |  | 295 | 6.2133 |  | 345 | 6.2416 |  | 395 | 6.3564 |
| 246 | 6.0443 |  | 296 | 6.1723 |  | 346 | 6.2786 |  | 396 | 6.3334 |
| 247 | 6.0661 |  | 297 | 6.1680 |  | 347 | 6.2724 |  | 397 | 6.3843 |
| 248 | 6.1200 |  | 298 | 6.1679 |  | 348 | 6.2887 |  | 398 | 6.3256 |
| 249 | 6.0531 |  | 299 | 6.2007 |  | 349 | 6.2707 |  | 399 | 6.3213 |
| 250 | 6.1394 |  | 300 | 6.1786 |  | 350 | 6.3137 |  | 400 | 6.3660 |

# Appendix D – Final Results

Graph of Results

Terminal Output from Visual Studio Code

PS C:\Users\harru> ${env:DEBUGPY\_LAUNCHER\_PORT}='31619'; & 'C:\Users\harru\AppData\Local\Programs\Python\Python36\python.exe' 'c:\Users\harru\.vscode\extensions\ms-python.python-2020.3.71659\pythonFiles\lib\python\debugpy\no\_wheels\debugpy\launcher' 'c:\Senior\_Project\_Repository\python\saida\_agent\_example\avoidReaver\avoid\_reavers\_PPO\_Ben\_v4.py'

Using TensorFlow backend.

C:\Users\harru\AppData\Local\Programs\Python\Python36\lib\site-packages\tensorflow\python\framework\dtypes.py:526: FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future version

of numpy, it will be understood as (type, (1,)) / '(1,)type'.

\_np\_qint8 = np.dtype([("qint8", np.int8, 1)])

C:\Users\harru\AppData\Local\Programs\Python\Python36\lib\site-packages\tensorflow\python\framework\dtypes.py:527: FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future version

of numpy, it will be understood as (type, (1,)) / '(1,)type'.

\_np\_quint8 = np.dtype([("quint8", np.uint8, 1)])

C:\Users\harru\AppData\Local\Programs\Python\Python36\lib\site-packages\tensorflow\python\framework\dtypes.py:528: FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future version

of numpy, it will be understood as (type, (1,)) / '(1,)type'.

\_np\_qint16 = np.dtype([("qint16", np.int16, 1)])

C:\Users\harru\AppData\Local\Programs\Python\Python36\lib\site-packages\tensorflow\python\framework\dtypes.py:529: FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future version

of numpy, it will be understood as (type, (1,)) / '(1,)type'.

\_np\_quint16 = np.dtype([("quint16", np.uint16, 1)])

C:\Users\harru\AppData\Local\Programs\Python\Python36\lib\site-packages\tensorflow\python\framework\dtypes.py:530: FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future version

of numpy, it will be understood as (type, (1,)) / '(1,)type'.

\_np\_qint32 = np.dtype([("qint32", np.int32, 1)])

C:\Users\harru\AppData\Local\Programs\Python\Python36\lib\site-packages\tensorflow\python\framework\dtypes.py:535: FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a future version

of numpy, it will be understood as (type, (1,)) / '(1,)type'.

np\_resource = np.dtype([("resource", np.ubyte, 1)])

Initialize...

Shared Memory create

SAIDA\_AR1036 Shared memory found.

WARNING:tensorflow:From C:\Users\harru\AppData\Local\Programs\Python\Python36\lib\site-packages\tensorflow\python\framework\op\_def\_library.py:263: colocate\_with (from tensorflow.python.framework.ops) is deprecated and will be removed in a future version.

Instructions for updating:

Colocations handled automatically by placer.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

input\_3 (InputLayer) (None, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1 (Dense) (None, 128) 4224

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_2 (Dense) (None, 128) 16512

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_3 (Dense) (None, 128) 16512

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

output (Dense) (None, 21) 2709

=================================================================

Total params: 39,957

Trainable params: 39,957

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

2020-04-27 21:50:30.291367: I tensorflow/core/platform/cpu\_feature\_guard.cc:141] Your CPU supports instructions that this TensorFlow binary was not compiled to use: AVX2

Starting training for 100000 steps ...

Training for 100000 steps ...

Interval 1 (0 steps performed)

10000/10000 [==============================] - 270s 27ms/step - reward: 6.3834

679 episodes - episode\_reward: 94.012 [59.300, 98.800]

Interval 2 (10000 steps performed)

10000/10000 [==============================] - 267s 27ms/step - reward: 6.3315

680 episodes - episode\_reward: 93.109 [54.500, 98.800]

Interval 3 (20000 steps performed)

10000/10000 [==============================] - 272s 27ms/step - reward: 6.3740

681 episodes - episode\_reward: 93.598 [64.200, 98.800]

Interval 4 (30000 steps performed)

10000/10000 [==============================] - 270s 27ms/step - reward: 6.3433

680 episodes - episode\_reward: 93.283 [64.300, 98.800]

Interval 5 (40000 steps performed)

7946/10000 [======================>.......] - ETA: 57s - reward: 6.3458WARNING:tensorflow:From C:\Users\harru\AppData\Local\Programs\Python\Python36\lib\site-packages\tensorflow\python\ops\math\_ops.py:3066: to\_int32 (from tensorflow.python.ops.math\_ops) is deprecated and will be removed in a future version.

Instructions for updating:

Use tf.cast instead.

10000/10000 [==============================] - 284s 28ms/step - reward: 6.3545

681 episodes - episode\_reward: 93.310 [64.200, 98.800]

Interval 6 (50000 steps performed)

10000/10000 [==============================] - 281s 28ms/step - reward: 6.3439

678 episodes - episode\_reward: 93.568 [59.600, 98.800]

Interval 7 (60000 steps performed)

10000/10000 [==============================] - 281s 28ms/step - reward: 6.2927

677 episodes - episode\_reward: 92.950 [54.500, 98.800]

Interval 8 (70000 steps performed)

10000/10000 [==============================] - 290s 29ms/step - reward: 6.3477

679 episodes - episode\_reward: 93.486 [54.600, 98.800]

Interval 9 (80000 steps performed)

10000/10000 [==============================] - 336s 34ms/step - reward: 6.3566

679 episodes - episode\_reward: 93.617 [54.400, 98.800]

Interval 10 (90000 steps performed)

10000/10000 [==============================] - 343s 34ms/step - reward: 6.3705

Training took 2893.2739730000003 seconds

done, took 2893.274 seconds

C:\Senior\_Project\_Repository\python\saida\_agent\_example\avoidReaver\save\_model\critic\avoid\_Reavers\_PPO\_20200427215025\_nsteps\_100000\_lr\_0.0001\_gamma\_0.99\_epo\_3\_move\_a\_18\_move\_d\_2\_critic\_20200427223843.h5f file is saved as a critic model for evaluation

C:\Senior\_Project\_Repository\python\saida\_agent\_example\avoidReaver\save\_model\actor\avoid\_Reavers\_PPO\_20200427215025\_nsteps\_100000\_lr\_0.0001\_gamma\_0.99\_epo\_3\_move\_a\_18\_move\_d\_2\_actor\_20200427223843.h5f file is saved as a actor model for evaluation