Pong Proximal Policy Optimization Self-Play Agent using Unity ML-Agents

Eric Buitrón López – A01704340
ITESM Campus Querétaro
Intelligent Systems

Abstract – This paper aims to showcase the difference between only using Proximal Policy Optimization (PPO) and using it with competitive self-play for creating an agent that plays Pong using the Unity ML-Agents toolkit.

I. Introduction

Reinforcement learning is a range of techniques used for learning based on experience. It is based on having an agent that explores a certain environment and gets a reward or penalty based on the action that it takes. Training involves a sequence of interactions between the agent and the environment for a certain amount of time. The goal of the training is for the agent to learn the optimal policy for the given environment [1].

Figure 1.- Reinforcement Learning sequence [2].

There are different algorithms that can be used for reinforcement learning. However, neural networks are often preferred.

II. State of the Art

One of the newest techniques for reinforcement learning is Proximal Policy Optimization (PPO). OpenAI released these algorithms in 2017 and they performed comparably or better than other state-of-the-art approaches while also being simpler to implement and tune [3]. PPO attempts to simplify the optimization process of previous policy algorithms while retaining their advantages [4].

Another technique used for reinforcement learning is competitive self-play. This consists of having the AI compete with itself to improve and learn from different play styles (previous versions of itself) [5]. This allows the agent to avoid overfitting to certain parameters of an environment since it is always changing with previous or current versions of the agent [6].

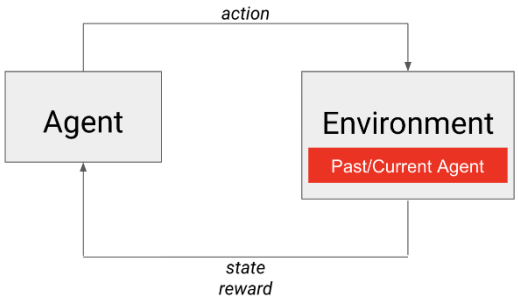

Figure 2.- Reinforcement Learning Self-Play sequence [6].

Unity released an open-source project called The Unity Machine Learning

Agents Toolkit (ML-Agents) that enables games and simulations to be used as environments for training agents. This toolkit provides implementations, based on PyTorch, of state-of-the-art algorithms, such as PPO & competitive self-play [7]. This is the toolkit that was used to create a Pong agent to showcase the difference between using PPO, competitive self-play, or both.

III. Implementation

The goal of the agent is to learn how to play Pong. This agent can then be used as an opponent AI for human players. To train the agent, a proper environment had to be built. This was done by creating a simple implementation of the videogame Pong in Unity. The environment was created in a way that allowed for duplication. Having several instances of the same environment allows the training to go faster. Each environment is made up of a field, a ball, a left paddle, and a right paddle.

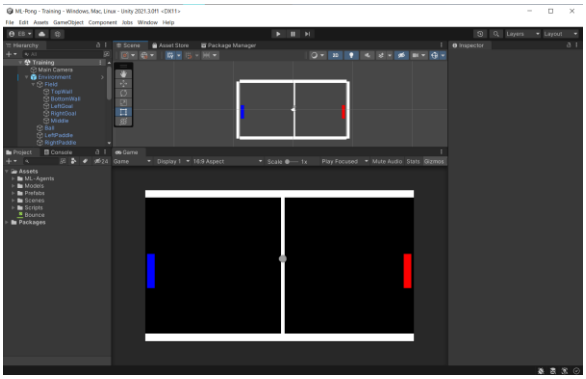

Figure 3.- Single environment setup.

The paddles act as the agents of the simulation. The left paddle was given a blue color while the right paddle was given a red one to differentiate them easily. However, it is important to note that both have the same script of the agent since the result will have the training of both paddles. The ball has a simple script that launches it in a random direction with a given speed to start the game.

The implementation of the agent consisted in creating a script with functions that handled the collection of observations from the environment and the actions that it should take based on these observations. Another script that handles the events of the environment (when a paddle scores a goal and ending the episodes after a certain number of steps) was created as a controller of the environment. The rewards given to the agent varied depending on the implementation. Besides this setup, a configuration file that specified parameters for the training algorithm was created. Finally, to speed up training, the environment was duplicated 19 times, for a total of 20 environments.

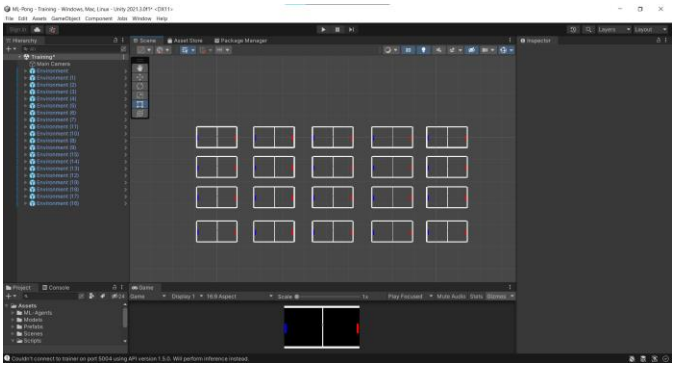

Figure 4.- Multiple environment setup.

The following implementations were made to train and test the agent:

Normal PPO:

This implementation consisted of using the base algorithm of PPO that

the ML-Agents toolkit provides. A reward of 1 was given to the agent when it collided with the ball and a penalty of -0.001 every time it moved. This was done with the intention of having the agent learn to hit the ball without it moving too much to avoid training an agent that just follows the ball since that would not look realistic.

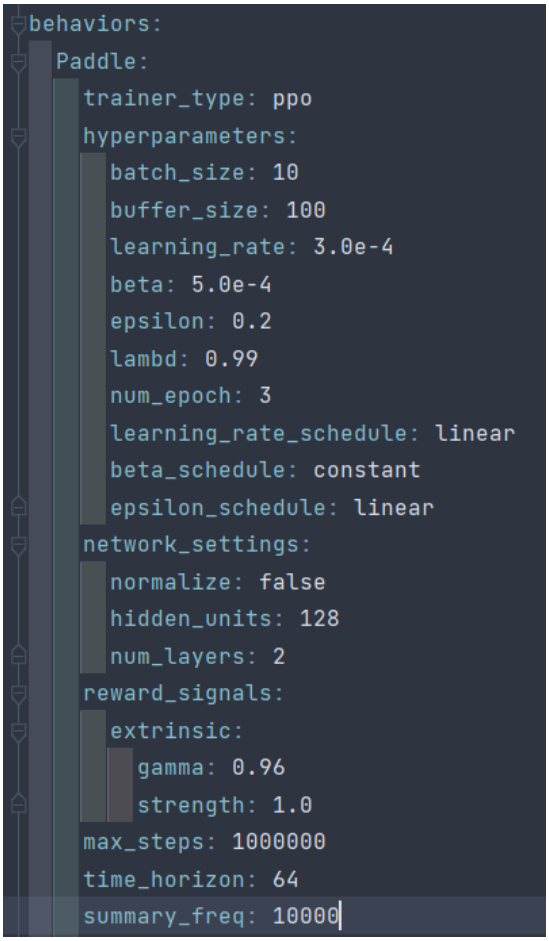

Figure 5.- Normal PPO configuration setting.

Competitive self-play:

To train a competitive self-play agent, a few modifications had to be made to the paddle agent and the awards that it received. First, the right paddle had to be marked with a different team id than the left paddle. Then, the function that handles the goals was given the following reward and penalty:

- Reward of 1 when it scored a goal.
- Penalty of 1 when the opponent scored a goal.

This allowed the agent to learn to be competitive and try new strategies throughout the episodes. Besides, the configuration had to be changed so that the algorithm included self-play steps.

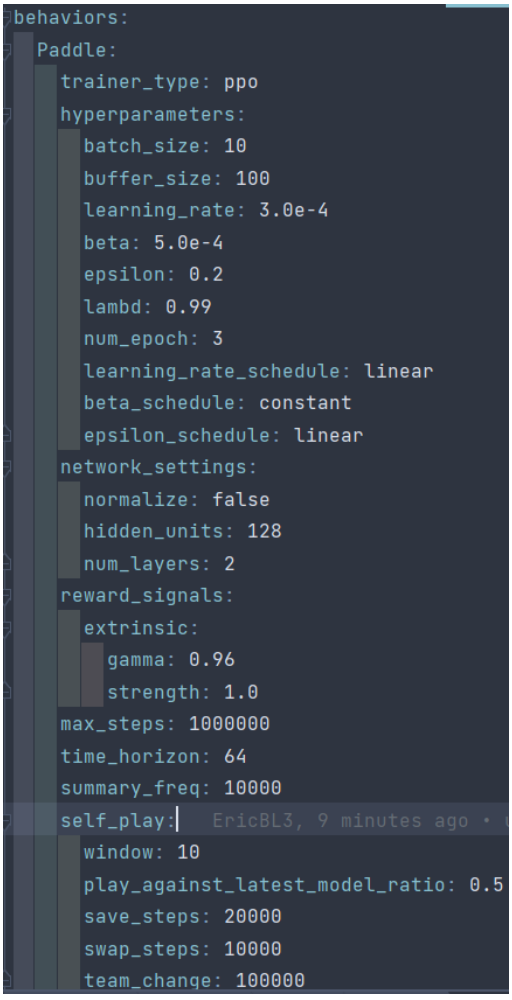

Figure 6.- Competitive self-play configuration setting.

Combined:

For the combined implementation, the same setting for normal PPO was used for 500 thousand steps and over that training the setting for competitive self-play was used for another 500 thousand steps. This gave the agent a total of 1 million steps of training which is the same as the other two implementations.

IV. Results

These were the results after training:

Normal PPO:

With this implementation, the agent tended to move to the bottom side of the field, and it only moves up as necessary to hit the ball.

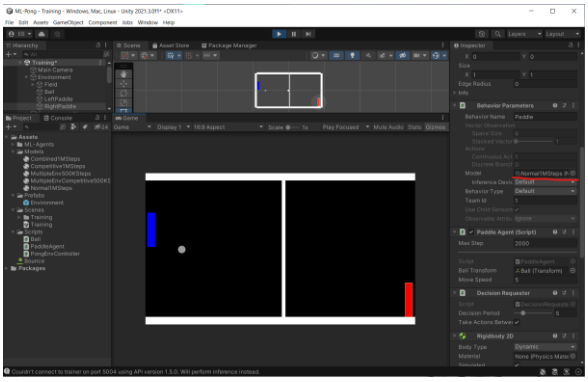

Figure 7.- Normal PPO agent in action.

The training process can be analyzed using TensorBoard. Figure 8 shows how the cumulative reward was increasing during training which means that the agent was learning. It can also be noted in figure 9, that the episode length started to increase which is a good sign that the agent started to hit more constantly the ball keeping it in play.

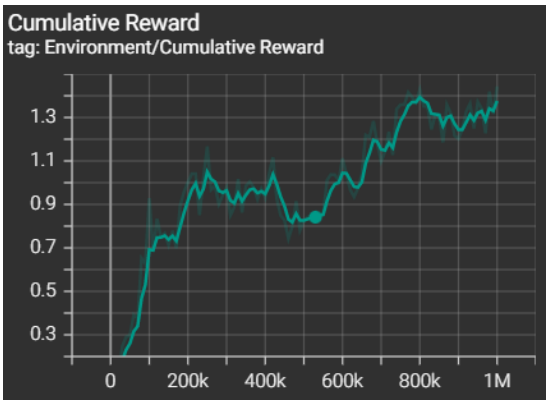

Figure 8.- Normal PPO cumulative reward.

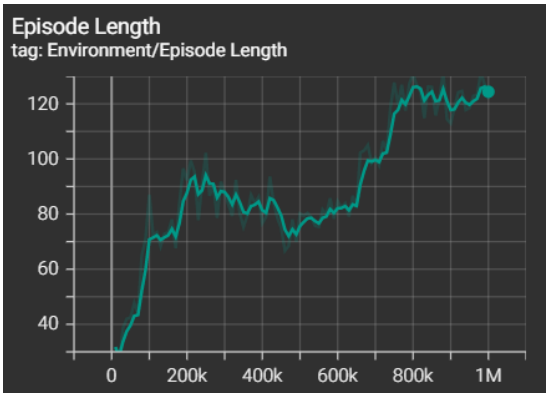

Figure 9.- Normal PPO episode length.

Competitive self-play:

In this implementation, the agent does not tend to go to the bottom side of the field, and it moves around more than the normal implementation. It sometimes hit the ball in more interesting ways, making it bounce in a different angle. However, it usually struggles with hitting the ball constantly which would make it a poor opponent against a human player.

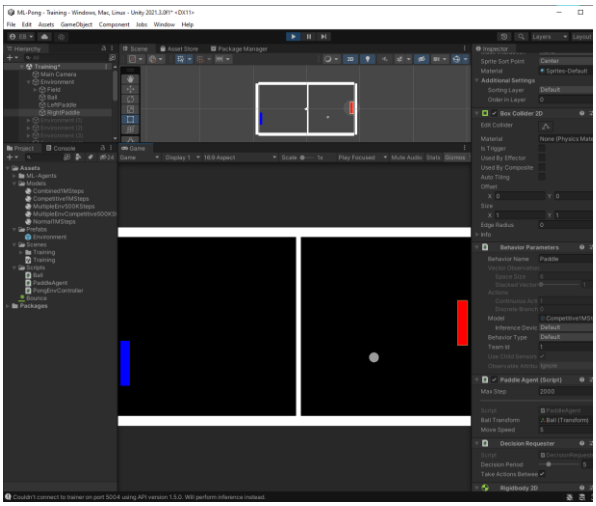

Figure 10.- Competitive self-play agent in action.

Figure 11 shows that the cumulative reward during this training went all over the place. The first 100 thousand steps the agent was learning but then it spiked down and up as training continued. The episode length also decreased after the 100 thousand step mark, and it never got up to the amount it reached during the first part of training.

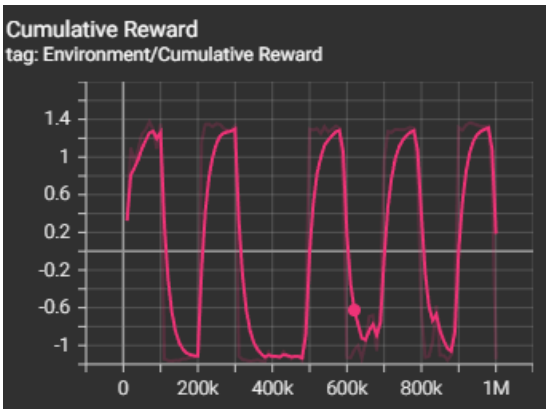

Figure 11.- Competitive self-play cumulative reward.

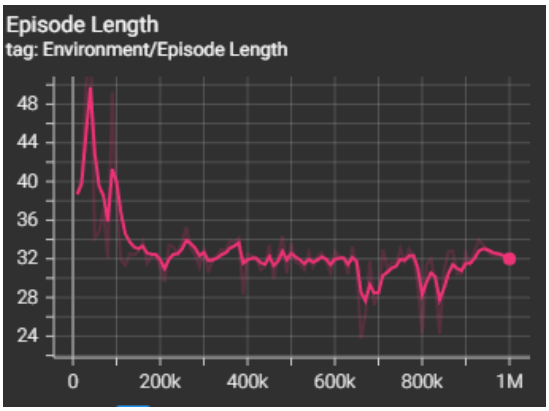

Figure 12.- Competitive self-play episode length.

All of this means that the agent did not learn well during training. This was probably since it tried to be competitive right from the beginning playing against itself before even knowing how to play. This can also be seen in the graph from Figure 13 which shows the relative skill level between two players (ELO rating). It started to quickly increase before rapidly decreasing and then it started to follow a similar pattern as the graph from Figure 11.

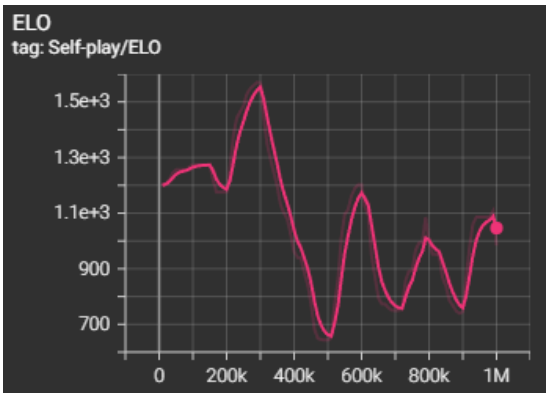

Figure 13.- Competitive self-play ELO.

Combined:

This implementation, which combined both methods for 500 thousand steps each, provided the best results. The agent manages to hit the ball more reliably than the competitive self-play

agent, while also hitting it in more interesting angles. This sometimes results in the agent not being able to hit the ball in occasions where the normal agent could, but it creates better gameplay.

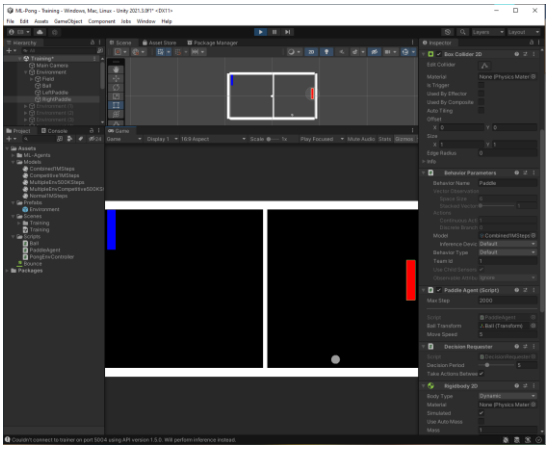

Figure 14.- Combined agent in action.

As seen in Figure 15, the cumulative reward had a similar behavior to the one from the normal training. When it changed to the self-play training, it decreased but then it started learning and increased to higher levels. The episode length also increased throughout the duration of training.

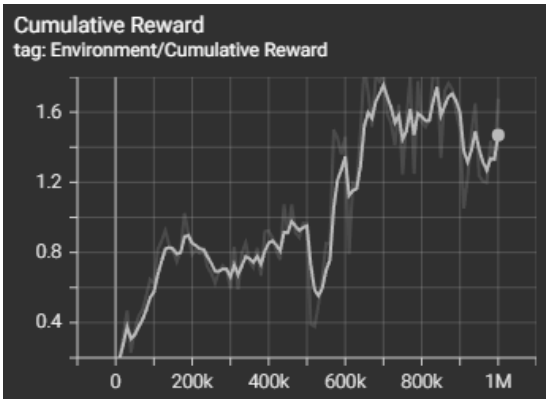

Figure 15.- Combined agent cumulative reward.

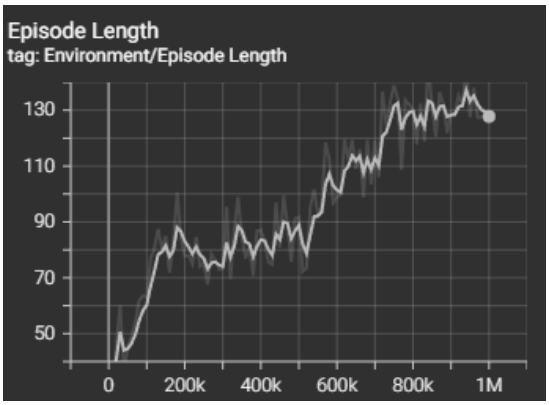

Figure 16.- Combined agent episode length.

The ELO rating of this training also steadily increased after a small learning curve between the 500 thousand and 600 thousand steps mark.

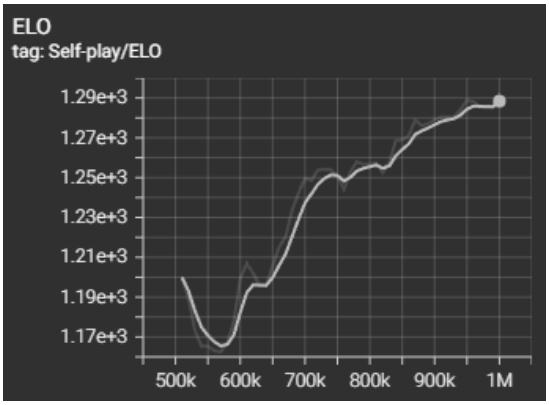

Figure 17.- Combined agent ELO.

When comparing the cumulative reward and the episode length of the normal PPO and the combined implementation, which were the successful training sessions, it can be observed that they are quite similar. The combined implementation performed slightly better in the end.

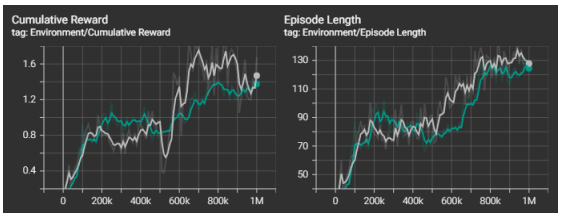

Figure 18.- Normal PPO & Combined agent comparison.

V. Conclusion

In conclusion, the best training was obtained by combining the normal PPO implementation with the competitive self-play one. This makes a lot of sense because first the agent needs to learn how to play and then it can try new strategies to become competitive, otherwise it will just start to behave poorly as seen in the competitive self-play implementation. It is important to note that even though training was done for double the default steps, they can still be increased to have longer training sessions and the other parameters can be modified to have different behaviors. It is very fascinating to see that a more in-depth study can be done by just changing a few things from the initial implementation that was done for this project.

References

1. Weng, L. (2022). A (Long) Peek into Reinforcement Learning. Retrieved 15 May 2022, from https://lilianweng.github.io/posts/2018-02-19-rl-overview/
2. YouTube. (2021). How to use Machine Learning AI in Unity! (ML-Agents) [Video].
3. Proximal Policy Optimization. (2017). Retrieved 15 May 2022, from https://openai.com/blog/openai-baselines-ppo/
4. Kim, T. (2021). Understanding and Implementing Proximal Policy Optimization (Schulman et al., 2017). Retrieved 15 May 2022, from https://towardsdatascience.com/understanding-and-implementing-proximal-policy-optimization-schulman-et-al-2017-9523078521ce
5. Competitive Self-Play. (2017). Retrieved 15 May 2022, from https://openai.com/blog/competitive-self-play/
6. Cohen, A. (2020). Training intelligent adversaries using self-play with ML-Agents | Unity Blog. Retrieved 15 May 2022, from https://blog.unity.com/technology/training-intelligent-adversaries-using-self-play-with-ml-agents
7. GitHub - Unity-Technologies/ml-agents: The Unity Machine Learning Agents Toolkit (ML-Agents) is an open-source project that enables games and simulations to serve as environments for training intelligent agents using deep reinforcement learning and imitation learning. Retrieved 15 May 2022, from https://github.com/Unity-Technologies/ml-agents