Multi-Agent Self-Learning Tank Game

Bidipta Sarkar
bidiptas@stanford.edu

Henry Ang
henryang@stanford.edu

I. Introduction

Many simple-player and multi-player games feature hostile entities that attempt to slow the player's progression. These entities often have very similar abilities to the players, but people can usually tell when they are playing a real opponent or an AI. Typically, these agents act in a very simplistic manner and can be outmaneuvered easily by experienced players. However, using reinforcement learning, we can train an agent to become a more effective adversary.

We have developed a simple tank game, where two players can move around the screen and shoot bullets at one another. Players have to learn to predict where their opponent will be to aim properly. Players also should learn how to dodge incoming bullets while also trying to get to a more advantageous position on the screen.

II. Literature Review

Traditional reinforcement learning (RL) uses a lookup table to store values and actions but this approach is slow as it learns the value of each state individually and it is memory consuming due to the curse of dimensionality. The solution is to estimate the value function using differentiable function approximations. There has been a tremendous amount of development in artificial intelligence with respect to RL over this past decade, in particular, "deep reinforcement learning (DRL)" – RL with function approximation by deep neural networks. For example, in the literature, one somewhat relevant paper to our project is the paper by Mnih et al. (2013) who developed a RL agent called Deep Q-Network (DQN) to play Atari that combined Q-learning with a deep convolutional ANN specialized for processing spatial arrays of data such as images. For learning each game, DQN used the same raw input, the same network architecture, and the same parameter values (i.e., step size, discount rate, exploration parameters, and other parameters more specific to their implementation). However, the Atari game has no adversarial agent which makes our problem different. Other papers more relevant to our project are from Fang et al. entitled "Applying Reinforcement Learning for the AI in a Tank-Battle Game" and from Smith et al. entitled "Continuous and Reinforcement Learning Methods for First-Person Shooter Games". Fang et. al. wrote a research paper on applying RL to a Tank-Battle Game. They used a game engine to create a Tank game and applied RL for the tank. Fuzzy logic was used as a concept to improve the performance of reinforcement learning. They performed several experiments based on their research to find the best fuzzy functions. Smith et al. wrote a paper on continuous learning models applied to first-person shooter bots. For example, by employing continuous learning, one can create bots that continuously adapt their behavior in response to their experiences, allowing them to change their actions and responses over time. There are generally three types of Deep Reinforcement Learning algorithms: 1) value optimization, 2) policy optimization, and 3) actor-critic. Examples of a value optimization algorithm include TD methods, i.e., TD-learning, Q-learning, DQN, etc. Policy-based methods include Proximal Policy Optimization (PPO), REINFORCE, etc. Actor-critic methods combine both value-based and policy-based approaches, and includes A3C, ACER, IMPALA, etc.

III. Dataset

Since our problem is fundamentally a game, we designed our own environment based on the OpenAI gym framework. Our environment is a square grid with two agents: a blue "agent" player and a red "opponent" player. Both players have full access to information about the state, so there is no hidden state information.

A. *Utility/IsEnd*

Formally, the environment is a 2-player zero-sum game. The players are spawned on random locations in the environment with 5 lives at the start of each match. The players move simultaneously from the perspective of the game as the environment asks both for their "actions" at the start of each timestep. An end state occurs when at least one player has run out of all lives or the two players collide directly. During a collision, both players lose a life until at least one of them has no lives left. The players' utility is 0 if neither has any lives left. If one player has lives while the other does not, the player with lives has a utility of +100 while the other has a utility of -100.

B. *Actions*

The action space for this environment has 4 characteristics. The first two characteristics control the movement of the tank as a direction and magnitude pair. The movement action is an acceleration instead of simply setting the velocity to a new value, so momentum is somewhat preserved between timesteps. The speed is capped so that the player does not have too high of a velocity, and tanks are also forced to stay in-bounds. The direction is represented as a radian between $-\pi$ and π, where 0 represents directly facing the opponent. The magnitude is a continuous value between $-m$ and m, where m represents the maximum acceleration at any timestep.

The last two characteristics of the action space control the firing of bullets. Bullets are circles that move at a constant velocity and are removed from the environment when they are out of bounds, or they interact with another bullet or the tank. If an enemy bullet hits a tank, the tank loses a life and the bullet disappears. If two bullets collide, one of them is randomly destroyed while the other continues moving at its original velocity. Tanks can fire bullets in any direction, but they have a cooldown of a set number of timesteps for firing bullets. The action controlling the direction is continuous with a minimum of $-\pi$ and a maximum of π. Last characteristic of the action is a boolean representing whether to fire at this moment, but this is ignored if the cooldown is still active.

Bullets are launched at a set speed relative to the launching tank. For example, a bullet being launched in the same direction as the tank's motion will move faster than it would if the tank was standing still.

C. Visual State Representation

As raw pixels, the environment has the tanks as their respective colors and the bullets have the same color as the tank they originated from. The lives of the tank are represented as white circles on the tank itself. This state representation is helpful for human observers, but we use numerical input for training and evaluation

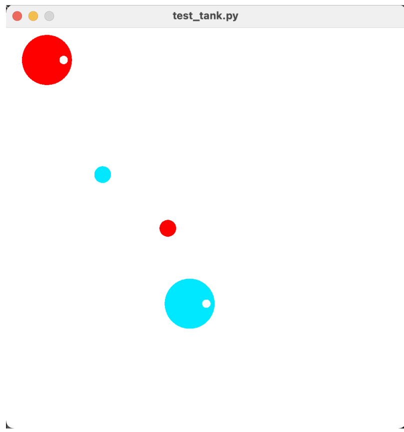

Fig. 1. Sample Visual Representation

D. Cartesian State Representation

For each player we can take note of the location (x, y), the velocity (v_x, v_y), the number of lives left, and the cooldown. Since there are 2 players in total, we have 12 continuous variables to represent the players in the environment. We also need to track the bullets in the scene and remember the position (x, y) and velocity (v_x, v_y) of each. Although there is a variable number of bullets in any given scene, we can explicitly calculate the maximum number of bullets possible on the scene using the fact that bullets move at a constant velocity greater than some minimum velocity and that there is a cooldown. When there are less bullets on screen than the maximum possible, we can repeat the expression of bullets in a cyclic manner. For example, if there are 5 possible bullets from a player but only 2 are launched, we can express the first bullet as a vector b_1 and the second bullet as b_2. To fill in the rest of the space, we can express the bullets as $[b_1, b_2, b_1, b_2, b_1]$.

We need to ensure that the policies designed for the agent also work for the opponent. We can rearrange the states to ensure that both of them follow a symmetric structure. For the agent, the observation is expressed as [agent_vars, opp_vars, agent_bullets, opp_bullets], while the opponent expresses the observation as [opp_vars, agent_vars, opp_bullets, agent_bullets].

Although this representation is technically complete, we have found that our training algorithms have struggled with forming strong actions using this representation. We believe that this is due to the difficulty inherent to converting cartesian coordinates to radian values. Both movement and bullet aiming are inherently angular measurements, so we decided to construct a different state representation that is better aligned with the form of these actions.

E. Relative Polar Representation

The relative representation imagines that the agent is stationary at the origin and the opponent is located on the axis of angle 0 (or on the positive x axis if we are thinking in terms of Cartesian coordinates). Note that since 0 represents facing the opponent in the action space outlined in section B as well. To achieve this result, the screen is essentially rotated and translated (but not scaled).

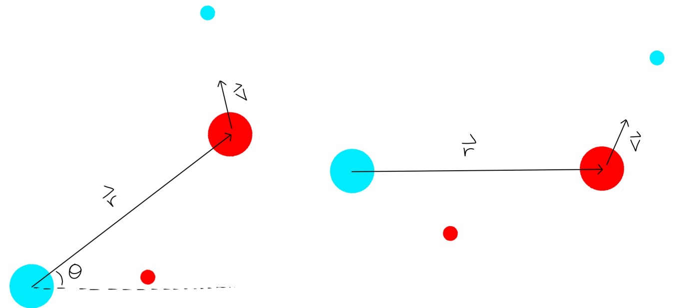

Fig. 2. Left: Original perspective. Right: Rotated perspective of Agent's observation. Vector $\vec{r}$ represents the displacement from the agent to the opponent, and $\vec{v}$ represents the relative velocity of the opponent.

The original angle θ, represented above, is calculated as:

$$\theta = atan2(y_{opp} - y_{agent}, x_{opp} - x_{agent}) \quad (1)$$

For all objects i, we need to recalculate the angles θ_i as:

$$\theta'_i = atan2(y_i - y_{agent}, x_i - x_{agent}) - \theta \quad (2)$$

Since the agent is considered stationary in this representation, all velocities must be modified to fit this frame. Therefore, for all objects i, we need to calculate new velocities ($\vec{v}$) and positions ($\vec{r}$):

$$\vec{v}'_i = \vec{v}_i - \vec{v}_{agent}$$
$$\vec{r}'_i = \vec{r}_i - \vec{r}_{agent} \tag{3}$$

Note that these new velocities and positions are not rotated to fit the new angle of the observation. We can rotate all of these, but the operations we will perform on them will not require this to be true.

We can calculate the new distance between an object i and the agent as:

$$l_i = ||\vec{r}'_i||_2 = \sqrt{\vec{r}'_i \cdot \vec{r}'_i} \tag{4}$$

With the values of θ_i and l_i, the agent can understand the positions of all the objects in the environment as polar coordinates. To encode velocity, we can calculate angular momentum and the dot product of velocity and position (which is proportional to the rate at which the object is approaching the player for a given distance).

$$L_i = (\vec{r}'_i \times \vec{v}'_i) \cdot \hat{k} = r'_{i,x} v'_{i,y} - r'_{i,y} v'_{i,x} \tag{5}$$

$$S_i = \vec{r}'_i \cdot \vec{v}'_i = r'_{i,x} v'_{i,x} + r'_{i,y} v'_{i,y} \tag{6}$$

Theoretically, the combination of θ_i, l_i, L_i, and S_i can be used to re-calculate all of the original information. However, to assist with training we can also make some extra calculations. Specifically, if we assume that all objects continue moving at their current velocity, we can calculate the distance from object's trajectory to the origin (d_i), and the time it would take to get there (t_i), which could be negative if the object is moving away from the origin. We can calculate them as follows:

$$d_i = \frac{|L_i|}{||\vec{v}_i||} \tag{7}$$

$$t_i = \frac{-v_{i,x} \frac{L_i}{||\vec{v}_i||^2} - r_{i,y}}{v_{i,y}} \tag{8}$$

An agent can use this information to perceive the threat of a collision. A large d_i implies that the trajectory will not intersect the other player and a large t_i means that the object will take a long time before it collides. Furthermore, a negative t_i means that there is no threat since the object is moving away from the player.

Since $v_{i,y}$ can be very small or 0, we clip the value of t_i such that its value is less than the maximum time a bullet can exist in the game. Similarly for d_i, we ensure that the magnitude of relative velocity is large enough as to not get an extremely large number when dividing.

For each bullet, the observation consists of $[\theta_i, l_i, L_i, S_i, t_i, d_i]$. For the opposing agent, we already know the angle is 0, so the observation includes $[l_{opp}, L_{opp}, S_{opp}, t_{opp}, d_{opp}, lives_{opp}, cooldown_{opp}]$. For the agent itself, we actually give the original Cartesian coordinates since the relative state leaves out some information. Specifically, it does not know when the agent's maximum speed is reached or how close it is to the game boundary. All of these sub-observations are combined to the full observation of the form [agent_vars, opp_vars, agent_bullets, opp_bullets].

Finally, to improve the stability of learning, we normalized all of the input states to be approximately between -10 and 10. Without this change, the values of L_i and S_i were extremely large since their maximum values are the product of the maximum velocity and the maximum distance between two objects.

IV. Baseline

For the baseline, we have developed 2 simple policies. The simplest is the random action policy (called L0), which randomly samples from the action space, moving and shooting randomly. The other policy moves randomly but shoots at the current position of the opponent (called L1). Note that since an angle of 0 always corresponds to the current direction of the opponent, the L1 policy is equivalent to L0 but it has a 0 for the aim direction and it always fires a bullet when possible.

Due to the complexity of the environment, we cannot create an effective oracle. However, a theoretically optimal agent would try to dodge incoming bullets while firing bullets at the expected future location of the opponent. Also, due to the nature of collisions, an oracle would likely try to collide with the opponent when they have a comfortable lead on lives but would flee otherwise. In our analysis, we will see how the learning agents compare to this idea of an oracle.

V. Main Approach

We choose to use the widely popular Proximal Policy Optimization (PPO) as the algorithm for training our agents because of its empirically strong performance in many RL tasks. PPO is a policy gradient RL algorithm whose objective function ensures that each gradient step does not deviate too much from the old policy, enhancing the training stability. We choose not to use value-based algorithms such as DQN because our Tank environment's action space is continuous, while valued-based methods typically works better with discrete action spaces.

The choice of the architecture for the policy network is rather straight-forward. Both of the initial Cartesian State representation and the latter Relative Polar representation during the development of our Tank environment have a bounded real-valued observation space. Because we do not require processing of pixel frames, we choose the simple Multilayer Perception (MLP) structure for the policy network.

The first type of learning we experiment with is training against a stationary opponent - the baseline policy, L1, which is simple but surprisingly strong in this game. We expect our agent to match or outperform the opponent after convergence.

The next step will be enabling self-training by taking advantage of the symmetry of the tank environment. The self-training process goes as follows: training starts with both the player and the opponent being randomly initialized PPO

agents. By training the player against the opponent, we expect the player to outperform the opponent. When the average reward of the player against an opponent is greater than a certain threshold, we will save the current player's parameters and make it the opponent. With this iterative training process, we let the player to continue to evolve by beating older versions of itself. This process will continue until the agent no longer improves or the rate of improvement becomes minuscule. With self-learning, our agent never sees a hard-coded policy during its training process. It learns purely from the environment. As a result, it is hard to predict the final performance of such an agent.

VI. Evaluation Metric

We evaluate the strength of our AI by playing many games against the other policies and determining the number of wins, losses, and ties. We also determine the mean and standard deviation of rewards for games between two environments to determine the consistency of wins or losses.

VII. Results & Analysis

A. Initial Attempt

We started our experiments with the first version of observation space - the Cartesian state representation. We trained our agent against the L1 baseline opponent, and after a few million timesteps of training and attempts to tune the hyper parameters, the agent does not seem to learn well.

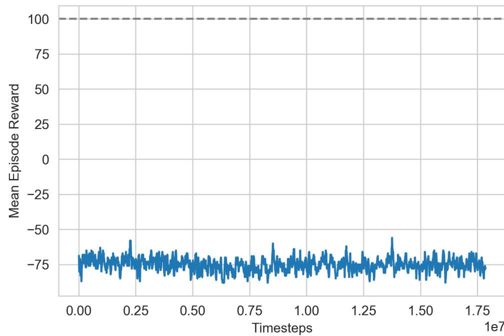

Fig. 3. Training Rewards in the First Training Attempt

The training rewards clearly showed that the algorithm made no progress in training the agent. We initially hypothesized that the L1 opponent might be too strong for the agent to receive any reward signals, so we tried to train it against the L0 random agent. However, it turned out that the learning curve is still flat even against the random opponent. We investigated possible bugs in our Tank environment, attempted hyper-parameter tuning and tried other algorithms such as A2C and TD3, but these attempts were all to no avail.

B. Changes to the Tank Environment

As a result of the failed attempts, we tried to make changes in the Tank environment that could potentially aid training. We made attempts on two fronts, observation space representation and rewards. The details of the Relative Polar Representation have been explained in the previous sections, so we make no further elaborations here. For rewards, the original environment only yields reward at the end of an episode when player wins, loses or ties. Such sparse rewards makes it difficult for the algorithm pick up useful signals for the majority of timesteps, especially earlier ones. To solve this problem, we decided to create a training environment with additional rewards that reflect mid-game events. For example, the player losing a life will get a negative reward, while the opponent losing a life will produce a positive reward. Some strategically beneficial events also receive rewards. For example, cancelling an opponent bullet with a player bullet will produce a small reward.

With these modifications, we managed to train a PPO agent that can outperform the L1 baseline opponent with a relatively small number of training timesteps.

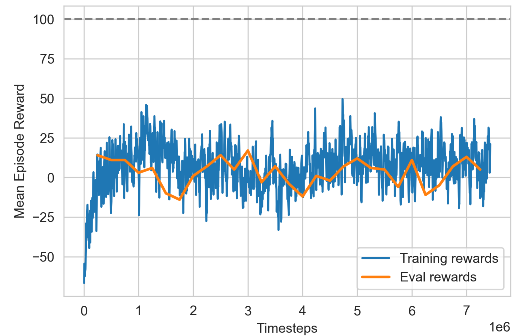

Fig. 4. Rewards of PPO agent against L1 opponent

It is important to note that training and evaluation of an agent took place in two different environments. The training environment has the additional rewards that help training of the agent, while the evaluation environment uses the original end-of-episode reward. In figure 4, the orange line shows the evaluation rewards.

C. Hyper-Parameter Tuning

While we finally trained an agent that beats the L1 baseline opponent, the hyper-parameters used in this model were the default values, so we still needed to perform hyper-parameter tuning for the PPO agent to perform better. Due to time limitation, we only attempted to a few hyper-parameters that we consider to be most relevant.

We first attempted tweaking the discount factor γ. Although the default 0.99 is relatively high already, the “big reward” in our Tank environment only appears at the end of each episode, so we tried to further increase it. After some attempts, we found that with $\gamma = 0.995$, the PPO agent outperforms the default value. Figure 5 shows that most of the evaluation rewards are above zero, and the maximum evaluation rewards is 31, which is above that of the PPO agent with default hyper-parameters.

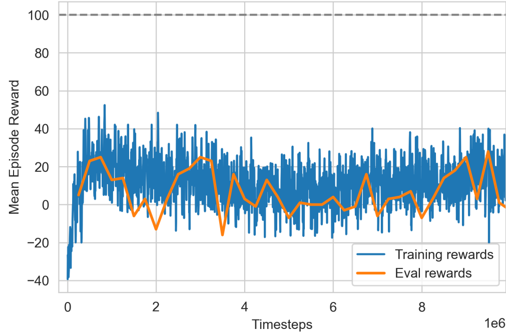

Fig. 5. Rewards of PPO agent against L1 opponent, $\gamma = 0.995$

We also tried some different batch sizes, but since we did not identify a value that surpasses the default, we will not elaborate here. From the experiments so far, we realized that the model first learned rapidly but then started oscillating. We expected that a smaller gradient steps might cause slower learning but higher rewards at convergence.

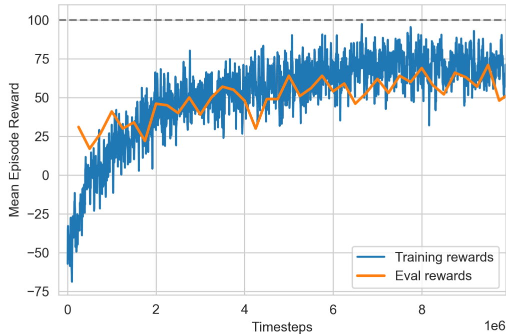

Fig. 6. Rewards of PPO agent against L1 opponent, stepsize = 3e-5

It turned out that with a smaller stepsize, training was indeed a little slower, but the end performance far exceeded the expectation. The evaluation rewards it obtained were a lot closer to the theoretical limit of 100, compared to all other agents so far.

D. PPO Self-Training

The learning curve of the self-training PPO agent looks quite different from the others, mainly because that the opponent is not stationary but frequently updated. This is the reason why the training rewards never went too much higher above zero; once the player became good enough, the opponent became a copy of the player and drove the training rewards down.

The red dashed line in figure 7 represents the evolution threshold. Whenever the evaluation reward is higher then the threshold, the opponent is replaced with a copy of the current player, and a new iteration begins.

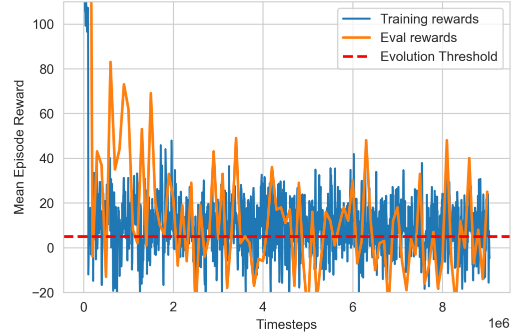

Fig. 7. Rewards of Self-Learning PPO Agent against an Evolving Version of Itself

E. Final Agent Rewards

After generating our optimal PPO and Self Training (ST) models, we can evaluate their strength by running games multiple times against another policy. Table 1 represents the rewards from running 300 games between two agents. As a reminder, L0 represents a completely random policy while L1 represents a policy that moves randomly but aims at the current position of the opponent.

TABLE I
REWARDS OF COMPETING POLICIES (TOP ROW VS LEFT COLUMN)

	L0	L1	PPO	ST
L0	-1.0 (4.91) 107-83-110	92.0 (1.767) 279-18-3	98.67 (0.66) 296-4-0	91.0 (1.65) 273-27-0
L1	-93.67 (1.48) 1-17-282	-2.0 (5.54) 135-24-141	55.67 (4.69) 229-9-62	41.67 (4.68) 181-51-62
PPO	-98.0 (0.81) 0-6-294	-57.0 (4.64) 60-9-231	-3.0 (5.38) 126-39-135	-3.0 (5.34) 124-43-133
ST	-93.0 (1.47) 0-21-279	-33.0 (5.04) 81-39-180	6.0 (5.34) 138-42-120	-7.33 (5.23) 113-52-135

Table 1: Each cell represents the result of 300 games of the player from the top row vs the player from the left column (where the score is from the perspective of the first player). The top value is the sample mean reward along with the standard deviation of the sample mean in parenthesis (sample standard deviation divided by $\sqrt{300}$). The bottom values represent wins-ties-losses.

The diagonal of the above table exists to demonstrate the "fairness" of the tank environment. In the diagonal, each policy is playing against itself and we find that 0 is always within 2 standard deviations of the each mean. Therefore, we cannot confidently state that the environment or learnt policies are biased in favor of one side over the other.

As expected, the random L0 policy is the weakest agent, losing to other opponents almost all the time. PPO seems to be the strongest opponent against this agent, winning around 98% of the games and having ties for the other 2%. The expected rewards for L1 and ST do not seem to have a statistically significant difference. However, ST never loses any match against L0 while L1 loses a few matches (approximately 1% of the time).

The hand-coded L1 policy is beaten by both PPO and ST in terms of average rewards. PPO performed much better than

ST on average, though this appears to be mostly because PPO can convert many of ST's ties into wins.

When PPO and ST are playing against one another, PPO tends to win more often, though not by a statistically significant margin when running for 300 games.

We can conclude that PPO trained against L1 is the strongest policy in the games above. Against all other opponents, PPO has an average positive reward, indicating that it wins more often than it loses.

F. Qualitative Analysis

The PPO agent appears to have many qualities expected of an optimal agent. The agent tends to move towards the opponent when it has more health but tries to move away when it has less health. It also appears to be able to dodge bullets when it is far away, but this is imperfect, especially when it has hit a wall. Its bullets also seem to hit the opponent more often than those from L1, which launches bullets based on the present location of the opponent. This accuracy could come from the fact that the agent tries to move closer to the opponent when it is winning (therefore decreasing the chances of missing) and because it is choosing where to aim based on some prediction of the future location of the opponent.

The ST agent also acts similarly to PPO with some interesting differences. Specifically, ST often tries to induce a tie at the start of the game by colliding with the other agent before either has the chance to launch a bullet.

When PPO and ST play against one another (or each play against themselves), we see that the agent with more lives tries to chase the agent with less lives while each aim directly at one another. We can't observe much bullet-dodging behaviour, but this is likely due to the fact that the winning agent always tries to stay close to the opponent, making neither side able to dodge.

G. Difference between PPO and ST

PPO directly trains against the L1 agent, which is why we can intuitively expect it to be the strongest opponent against L1. However, this also means that PPO implicitly assumes that the opponent will be similar to the L1 agent. We see that, in practice, this was a good assumption as PPO ended up being the strongest agent of the four, and was extremely effective against the L0 agent.

On the other hand, ST trains against the strongest older version of itself, so it implicitly assumes that the opponent will act in a similar way to the agent it has trained against. This fact could explain why ST is more tempted to induce ties at the start of the rounds. Against a strong opponent, getting a guaranteed tie (with a score of 0), is more favorable than having the high possibility of losing.

Since neither PPO nor ST have trained against the other, the fact that PPO wins indicates that its policy generalizes better.

VIII. Error Analysis

One unexpected outcome related to the self-training process is that the final model has a lower performance then some of the intermediate versions. Specifically, we found that the model generated in the 15-th iteration had the strongest performance against other policies, and we actually used that model instead of the final model when evaluating the final performance.

To understand this result, we need to revisit figure 7. We can see that in the earlier timesteps, most of the evaluation rewards are above the threshold, meaning that the agent managed to surpass the opponent in just one evaluation cycle. However, as training progresses, more and more evaluation rewards are below the threshold. This is likely because that the opponent have gotten stronger, and it is harder to gain an advantage against it.

One hypothesis behind a declining performance after some iterations is that the agent starts to overfit itself. As the agent gets stronger, it needs to develop strategies specifically against itself, which might be effective only against itself but not other opponents.

Another hypothesis is that in the later half of the training process, all the improvements gained by the agents were merely noises. There were considerable oscillations around the threshold in the evaluation rewards towards the end of the training process, and an agent might just have got lucky to be above the threshold.

IX. Future Work

A. Tank Environment

The current Tank environment we made is relatively simple. As a result, a policy that merely shoots at the opponent is already quite strong. To make the environment more interesting, we might want to add more complexity to the game so that the agent has be more strategic to win. For example, we could add walls in the game so that a good agent could learn to use a wall as a cover and "play the corner".

Another possible area of future works is training with a different type of observation space. We only experimented with the numerical representations but not a pixel one in our project. In the future, it would be interesting to see if we can train an agent with just the visual inputs.

B. Training

With more time, we could perform better hyper-parameter tuning. In the project we only tuned hyper-parameters individually and used the one that produced the best result. A better approach would be doing a grid search over different combinations of hyper-parameters, and we might have achieved a better performance. We also just used the same hyper-parameters we tuned for the normal PPO on the self-learning PPO. With more time we could tune hyper-parameters of self-play PPO separately.

As explained in the Error Analysis section, one hypothesis we had for the self-learning agent's decline in performance is the noises in the latter training process. One potential solution is to raise the evolution threshold, so that each iteration takes longer, but improvements might be more robust.

Last but not least, we only tried training the self-play agent from scratch. One alternative that we could explore is starting with a strong opponent instead of a random one. This allows us to see if self-learning can improve further upon an already converged PPO agent.

X. Code

Link to our Github repo:
https://github.com/guohaoang/TankGym

The sandbox repo with extra files along with the commits during development:
https://github.com/barrybrianbarrios/slimevolleygym

Special thanks to David Ha, the author of SlimeVolleyGym environment, which we used as a starting point for developing our Tank environment and the training process.

References

[1] Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., . . . Silver, D. (2017, October 06). Rainbow: Combining improvements in deep reinforcement learning. Retrieved April 30, 2021, from https://arxiv.org/abs/1710.02298

[2] Fang YP, Ting IH. Applying Reinforcement Learning for the AI in a Tank-Battle Game. JOURNAL OF SOFTWARE, VOL. 5, NO.12, DECEMBER 2010.

[3] Mnih, V., Badia, A., Mirza, M., Graves, A., Lillicrap, T., Harley, T., . . . Kavukcuoglu, K. (2016, June 16). Asynchronous methods for deep reinforcement learning. Retrieved April 30, 2021, from https://arxiv.org/abs/1602.01783

[4] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017, August 28). Proximal policy optimization algorithms. Retrieved April 30, 2021, from https://arxiv.org/abs/1707.06347

[5] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hassabis, D. (2017, October 19). Mastering the game of go without human knowledge. Retrieved April 30, 2021, from https://www.nature.com/articles/nature24270

[6] Smith, Tony C., and Jonathan Miles. ”Continuous and Reinforcement Learning Methods for First-Person Shooter Games.” GSTF Journal on Computing (JoC) 1.1 (2014).

[7] David Ha. 2020. Slime Volleyball Gym Environment. https://github.com/hardmaru/slimevolleygym. (2021).

[8] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.