Learn From Your Enemy: An A-MARL Framework Demonstrating Adversarial Learning Abilities

Kaiwen Bian*
Halıcıoğlu Data Science Institute
University of California, San Diego
kbian@ucsd.edu

Andrew Hudson Yang
Halıcıoğlu Data Science Institute
University of California, San Diego
any012@ucsd.edu

Zhenghao Gong
Halıcıoğlu Data Science Institute
University of California, San Diego
z3gong@ucsd.edu

Weijie Zhang
Halıcıoğlu Data Science Institute
University of California, San Diego
wez042@ucsd.edu

Ziyu Huang
Halıcıoğlu Data Science Institute
University of California, San Diego
zih029@ucsd.edu

Qirui Zheng
Halıcıoğlu Data Science Institute
University of California, San Diego
q7zheng@ucsd.edu

Abstract

In this study, we introduce AlphaTank, an Adversarial Multi-Agent Reinforcement Learning (A-MARL) framework designed to explore reinforcement learning (RL) agents' capabilities in adversarial environments. Our framework provides a fully integrated RL pipeline, enabling seamless training with minimal external dependencies while allowing extensive customization for multi-agent cooperation and competition. We demonstrate that RL agents develop diverse strategies by adapting to dynamic environments and opponent behaviors, learning adversarially in response to different bot characteristics. Our best performing agent is PPO agent against aggressive bot with win rate of 79%. However, our findings reveal fundamental limitations of traditional RL algorithms in adversarial settings, particularly their dependency on strong opponents as implicit "teachers" for effective learning. Weak adversaries hinder strategy development, emphasizing the need for curriculum learning, transfer learning, or cycle learning to facilitate training progress. Our cycle learning agent addresses this issue and achieves the best win rate (average win rate of 47.75% across all tasks) other than the original corresponding agent on each task. Additionally, our experiments show that RL agents struggle with team-based collaboration with win rate of 58% using PPO agents against smart bots, highlighting the need for algorithmic advancements such as shared parameters or explicit coordination mechanisms to enable cooperative behaviors. Our findings underscore key challenges in adversarial RL, such as reward function design, convergence dynamics, and emergent complexity in multi-agent interactions, paving the way for future research on scalable and robust adversarial learning frameworks. Code is available at: https://github.com/georgong/alphaTank.

*Codebase at this repository and sample Wandb run at this report.

Preprint. Under review.

1 Introduction

Reinforcement learning (RL) has demonstrated remarkable success in solving complex sequential decision-making problems across various domains, ranging from robotics and autonomous driving to strategic games and scientific discovery. However, the majority of RL research has focused on single-agent environments, where an individual agent learns an optimal policy through interactions with a static environment. In contrast, multi-agent reinforcement learning (MARL) presents additional challenges, as agents must not only optimize their own rewards but also adapt to the dynamic strategies of other learning agents in cooperative, competitive, or mixed settings. This challenge is over-complicated even more for our agent when the environment becomes adversarial and when the other agent or team of agents' goal is to kill this learning agent. They need to learn from each other and develop strategies to defeat one another.

Adversarial Multi-Agent Reinforcement Learning environments (A-MARL) introduce non-stationarity, making it difficult for individual agents to model the underlying dynamics due to constantly evolving opponent behaviors. Traditional independent Q-learning and decentralized policy learning approaches struggle in such settings, often resulting in suboptimal performance due to policy instability and overfitting to specific opponent strategies. To address these challenges, recent research has explored centralized training with decentralized execution (CTDE), adversarial self-play, and curriculum-based opponent selection, enabling agents to generalize better and exhibit robust decision-making in diverse scenarios.

In this work, we introduce a novel A-MARL framework designed to investigate the learning dynamics of RL agents in adversarial competition, possibly demonstrating learning abilities of traditional RL algorithms. Our environment features multiple agents with distinct objectives and rule-based AI-controlled opponents, ensuring a diverse and evolving training landscape. We incorporate cyclic learning and adaptive opponent selection to systematically expose agents to varied strategies, fostering adaptability and resilience in learned policies. By leveraging adversarial interactions and strategic adaptation, our study aims to provide insights into how RL agents develop robust policies in competitive multi-agent settings.

2 Related Work

Multi-agent reinforcement learning (MARL) has emerged as a vibrant research area, addressing agents' learning problems in adversarial and/or cooperative environments. Early work in MARL focused on decentralized learning and coordination among agents, leading to frameworks such as independent Q-learning [12]. The field has advanced to include centralized training with decentralized execution, enabling agents to use global information during learning while retaining autonomy during inference. [8]

Policy gradient methods such as Proximal Policy Optimization (PPO) [11] and Soft Actor-Critic (SAC) [3] have been the SOTA RL algorithms that demonstrated strong performance in both single-agent and multi-agent settings. PPO's clipped surrogate objective provides a stable update process that is well-suited to multi-agent interactions. Similarly, SAC's maximum entropy regularization framework encourages exploration by optimizing a stochastic policy that balances reward maximization with policy entropy. Both methods have been successfully extended to multi-agent settings, with centralized critics or opponent modeling [8].

A well-designed RL environment is important for effectively training agents, as it defines the rules, physical dynamics, actions, and goals that shape the learning process. For instance, platforms such as OpenAI's Gymnasium [1] offer standardized APIs for a wide range of environments. Our custom tank battle environment leverages OpenAI's Gymnasium API to create a standardized environment wrapper that makes it much easier to bridge to different RL algorithms. The crafting of reward functions is also critical for effective agent training. Previous studies have employed dense and shaped rewards to guide agents toward complex behaviors, such as coordinated team play or strategic positioning [4]. We used a combination of spare reward and dense penalty to help the agents learn different goals.

In an adversarial setting, self-play and competitive training have been crucial for developing a robust policy. Training techniques such as cyclical and curriculum learning have been widely adopted in advanced RL systems such as AlphaStar [13]. The idea of cycle learning is to rotate the agent training

among a set of opponents with various strategies, ensuring that an agent does not over-specialize to defeat a particular strategy. This approaches allow the agents to gradually improve their policy, learning the advantages of different strategies. Thus, our work draws inspiration from these methods by incorporating a cyclic learning approach with a pool of rule-based and heuristically designed bot opponents, ensuring that agents are exposed to diverse strategies throughout the training process.

3 Methods

In this paper, we apply an very unique way of demonstrating RL's learning capability through using our unique multi-agent environment and learning via adversarial competition. We have provided an simplified pipeline of our framework in the below figure 1.

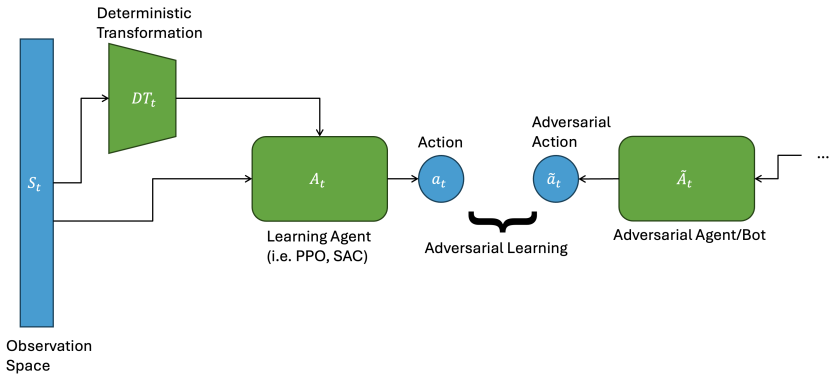


Figure 1: Pipeline overview of adversarial training procedures.

Under single agent-to-agent or agent-to-bot scenario (non-team mode), each learning agent is taking in observations both from the environment itself described in section 3.2 as well as observations after deterministic transformations described in section 3.2.2. After that, each agent perform its own update based on the RL learning algorithms and experiences collected by interacting and competing with the adversarial agent/bot with adversarial objective.

3.1 Hardware & Wandb Supports

All RL algorithms and trainings are ran using Apple Silicon CPUs. The training process is fast enough without CUDA distributed processing. All training metrics, media rendering, and final results are logged synchronously into the Weights & Biases (Wandb) platform, enabling real-time monitoring and hyperparameter tuning. We provide this Wandb report as a demonstration or our training process.

3.2 Fully Customized Multi-agent Competition & Collaboration Environment

We will introduce in the below section about our own unique environment that support multi-agent cooperative/adversarial learning, which we will be describing it through the MDP formulation, deterministic transformation, reward engineering, and team collaboration mode.

3.2.1 MDP Formulation: Observation Space & Action Space

In our multi-agent reinforcement learning setups, the entire tank competition competition is formulated as a Markov Decision Process (MDP) [9], following the standard definition:

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma) \tag{1}$$

State Space The state space $\mathcal{S}$ captures the complete observations of the environment at any given time frame. In our tank battle environment, a state $s \in \mathcal{S}$ includes the following information, which is an vector of length 260 per agent for single agent-to-agent or agent-to-bot trainings, which is constitute of raw observation as well as observations after deterministic transformations in section

3.2.2. We provide a detailed breakdown of our observation space in 1. In addition, all information is fully observable by any agents in our environment.

Component	Description	Dimension
Agent Tank Information (single instance)		
Position	(x, y)	2
Angle-based movement vector	(dx, dy)	2
Corner positions	-	4
Hitting wall (Boolean)	-	1
Alive status (Boolean)	-	1
Agent Bullet (multiplied by max number of bullets)		
Position	(x, y)	2
Velocity	(dx, dy)	2
Enemy Information (multiplied by number of enemy agents)		
Position	(x, y)	2
Relative position to agent	(rel_x, rel_y)	2
Distance to agent	-	1
Corner positions	-	4
Velocity	(dx, dy)	2
Hitting wall (Boolean)	-	1
Alive status (Boolean)	-	1
Enemy Bullet (multiplied by enemy bullet count)		
Position	(x, y)	2
Velocity	(dx, dy)	2
Relative position to agent	(rel_x, rel_y)	2
Distance to agent	-	1
Wall Information (multiplied by wall count)		
Corner positions	(x_1, y_1, x_2, y_2)	4
Buffer Information (unique in environment)		
Buff Position	(x, y)	2
Debuff Position	(x, y)	2
Agent in buff zone (Boolean)	-	1
Agent in debuff zone (Boolean)	-	1

Table 1: Observation Space Components and Dimensions

Action Space The action space $\mathcal{A}$ consists of all possible controls available to each tank agent. For our environment, actions include movement (speed adjustments), rotation (changes in the tank's orientation), and shooting (commands to fire bullets).

Environmental Dynamics The dynamics of the environment is captured by the transition probability function P, namely, $P(s'|s, a)$, which defines the probability of transitioning from state s to state s' when the agents execute a joint action $a \in \mathcal{A}$. The transitions are ruled by the physics of the environment simulation (e.g., bullet bouncing, buff zone, wall) and incorporate any stochasticity in the environment.

Rewards The reward functions $R(s, a, s')$ assign a scalar reward for each state transition. Our reward functions are carefully engineered and multifaceted to balance short-term tactical behavior with long-term strategic goals. The specific reward functions are described in section 3.2.3.

3.2.2 Deterministic Transformations

In the multi-agent Tank Battle environment, deterministic transformations ensure reproducibility, consistency, and fairness in agent interactions. These transformations define precise mechanisms for detecting environmental constraints, handling interactions, and computing relevant metrics in a manner that guarantees the same outputs for the same inputs.

Deterministic transformations are fundamental to collision handling, hit detection, and agent movement analysis. This section elaborates on three key deterministic transformation components: corner detection, hit detection, and distance metrics.

Corner Detection Corner detection ensures that tanks correctly interact with walls and obstacles, preventing unrealistic movement and unintended clipping. Each tank is represented by an Oriented Bounding Box (OBB) with four corners P_i computed using its position (x, y), width w, height h, and rotation angle θ:

$$P_i = (x + \cos(\theta) \cdot \Delta x - \sin(\theta) \cdot \Delta y, y + \sin(\theta) \cdot \Delta x + \cos(\theta) \cdot \Delta y)\,, \tag{2}$$

where $\Delta x, \Delta y \in \{\pm w/2, \pm h/2\}$. For static obstacles, an Axis-Aligned Bounding Box (AABB) is used, defined as:

$$AABB = \{(x_{\min}, y_{\min}), (x_{\max}, y_{\max})\}, \tag{3}$$

where $x_{\min} = x - w/2$, $x_{\max} = x + w/2$, $y_{\min} = y - h/2$, and $y_{\max} = y + h/2$. A collision is detected when an OBB and AABB overlap based on the Separating Axis Theorem (SAT), ensuring deterministic collision detection.

Hit Detection Hit detection determines whether a bullet collides with a tank by checking bounding box intersections. A bullet at position (x_b, y_b) with width w_b and height h_b is represented as:

$$\text{bullet\_rect} = \{(x_b - w_b/2, y_b - h_b/2), (w_b, h_b)\}. \tag{4}$$

A tank at (x_t, y_t) with dimensions w_t and h_t has its bounding box defined as:

$$\text{tank\_rect} = \{(x_t - w_t/2, y_t - h_t/2), (w_t, h_t)\}. \tag{5}$$

A collision occurs if the bounding boxes overlap:

$$x_b - \frac{w_b}{2} < x_t + \frac{w_t}{2}, \quad x_b + \frac{w_b}{2} > x_t - \frac{w_t}{2}, \tag{6}$$

$$y_b - \frac{h_b}{2} < y_t + \frac{h_t}{2}, \quad y_b + \frac{h_b}{2} > y_t - \frac{h_t}{2}. \tag{7}$$

Distance Metrics of Enemy We provide the Euclidean distance between the current agent tank with all other enemy tank as well as the distance between the current agent tank with all enemy bullets.

for the enemy tank:

$$d_{\text{tank}}(a, e) = \sqrt{(x_a - x_e)^2 + (y_a - y_e)^2}. \tag{8}$$

for the enemy bullet:

$$d_{\text{bullet}}(a, b) = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}. \tag{9}$$

3.2.3 Reward Engineering

Since we developed a new reinforcement learning environment, we need to implement reward functions according to our understanding of the environment dynamics and our expectation for agents' learning goals. Thus, we attempted the following reward engineering approaches for multi-agent learning.

Victory Reward The victory reward is applied when only one team survives in the game. It is processed by handling the bullet hit during the environment update. If a bullet collision occurs, the shooter will gain a hit reward, while the victim will receive a hit penalty. If an entire team was eliminated during the game play, then the surviving game will gain a victory reward. This simple reward mechanism encourages the primary objective of the gameplay for adversarial learning. The reward function can be defined as:

$$R_{\text{victory}} = \begin{cases} 100 & \text{if only one team survives} \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

Wall Hit Penalty The Wall Hit Penalty is designed to prevent the agent from repeatedly attempting to move through blocked areas, which would hinder learning and exploration. Initially, it applies a mild penalty for a limited number of collisions, but if the agent continues to collide with walls multiple times, the penalty increases in severity. This approach ensures that the agent is not discouraged from exploring the environment while simultaneously encouraging it to learn how to navigate around obstacles effectively.

$$P_{\text{wall}} = \begin{cases} P_{\text{low}}, & 0 < C_{\text{wall}} \leq T_{\text{low}} \\ P_{\text{high}}, & C_{\text{wall}} > T_{\text{low}} \end{cases} \tag{11}$$

$$C_{\text{wall}} = 0, \quad \text{if agent stop hitting the wall, reset the count} \tag{12}$$

Stationary Penalty The stationary penalty is designed to discourage the agent from staying in the same position for too long, thereby encouraging active exploration of the environment. We define stationary status based on grid size. If the tank remains within the same grid for 20 frames, the stationary penalty starts to apply.

$$P_{\text{stationary}} = \begin{cases} P_{\text{stationary}} + S, & \text{if } \lfloor x/G \rfloor = \lfloor x_{\text{last}}/G \rfloor \text{ and } \lfloor y/G \rfloor = \lfloor y_{\text{last}}/G \rfloor \text{ and } C_{\text{stationary}} \geq T_{\text{stationary}} \\ 0, & \text{otherwise} \end{cases} \tag{13}$$

Breadth First Search (BFS) Reward During training, we observed that the existing rewards were not effective in helping the tank locate its opponent. Additionally, due to the complexity of the maze, using a simple reward based on minimizing Euclidean distance led to unintended learning behaviors. To address this issue, we implemented a Breadth-First Search (BFS) algorithm to help the tank recognize the shortest path to its opponent. With this new feature, we designed a more structured penalization strategy and an improved reward mechanism.

Let D^t represent the number of grid cells explored by BFS at time step t, which reflects the shortest path distance between the two opponents. Additionally, let d^t denote the Euclidean distance between the tank

The new BFS-based reward function is defined as:

$$R^t_{\text{bfs}} = 0.05(D^{t-1} - D^t) + 0.05(d^{t-1} - d^t) \tag{14}$$

where:

- $(D^{t-1} - D^t)$ encourages the tank to reduce the shortest-path distance to its opponent over time.(global perspective)
- $(d^{t-1} - d^t)$ rewards the tank for moving closer to the nearest BFS-explored grid point. (local perspective)
- The scaling factor 0.05 ensures that the reward contributes to learning without overwhelming other reward signals.

This approach provides a more structured way to guide the tank's movement, leading to more effective learning and better navigation in complex environments.

Bullet Trajectory Reward/Penalty The bullet trajectory reward/penalty evaluates whether the predicted bullet path, based on the agent's current aiming location and direction, will hit the target. A simulation of the bullet trajectory will be performed for each bullet shot. The distance between the endpoint of the trajectory and the target opponents is calculated to determine a reward or penalty. If the calculated distance is within the close distance threshold, a reward is given that is proportional to that distance. If the calculated distance is outside the far distance threshold, a penalty is given that is proportional to that distance. The reward can be defined as:

$$R_{\text{bullet trajectory}} = \begin{cases} R_{\text{reward}} \cdot (1 - \frac{d}{D_{\text{thresh}}}) & \text{if } d < D_{\text{thresh}} \\ P_{\text{penalty}} \cdot \min(\frac{d - D_{\text{far}}}{D_{\text{far}}}) & \text{if } d > D_{\text{far}} \end{cases} \tag{15}$$

where D_{thresh} is the close distance threshold for granting reward, D_{far} is the far distance threshold for receiving penalty, and d is the euclidean distance between the endpoint of the bullet trajectory and all opponents, namely,

$$d = \min_{j \in O} \sqrt{(x_{\text{traj}} - x_j)^2 + (y_{\text{traj}} - y_j)^2} \tag{16}$$

where O is the set of all opponents' positions.

Tank Aiming Reward The tank aiming reward specifically incentivizes aligning the tank such that its projected bullet path intersects the opponents. It gives rewards if a tank consistently maintains its bullet direction toward an opponent. This encourages the agent to maintain alignment with the target over consecutive frames. The reward can be defined as:

$$R_{\text{aim}} = R_{\text{reward}} \cdot \mathbb{1}_{\{I=1 \text{ and } c+1 \geq T_a\}} \tag{17}$$

where $\mathbb{1}_{\text{condition}}$ is an indicator function. Within the condition, I is another indicator function representing whether the simulated trajectory will hit a target. T_a is a predefined aiming frame threshold for receiving a reward. c is an aim counter that either increases over consecutive frames or resets to 0 if the state of I changes or it goes above T_a.

Control Penalty The control penalty is designed to reduce erratic behavior by penalizing rapid or inconsistent actions, such as jittery movement, abrupt rotations, or spamming shots. For each type of action (movement, rotation, shooting), a counter is used to track action change, and if the change exceeds a predefined threshold, a penalty is applied. The reward can be defined as:

$$R_{\text{control}} = \sum_{i \in \{\text{movement, rotation, shooting}\}} P_{\text{penalty}} \cdot \mathbb{1}_{\{c_i \geq T_{\text{control}}\}} \tag{18}$$

where $\mathbb{1}_{\text{condition}}$ is an indicator function. Within the condition, c_i is an action consistency counter for action i. It increases if the action at the current frame is not consistent with the action at the previous frame. T_{control} is a predefined control threshold for receiving a penalty.

Rotation Penalty The rotation penalty addresses excessive rotation without sufficient movement. The agent's cumulative rotation angle is tracked. If it exceeds a predefined threshold without the tank moving more than a minimum distance threshold, a penalty is applied. The reward can be defined as:

$$R_{\text{rotation}} = P_{\text{penalty}} \cdot \mathbb{1}_{\{\theta \geq T_{\text{rotate}} \text{ and } d < T_{\text{reset}}\}} \tag{19}$$

where $\mathbb{1}_{\text{condition}}$ is an indicator function. Within the condition, θ is the agent's cumulative rotation angle, and d is the distance moved. T_{reset} is a minimum distance threshold for reset distance moved and T_{rotate} is a rotation threshold for receiving a penalty.

Final Rewards Not all reward/penalty functions are used in our final training phases as we don't want to impose too much constraint on the algorithm to learn what we human think is correct, but rather giving them a opportunity to learn novel ways of doing things and achieving the task.

We only use rewards according table 5. The rest of the reward functions are left as part of the environment API that allows other developers to use them for different agent training goals.

3.2.4 Team Player Mode

Notice that tea competition is essentially a form of cycle learning as well. However, instead of dealing with each type of strategic bot one by one and swapping out during training, our agent need to learn how to deal with multiples of them all at once, adapting to different strategies.

Our Reinforce Learning can also be applied on many-vs-many scenarios, where multiple agents cooperate as teams and compete against opposing teams. Under this setting, we add an additional team identification boolean value for each tanks and bullets in the observation value, which helps our agent to identify the teams.

3.3 Your Opponent is Your Teacher

The key idea behind any of the “intelligent" occurred in our setup is to rely on interaction with the environment and with the opponent or teacher, in some sense. This is different fro traditional adversarial generation learning that is often discussed in machine learning literature [2], we specify the differences in section 5.3. In addition, this learning is not purely mimicking the expert/teacher like in imitation learning [14] since the teacher has a different objective (exact opposite) comparing to the student/agent's objective. Hence, the agent need to learn from the teacher/opponent's behaviors to first understand what constitute "good" actions, then deduct out the appropriate action that it need to do to counter strike the teacher. This paradigm would face different problems mentioned in 5.

This type of learning paradigm poses a much broader discussion than what we can cover in this paper. However, our framework here proposed and showcased the needs of developing more advance RL algorithms towards this area of research.

3.4 Multi-agent Style Reinforcement Learning Algorithms

Through out this project, we will be mainly using two main RL algorithms to train adversarially against our other agent, namely Proximal Policy Optimization (PPO) [11] and Soft Actor-critic (SAC) [3]. We re-implemented these algorithm following Clean-RL's implementation [6] since most of the algorithm provided in standard RL packages (i.e. Stable-Baseline-3 [5], Clean-RL) are for individual agents in certain environment and we sometimes need two or more agents to be trained simultaneously in the same time (multi-agents conditions). We will go into the details of these algorithms in this section.

3.4.1 Proximal Policy Optimization

We adopt PPO as the first learning algorithm for training an agent in a multi-agent environment. PPO is an actor-critic method that optimizes a stochastic policy while constraining updates to prevent excessive policy shifts. Below, we detail the policy optimization process, advantage estimation, and loss function used in our implementation.

Policy Representation The policy is parameterized by a neural network $\pi_\theta(a_t|s_t)$, where θ denotes the policy parameters, and it outputs a categorical distribution over discrete action choices. The critic network estimates the state value function $V_\phi(s_t)$, where ϕ denotes the value function parameters. At each timestep t, the policy generates an action by sampling from the action distribution:

$$a_t \sim \pi_\theta(a_t|s_t) \tag{20}$$

The environment then returns a reward r_t and the next state s_{t+1}. The trajectory is stored in a replay buffer for T timesteps before performing a policy update.

Generalized Advantage Estimation (GAE) Following Clean-RL's implementation, we employ Generalized Advantage Estimation (GAE) [10] to compute the advantage function of the following where γ is the discount factor, λ controls bias-variance trade-off, and δ_t is the temporal difference (TD) error. We will be using the neural network representation of the TD error, essentially just MSE loss, with our value function, which we will be introducing in the following section.

$$A_t = \sum_{l=0}^{T-t} (\gamma\lambda)^l \delta_{t+l} \quad \text{or} \quad A_t = \delta_t + (\gamma\lambda) A_{t+1} \tag{21}$$

$$\delta_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t) \tag{22}$$

In addition, we dynamically normalizes our advantage value for stable training, represented as the following where μ_A and σ_A are the mean and standard deviation of the advantage that dynamically changes from the estimates over different batch.

$$\hat{A}_t = \frac{A_t - \mu_A}{\sigma_A} \tag{23}$$

Clipped Surrogate Objective The policy is updated using the PPO clipped surrogate objective, which prevents large policy updates by bounding the probability ratio:

$$L_{\text{clip}}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right] \tag{24}$$

where the probability ratio $r_t(\theta)$ is defined as the following where ϵ is a hyperparameter that controls the update step size.

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \tag{25}$$

Essentially, prior to PPO, many policy gradient based RL algorithm suffered from drastic changes in gradients, leading to unstable policy. The ratio is a method to prevent too far deviation o policy from the current with previous policy and the clipping is a strategy that prevents the ratio from deviating too far from 1, reducing variance and stabilizing training.

Value Function Update The value function is updated using a mean squared error (MSE) loss between the predicted state value $V_\phi(s_t)$ and the computed return $R_t = A_t + V_\phi(s_t)$, which can be shown as similar deriviation with traditional TD update using Bellman equation.

$$L_V(\phi) = \mathbb{E}_t \left[\frac{1}{2} (V_\phi(s_t) - R_t)^2 \right] \tag{26}$$

Loss Function To encourage exploration of our agent, we incorporated an entropy bonus, which is an standard practice in PPO, denoted as the following:

$$L_{\text{entropy}} = \mathbb{E}_t \left[-\sum_a \pi_\theta(a|s_t) \log \pi_\theta(a|s_t) \right] \tag{27}$$

Hence, the overall loss function combines the three components, where c_1 and c_2 are coefficients for balancing value function loss and entropy regularization.

$$L(\theta, \phi) = L_{\text{clip}}(\theta) - c_1 L_V(\phi) + c_2 L_{\text{entropy}} \tag{28}$$

The PPO training will be using the hyperparameter referenced in the table in the appendix [REFERENCE].

3.4.2 Soft Actor-critic

Very similar to PPO, SAC also uses an actor-critic style updates for the core learning to happen. Though it seems like that SAC is only vanilla actor-critic with extra entropy coefficient to promote exploration, it is actually theoretically derived from the paradigm known as RL as inference [7]. Below, we provide a detailed explanation of the policy updates, value function estimation, and loss functions used in our implementation.

Policy Representation Similar to PPO, SAC employs a parameterized policy $\pi_\theta(a_t|s_t)$ with a Gaussian distribution. It is then transformed via a `tanh` function to enforce action bounds. The policy network outputs a mean $\mu_\theta(s_t)$ and standard deviation $\sigma_\theta(s_t)$, parameterizing a Gaussian distribution like the following:

$$a_t \sim \tanh(\mathcal{N}(\mu_\theta(s_t), \sigma_\theta(s_t))) \tag{29}$$

Entropy Regularized Policy Optimization Sharing similar thoughts, SAC maximizes a modified objective that includes an entropy term. The entropy term $\alpha \log \pi_\theta(a_t|s_t)$ ensures sufficient exploration.

$$L_\pi(\theta) = \mathbb{E}_{s_t \sim D, a_t \sim \pi_\theta} \left[\alpha \log \pi_\theta(a_t|s_t) - Q_\phi(s_t, a_t) \right] \tag{30}$$

Again, this expression has a very deep theoretical root [7] that we will not be going into in this project, we will just be using the result.

Soft Q-Network The Q-value function is parameterized by two critic networks $Q_{\phi_1}(s_t, a_t)$ and $Q_{\phi_2}(s_t, a_t)$ to mitigate overestimation bias. The target Q-value is computed using the soft Bellman equation where d_t is the done signal, α is the entropy coefficient, which controls the balance between exploration and exploitation, and $\bar{\phi}_1, \bar{\phi}_2$ are target network parameters that are updated using an exponential moving average.

$$Q_{\text{target}}(s_t, a_t) = r_t + \gamma(1 - d_t) \left(\min_{i=1,2} Q_{\bar{\phi}_i}(s_{t+1}, a_{t+1}) - \alpha \log \pi_\theta(a_{t+1}|s_{t+1}) \right) \tag{31}$$

Essentially, the core intuition behind this "soft" update is to play conservative on the estimated values that the agent finds, sending out different "explore Q values" and use the most conservative value found. The actual critic networks are then trained to minimize the squared loss between the estimated Q-values and the soft Bellman target:

$$L_Q(\phi) = \mathbb{E}_{(s_t, a_t, r_t, s_{t+1}) \sim D} \left[\left(Q_\phi(s_t, a_t) - Q_{\text{target}}(s_t, a_t) \right)^2 \right] \tag{32}$$

Notice that this is the exact same updating logic in PPO with the difference being that PPO has the targeted value coming from the returned calculated by the GAE function and in here we have it coming from the soft Bellman update function.

Training and Implementation Details Architecture and hyperparameter used to train SAC is described in detail in table 8. We later trained SAC vs PPO, the hyperparmeters used for that setting is described in table 9.

Continuous-to-Discrete Wrapper. The environment originally operates in a discrete action space, where each action dimension corresponds to a finite set of discrete choices. To enable smooth exploration and learning with Soft Actor-Critic (SAC), we use a `ContinuousToDiscreteWrapper`, which maps a continuous action space $\mathcal{A}_{\text{cont}} \in [0, 1]^d$ into the original discrete space $\mathcal{A}_{\text{disc}}$ using a rounding and clipping function. The transformation is defined as follows:

$$a_i^{\text{disc}} = \min \left(\max \left(\left\lfloor a_i^{\text{cont}} \times (n_i - 1) + 0.5 \right\rfloor, 0 \right), n_i - 1 \right), \tag{33}$$

where $a_i^{\text{cont}} \in [0, 1]$ represents the continuous action for the i-th dimension, n_i is the number of discrete bins for that dimension, and a_i^{disc} is the corresponding discrete action. This mapping ensures that the transformed actions remain within the valid discrete space while preserving smooth control over discrete choices. The wrapper applies this transformation independently to each agent in the multi-agent system.

Replay Buffer. Each agent in our multi-agent system maintains an independent experience replay buffer, implemented as a fixed-size cyclic buffer. The buffer stores tuples of the form $(s_t, a_t, r_t, s_{t+1}, d_t)$, where s_t and s_{t+1} represent the agent's observed states at consecutive time steps, a_t is the executed action, r_t is the received reward, and d_t is a binary termination signal.

Transitions are stored in a rotating fashion, ensuring that newer experiences overwrite the oldest ones when the buffer is full. During training, agents uniformly sample mini-batches of experiences from their respective buffers to compute gradient updates. This off-policy storage mechanism facilitates sample reuse, stabilizes training dynamics, and enables the SAC agents to learn from temporally uncorrelated transitions.

3.4.3 Rule-Based Training Opponents

To supplement the regular training algorithms, we employ a suite of rule-based bots as training opponents. These deterministic agents, each implementing specific behavioral strategies, serve multiple purposes in our training frameworks. First, they provide consistent and interpretable baseline opponents, helping to guide the learning agent toward mastering fundamental combat techniques. Second, by maintaining a diverse set of bot behaviors - from aggressive pursuit to defensive positioning - we can expose the learning agent to a broad spectrum of strategies, potentially preventing the development of exploitable weaknesses that might arise from training against a single opponent type.

However, this approach introduces the challenge of "cyclic learning", where an agent might optimize its strategy against one bot type at the cost of performance against others. This phenomenon is particularly relevant in our tank combat scenario, where strategies effective against an aggressive opponent might prove counterproductive against a defensive one. This challenge motivates our subsequent implementation of cycle learning and curriculum-based training approaches.

3.4.4 Individual Bot Characteristics

The environment implements a flexible bot system through a BotFactory design pattern, which facilitates the creation and management of diverse bot behaviors. Each bot inherits from a BaseBot class and implements specific strategies following certain human heuristics (i.e. find advantage position and shoot):

RandomBot Executes random actions with temporal consistency, maintaining each action for 10-30 frames to create coherent movement patterns.

AggressiveBot Employs a pursuit-focused strategy, actively chasing opponents while maintaining optimal shooting distance. This bot is different from the SmartBot described later as this bot does not pursue in BFS manner, making it only available for fighting our RL agent in "no wall" battle ground conditions.

DefensiveBot Utilizes a human heuristics position-based strategy, seeking buff zones for tactical advantage and maintaining safe distances while shooting at targets with precise aim.

DodgeBot Implements an evasion-focused approach with a threat detection radius, calculating direction vectors to avoid both enemy tanks and incoming projectiles.

SmartBot An breadth first search agent with the goal to get into the shooting range of the enemy agent and shoot directly, used for battles with walls.

3.4.5 Cycle Learning Challenge

The cycle learning system builds on the regular training cycles by implementing a progressive training regime where the agent faces different bot types in sequence. Three rotation strategies are available:

- Fixed: Cycles through bot types in a predetermined order.
- Random: Selects opponents randomly from the available bot pool.
- Adaptive: Dynamically adjusts bot selection probabilities based on the agent's performance against each type.

The system maintains performance metrics through a rolling window to track win rates against each bot type. Victory rewards are scaled based on the agent's overall mastery of different bot types.

During the initial training phase, victory rewards are weighted by the proportion of bot types against which the agent has achieved the target win rate. After this phase, victory rewards are only granted when the agent has mastered all bot types, encouraging comprehensive skill development rather than specialization against specific opponents.

3.4.6 Curriculum Learning Setup

The training process implements a curriculum through a weakness parameter that modulates bot behavior. This parameter controls the frequency of bot actions:

- Initial Phase: Bots start with low action frequency (acting 10% of the time).
- Progressive Scaling: Action frequency increases either linearly or following a sigmoid curve.
- Final Phase: Bots reach near-full capability (acting 80% of the time)

This approach is particularly crucial in reinforcement learning, where agents often struggle with the exploration-exploitation dilemma when faced with complex tasks. By starting with simplified versions of opponents (bots that act infrequently), the agent can first develop basic strategies and fundamental skills. As the curriculum progresses and bot action frequency increases, the agent is gradually exposed to more sophisticated and challenging scenarios, preventing the common problem of policy collapse that can occur when training directly against fully capable opponents. This progressive learning approach helps maintain a manageable difficulty curve while ensuring the agent develops robust and adaptable strategies.

4 Experimental Results

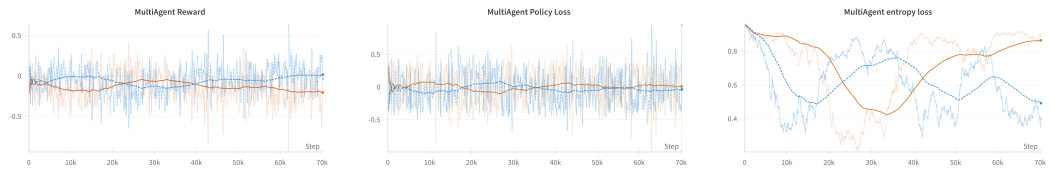
For this section, we have created a interactive Wandb report to show the training process. We have also uploaded all the rendering results in this video drive. We will also be going over a more comprehensive review of our result in the following section followed by video rendering of the performance of our agent on each task. We will be describing the performance from 6 different sections, including agent vs agent, agent vs bots, team agents vs team bots, team agents vs team agents, cycle agent vs bots, and a final section reporting inference time win rate of our agents.

All of the results below are trained with the reward/environment configuration in table 5, the PPO training setting in table 7, the SAC training setting in table 8, the SAC against PPO training setting in table 9, and the team setting with table 6.

4.1 Agent vs Agent

In general, we have observed that it is extremely hard for agent (no matter SAC or PPO agent) to defeat or learn from agent from the scratch since the RL algorithm rely highly on the interaction it has with the environment and its opponent is a big part of the environment loop. Hence, when the opponent is performing bad during instantiation (for both agents from a relative angle), the training becomes extremely hard, we observe similar effects in section 4.2.1. We will go over this phenomenon more thoroughly during section 5.

4.1.1 PPO vs PPO



(a) PPO Reward Plot (b) PPO -Policy Loss Plot (c) PPO- Entropy Loss Plot

Figure 2: PPO Agent vs. PPO Agent

4.1.2 SAC vs SAC

(a) SAC - Reward Plot (b) SAC - Policy Loss Plot (c) SAC - Alpha Loss Plot

Figure 3: SAC Agent vs. SAC Agent

4.1.3 SAC vs PPO

(a) PPO - Reward Plot (b) PPO - Policy Loss Plot (c) PPO - Entropy Loss Plot

Figure 4: SAC Agent vs. PPO Agnet

(a) SAC - Reward Plot (b) SAC - Policy Loss Plot (c) SAC - Alpha Loss Plot

Figure 5: SAC Agent vs. PPO Agnet

4.2 Agent vs Bots

We have numerous agents vs bots setting described in details for each sections.

4.2.1 PPO vs Smart Bot

(a) Reward Plot (b) Policy Loss Plot (c) Entropy Loss

Figure 6: PPO Agent vs. Smart Bot

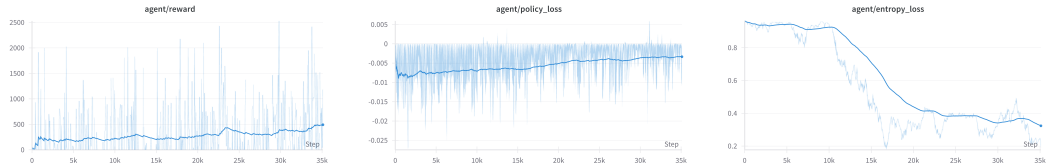
As mentioned above in section 4.1, when the opponent is performing terribly, it makes the learning process extremely hard, we observe a similar process here as well. When our smart bot has trajectory bounce turned off (meaning that it cannot "see" the other when there is a wall, hence decreasing performance), the RL agent has an extremely hard time to learn the correct strategy to defeat this bot agent.

4.2.2 PPO vs Aggressive Bot

(a) Reward Plot (b) Policy Loss Plot (c) Entropy Loss

Figure 7: PPO Agent vs. Aggressive Bot

4.2.3 PPO vs Dodge Bot



(a) Reward Plot (b) Policy Loss Plot (c) Entropy Loss

Figure 8: PPO Agent vs. Dodge Bot

4.2.4 PPO vs Defensive Bot

Training with Defensive Bot requires Our PPO Agent to learn to aim from a far distance. So this experiment is running under a faster bullets speed and a higher bullets max distances.

Table 2: Comparison of two bullet parameter settings

Environment Settings	Bullet Max Distance	Bullet Speed
Aggressive/Smart Bot	300	1
Dodge / Defensive Bot	800	4

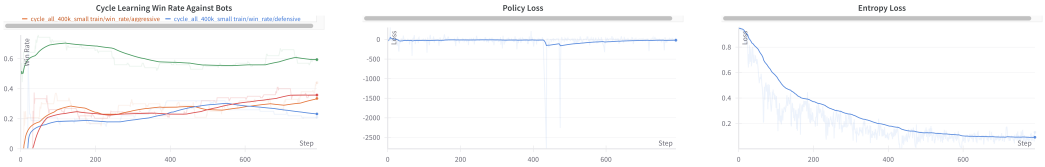
(a) Reward Plot (b) Policy Loss Plot (c) Entropy Loss

Figure 9: PPO Agent vs. Defensive Bot

4.3 Cycle Learning Agent vs Bots

The focus of cycle learning was to test the generalization of agents, as well as trying to solve the issue of having weak teachers. We believe that if one bot is an weak teacher, all bots together would be a strong teacher. Specifically we wanted to see if an agent trained on cycle learning will have better general performance than specialized agents.

To conduct our experiment, we trained an agent on each bot individually, then trained an agent on all four bots using cycle learning approach. Then we measured the win rate of each agent on each bot type. The results can be found in 4.



(a) Win Rate Plot (b) Policy Plot (c) Entropy Loss Plot

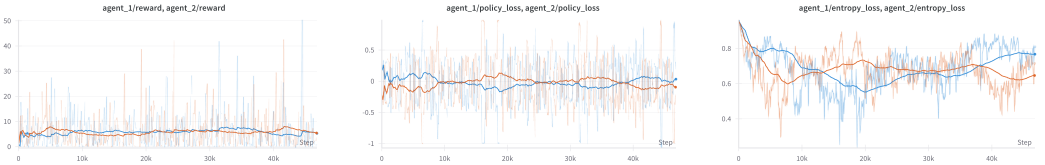
Figure 10: Cycle Learning Agent vs Four Other Bots

All the training configuration for cycle learning is the same for PPO training.

4.4 Team Agent vs Team Bots

We will be showcasing our agents team against various different bots team with different configurations to demonstrate team collaboration abilities.

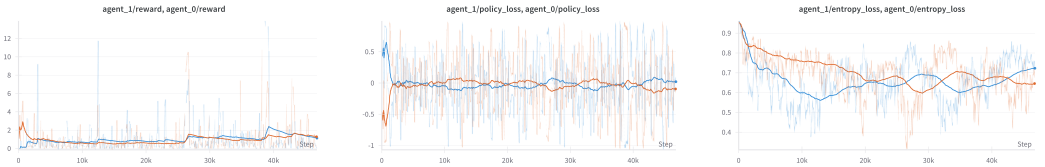
4.4.1 2 PPO Agents vs 2 Smart Bots



(a) Reward Plot (b) Policy Loss Plot (c) Entropy Loss

Figure 11: 2 PPO Agent vs. 2 Smart Bots

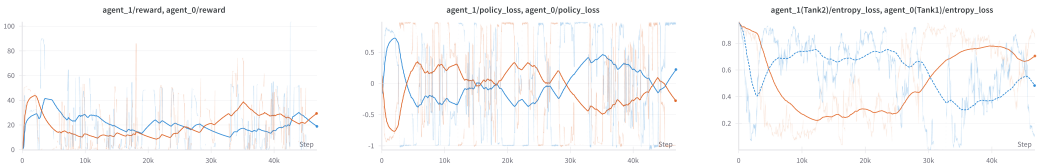
4.4.2 2 PPO Agents vs 1 Defensive Bot



(a) Reward Plot (b) Policy Loss Plot (c) Entropy Loss

Figure 12: 2 PPO Agent vs. 1 Defensive Bot

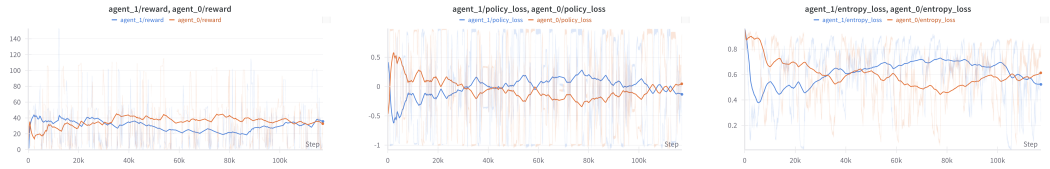
4.4.3 2 PPO Agents vs 1 Defensive Bot + 2 Smart Bots



(a) Reward Plot (b) Policy Loss Plot (c) Entropy Loss

Figure 13: 2 PPO Agent vs. 1 Defensive Bot + 2 Smart Bots

4.4.4 2 PPO Agents vs 1 Defensive Bot + 1 Aggressive Bot + 2 Smart Bots



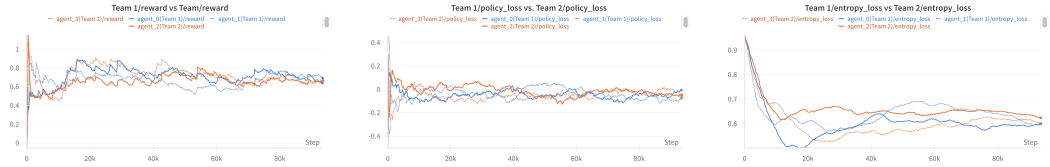
(a) Reward Plot (b) Policy Loss Plot (c) Entropy Loss

Figure 14: 2 PPO Agents vs. 1 Defensive Bot + 1 Aggressive Bot 2 Smart Bots

4.5 Team Agent vs Team Agent

We have also conducted team battling mode by comparing two PPO teams.

4.5.1 2 PPO Agents vs 2 PPO Agents



(a) Reward Plot (b) Policy Loss Plot (c) Entropy Loss

Figure 15: 2 PPO Agents vs. 2 PPO Agents

4.6 Inference Time Performance Against Bots

We provide the following table 3 to highlight the inference performance of different agents. All experiments are performed for 100 game runs. Notice that this result

Table 3: Inference time performance for RL agents vs bots in single mode and team mode.

Experiment Name	Agent Wins	Bot Wins	Agent Win Ratio
PPO Agent vs Smart Bot	60	40	60%
PPO Agent vs Aggressive Bot	79	21	**79%**
PPO Agent vs Dodge Bot	47	53	47%
PPO Agent vs Defensive Bot	41	59	41%
2 PPO Agents vs 2 Smart Bots	58	42	**58%**
2 PPO Agents vs 1 Defensive Bots	26	74	26%
2 PPO Agents vs 1 Defensive + 2 Smart Bots	40	60	40%
2 PPO Agents vs 1 Defensive + 1 Aggressive + 2 Smart Bots	51	49	51%

4.7 Cycle Learning Inference Performance

From the table 4 below, we can see that when the agent is trained against their corresponding bots, the performance is the best during inference. However, cycle learning maintains the second place on the performance of each different bot, demonstrating the capability of cycle learning.

Table 4: Inference time performance for agents vs bots in isolated learning and cycle learning

Bot used in Training	Random	Aggressive	Defensive	Dodge	Average Win Rate
Random Bot	**72**	47	41	29	45.50%
Aggressive Bot	37	**81**	26	20	41.00%
Defensive Bot	55	32	**45**	28	38.75%
Dodge Bot	40	48	25	26	34.75%
Cycle Learning	**62**	**51**	**43**	**34**	**47.75%**

4.8 Team PPO vs Human (None Empirical)

In addition to all the metrics we have tested so far, we have also conducted testing against human players who previously have no experiences with game. Usually speaking, human participants are not able to defeat agents that we have trained. You visit our repository to play against our agent yourself.

5 Discussion

In this section, we will be going over the hypothesis behind the general findings we have shown in section 4 and some future directions to take. In this study, we created AlphaTank, which is an A-MARL framework capable of demonstrating reinforcement learning agent's adversarial learning abilities. Particularly, we have delivered:

1. An full RL pipeline from environment to training, creating an end-to-end training environment with low external dependency. We envision that this environment may be utilized by other researchers to investigate more on the algorithmic side of running multi-agent corporations/competitions trainings since the environment we have created is highly simple to customize but yet the RL problem itself is tricky to solve in nature.
2. A unique learning environment to test the capability of RL's learning ability under highly dynamic environments, providing an opportunity for artificial agents to fight against "human" intelligence and learn adversarially.
3. A showcase of how traditional RL algorithm may learn to use characteristics of the environment and the opponent agent/bot to fight given that their opponent or teacher has well performance.
4. A showcase of how traditional RL algorithms may fail in special cases of adversarial learning when their opponents are weak or when team collaboration is needed, highlighting the needs of more algorithmic development towards RL algorithm that can handle conditions when their opponent is weak (i.e. curriculum learning, cycle learning, or tansfer learning?) as well as when team collaboration is needed (i.e. shared parameters?)

We will be elaborating more on the above section with the detailed discussion of the following sections.

5.1 Challenging control & reasoning tasks

In nature, this tank controlling task that our agent is learning is quite complicated in nature. Though it is a discrete state space environment, the observation space grows quadratically with the number of agent that is presented in the environment, making a vast state space during team playing mode. Our agent need to reason about the opponent's action and then make optimal moves based on that, making it a very challenging task to solve, even ion discrete conditions.

In addition, the agent does not receive an image of the game board but rather solo rely on proprioceptive signals, discussed in 1 to operate, making the learning more complicated since the agent need to parse out the signals and the information it need from a vast state space.

Though discrete action control, the agent need to learn how to output the action combination of rotation + movement + shooting, which is equivalence of, making an analogy, drifting to move while shooting. This is a very hard task, especially when there are walls presented in the environment.

5.2 Bumpy reward & value curves

From section 4, we have seen that all the reward curves and value curves (not shown) seems to be more bumpy than what we would expect. We suspect that this is caused by the highly dynamic environment since most of the reward is determined by the action of this other entity in the environment as well. However, we do see some overall gradual increases in reward in some of our agents.

5.3 Adapt to the characteristic of the environment & its enemy: learning whatever the teacher is teaching

We have showcased that each of the RL agent may develop very different strategies towards not only the environment that it is in but also the opponent that it is fighting against. The opponent is been deemed as part of the environment that the agent need to learn to know how to interact with. We will be describing a few iconic strategies that our agent have developed to utilize the characteristics fo the opponent agent or the condition given by the environment. We have also attached a video rendering of each of the strategies.

- **Lure & circle to shoot**: this strategy is usually developed when our agent is fighting an aggressive bot. Since, after iterations, the agent learned that the bot will rush towards itself and shoot, it learns to wait until the bot comes to itself and then start to circle around the bot while shooting, which empirically has seem to be working in hitting the bot.
- **Backtrack & shoot**: similarly, this is also a strategy developed when fighting against aggressive bots. Our agent developed an alternative other than the first strategy mentioned above, actually a smarter one. Instead of just circling around the bot and hope that the bullet will hit it, the agent learns to move backward while the bot is chasing it and shoot directly.
- **Wait & set traps**: this strategy is usually developed during fights with the smart agent with an octagon off condition, meaning that there are walls in the environment. When this is given by the environment, our agent learned to use this characteristics and set "traps" waiting for the smart agent that comes to pursue itself, making an easy win. Very interestingly, a different strategy is learned (ones similar to how our agent fights against aggressive bots) when octagon setting is off, showcasing how the agent is actually learning to use the environment characteristics.
- **Dodging**: this strategy is developed by our agent when fighting against the dodge bot.It doesn't learn to attack, but it does learn how to dodge the bullet from the opponent agent pretty well, confirming with our previous perspective that the agent will learn "whatever the teacher is teaching".
- **Circle to dodge**: this strategy is developed while fighting against one of our hardest bot, the defensive bot. Though the performance is not as ideal,the agent develop the strategy to circle around the bot and try to find angles to attack.
- **Camping & re-bounce**: similarly, this strategy is also developed while fighting against the defensive bot. It learned to use the wall re-bounce mechanism to try to bounce off the bullet to hit the bot. In video gaming, this type of strategy is called camping, a tactic where a player obtains an advantageous static position, which may be a discreet place which is unlikely to be reached.
- **Distract & attack**: under team settings our agents sometime exhibit strategies involving collaboration, having one agent to distract the bot and the other to attack.

We have been observed that there aren't particular team strategies that was developed from the the training, which we will be discussing more in section 5.6, highlighting needs for developing more algorithms.

5.4 Teacher & enemy needs to be smart

Currently, traditional RL algorithm seems to suffer from needing a really good enemy to serve as their "teacher" in some sense, to help them know how to interact with the environment fully.

When the teacher itself is weak, it becomes much harder for the agent to learn the correct strategy given an environment. We provide thisexample of this weak smart agent fighting our RL agent as a

show case of this effect. This weakened bot agent has bounce trajectory turned off, making it only able to "see" what ever is in front of it and not being able to estimate how the bullet may re-bounce off the wall to hit its opponent. As a result, the agent learned is also weaker when comparing to regular trainings between smart bots and RL agent. The same problem manifest when training two PPO agent from scratch directly (using octagon battlefield), due to inefficient opponent performance, the agent performance itself would not be ideal either.

One potential method to mitigate this issue is through transfer learning,, by pre-training two agent first against bot, then transfer the parameters to train further against each other. Our cycle learning setup described in section 3.4.5 may be used to try to solve this issue. However, one of the challenges that might still occur under transfer learning would be what we describe in section 5.5.

Another potential method is to use team mode training. Empirically from our result, it seems that when training under a team mode, the agent seems to be learning relatively better when fighting another team of agents training from scratch, which we suspect might be due to the effect of more opportunity to interact since there are more agents presented.

5.5 Combating defensive bot

We observe the phenomenon during training that it is relatively hard to defeat our defensive bot agent as they rely on the strategy of finding an buff zone to constantly aiming and shooting at our agent.

We attempted to mitigate this issue with two different methods:

- Changing the training configuration, specifically making bullet speed faster to increase the range of attack.
- Using team setting to provide extra help the agents against bots.

From our result, seems like method one achieves higher performance, again showing the significance of how the environment setting effects the strategy that the agent can learn. We suspect that the team collaboration setting did not work as well was due to the nature of these RL algorithms that we used being single agent.

5.6 Failed to collaborate under team setting

Importantly, we found that, though team mode benefit agent to learn from agent vs agent setup, there is only little sign of team collaborations (i.e. distract and attack) that distinguishes the performance of team playing from single agent playing. Traditional RL algorithm doesn't seem to understand the concept of "team mates" and still operates solely when doing the task. More advance team playing RL algorithms may be needed to mitigate this issue and infuse the idea of team playing into the learning algorithm itself (i.e. shared parameters across agents).

5.7 Cycle learning to boost learning

We observed that cycle learning encourages the agent to develop general strategies effective against multiple opponents. This is evident in the win rate trends shown in Figure 10. Initially, the agent performs very well against random bot—the easiest opponent—but its win rate against this bot gradually declines over time. In contrast, win rates against more strategically capable bots steadily increase. This pattern suggests that as the agent is exposed to a rotating set of diverse bots, it shifts away from exploiting a single weak strategy and instead learns a more balanced policy that attempts to handle a wider range of challenges.

Further support for this comes from Table 4, which displays win rates of agents trained against specific bots. Agents trained in isolation tend to overfit to their training opponents. For instance, the agent trained solely on aggressive bot performs extremely well against it but shows significantly lower win rates against other bot types. This pattern holds true for most single-bot agents, with the exception of the agent trained on dodge bot, which generalizes slightly better—possibly due to its inherently evasive nature.

In contrast, the agent trained using cycle learning consistently performs as the second-best agent against each individual bot, and notably, achieves the highest win rate overall when averaged across all opponents. Against dodge bot, it even outperforms the specialized agent. These results suggest

that there exist strategies in the environment that are effective across multiple bot types. Cycle learning promotes such strategies by trading off peak performance against individual bots in favor of more robust, general-purpose behavior.

Another key observation underscoring the impact of cycle learning appears in the policy loss plot in Figure 10. Around the midpoint of training, there is a notable dip in loss, which corresponds to a shift in the reward structure: starting at 50% of training, the agent only receives maximum rewards if it can defeat all four bot types consecutively. This marks a turning point where the agent must move beyond narrow strategies and adopt a more generalized policy. The dip in loss reflects this adjustment, signaling that the agent is adapting to meet the stricter, more comprehensive success criteria enforced by cycle learning.

5.8 SAC performance against PPO performance

During post-training evaluations, a divergence in learned behavior exits between agents trained with PPO and SAC. Specifically, PPO-trained agents are more conservative, defensive strategy characterized by continuous rotational movement while maintaining target alignment. This behavior suggests that the PPO objective, with its clipped surrogate loss and emphasis on stable policy updates, encourages incremental refinements.

In contrast, SAC-trained agents developed a markedly more aggressive rotational strategy. The faster spinning behavior observed suggests that SAC's entropy-augmented objective fosters greater exploration and action variability. While this high-frequency spinning enabled rapid reaction to changing opponent positions, it also introduced an increased likelihood of misaiming, particularly in scenarios requiring precise alignment. This misaiming can be the result of the high stochasticity preserved in SAC's policy even at convergence, which may hinder fine-grained control in tasks with narrow aims.

5.9 Let the agent learn its own representation

Through out this whole project, one common phenomenon that we keep observing is that when we poses too many constraint upon the agents (given too many rewards, given too many observations), it essentially hinders the learning algorithm to actually learn their own computational representation. **After all, we should not impose human intelligence upon machine intelligence.** Their way of acting may not make sense to us, but it works.

5.10 Differences between our adversarial leaning & traditional generative adversarial learning (GAN)

The learning dynamics observed in our experimental setup closely resemble the Generative Adversarial Network (GAN) framework, a widely used machine learning approach, particularly in image generation. In a typical GAN, two key components interact adversarially: the Generator, which produces synthetic images designed to mimic real data, and the Discriminator, which evaluates whether an image is real (from the actual dataset) or fake (generated by the Generator). Through iterative training, the Generator refines its output to better deceive the Discriminator, while the Discriminator simultaneously improves its ability to distinguish between real and generated images. This co-evolutionary process leads to increasingly realistic outputs over time.

A similar competitive structure emerges in our experimental environment, particularly when multiple agents interact and compete. Unlike the traditional GAN framework, where competition occurs between two agents, our setup features a distributed adversarial learning process in which each agent takes on multiple roles simultaneously. As Generators, agents create strategies and behaviors to optimize their performance, while the reward function serves as the Discriminator, evaluating and guiding their learning. Additionally, each agent also functions as a teacher or real data source for others, presenting challenges that force their competitors to adapt and refine their strategies. This setup fosters an evolving learning process in which agents improve not only through their direct interactions with the environment but also through competition and adaptation in response to one another.

The expectation is that this competitive interaction will drive agents to iteratively refine their strategies, much like how a GAN's Generator improves by continuously learning to outsmart the Discrimina-

tor. However, several challenges arise in implementing this approach effectively. One key issue is balancing competition and convergence—in GANs, an overly dominant Discriminator can prevent the Generator from learning effectively, and a similar imbalance in a multi-agent setting may hinder meaningful learning. Additionally, reward function design is critical, as a poorly structured reward system could lead to agents exploiting unintended strategies rather than genuinely improving performance. In other words, the two agents will agree with a local minima and stops improving or learning. Finally, the emergent complexity of multi-agent interactions introduces unpredictability, making it difficult to control or anticipate the resulting behaviors.

6 Author's Contribution

Kaiwen Bian

- Codebase: implementation& debugging of multi-agent PPO, variational training loop adapting to fighting bots, basic implementation of SAC, basic inference functions, Wandb support, modification to base environments and reward system, modularization and structure of the codebase, communication and management of project.
- Paper: abstract, introduction, methods-general, methods-algorithms (3.3, 3.4), experiment-results, discussion-general, discussion (5.1-5.6, 5.9).

Qirui Zheng

- Codebase: SAC implementation, Video saving on cuda based systems, SAC vs PPO training implementations
- Paper: methods, results, discussion.

Ziyu Huang

- Codebase: Reward Engineer(mainly stationary, BFS, distance rewards), modified PPO, code and design BFS algorithm in tank environment.
- Paper: Discussion, results.

Weijie Zhang

- Codebase: Training time video recording with wandb logging, reward engineering, added functionality related to bullet trajectory, modified some of the single vs single inference loop.
- Paper: Related work, methods (reward engineering, MDP), results.

Zhenghao Gong

- Codebase: Environment, Gym Interface, Reward Engineering, Observation Engineering, Multi-agent Environment, debug and refactor of training code and inference code.
- Paper: Deterministic Transformations, Team Player Mode, Results.

Andrew Hudson Yang

- Codebase: Some Reward Engineering, Rule-Based Bots, BotFactory, Cycle Learning, Curriculum Learning, Debugging training code.
- Paper: Cycle Learning, Curriculum Learning, Results, Discussion.

References

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint*, 2016.

[2] Ian J Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *arXiv preprint arXiv:1406.2661*, 2014.

[3] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint*, 2018.

[4] Pablo Hernandez-Leal, Bilal Kartal, and Matthew E. Taylor. A survey and critique of multiagent deep reinforcement learning. *Autonomous Agents and Multi-Agent Systems*, 33(6):750–797, October 2019.

[5] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. `https://github.com/hill-a/stable-baselines`, 2018.

[6] Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, and João G.M. Araújo. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research*, 23(274):1–18, 2022.

[7] Sergey Levine. Reinforcement learning and control as probabilistic inference: Tutorial and review. *arXiv preprint*, 2018.

[8] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. *arXiv preprint*, 2017.

[9] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, 1994.

[10] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint*, 2016.

[11] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint*, 2017.

[12] Ming Tan. Multi-agent reinforcement learning: independent versus cooperative agents. In *Proceedings of the Tenth International Conference on International Conference on Machine Learning*, ICML'93, page 330–337, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[13] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, et al. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

[14] Maryam Zare, Parham M. Kebria, Abbas Khosravi, and Saeid Nahavandi. A survey of imitation learning: Algorithms, recent developments, and challenges. *arXiv preprint arXiv:2309.02473*, 2023.

A Appendix

A.1 Game Settings and Reward Configurations

Table 5: Game/Reward Configurations

Parameter	Value
Game Settings	
Width, Height	770, 770
Maze Width, Height	11, 11
Grid Size	Width / Maze Width
Epsilon	0.01
Rotation Speed	2
Bullet Speed	1
Bullet Max Bounces	6
Bullet Max Distance	300
Max Bullets	6
Bullet Cooldown	300 ms
Stationary Epsilon	3
Map & Tank Settings	
Use Octagon	True/False (depends on task)
Rotation Degree	3
Tank Speed	10
Tank Width	30
Tank Height	24
Human Tank Configurations	
Tank1 - Team	TeamA
Tank1 - Color	Green
Tank1 - Mode	Human
Tank1 - Keys	A (Left), D (Right), W (Up), S (Down), F (Shoot)
Tank2 - Team	TeamB
Tank2 - Color	Red
Tank2 - Mode	Human
Tank2 - Keys	Left (Left), Right (Right), Up (Up), Down (Down), Space (Shoot)
Reward Configurations	
Hit Penalty	-50
Team Hit Penalty	-5
Opponent Hit Reward	50
Victory Reward	200
Wall Hit Penalty	0
Stationary Penalty	0
Move Reward	0
Trajectory Hit Reward	40
Trajectory Distance Reward	10
Bullet Away Penalty	0.02
Bullet Close Reward	0.01
Visualization & Rendering	
Visualize Trajectory	False
Render Aiming	True
Render BFS	True
Visualize Explosion	True

A.2 Team Player Configurations

Table 6: Team Player Configurations

Configuration	Tank	Team	Mode	Bot Type / Keys
Team Configurations				
Team Configs	Tank1	TeamA	Agent	-
	Tank2	TeamA	Agent	-
	Tank3	TeamB	Agent	-
	Tank4	TeamB	Agent	-
Pair Configs	Tank1	TeamA	Agent	-
	Tank2	TeamB	Agent	-
Team vs Bot Configurations				
Team vs Bot	Tank1	TeamA	Agent	-
	Tank2	TeamA	Agent	-
	Tank3	TeamB	Bot	Smart
	Tank4	TeamB	Bot	Smart
Team vs Bot (Hard)	Tank1	TeamA	Agent	-
	Tank2	TeamA	Agent	-
	Tank3	TeamB	Bot	Aggressive
	Tank4	TeamB	Bot	Smart
	Tank5	TeamB	Bot	Defensive
Team vs Def. Bot	Tank1	TeamA	Agent	-
	Tank2	TeamA	Agent	-
	Tank3	TeamB	Bot	Defensive
Bot Team Configurations				
Bot Teams	Tank1	TeamA	Bot	Smart
	Tank2	TeamA	Bot	Smart
	Tank3	TeamB	Bot	Aggressive
	Tank4	TeamB	Bot	Aggressive
	Tank5	TeamC	Bot	Defensive
	Tank6	TeamC	Bot	Defensive
Mixed Team Configurations				
Mixed Teams	Tank1	TeamA	Human	WASD + F (shoot)
	Tank2	TeamA	Bot	Smart
	Tank3	TeamA	Agent	-
	Tank4	TeamB	Human	IJKL + P (shoot)
	Tank5	TeamB	Bot	Smart
	Tank6	TeamB	Agent	-
	Tank7	TeamC	Bot	Aggressive
	Tank8	TeamC	Bot	Smart

A.3 PPO Agent Architecture and Hyperparameters

Table 7: PPO Configurations

Component	Details
PPO Agent Architecture	
Observation Dimension	`obs_dim`
Action Dimension	`act_dim`
Critic Network	
Layer 1	Linear(`obs_dim`, 128), Tanh()
Layer 2	Linear(128, 128), Tanh()
Layer 3	Linear(128, 1)
Actor Network	
Layer 1	Linear(`obs_dim`, 128), Tanh()
Layer 2	Linear(128, 128), Tanh()
Layer 3	Linear(128, `act`) for each action dimension
Functions	
Get Value	Returns critic output for input tensor
Get Action and Value	Computes logits, samples actions, calculates log-probs, entropy, and critic value
Hyperparameter Configuration	
Project Name	`multiagent-ppo-bot`
Learning Rate	3×10^{-4}
Discount Factor (γ)	0.99
GAE Lambda	0.95
Clipping Coefficient	0.2
Entropy Coefficient	0.01
Value Function Coefficient	0.3
Max Gradient Norm	0.3
Number of Steps	512
Number of Epochs	60
Total Timesteps	300,000
Auto Reset Interval	10,000
Negative Reward Threshold	0
Epoch Check Interval	50

A.4 SAC Agent Architecture and Hyperparameter

Table 8: SAC Agent Architecture and Hyperparameters

Component	Details
Observation Dimension	`obs_dim (env.observation_space.shape[0] / num_tanks)`
Action Dimension	`action_dim (env.action_space.shape[0])`
Critic Network (QNetwork)	
Layer 1	`Linear(state_dim + action_dim, 256)`, ReLU
Layer 2	`Linear(256, 256)`, ReLU
Output	`Linear(256, 1)`
Actor Network (PolicyNetwork)	
Layer 1	`Linear(state_dim, 256)`, ReLU
Layer 2	`Linear(256, 256)`, ReLU
Mean Head	`Linear(256, action_dim)`
Log Std Head	`Linear(256, action_dim)`
Action Squash	`torch.sigmoid()`
Key Functions	`select_action(state, evaluate=False)`: returns action for given state `update(replay_buffer, batch_size)`: performs one SAC update step `get_action_and_value(state, action=None)`: returns action, log_prob, value
Hyperparameters and Configuration	
Project Name	`multiagent-sac`
Learning Rate (actor, critic, alpha)	`3e-4`
Discount Factor (γ)	0.99
Target Smoothing Coefficient (τ)	0.005
Batch Size	256
Num Steps	512
Total Timesteps	200,000
Start Steps	1,000
Update After	1,000
Update Every	50
Auto Reset Interval	20,000
Negative Reward Threshold	0.1
EPOCH_CHECK	10
Replay Buffer Capacity	1,000,000

A.5 SAC vs. PPO Hyperparameter

Table 9: Hyperparameter Configuration for SAC vs PPO

Hyperparameter	**Value**
PPO Hyperparameter	
Learning Rate	3×10^{-4}
Discount Factor (γ)	0.99
GAE Lambda	0.95
Clipping Coefficient	0.1
Entropy Coefficient	0.02
Value Function Coefficient	0.3
Max Gradient Norm	0.3
Number of Steps	512
Number of Epochs	20
SAC Hyperparameter	
Actor Learning Rate	3×10^{-4}
Critic Learning Rate	3×10^{-4}
Alpha Learning Rate	3×10^{-4}
Discount Factor (γ)	0.99
Target Smoothing Coefficient (τ)	0.005
Batch Size	256
Update After	1000
Update Every	50
Replay Buffer Capacity	1,000,000
Shared Training Settings	
Total Timesteps	200,000
Start Steps	1,000
Auto Reset Interval	20,000
Negative Reward Threshold	0.1
Epoch Check Interval	10