Building machine learning bot with ML-Agents in Tank Battle

Van Duc Dung

A thesis submitted in part fulfilment of the degree of BSc. in Computer Science with the supervision of Assoc. Prof. Phan Duy Hung.

Bachelor of Computer Science

Hoa Lac campus - FPT University

April 2022

Acknowledgments

Thanks to Assoc. Prof. Phan Duy Hung for continuous support and insightful comments. Your great assistance pointed me in the right direction and brought my work to a higher level.

Contents

List of Figures

List of Table

Abstract

In recent years, Deep Reinforcement Learning (DRL) has made great progress in video games, including Atari [1], ViZDoom [2], StarCraft [3], Dota2 [4], and so on. Those successes, coupled with the release of the Machine Learning Agents Toolkit (ML-Agents Toolkit), an open-source that helps users create simulated environments, show that DRL can now be easily applied to video games. Therefore, stimulating the creativity of developers and researchers. This research aspires to develop a new video game and turn it into a simulation environment for training intelligent agents. Experienced it with tuning the hyperparameters to get the agent the best performance for a final commercial video game product.

***Keywords**:* Reinforcement Learning, Proximal Policy Optimization, Unity, ML-Agents, Tank-game.

1. Introduction

Video games are inherently an extremely fertile ground for AI research. It is computationally complex, has rich human-machine interactions, and can generate tons of data to study. Creating agents that play as well or even as excellent as humans is one of the critical milestones marking the development of RL.

In recent years, we have seen many breakthroughs in artificial intelligence. Almost 25 years ago, an AI had defeated the strongest chess player for the first time in history, surprising the whole world [5]. Twenty years later, in 2016, AphalGO, a computer once again beat humans at Go. A board game whose total number of moves could be more than the number of atoms in the universe, a thing that was once thought to be impossible [6]. Not stopping there, two years later, OpenAIFive was developed to play a game even more hardened: Dota2. A real-time strategy game with a complexity of several tens to several hundred times Go and chess [4]. OpenAI has opened a new era for the artificial intelligence industry with many possibilities.

However, the road to applying Reinforcement Learning to commercial games is still quite far. First, the goals of game developers and scientific researchers are different. Game developers want to provide a perfect and great experience for users so that they can profit rather than build an AI that out-performs human. Even artificial intelligence agents designed to beat humans in games like OpenAI in Dota 2 are only for research purposes, not money-making games. Second, the existing AI-building methods are already good enough to create incredible games. Therefore, investing in this field is an application that game developers don't want to aim for yet.

Although artificial intelligence, specifically RL in video games, is mainly for research purposes, applying them for commercialization in video games still has a

lot of potential. This thesis aims to study the possibilities and performance of agents trained under the ML-Agent Toolkit. We designed a new game environment and made incremental improvements when we had DRL in the problem. Specific tasks include target shooting, objects collection, and obstacle avoidance. Implementations include environment design, learning process, and algorithm tuning for the best possible results. Then, we consider the possibility of trained intelligent agents as an alternative to hand-scripted bots for diverse interactions with players for a better commercial video game product.

2. Related Works

Reinforcement learning (RL), one of a training method of machine learning that is inspired by how humans and animals learn and adapt to the environment. The basic working principle of this method is based on the reward and agent received through the results of a sequence of actions. That is to say, the agent learns by trial and error, and the reward guidance behavior obtained through interaction with the environment aims to make the agent get the maximum reward [7]. In some aspects, it is comparable to supervised learning in that developers must offer algorithms well defined goals as well as set rewards and punishments. Therefore, explicit programming is a much more mandatory requirement. In the training process, the algorithm will be provided with very little information. So, RL usually has a longer time to reach the optimal solution than other methods. In this way, RL improves the strategy mainly through its experience in exploring the environment and making mistakes [8].

Given an environment that delivers valuable and realistic observations for an agent, reinforcement learning produces excellent results. The environment design requires an easy and highly configurable tool to imitate real-world ideas and test researchers' theories. Unity, one of the most popular gaming engines globally, bills itself as an ecosystem that offers a global real-time platform with detailed physics and complete usability to meet research demands. Engineering, entertainment, customer service, and other fields use the research outputs, which subsequently appear in instructional simulators and mobile or VR applications with multi-platform compatibility [9].

In order to provide all the necessary information for agents and meet the needs of research and easy environment creation, Unity has published ML-Agents toolkits. It is open-source that allows researchers and developers to create an emulator environment on the Unity editor for interacting with them through a python API. The toolkit helps us define objects and events in the environment handled by C# scripts which then log and connect to the python algorithm. One of the critical components of the toolkit is Soft Actor-Critic (SAC) and PPO, which this research will utilize [10]. Although PPO is a state-of-the-art approach, in many cases, especially when the interaction in the environment becomes complex, it will be difficult for the agent to find the optimal solution. For example, in the very first learning stage, the agent exploration is represented by random actions, which may lead to sparse rewards. In numerous instances, the sparseness of the rewards can make the agent hardly improve its policy and get stuck in random actions loop. We can add more rewards to instruct the agent on such complex problems. Or we can start from a simpler environment and then gradually increase its complexity. This concept, called Curriculum Learning, has been shown to significantly reduce training time and local minima quality [11]. In ML-Agents Toolkit, environment parameters may be added and changed during the training process. A curriculum is made of a sequence of lessons triggered by certain completion requirements. Each criterion should have a threshold to decide when the lesson ends for the chosen measure (e.g., cumulative reward or step progress). It is also possible to choose a minimum lesson duration and signal smoothing. Overall, a good curriculum lesson will result in less training time and better optimal behavior.

3. Background

3.1. Proximal Policy Optimization (PPO)

To create OpenAIFive, the OpenAI team introduced a new class of reinforcement learning algorithms called Proximal Policy Optimization (PPO), which outperforms state-of-the-art techniques while being significantly easier to deploy and tweak [12]. PPO is a Policy Gradient approach that makes use of the actor-critic method. The Policy Gradient equation is defined as below:

$$L^{PG}(\theta) = \hat{\mathbb{E}}_t\left[\log \pi_\theta(a_t|s_t)\hat{A}_t\right]$$

Policy $\log \pi_\theta(a_t|s_t)$ is the neural network that receives observations from the environment and makes outputs as actions. Through the rewards, we will calculate $\hat{A}_t$ which is the estimation of the relative value of the selected action. As appealing as it is to perform multiple gradient descent steps in the same trajectory, it frequently changes the policy outside the range that often lead to a destructively large policy.

To keep the policy gradient steps from deviating too far from the initial policy, the OpenAI team considered an algorithm called TRPO. In this method, $r_t(\theta)$ is the probability ratio between the action under the current policy and the action under the previous policy and is defined as:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}, \qquad so\; r(\theta_{old}) = 1$$

$r(\theta)$ will represent one of many action sequences of the current policy that are more likely to occur than the old policy if its value is greater than 1. Otherwise, the action has less chance of occurring if the value is between 0 and 1. Then, if we multiply $r(\theta)$ with $\hat{A}_t$ from the above, and adding KL constraint to limit the gradient step, we get the TRPO's objective function:

$$\underset{\theta}{\text{maximize}} \quad \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t \right]$$

$$\textit{subject to} \quad \hat{\mathbb{E}}_t \left[KL\left[\pi_{\theta_{old}}(\cdot\,|s_t), \pi_\theta(\cdot\,|s_t) \right] \right] \leq \delta$$

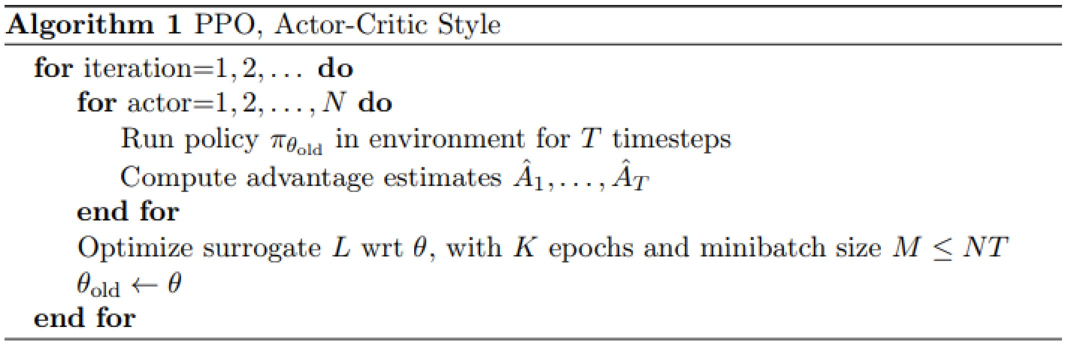
Based on the above theoretical method, Proximal Policy Optimization's main objective is clipped surrogate:

$$L^{CLIP}(\theta) = \hat{E}_t\left[\min(r_t(\theta)\,\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\right],$$

where the key idea is 'clip' the objective to prevent the gradient policy from going too far by flattening the loss function out when the probability ratio is too high. All of these terms can be obtained in the following final objective:

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_1 S[\pi_\theta](s_t)]$$

Finally, Proximal Policy Optimization algorithm works by repeating the sequence of data collection actions, compute advantage estimate and sample mini-batches in order to update the policy and fit the value function. A stochastic gradient ascent optimizer is used to update the policy, while a gradient descent technique is used to fit the value function. This method is repeated for K epochs until the environment is resolved.



```
Algorithm 1 PPO, Actor-Critic Style
for iteration=$1, 2, \ldots$ do
    for actor=$1, 2, \ldots, N$ do
        Run policy $\pi_{\theta_{\text{old}}}$ in environment for $T$ timesteps
        Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
    end for
    Optimize surrogate $L$ wrt $\theta$, with $K$ epochs and minibatch size $M \leq NT$
    $\theta_{\text{old}} \leftarrow \theta$
end for
```

Figure 3.1: **PPO Algorithm** (Schulman et al., 2017)

3.2. Curriculum Learning

One of the ways to reduce training time or even improve policy performance is to use the Curriculum Learning approach. Let's take the example of a toddler who hardly knows any math. Yet we cram into their head derivatives and integrals, hoping that they will become experts in a few years. It seems almost impossible or takes a long time to teach them. Just like humans, if an agent is trained in an environment with extremely complex tasks, it will take a long time for the agent to find meaningful actions. In fact, agents always start with random sequences of actions. In some cases, agents get stuck and cannot improve policy because the rewards are too sparse. Bengio et al. (2009) indicate that RL with curriculum strategies helps the training convergence faster or provides a better policy [11].

To summarize, the idea of Curriculum Learning is to create an environment that can vary in complexity based on the agent's current policy. For example, in the problem of moving to a goal, the environment initially has only agents and the goal. After the agent successfully completes this task, the environment increases its complexity by adding obstacles, forces, traps, etc.

3.3. Self-Play

Although in environments like Mario or Pac-man, agents act and learn through generals interacting with the environment. Competitive games like football and chess require the agent to interact with both the environment and the opponent. In that case, opponents can be hand-scripted bots or humans. However, reinforcement learning is a process that requires a large amount of data and training time. The data set is often not diverse if the algorithm only practices with standard bots because it may easily lead to overfitting. And if the algorithm practices with humans, we can't spend thousands of hours or even many years playing to provide data. Self-Play is not only an optimal solution to overcome the above disadvantages but also creates behaviors that are more complex than the environment [13]. The idea of the self-play approach is training multi-agent in a competitive environment without direct supervision; all the data that the algorithm collect is entirely from itself or an older version of itself. This approach's success is represented through many complex multi-player games, particularly in tackling StarCraft [3] and Dota 2 [4].

3.4. Unity

Unity is a 3D ultimate game engine development platform published in 2005. Not only contain lots of assets, huge community, and tutorials for attracting beginners. Unity also is a very powerful cross-platform for experts. Making it become one of the most popular among all the game platforms.

3.5. ML-Agent Toolkit

Unity is very friendly to use. It also can simulate physics and create simulator environments from simple to complex. Unity Technologies realizes the game engine as a platform with great potential for the research of Intelligent Agents and released ML-Agent Toolkit [10]. After researchers and game developers finish designing the environment, they need to define three components [14]:

Observations – are the information that the agent will collect and feed into the algorithm. To design reasonable observations, researchers can first imagine how we would solve this task if we were humans: what we need to know and what information is unnecessary. Having a sufficient and clear observation will shorten the training time and have good behavior. On the contrary, if there are too many redundant observations, it can prolong the training time and have undesirable behaviors.

Actions – which defines how the agent can interact with the environment. Actions can be continuous or discrete depending on the goal of the environment or the complexity of the environment is low or high.
Rewards signal – to let an agent know it is making correct decisions, we will add rewards every time they complete a good action. However, the reward is usually applied only after the agent has performed a sequence of actions that yielded a positive result rather than every action that led to the goal.

The ML-Agents Toolkit has four primary components:

- **Learning Environment:** which is the Unity scene that researchers and game developers create through Unity Editor. This is where the agent will be included and receive observations, make actions decision, and added rewards.
- **Python Low-Level API:** This external component is not part of Unity. It connects the environment agent to the learning trainers
- **External Communicator:** which is part of the Learning Environment and manages the connection between the Low-level Python API and agent's policy.
- **Python Trainers:** which provides the learning algorithm and allow users to select option for training.

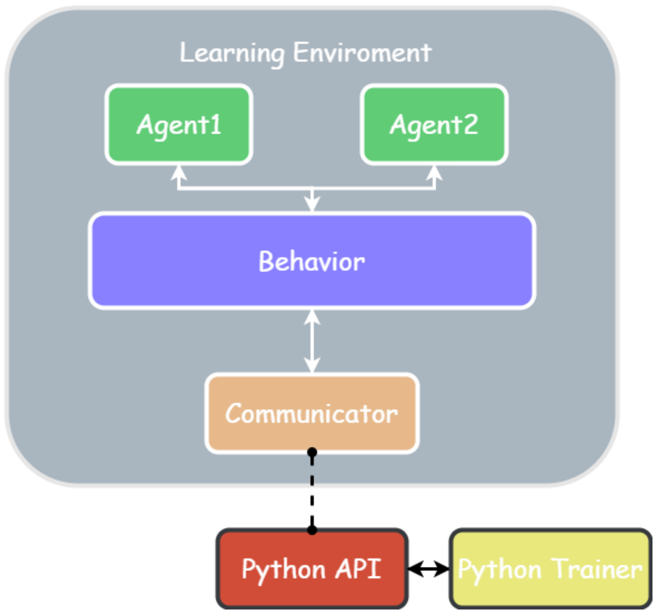
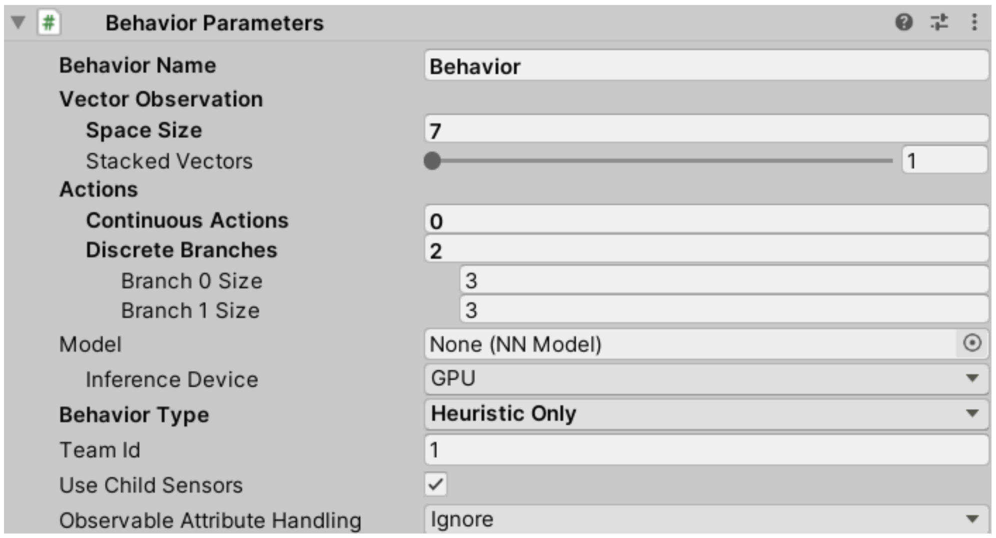


Figure 3.2: **A diagram of ML-Agents Toolkit in an Environment Learning.**

3.5.1 Behavior Parameters

When creating the agent, a script called Behavior Parameters is required to attach to it. In this component, we will define various compositions. A policy can also be implemented for the agent to perform after training.



Behavior Parameters

Behavior Name	Behavior
Vector Observation	
Space Size	7
Stacked Vectors	1
Actions	
Continuous Actions	0
Discrete Branches	2
Branch 0 Size	3
Branch 1 Size	3
Model	None (NN Model)
Inference Device	GPU
Behavior Type	Heuristic Only
Team Id	1
Use Child Sensors	✓
Observable Attribute Handling	Ignore

Figure 3.3: **Example of Behavior Parameters**

These are the concepts that are primary consider since they are in the scope of this study:

- *Behavior name:* This is where the name of the policy (behavior) is named. It must have the same as defined in hyperparameter configuration files.
- *Vector Observation:* Where we define the observations variable by C# script, Space Size represents how many of them that the agent observes. If we want the agent to have "limited" memory, we can increase the Stacked Vectors. It corresponds to the number of all last frames since the moment agents feed the observations to its algorithm.
- *Vector Action:* Where the number of Continuous Actions and Discrete Actions are defined. Note that the algorithm does not understand their action to the environment. The agent simply just tries all the actions and adjusts them through training and reward feedback.
- *Model:* For running the agent as a test or product, a policy that is already trained can be attached to this.
- *Behavior Type:* Depending on which purpose of the users, the policy will perform according to three types of behavior:
  - Default: usual mode for training.
  - Heuristic Only: Allow users to test the actions of the agent in the environment by manually controlling it.
  - Inference Only: Running trained policy.
- *Team ID:* It always is an integer number equal to or greater than 0. For training multi-agent in competitive games, set different values for any agent that is not in the same team.

4. Methodology

Self-play can be used with implementations of both Proximal Policy Optimization and Soft Actor-Critic. However, because the opponent is always changing, many scenarios appear to exhibit non-stationary dynamics from the viewpoint of a solitary agent. Self-play has a high risk of causing serious problems with SAC's experience replay system. As a result, users are advised to utilize PPO [15].

4.1 Agent Environment

Tank Battle plays out on a square map surrounded by four walls with two tanks shooting each other. Each tank has to move around the map to find the enemy, avoid rocks, take health packs, and align the cannon angle accurately (Figure 4.1). The game ends when one of them is eliminated or the time runs out. When the time runs out, that match is considered a draw. To give more varied aiming behavior, we designed the terrain with certain unevenness to create significant noise that challenges the agent's abilities. For the aiming behavior to be more varied, we designed the terrain with certain unevenness to create significant noise that challenges the agent's shooting task (Figure 4.2).

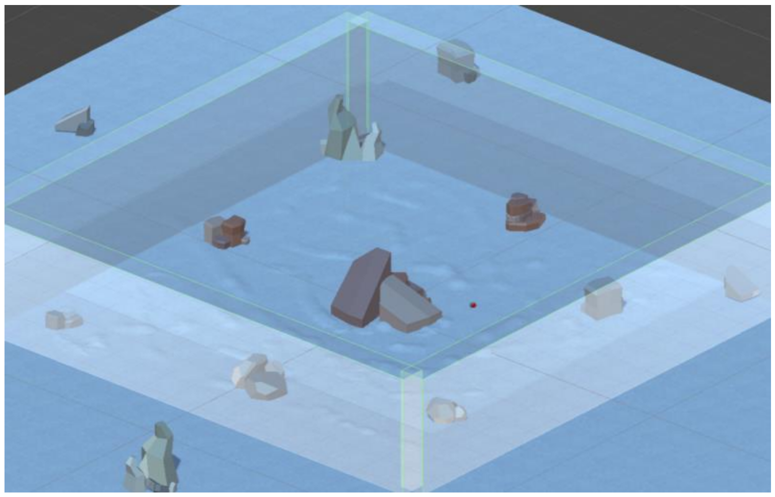


Figure 4.1: **Tank Environment**

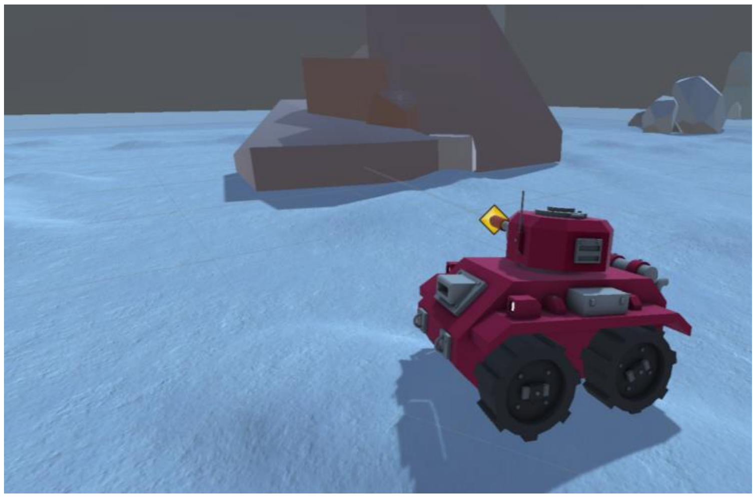


Figure 4.2: **Even Terrain**

There are two main parts of the tank, the body and the turret (Figure 4.3).

- **Body.** The tank can move like a standard 4-wheel car, including actions: forward, backward, turn left, turn right. However, in this study, to reduce the complexity, the agent will always move forward and cannot stop and only automatically goes back for a fixed time after colliding with an obstacle. It also can turn left or right 20 degrees.

- **Turret.** The turret is fixed on the vehicle's body and can rotate 360 degrees. Include two actions: rotate clockwise and counterclockwise. In addition, there is a cannon on the turret, from which the bullets are fired. Cannon can adjust the angle up and down to 5 and -5 degrees. Therefore, to accurately shoot the target, the agent needs to skillfully align both the angle of the turret and the cannon. To aid in an accurate aim, a ray cast from the cannon beams straight in the direction it is facing to the first object it hits, indicating the distance from the cannon to that object (Figure 4.4). When shooting, a bullet gameObject will be created and added a force to move straight forward of cannon direction, it also unaffected by the physic systems.

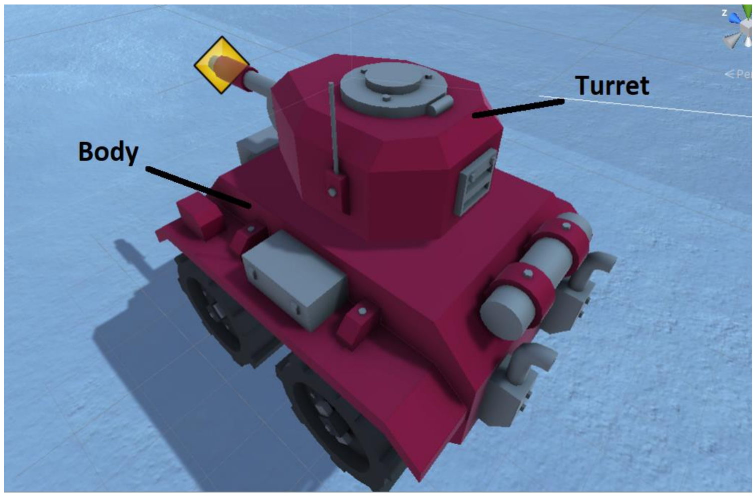


Figure 4.3: **Turret and body of the tank**.

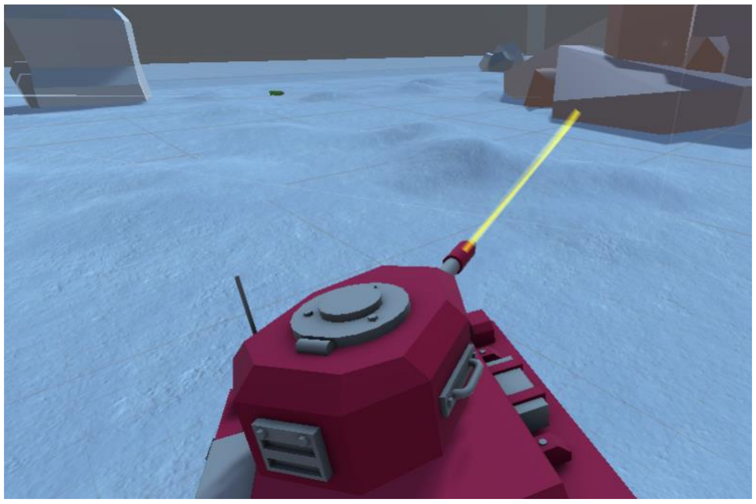


Figure 4.4: **Raycast Aiming**

4.2 Environment Learning

Although the game is designed for humans to receive information through visual input (Figure 4.5), the agent observes the environment through numbers to minimize calculation and neural networks complexity. The game is designed for players to control the tank from a third-person perspective using input devices like mouses and keyboards. On the other hand, the agent observes the environment through position, vector to the enemy, and distance provided by the Unity game engine at each time step (Table. 1). It is considered to normalize all components of the agent’s Vector Observations for a best practice when using neural networks, so all information is adjusted to range [-1, +1]. For a sequence of acts that lead to a match win, we give the agent a reward (or a punishment). Table 2 lists all of the outcomes rewards that we identify. In experiments, we maximize the reward function that includes extra signals such as colliding with obstacles and collecting health packs. When computing the reward function, we also use a method to take advantage of the problem's zero-sum construction — for example, we symmetrize rewards by deducting the reward gained by the enemy.

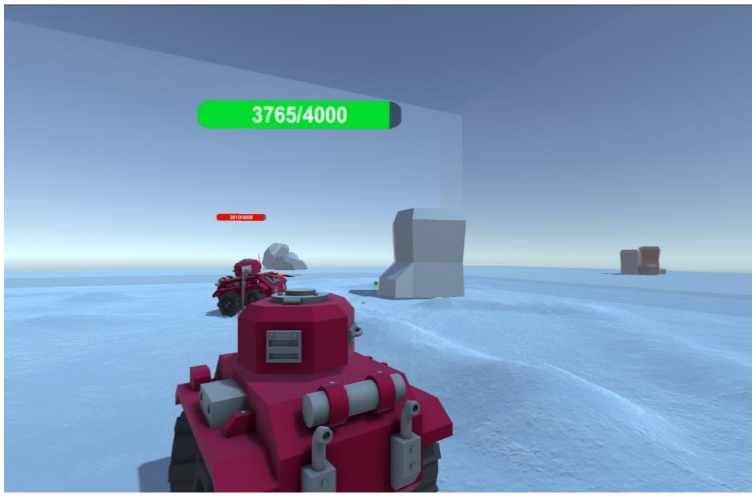


Figure 4.5: **Tank Battle’s human “Observation Space”.**

Table 1: **Vector observation.**

Current position (x, z)	2
Current health percent	1
Turret's vector direction (x, z)	2
Vector from itself to enemy (x, z)	2
Fire bullet cooldown	1
Distance from the cannon to the first object that raycast hits	1
Cannon angle	1
Enemy's current health percent	1
Enemy's velocity (x, z)	2
Distance to enemy	1
Total	14

Table 2: **Shaped Reward Weights.**

Name	reward	Description
Shooting accurately	0.1	Each bullet that hits the enemy will get a reward.
Collect a health pack	3	
Collide with obstacle	-1	Collide with walls or rocks.
Turret direction	0.003	Every step if the turret's direction is facing the enemy.
Penalty per step	-0.0001	This penalty is applied every step for making the agent kill the enemy faster.
Win	2	

In additional, for tracking obstacles and finding health packs, the agent used RayPerception Sensor whose total size of: (Observation Stacks) * (1 + 2 * Rays Per Direction) * (Num Detectable Tags + 2) = 1 * (1 + 2 * 5) * (2 + 2) = 44 (Figure 4.6).

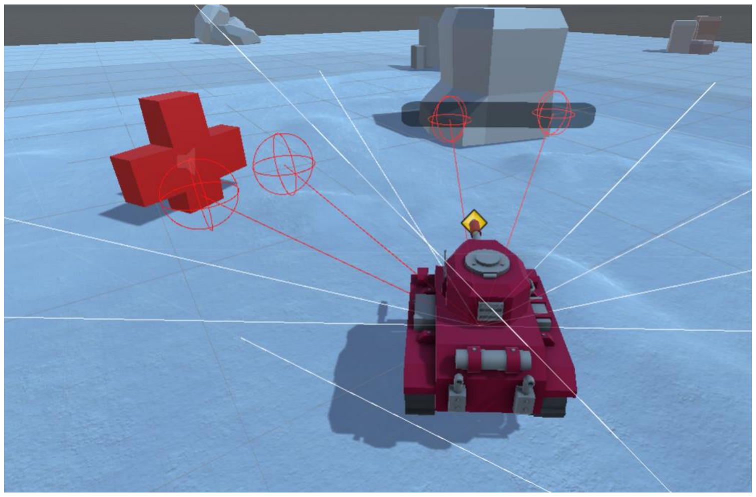


Figure 4.6: **Ray Perception Sensor.**

During inference mode, the agent's policy will determine the actions that map the current situation based on the information gathered from Vector Observation and Ray Perception Sensor. The reward in reinforcement learning is an indication that the agent has made the right series of actions. According to these rewards, the PPO algorithm optimizes the agent’s decision to maximize the cumulative reward over time. The training is divided into Episodes, each Episode is a Tank Battle match. When a match ends, all environments and reward points will be reset, and a new Episode begin.

If the Self-Play hyperparameters is not defined in configuration files, the model will train the agent with its current self. But in an environment, only one agent will learn. The other just runs the latest model to act as an opponent. Since the agent’s opponents are rapidly changing as the policy is updated every moment, it may lead to an unstable training process [16]. To have a set of slowly or unchanging adversaries with low diversity, we define Self-Play hyperparameters as in Figure 4.7

```
self_play:
    save_steps: 50000
    team_change: 200000
    swap_steps: 10000
    window: 10
    play_against_latest_model_ratio: 0.6
```

Figure 4.7: **Self-Play hyperparameters**

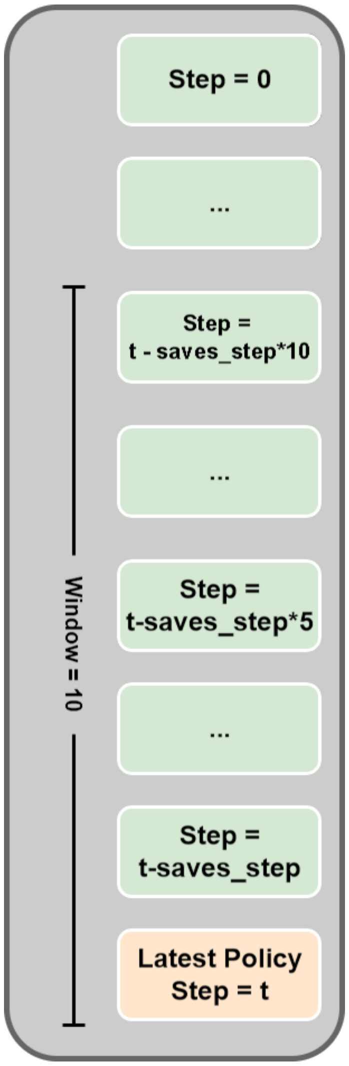


Figure 4.8: **Self-Play Snapshots**

According to the Self-Play hyperparameters, the trainer will save the policy every 50k steps, each of them called a snapshot. When learning, the agent's opponent will be chosen from the 10 (`window:10`) latest snapshots. The play_against_latest_model_ratio parameter is set to 0.6 means there are 60% probability of the fixed opponent is the agent latest model and 40% from its past version (Figure 4.8). Every 10k steps (`swap_steps:10000`), the opponent's policy will be swapped with a different snapshot. And after 200k (`team_change:200000`) steps, the learning agent and opponent teams will be switched.

5. Experiments & Results

The statistics were saved by ML-Agents Toolkit and monitored via TensorBoard during the learning lesson. It gives us the ability to track and evaluate the learning process through data that has been visualized. Over the whole step count, a graph illustrates each separate training run with chosen metrics.

In the first lesson of Curriculum Learning, the environment will not contain rocks as obstacles. So, the early task of the agent is learning to shoot and not hit walls. In this period, Self-Play is applied by the latest version model meaning that the agents are trained with its current self. After about 3 million steps, the mean reward is at its peak. The environment starts to add some obstacles, increasing the amount gradually proportional to the mean reward. (Figure 5.1). Since the agent started to grasp the rule of the game, we removed the turret direction reward and added Self-play hyperparameters so that the agent learns with its past version of itself to provide a diversity of opponents.

In Figure 5.1., the reward starts from 0, gradually increases to a peak of 4 in between steps 1M and 2M, then gradually stabilizes and maintains the oscillation amplitude from around 3. This result happens because there are not only the rewards received after each right action. The agent also gets a +2 reward for each game they win. When the policy improved, the agent's opponents grew more assertive, making each episode ending in win/lose more pronounced.

Because of this reason that it is not reliable to evaluate policy improvement through the Cumulative Reward metric. The ML-Agent toolkit provides users with another metric to evaluate agents in self-play called the ELO rating system. However, to use it, the agent's reward must be designed in a zero-sum game, and the structure of winners with a positive reward, negative for losers, and 0 for a tie. This type of reward has been implemented by using 'SetReward()' to negative two if the agent loses. Unfortunately, this implementation makes the learning unstable. Experiments show that after training the agent to learn the game's basic rules in the first lesson of Curriculum Learning, the agent knew to turn the cannon at the enemy and avoid the wall to optimize the reward. But later on, somehow, the above reward shape made the gent behavior weird. They did not spin the turret in the right direction of the enemy anymore. They just roamed around in the environment and shot aimlessly. Agent evaluation becomes more difficult without the ELO metric because empirical observations must be applied more frequently. The mean length of the episode (Figure 5.2) shows that agents are killing each other much faster, meaning they are learning to shoot more precisely. However, after adding obstacles, projectiles are regularly blocked, causing the episode's length to increase dramatically and decrease over time.

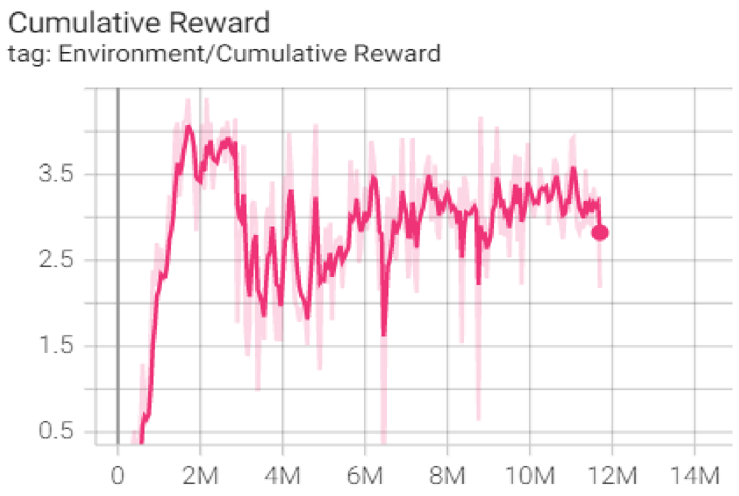


Figure 5.1: **Cumulative Reward.**

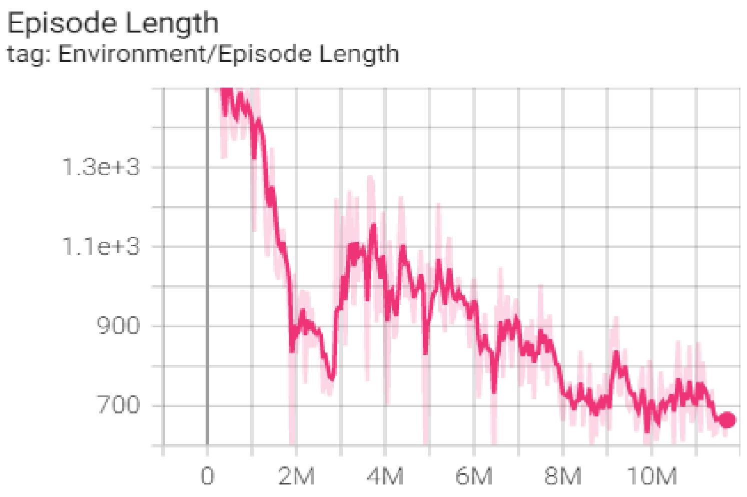


Figure 5.2: **Episode Length.**

Entropy, which measures the unpredictability of agents' decisions, is another critical metric for evaluating the policy. As the training progresses, it steadily declines, indicating a well-selected beta hyperparameter. According to Figure 5.3, the more training agent has, the fewer random actions agent will have.

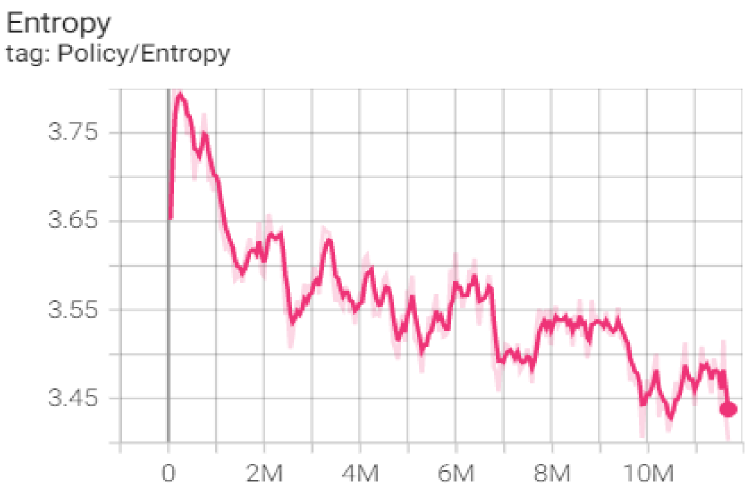


Figure 5.3: **Entropy Metrics.**

One important note is the “Normalize” hyperparameter in the configuration file. This hyperparameter is recommended to use only when there are continuous actions. It is even said to be harmful with more straightforward discrete control problems. In comparison, all of the actions in this study are purely discrete actions.

Experiments show that, after only about the first 250k steps, the neuron network somehow converges fast to some weird local minimum. Making the agent's behavior selects only one action in each action branch. Expressly, they only turn in one direction, go in a circle, and constantly rotate the turret clockwise. They do not even fire any bullets. This issue is entirely resolved after the hyperparameter switches to True.

6. Conclusions & Future Works

This study demonstrates the performance and possibilities of intelligent agent training by ML-Agents Toolkits, which is remarkably easy to use and to set-up in order to conduct learning of policies. The agent was able to learn the basic rules of the game quickly. It can avoid obstacles and walls, collect health packs, and face its turret toward the enemy. However, the way the agent observes their surroundings is not visual observations, which is very costly, making shooting a complex problem. As humans play the game through a screen and control their tank by keyboard and mouse, they can effortlessly aim and shoot precisely to trounce the agent. Although we can make the agent do even better if we increase the hidden units and improve its observations, it is pretty hard for the agent to play the game as well as humans. The reason is due to the limitations of ML-Agents itself. We can configure the training by changing Hyperparameters in the configuration file, but interfering too deeply in the neural network is not allowed. Therefore, we can conclude that ML-Agents Toolkit and Unity engine still have a high potential for commercial in video games. However, the more complex the environment, the harder the agent can learn. So causal games are most likely the best suit for this commercial due to their simplicity.

We would like to add more agents and make Tank Battle a Cooperative game for further work. In addition to shooting each other and collecting health packs, agents on the same team can also fire special bullets to heal teammates and diversify interactions and tactics. Moreover, we will also alternate entirely current the vector observation with visual observations by adding a camera following the turret so that the agent can learn the ability to aim more precisely.

References

1. Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." *arXiv preprint arXiv:1312.5602* (2013).
2. Guillaume, L., Chaplot, D.S.: Playing FPS games with deep reinforcement learning. In: Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI'17). AAAI Press, 2140–2146 (2017)
3. Arulkumaran, Kai, Antoine Cully, and Julian Togelius. "Alphastar: An evolutionary computation perspective." Proceedings of the genetic and evolutionary computation conference companion. 2019.
4. OpenAI et al.: Dota 2 with Large Scale Deep Reinforcement Learning, arXiv:1912.06680 (2019).
5. Hsu, F-H.: Behind Deep Blue: Building the computer that defeated the world chess champion. Princeton University Press, 2002.
6. Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., Hassabis, D.: A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. Science 362.6419, pp.1140-1144 (2018)
7. Sutton, Richard S., Andrew G. Barto, and Ronald J. Williams. "Reinforcement learning is direct adaptive optimal control." *IEEE control systems magazine* 12.2 (1992): 19-22.
8. Li, Y.: Deep reinforcement learning: An overview. arXiv:1701.07274 (2017)
9. Xie, J.: Research on key technologies base Unity3D game engine. In: Proceedings of the 7th International Conference on Computer Science & Education (ICCSE), pp. 695-699, doi: 10.1109/ICCSE.2012.6295169 (2012)
10. Juliani, A. et al.: Unity: A General Platform for Intelligent Agents. arXiv:1809.02627 (2020).
11. Bengio, Y., Louradour, J., Collobert, R., Weston, J.: Curriculum learning. In: Proceedings of the 26th Annual International Conference on Machine Learning (ICML '09). Association for Computing Machinery, New York, NY, USA, 41–48. DOI: https://doi.org/10.1145/1553374.1553380 (2009)
12. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O.: Proximal Policy Optimization Algorithms. arXiv:1707.06347 (2017)
13. Bansal, T., Pachocki, J., Sidor, S., Sutskever, I., Mordatch, I.: Emergent complexity via multi-agent competition. arXiv:1710.03748 (2017)
14. Unity-Technologies. “ML-Agents/ML-Agents-Overview.md at Main · Unity-Technologies/ML-Agents.” *GitHub*, 15 Apr. 2021, github.com/Unity-Technologies/ml-agents/blob/main/docs/ML-Agents-Overview.md.
15. Foerster, J., Nardelli, N., Farquhar, G., Afouras, T., Torr, P.H.S., Kohli, P., Whiteson, S.: Stabilising experience replay for deep multi-agent reinforcement learning. In: Proceedings of the 34th International Conference on Machine Learning - Volume 70 (ICML'17). JMLR.org, 1146–1155 (2017)
16. Unity-Technologies. “ML-Agents/Training-Configuration-File.md at Main · Unity-Technologies/ML-Agents.” *GitHub*, 15 Dec. 2021,

https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Training-Configuration-File.md.