ARITIFICIAL INTELLIGENCE | COURSE 2023-2024

Team Project

IMPLEMENTATION OF REINFORCEMENT LEARNING ALGORITHM FOR SOLVING THE BATTLE CITY ARCADE GAME

Group:

Alberto Ibernón Jiménez
Daniel Sotelo Aguirre
Jiajun Xu
Vladyslav Korenyak

INDUSTRIALES
ETSII | UPM

January 15th, 2024

Contents

1. Introduction

This report details the final project for the Artificial Intelligence course within the MSc in Automation and Robotics at UPM. The project centers on developing a Reinforcement Learning (RL) agent capable of playing the Battle City arcade game. It provides an overview of the project, including objectives, problem definition, methodology, and key design decisions. Detailed descriptions of each development stage, the final system architecture, and the results are also presented. Conclusions and future directions are discussed in the closing sections. The project's GitHub repository, containing installation instructions and user guidance, can be found at https://github.com/danisotelo/RL_tank_battalion.

1.1 Project Objectives

The goal of this project is to create an AI agent using reinforcement learning that can proficiently play Namco's classic game, Battle City. The agent's training involves shaping a reward system that aligns with the game's objective: defending the castle and eliminating enemy tanks.

1.2 Main Challenges

In this section we briefly describe the main challenges posed by this project:

- **Game Complexity:** A detailed discussion on the intricacies of Battle City is provided in chapter 2.1. The game complexity affects every aspect of the project.
- **Adaptation of the Game to Reinforcement Learning Training Environment:** Adapting a game to a RL environment involves several steps such as: creating a wrapper to translate the game's state and player's actions into a format understandable by RL algorithms; defining the action and observation spaces to specify the possible actions and what the agent perceives from the game; implementing the step function to update the game state and calculate rewards based on agent actions; and creating a reset function to reinitialize the game to its starting state after each episode. A more detailed guide, used as a template for our work, can be found at https://gymnasium.farama.org/tutorials/gymnasium_basics/environment_creation/.
- **Learning Reinforcement Learning from Scratch:** Another remarkable challenge is the learning curve associated with understanding and applying reinforcement learning from a null level. This includes not only understanding the theoretical aspects but also translating them into a practical application that successfully trains the AI agent. It will be essential to understand which are the best RL algorithms that exist and that are applicable to this game in order to achieve the best performance (PPO, DQN, etc.).
- **Observation Space Determination:** Deciding on the RL agent's input is crucial as it significantly influences its performance. The design choice could range from providing a single image of the game to multiple images, varying resolutions, or even a vector containing additional game information. The nature of this observation space is pivotal in determining the speed and efficiency of the agent's learning process, and in some cases, its ability to learn at all.
- **Reward Policy Formulation:** Crafting an effective reward policy is vital in reinforcement learning. This task entails strategically designing rewards and penalties to steer the agent's actions towards the desired outcomes. An inaccurate or poorly structured reward policy can lead to extensive, unproductive training periods. Such inefficiency is particularly critical given the project's time constraints and deadlines.
- **Computational Restrictions:** Reinforcement learning, particularly in its demand for GPU resources, is inherently computationally intensive. Crafting an efficient system becomes a paramount challenge when the team lacks access to high-performance computers. This hardware limitation can significantly extend training times, impacting the overall efficiency and timeline of the project.

1.3 Initial Academic Literature Review

Since the beginning of Reinforcement Learning (RL), there have been significant advancements and noteworthy contributions in the field. Key developments include the application of Deep Q-Network (DQN) in Atari games [1] where Mnih et al. demonstrated that the DQN agent was able to surpass the performance of all previous algorithms and achieve a level comparable to a professional human gamer across a set of 40 games, using the same algorithm, network architecture and hyperparameters. Another groundbreaking achievement was AlphaGo, developed by Silver et al. [2], which used a combined approach of RL and Monte Carlo tree search to defeat a world champion in the game of Go. Recent advancements have also been seen in multiplayer online gaming such as StarCraft II by Vinyals et al. [3], marking a significant step in applying RL to real-time strategy games.

Nowadays, the field of RL is rapidly evolving, with thousands of papers being published on it daily. It is worth mentioning that on the first day of 2024 alone, Science Direct reported a publication of 2056 papers related to reinforcement learning! For this reason, even though the general public only has knowledge of the most remarkable results, there are plenty of advances such as the study published last October in [4] where it understands the vulnerabilities of Deep Reinforcement Learning (DRL) and introduces the "E-value" concept to mitigate them which ended up showing positive experimental results. Such ongoing research underscores the dynamic and ever-expanding nature of RL studies.

Thanks to these advances, very useful tools have been developed to facilitate research in this field. The most relevant ones for this project include OpenAI's Gym and Stable Baselines libraries. **OpenAI's Gym**, now rebranded as **Gymnasium**, provides a standardized set of environments for testing and comparing RL algorithms, offering a diverse range of scenarios from simple control tasks to complex strategy games. It is designed to simplify the process of developing and testing new RL algorithms, making it an invaluable resource for researchers and developers alike. The link to their website is: https://gymnasium.farama.org/. **Stable Baselines** library, on the other hand, is a collection of high-quality, efficient, and easy-to-use RL implementations. It is build to be compatible with Gymnasium, providing improved interfaces and documentation, which makes the development and testing of RL agents more accessible, especially for those new to the field. The link to their website is: https://stable-baselines3.readthedocs.io/en/master/index.html.

Alternative tools and libraries that also contribute significantly in this realm include TensorFlow Agents (TF-Agents), a flexible library for Reinforcement Learning in TensorFlow, and Ray RLlib, a scalable RL library. TF-Agents offers a wide variety of algorithm implementations and is particularly useful for those already familiar with TensorFlow. Ray RLlib, meanwhile, is designed for high-performance RL and can scale from a single CPU to a large cluster, making it suitable for projects with substantial computational demands.

1.4 First Decisions

One of the primary decisions we faced at the outset of our project was selecting the most appropriate programming language and tools for adapting the game for reinforcement learning and developing the RL agent. After careful consideration, we chose **Python** due to its widespread use in the AI and ML communities, as well as its compatibility with leading RL libraries. Our decision was significantly influenced by the discovery of **Gymnasium and Stable Baselines**, which offer a robust and user-friendly framework for RL applications. One team member's previous experience with the challenges of developing RL models from scratch, particularly the complexity of using PyTorch for model implementation, guided us towards these tools. Stable Baselines, known for its simplicity and ease of use compared to building custom models in PyTorch, emerged as the ideal choice. Additionally, the integration of **TensorBoard** with Stable Baselines was a decisive factor, as it provides valuable insights into training metrics and agent performance. This blend of accessibility, comprehensive features, and informative analytics made Gymnasium and Stable Baselines the optimal choices for our project, aligning well with our team's skill set and the project's technical requirements.

The next decision taken was the particular RL technique we are going to use in our project. The Stable Baselines library currently provides 13 different models, with the most popular being PPO and DQN. After evaluating the range of models available in the Stable Baselines library, our team decided to implement the **Proximal Policy Optimization (PPO)** technique for our project. PPO's has a widespread recognition for its balance between simplicity, ease of implementation, and strong performance, which made it a compelling choice. The decision was further reinforced by an OpenAI article (https://openai.com/research/openai-baselines-ppo), which highlighted PPO's status as the default RL algorithm at OpenAI due to its user-friendliness and effectiveness.

Apart from its recognition, two relevant factors influenced our decision: PPO's **stability during training** and its **lower memory requirements**. PPO uses a clipping mechanism that prevents the policy from changing too much, ensuring a more consistent learning trajectory by avoiding drastic policy updates. This leads to a more stable training experience, reducing the likelihood of significant performance variances between iterations. Additionally, PPO is more memory-efficient compared to other models, especially those requiring large replay buffers like DQN. This efficiency is due to PPO's on-policy learning approach, which operates on smaller batches of data and continually updates the policy based on recent experiences rather than relying on a vast history of past data. This is a crucial aspect since our observation includes an image, and our computers do not have the capability to store the typical 1M observations to properly train the DQN. These aspects of PPO, combining stable training with reduced memory demands, significantly align it with our project's constraints and objectives.

Several papers influenced the next decisions we took at the start of the project. The first reference that approached us to the RL problem understanding has been *"Playing Atari with Deep Reinforcement Learning"*, where it focus on **reward policies and convolutional neural networks trained with Q-learning**. More info in **ataripaper**.

Knowing the technique which will be applied and that there are a huge amount of daily publications, several proposals stand out for enhancing PPO's capabilities. One such proposal is the **Exploratory Intrinsic with Mission Guidance Reward (EMR)** approach, which seamlessly integrates intrinsic rewards from exploration mechanisms with reward redistribution in RL. This approach, detailed in [5], is particularly effective in balancing exploration and exploitation in RL tasks with sparse and delayed rewards. This paper inspired us to adopt the following strategies:

- **Intrinsic Reward Integration:** Implementing entropy-based intrinsic incentives to encourage exploration, such as rewarding the agent for exploring different map areas in Battle City.
- **Uniform Reward Redistribution:** Addressing sparse and delayed rewards with a uniform redistribution method, providing frequent and smaller rewards for intermediate achievements like destroying an enemy tank.

Another noteworthy approach is the **Motivational Curriculum Learning Distributed Proximal Policy Optimization (MCLDPPO)**, as discussed in [6]. This approach applies curriculum learning to enhance an agent's combat capabilities in aerial combat scenarios, illustrating its potential for application in gaming environments like Battle City. This paper inspired us to adopt the following strategies:

- **Motivational Curriculum Learning:** Based on the unsatisfactory performance of the agent, it is proposed to monitor the performance to guide reward adjustments, such as increasing rewards for maintaining base security or destroying enemy tanks. Additionally, higher penalties can be defined for critical errors such as losing the base, to emphasize the importance of key actions.
- **Interruption Mechanism:** If the agent faces high-risk situations, like being surrounded by enemies or bullets, this mechanism activates to reassess the situation, enabling more informed decision-making, such as changing tactics or retreating.

Finally, a reference that has served as a very valuable support has been the video tutorial called *"Training AI to Play Pokemon with Reinforcement Learning"* by Peter Whidden which provides a deep understanding about the RL capabilities to resolve role games just by frame recording. In this case, the agent starts from totally random actions. After 5 years of training and different reward policies attends, the RL agent is able to excess completing the Pokemon game. More info in: https://www.youtube.com/watch?v=DcYLT37ImBY

2. Methodology and Further Relevant Design Decisions

In the present section, the methodology followed during the project is briefly described, as well as each of the steps that have been taken in order to reach the final program solution. Additionally, the most relevant design decisions that have been taken in each of the program development steps are detailed.

2.1 Battle City Rules

An overview of the game's rules is necessary to configure the main aspects of our RL algorithm. Battle City is a 2D shooter video game where the player controls a tank to defend its base (also called castle). The player must shoot the enemy tanks while protecting the base from them until eliminating all of them. The game was adapted from the following GitHub repository: https://github.com/raitisg/battle-city-tanks. We invite the reader to play the game a few round to get the feeling of how it works.

Player's tank: states and actions

The player tank is destroyed when impacted by an enemy tank bullet. It has a number of bonus lives that give extra opportunities after being destroyed. By default, this parameter has been set to 3 lives. The player tank admits four states: (1) alive; (2) dead, a bonus life is required to continue the game; (3) exploding, graphical states when a enemy bullet is received before dying, (4) spawning, starting state when no enemy bullets can be received. Tank position and rotation are updated every step. Player's tank admits the following actions:

- **Move Direction:** up / down / right / left / stay.
- **Shoot:** shoot a bullet / not shoot a bullet.

Enemy's tanks

The maximum number of simultaneous enemy tanks on the map is constrained by default to 4 tanks. Their behaviour is very simple. If the tank does not collide with any obstacles, it continues forward. Otherwise, a random direction is chosen to avoid the obstacle. There are 3 main attributes applicable to both enemy and player tanks:

- **Speed:** continuous value that defines the tank velocity (pixel per step). By default, speed = 2 (2 pixels per step).
- **Superpower:** (0) no superpowers; (1) faster bullets; (2) fire 2 simultaneous bullets; and (3) can destroy steel. By default, "0".
- **Health:** parameter that indicates the number of bullets needed to destroy the tank. "0" indicates dead and "400" requires 4 standard bullets to die. By default, "1" for enemy and player tanks.

There are four types of enemy tanks: (a) Basic: speed = 1, (b) Fast: speed = 4, (c) Power: superpower = 1 and (d) Armor: health = 400.

Obstacles (tiles)

There are some obstacles that complicate the tank motions and obstruct the interaction among the bullets. The types of obstacles are (1) indestructible stone walls; (2) destructible brick walls; (3) grass, which reduce visibility of the tanks motion, (4) water, which is non traversable but bullets can pass through it, and (5) frozen.

Base (castle)

The game ends once the enemy tanks destroy the player's castle, which is protected by brick walls by default. It is always located in the bottom-middle part of the map.

Bullets

Both player and enemy tanks fire bullets in the direction they last moved. Each tank is limited to firing one bullet at a time. A new bullet can only be fired after the previous one has made an impact. Within the game's code, player and enemy bullets are differentiated, interacting with various elements on the map as follows:

- **Collisions with other bullets:** Enemy bullets can destroy the player's bullets but do not interact with other enemy bullets; similarly, player bullets can destroy enemy bullets but not other player bullets.
- **Collision with tanks:** Bullets from enemy tanks can destroy the player's tank, subtracting a life. Conversely, bullets fired by the player can destroy enemy tanks.
- **Collision with obstacles:** Bullets can destroy brick wall obstacles, creating empty spaces through which tanks and other bullets can pass.
- **Collision with the base:** Any bullet that hits the player's base destroys it with a single shot, thereby ending the game. This applies to both enemy and player-fired bullets.

Levels

The game presents 35 levels where each level contains different obstacle distributions which provide dynamics environments. A couple of levels of the game can be seen in Figure 2.1.

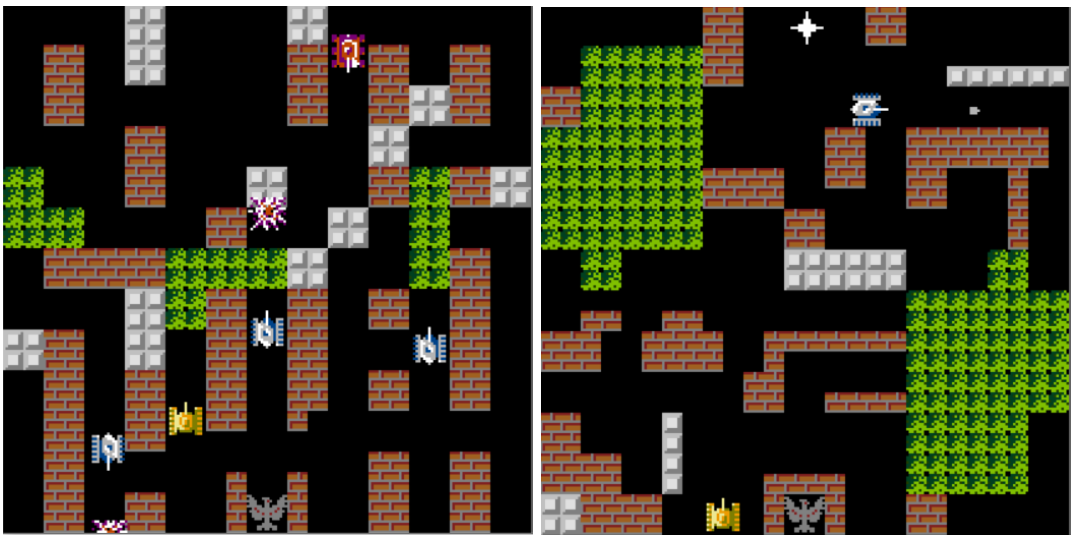

Figure 2.1: Battle City elements at: (a) Level 2 ; (b) Level 3. Game code accessed from external repo in Github: https://github.com/raitisg/battle-city-tanks.

Bonuses

Bonuses are temporary special features that enhance the player's tank for a short duration. To acquire these bonuses, the player's tank must collide with bonus icons scattered across the map.

- **Grenade:** Activating the grenade power-up instantly eliminates all enemies currently on the screen, including Armor Tanks.
- **Helmet:** The helmet power-up grants the player's tank a temporary force field, making it invulnerable to enemy shots.
- **Shovel:** The shovel power-up transforms the walls around the castle from brick to stone, preventing enemies from penetrating and destroying the player's base.
- **Star:** Each star power-up enhances the offensive capabilities of the player's tank, with a maximum of three upgrades. The first star allows the tank to fire bullets as rapidly as power tanks. The second star enables firing up to two bullets simultaneously. The third star allows the player's bullets to destroy steel walls.
- **Tank:** The tank power-up awards an extra life.
- **Timer:** The timer power-up temporarily freezes time, enabling the player's tank to safely approach and eliminate enemy tanks until the effect dissipates.

2.2 Initial Adaptation to Gymnasium and the PPO Algorithm

The code for the initial adaptation is available in the `tanks_initial.py` file. This adaptation was guided primarily by the Gymnasium documentation and greatly assisted by the (much appreciated) YouTube series "Reinforcement Learning with Stable Baselines 3" by the user *sentdex*. Key aspects of this adaptation include:

- Translating of the core logic of the game into the `step()`, `reset()`, and `_get_obs()` functions of the Gymnasium environment. This task presented a significant challenge due to our lack of prior experience in this area.
- Defining the action space, observation space, and initial reward policy.
- Implementing the Stable Baselines' default PPO model, with functionalities for **training and loading models** provided in the `agent_train.py` and `agent_load.py` files.

The main aspects of our PPO algorithm were configured as it follows:

- **Action Space:** This defines the set of possible actions that an agent can take in a given environment and defines what the agent can do at each step. In our case, the decision was straightforward; we chose a **discrete two-dimensional vector**: the first number (1/0) for shoot or not shoot, and the second number (ranging from 0 to 4) for moving up, right, down, left, or staying in the same place.
- **Observation Space:** This defines the set of possible observations the agent can perceive from the environment, encompassing everything the agent uses as input for decision-making. Initially, we used the **grayscaled game image at full resolution**, as it contained all necessary information for gameplay. More sophisticated approaches were developed subsequently (see following sections).
- **Reward Policy:** These are the rules that assign rewards or penalties to the agent based on its actions and the state of the environment. This component is crucial as it provides feedback to the agent on the effectiveness of its actions in achieving the objectives. Similar to the observation space, we began with a basic approach and continually refined it, as will be discussed in future sections.
- **Hyperparameters:** These are configuration variables that detail the learning process of the model. We started with the **default hyperparameters**, as initially, we lacked the expertise for proper tuning. Also, based on our initial research, we determined that tuning the parameters might impact the results by at most 10% relative to the predefined hyperparameters set by Stable Baselines.

The flow of the whole training process was established as it follows:

1. The environment is loaded using the `gymnasium.make()` function.
2. The agent is initialized with either `PPO()` for a new model or `PPO.load()` for loading a previously saved model.
3. The learning process starts with the `model.learn()` function, which encompasses the following steps:
   (a) The `reset()` function is called at the start of each game to initialize all game variables.
   (b) An action, generated by the PPO model, is used as input for the `step()` function. This function executes one frame of the game at a time, using the action as though it were the player's input.
   (c) During the `step()` function, the reward is determined by modifying the `self.reward` variable.
   (d) After updating the game state, the `_get_obs()` function is called to obtain the new state of the game. This new state, along with the reward and a flag indicating whether the game has ended, is stored in RAM for subsequent training, and a new action is generated.
   (e) The model is updated every fixed number of steps, as defined by the `TIMESTEPS` variable.
4. The model is saved at intervals specified by the `SAVE_INTERVAL` variable. A new learning session then begins, continuing until the `TOTAL_TRAINING_TIMESTEPS` is reached or the program is manually stopped.

Next, some details about the algorithm and monitoring tools are discussed.

Randomness and Entropy of Decisions

Initially, the RL agent generates random actions following a uniform distribution. This means that there is an equal probability of choosing any of the 5 move actions [up, left, down, right, stay] or deciding between firing and not firing. The agent learns the consequences of each decision through the observation space and the rewards obtained. It uses a gradient descent method to update the weights of the neural network that defines the model, thereby altering the future probability of choosing each action in a given context. As a result, the learning process is not deterministic and may yield different results in separate runs.

The level of randomness for each action can be adjusted by modifying the `ent_coef` parameter when initializing the model for training. Additionally, setting the "deterministic" parameter to true when using the `model.predict()` function in `agent_load.py` removes the random element completely.

TensorBoard and Metrics

During the training of the RL agent, which can take extended periods, having a visualization tool is crucial for monitoring progress and determining if modifications are needed, or to identify the model achieving the best performance for later use. Two primary PPO metrics indicate the agent's performance and learning progress:

- **Episode Length Mean (`ep_len_mean`):** This metric represents the average duration of games before the game is finished. The episode length refers to the number of time steps required for the agent to complete an episode.
- **Episode Reward Mean (`ep_rew_mean`):** This metric signifies the average reward the agent obtains across the training episodes. The episode reward is the total sum of rewards the agent receives from the environment during an episode.

TensorBoard provides a comprehensive suite of tools for efficient and effective result representation. To visualize training results, open a new terminal and execute:

```
tensorboard --logdir=logs
```

This command will extract and plot the PPO metrics gathered during RL training. Episode Length Mean and Episode Reward Mean are plotted against the number of iterations (game runs). These metrics serve as indicators of the RL agent's learning convergence. A process is typically considered complete when variations are observed to be less than 1% of the nominal value.

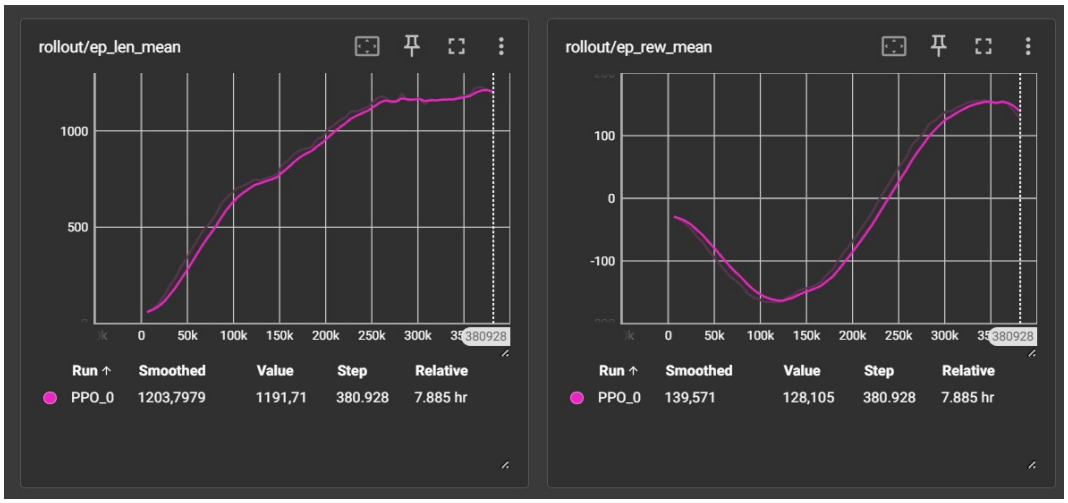

Figure 2.2: Example of RL ELM and ERM metrics visualizer in TensorBoard.

2.3 Initial Reward Policy and First Training Trials

One of the fundamental challenges in Reinforcement Learning is the selection of an effective reward policy, as it plays a crucial role in the success of the training process. While the desired behavior of the player's tank may be obvious to a human player familiar with the game rules, or even to a manually coded AI bot that can interpret the current state of each game element, the RL agent operates differently. It initially lacks this contextual knowledge and must infer it from interactions with the environment. Therefore, the RL agent must learn through trial and error gradually understanding the consequences of its actions based on the rewards received. Initially, the reward policy may not be fully optimized but serves as a foundation for the agent to start learning which actions are beneficial or detrimental. Table 2.1 outlines the initial rewards at the start of the training.

Table 2.1: Preliminary Reward Policy at the Beginning of the RL Training Process

Category	Description	Value
Positive	Select different direction from previous frame	+0.05
	Stay alive	+0.05
	Fire in forward direction	+0.1
	Kill an enemy tank	+50
	Kill all enemy tanks to finish the stage	+100
	Complete the stage in efficient time	+50
Negative	Game Over	−80
	Select same direction from the previous frame	−0.05
	Fire in forward direction after unnecessary fire in previous frame	−0.05
	Destruction of player's tank (lose a life)	−5
	Complete the stage in very long time	−50

Frame Skipping

During the initial training trials, it was observed that **training was very slow**, primarily because the agent was learning from every single frame. This inefficiency necessitated the adoption of strategies to accelerate the training process, which we identified during our initial research phase. Many frames in each game are visually appealing for human players but offer redundant information for an RL agent, thus not contributing significantly to its learning. The technique of frame skipping involves the algorithm selectively discarding a portion of these frames to enhance computational efficiency. This process requires careful customization to balance learning effectiveness with processing demands. We experimented with various frame skipping intervals, such as discarding every second or every fourth frame. Ultimately, we moved away from this initial frame skipping approach due to its limitations. Instead, we increased the velocities of tanks and bullets by fivefold. This adjustment not only made the game execution faster but also had the beneficial side effect of accelerating the training process.

Frame Stacking

In addition to the previous adjustments, we observed through some initial failed trials that the agent seemed to be only aware of the game's state at the moment of a specific frame. Possibly due to the limitations of the internal convolutional networks and the resolution of the input image, the agent struggled to discern crucial details, like the direction of bullet or enemy movements. Frame stacking, a technique that accumulates consecutive frames, was employed to address this issue. We set the number of consecutive frames for the RL agent to four. This was feasible without overburdening the model because we also reduced the resolution of the map. In fact, this approach significantly decreased the size of the model because we stacked the four frames on different channels of the same Box element in the observation space. This allowed the PPO model to use the same convolutional network for processing the frames, resulting in much lighter final models (reduced from around 400MB per model to 100MB). Frame stacking enables the RL agent to comprehend not just the current positions of game objects but also their positions in previous frames (time steps), providing the agent with vital information about the temporal evolution of player's tank, enemy tanks, and bullet positions.

First Major Learning

Further trials led to additional insights. We noticed recurrent behaviors and issues during training that required more refinements to the reward function and the observation space. Figure 2.3 (a) presents a case where the player's tank remained stationary near the base. The tank was excessively penalized for various actions, leading it to learn a strategy of avoidance and minimal action, engaging only enemy tanks that came directly over it. This behavior highlights a common challenge in RL: **the balance between punishment and reward**. We learned that excessive punishment or undue rewards for inaction can lead to suboptimal learning outcomes.

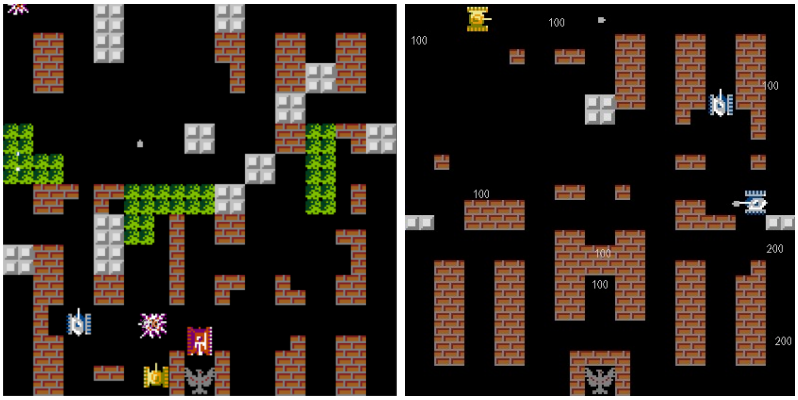

Figure 2.3: Different behaviours observed along the training process.

Second Major Learning

In an attempt to resolve the previously observed issue of the player's tank's inactivity, we introduced a **reward for completing the stage** (eliminating all enemy tanks) quickly (as detailed in Table 2.1). This was designed to motivate the player's tank to move and seek out new targets. However, this adjustment led to another unforeseen behavior: the player's tank began to position itself statically in the upper part of the map, where new enemy tanks spawn. Intriguingly, the tank would immediately move to the top of the map at the start of the game and then become immobile. While this strategy resulted in the rapid destruction of new enemy tanks, it led to the neglect of its base, ultimately yielding unsuccessful outcomes (as shown in Figure 2.3 (b)).

New Training Strategies

For resolving this new issue, we decided to train the agent with the player's tank **initialized at random locations**, diverging from its static starting position in the original game. This change was intended to encourage the tank to explore the map and exhibit varied behavior, fostering adaptability. We hoped this variation would teach the RL agent to make strategic decisions, like whether to pursue an enemy tank or return to defend its base.

In parallel with the random spawn strategy, we explored another approach: **gradual training on individual, simpler tasks**. This method aimed to streamline the learning objectives, allowing the agent to acquire skills and strategies incrementally. By breaking down the overall learning goal into smaller, more manageable tasks, we aimed to facilitate **more effective** and **step-by-step learning outcomes**.

One such strategy involved creating an environment with a **fortified base**, **disabling bullets**, and introducing only **one enemy**. The objective was to encourage the agent to learn **map navigation** while penalizing collisions and rewarding the tracking of enemy movements. However, this approach initially led to an unintended behavior where the agent opted to **remain stationary**, primarily due to excessive penalties for collisions. Adjusting the penalties to discourage inactivity didn't yield the desired results either; the agent continued to show a preference for staying immobile.

Another exploration focused on fortifying the base and training the agent to **eliminate tanks**. In this setup, the rewards were exclusively tied to destroying enemy tanks, with penalties for the agent's destruction and for exceeding a high threshold of steps per level. The penalty for taking too many steps was minimal but cumulative, nudging the agent towards faster completion of objectives. Additionally, a dynamic reward system was implemented, where the speed of eliminating all enemies directly influenced a final reward magnitude. The quicker the agent cleared a level, the higher this final reward. This system aimed to balance the agent's aggressiveness with efficiency, motivating it to eliminate enemies swiftly while still being cautious about survival.

During this phase, an interesting behavior was observed. The agent developed a technique of **alternating movement up and down while firing randomly**, hoping that enemy tanks would cross its line of fire. However, this developed technique was not particularly effective for several reasons. Firstly, it relied heavily on the chance that enemy tanks would move into the firing path, making the process **slow** and **inefficient**. Secondly, this approach doesn't include the **agent avoidance of enemy bullets**, which was expected to be learnt due to the heavy penalties for dying.

It was also tested policies focusing on enemy tank tracking in an environment featuring only one enemy tank and a fortified base. The agent was rewarded minimally through the **Manhattan distance** for approaching the enemy tank. Additional rewards were allocated for **aligning the agent's aim towards the enemy tank** and successfully shooting and destroying it. Penalties were imposed for the **agent's destruction**.

This policy led to an unexpected learning outcome. The agent discovered that **remaining stationary** was a viable strategy to accumulate points, as the enemy tank, moving randomly, often came

within proximity, inadvertently increasing the agent's reward. This behavior was further reinforced by the fact that movement carried the risk of death and subsequent penalties. As a result, the agent did not effectively learn to actively hunt and kill the enemy tank. Instead, it adopted a **passive strategy** where it remained stationary, earning points with minimal effort and avoiding the risk associated with more aggressive tactics.

Throughout our experimentation, we have tested numerous policies and variations, including combinations of different strategies or variations of the values of the rewards or penalties. For instance, we implemented a policy focused on protecting the base, penalizing the agent if an enemy tank entered a certain radius around the castle. This policy was blended with previously mentioned strategies, while also adjusting the values of penalties and rewards to evaluate potential improvements. However, these extensive training efforts, spanning several days, did not yield fruitful results.

Without achieving satisfactory outcomes, we hypothesized that the challenges might be attributed to two main factors: either the agent was not receiving sufficiently frequent or precise rewards for each step (the rewards were too sparse), or the observation space provided was inadequate for the agent to learn effectively within our required timeframe.

After considerable training and numerous variations by the team, it became apparent that it was time to explore more advanced options.

2.4 Advanced Training Trials

2.4.1 Code Adaptation for External AI Bot-Based Reward Policy

For the first advanced training trials, it was explored the concept of RL assisted by a functional AI bot. The idea was to have the AI bot indicate the RL agent the necessary actions to complete the game up to a certain point where after reaching this point, the RL agent would continue with a simple reward policy to refine its gameplay and become capable of completing the game at all levels.

Since the core of our project is focused on RL and not the AI bot itself, we sought out relevant literature and found a GitHub repository https://github.com/munhouiani/AI-Battle-City/blob/master/ai.py that featured an AI bot capable of playing the game up to level 4. This bot was tested and the execution was impressive, with precise movements and a smooth navigation through the map. Simplifying, using the current state of the game world, the AI bot operation consisted in the following:

- **Decision Making**: The bot sorts enemies based on their distance to the player and the castle to prioritize the enemies who are more prompt to destroy the castle.
- **Path Finding**: By employing A* algorithm, the bot finds the most efficient path to its target. The algorithm uses a heuristic that consists on the Manhattan distance to estimate the shortest path and a cost funcion to keep track of the total travel cost based on the steps of the bot. It explores neighboring nodes and chooses the path with the lowest combined cost and heuristic score.
- **Combat Strategy**: The bot uses a function to determine if it is aligned with an enemy to decide when to shoot. Additionally, there is a function to avoid incoming fire while considering the opportunity to return fire.

However, as the bot only reached level 4, improvement was programmed, enabling it to complete up to level 5 with the following modifications:

- **Enhanced Navigation and Positioning**: This improvement focuses on dynamically calculating the bot's new position based on its current direction and speed. The enhancement not only reduces collisions in a more advanced and sophisticated manner but also seamlessly integrates with the mechanism for avoiding incoming fire. This dual functionality ensures smoother navigation and more effective evasion strategies in complex game scenarios.

- **Advanced Combat Strategy**: Building upon the enhanced navigation capabilities, the bot is now adept at assessing potential collisions with various game elements such as walls, enemies, obstacles, and even the game's boundaries. This improved awareness significantly refines both navigation and evasion tactics. The bot can now alter its movement paths during evasion to avoid unintended collisions, thereby reducing the likelihood of in-game fatalities. Additionally, a counterattacking strategy has been implemented, enabling the bot to respond with fire when strategically aligned with an enemy.
- **Preventing Friendly Fire on the Castle**: The AI agent now incorporates a predictive bullet path functionality that allows it to anticipate the trajectory of its shots, a critical feature for evaluating whether the fire might accidentally hit the base. This predictive analysis is key in preventing unintended damage to the base, ensuring that offensive actions do not compromise the mission's success.

Having the AI bot prepared, its integration was the next step which consisted in modifying the environment of the RL to capture the actions taken by the AI bot to define the reward policy in the RL training process. It is important to highlight the use of multiprocessing for the integration as it allows AI bot to operate asynchronously from the RL agent, ensuring the interaction of RL agent and AI bot with the game environment simultaneously.

The first reward policy implemented in this setup involved assigning rewards based on the alignment of the RL agent's actions with those of the AI bot. Specifically, a reward of +0.2 is given if the RL-controlled agent **shoots in the same scenario as the AI bot**, and +0.1 if it **moves in accordance** with the AI bot's movement. This reward system's objective was to encourage the RL agent to mimic the strategic choices of the AI bot. This reward policy was seen to not be appropriate because it was observed that the tank never fired, since the AI bot most frequent decision about shooting is not to shoot (0), so the **tank learned not to shoot**.

After eliminating this part of the reward policy, it was observed that the tank performed the same strategy mentioned in a previous section where it involved the technique of alternating movement up and down while firing randomly, making a sort of vertical "sweeping" to try to kill all the enemy tanks that cross its line of fire. Though it was observed the tank won the game in some occasions, it was still not taking into account its base, which resulted in premature game end.

2.4.2 Enhanced Observation Space

In subsequent trials, we enhanced the observation space to better assist the PPO algorithm in comprehending the game state. This enhancement was partly inspired by a linear model from a previous project titled *"AI Agent for Battle City"* (https://web.stanford.edu/class/archive/cs/cs221/cs221.1192/2018/restricted/posters/diaozh/poster.pdf), and it evolved gradually alongside our project's development. The new observation space, as depicted in Figure 2.4, includes the following modifications:

- **obs_frames:** This is the stack of 4 consecutive frames, as already discussed in Section 2.3.
- **player_position:** Encodes the state of the player's tank, including x and y position, its direction, remaining lives, distance to castle, distance to nearest enemy tank, and direction to avoid bullets.
- **enemy_position:** This encodes the state of up to four enemy tanks (the maximum number that can be present simultaneously). For each tank, the encoding includes its x and y positions, direction, status (1 for alive, 0 for dead or non-existent), distance to the castle, distance to the player, and the direction relative to the player if it is in line with him.
- **bullet_position:** This encodes the states of up to six bullets currently active on the map (the maximum number that can be present simultaneously in our setup). For each bullet, the encoding includes its x and y positions, direction, owner (0 for player, 1 for enemy), distance to the castle, distance to the player, and the direction relative to the player if it is in line with him.

```
self.observation_space = gymnasium.spaces.Dict(
    {
        "obs_frames": gymnasium.spaces.Box(low=0.0, high=1.0, shape=(4, self.width, self.height), dtype=np.float64),

        "player_position": gymnasium.spaces.Box(low=0.0, high=1.0, shape=(7,), dtype=np.float64),  # Normalized values
        "enemy1_position": gymnasium.spaces.Box(low=0.0, high=1.0, shape=(7,), dtype=np.float64),
        "enemy2_position": gymnasium.spaces.Box(low=0.0, high=1.0, shape=(7,), dtype=np.float64),
        "enemy3_position": gymnasium.spaces.Box(low=0.0, high=1.0, shape=(7,), dtype=np.float64),
        "enemy4_position": gymnasium.spaces.Box(low=0.0, high=1.0, shape=(7,), dtype=np.float64),

        "bullet1_position": gymnasium.spaces.Box(low=0.0, high=1.0, shape=(7,), dtype=np.float64),
        "bullet2_position": gymnasium.spaces.Box(low=0.0, high=1.0, shape=(7,), dtype=np.float64),
        "bullet3_position": gymnasium.spaces.Box(low=0.0, high=1.0, shape=(7,), dtype=np.float64),
        "bullet4_position": gymnasium.spaces.Box(low=0.0, high=1.0, shape=(7,), dtype=np.float64),
        "bullet5_position": gymnasium.spaces.Box(low=0.0, high=1.0, shape=(7,), dtype=np.float64),
        "bullet6_position": gymnasium.spaces.Box(low=0.0, high=1.0, shape=(7,), dtype=np.float64),

        "prev_action": gymnasium.spaces.MultiDiscrete([2, 5]), # [no shoot, shoot], [move up, down, right , left, no move]
        "ai_bot_actions": gymnasium.spaces.MultiDiscrete([2, 5]),
        "flags": gymnasium.spaces.MultiBinary(5),
        "enemies_left": gymnasium.spaces.Box(low=0.0, high=1.0, shape=(1,), dtype=np.float64),
        "heatmap": gymnasium.spaces.Box(low=0.0, high=1.0, shape=(13, 13), dtype=np.float64),


        #celdas alrededor que están libres?

    }
)
```

Figure 2.4: Complete observation space definition Gymnasium.

- **ai_bot_actions:** These are the actions that an AI bot would take in the current scenario, serving as a potential guide for the RL agent.
- **flags:** This encodes circumstantial information about the game state with the following tags:
  - **castle_danger:** Set to 1 when an enemy tank is close to the base. This tries to address a situation that often arises when the player's tank, focusing on eliminating enemy tanks near their spawn area, fails to destroy one, allowing it to travel towards the player's base. The introduction of this flag is meant to guide the agent towards balancing offensive actions with the strategic defense of its base. In Figure 2.3 (b) this situation can be seen, where the player's tank positions itself at the top of the map, potentially neglecting the defense of its own base.
  - **stupid:** Set to 1 if any of the player's bullets are on a trajectory to hit the castle. This flag is used to generate a negative reward when the player's tank inadvertently targets its own base. In early training, it was noted that self-destruction of the player's base was a common reason for losing.
  - **player_collision:** Set to 1 if the player collides with an obstacle.
  - **hot:** Set to 1 if the temperature is too high.
  - **bullet_fired:** Set to 1 if there is a player's bullet active on the map.
- **enemies_left:** Encodes the number of enemy tanks remaining in the level. The level is completed when this count reaches 0.
- **heatmap:** Represents a fictitious map providing a 'temperature' value for each cell, which changes based on the player's tank movements. The temperature in a cell decays when the tank leaves and increases when the tank remains stationary. This mechanic is designed to discourage the behavior of camping in one spot by implementing a penalty for inactivity.

An integral part of enhancing the observation space is the normalization of observations. **Normalization** is crucial as it ensures that all input values fed to the PPO algorithm are within a similar scale, typically between 0 and 1. This process helps in stabilizing the learning process and improving the convergence speed of the model. In our approach, each element of the observation space is divided by its maximum possible value, as demonstrated in Figure 2.5. This scaling down fits the values into the 0-1 interval, which aligns with the defined space Box element in the Gym environment. For instance, positions

```
################## OBSERVATION DEBUGGING ####################

return {
    "obs_frames": np.array(self.obs_frames) / 255.0,

    "player_position": np.array(self.grid_position) / np.array([29*16, 29*16, 4, 4, 59*16, 59*16, 4]),
    "enemy1_position": np.array(self.enemy_positions[0]) / np.array([29*16, 29*16, 4, 1, 59*16, 59*16, 4]),
    "enemy2_position": np.array(self.enemy_positions[1]) / np.array([29*16, 29*16, 4, 1, 59*16, 59*16, 4]),
    "enemy3_position": np.array(self.enemy_positions[2]) / np.array([29*16, 29*16, 4, 1, 59*16, 59*16, 4]),
    "enemy4_position": np.array(self.enemy_positions[3]) / np.array([29*16, 29*16, 4, 1, 59*16, 59*16, 4]),

    "bullet1_position": np.array(self.bullet_positions[0]) / np.array([29*16, 29*16, 4, 1, 59*16, 59*16, 4]),
    "bullet2_position": np.array(self.bullet_positions[1]) / np.array([29*16, 29*16, 4, 1, 59*16, 59*16, 4]),
    "bullet3_position": np.array(self.bullet_positions[2]) / np.array([29*16, 29*16, 4, 1, 59*16, 59*16, 4]),
    "bullet4_position": np.array(self.bullet_positions[3]) / np.array([29*16, 29*16, 4, 1, 59*16, 59*16, 4]),
    "bullet5_position": np.array(self.bullet_positions[4]) / np.array([29*16, 29*16, 4, 1, 59*16, 59*16, 4]),
    "bullet6_position": np.array(self.bullet_positions[5]) / np.array([29*16, 29*16, 4, 1, 59*16, 59*16, 4]),

    "ai_bot_actions": np.array(game.ai_bot_actions),
    "prev_action": np.array(self.prev_action),

    "flags": np.array([obs_flag_castle_danger, obs_flag_stupid, obs_flag_player_collision, obs_flag_hot, obs_flag_bullet_fired]),
    "enemies_left": np.array([len(enemies) / 20]),
    "heatmap": np.array(self.heat_map) / 25,

}
```

Figure 2.5: Complete observation space in `_get_obs()` function.

and distances, which can vary widely, are normalized to prevent disproportionately large values from dominating the learning process. By applying this normalization uniformly across all observations, we ensure that the RL agent receives a balanced and consistent representation of the game state, facilitating more efficient and effective learning.

2.4.3 Game Simplification: Random Map Generator and Virtual Restrictions

In an effort to streamline the training process and focus on key learning aspects, we implemented a significant simplification of the game environment. One component of this simplification was the development of a **random map generator**, as facilitated by the `generate_random_map.py` script. This generator creates maps composed exclusively of brick blocks. Additionally, we experimented with **varying the initial player position**, either placing it randomly across the map or restricting it to the lower part, as illustrated in Figure 2.6.

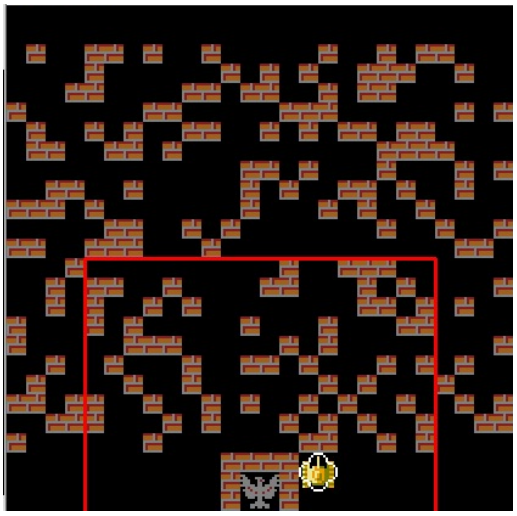

Figure 2.6: Random map with a virtual restriction for player's tank

To accelerate the pace of each game and enhance learning efficiency, we **halved the enemy spawn time** and significantly **reduced the number of enemy tanks**—ranging from the original 20 down to a minimum of 2, depending on the specific trial. Simplification extended to enemy types as well; **only basic tanks** were included, deliberately excluding armored, fast, or power tanks to reduce complexity. Furthermore, we **removed all bonuses** from the game, ensuring a more straightforward and controlled learning environment for the RL agent. These modifications collectively aimed to create a more focused and effective training setup, concentrating on the core mechanics and strategic elements essential for the agent's development.

2.4.4 Hyperparameter Tuning

Hyperparameters are configuration settings used to control the behavior of machine learning algorithms. Unlike model parameters, which are learned from the training data, hyperparameters are set prior to the training process and generally remain constant throughout it.

To enhance our agent's performance, we delved into the plethora of hyperparameters available for PPO, aiming to identify the most effective settings for our project. The default parameters, typically optimized for Atari games, may not be ideal for our more complex game. We consulted multiple resources to determine which hyperparameters were most relevant, with a particular focus on:

- This repository, which outlines the meaning for most hyperparameters: https://github.com/gzrjzcx/ML-agents/blob/master/docs/Training-PPO.md.
- The paper **Hyperparameters in Reinforcement Learning and How To Tune Them [7]**, which provided insights into the impacts of adjusting various hyperparameters. We paid special attention to their findings related to the MiniGrid game, as it is the most similar one to our game.

After this research, we decided that the best hyperparameters would be the following:

- `learning_rate` = $\mathbf{10^{-5}}$: The learning rate determines how much the weights of the model change at each iteration of the training process, influencing how quickly or slowly a machine learning model converges to a solution. The default learning rate was 3e-4. We chose to lower it because we observed that the agent converged too quickly into a suboptimal policy.
- `gamma` = **0.995**: Gamma represents the discount factor for future rewards, indicating the importance the agent assigns to future versus immediate rewards. A default value of 0.95 implies that after 10 steps, the model values future rewards at $0.95^{10} \approx 60\%$ of their original value. However, in our strategic game context, such as firing a shot or protecting the base, rewards or consequences often manifest beyond a 10-step horizon. Therefore, we increased gamma to 0.995, allowing the agent to more effectively consider the delayed rewards and penalties inherent to the game and emphasize longer-term strategies.
- `gae_lambda` = **0.85**: Gae_lambda is a complex parameter that helps to balance the accuracy and consistency of the training process by controlling how future rewards are estimated and used to guide the agent's learning. We found in the [7] paper that the default value of 0.95 was not optimal for most situations and that 0.85 seemed to produce better learning outcomes.
- `ent_coef` = **0.001**: The entropy coefficient encourages exploration by forcing the agent to make more varied actions, preventing premature convergence to suboptimal policies. Based on [7] and [8] papers and also on the experience of other users (https://youtu.be/1ppslywmIPs and https://www.reddit.com/r/reinforcementlearning/comments/), the default value of 0 was not appropriate. We chose 0.001 as a sweet spot between 0 and higher values like 0.1.
- `n_steps` = **24576**: This parameter defines the number of steps the agent takes before updating the model, and it is equal to the `TIMESTEPS` variable in our code. It is directly related to the RAM memory necessary to train the model. Because of this, each of us used a different value of `TIMESTEPS`, being 24576 the one we used to reached the results in section 2.5.

- `batch_size` = **6144**: This refers to the number of samples used in each update of the model. We chose an unconventionally big size of 6144 to provide a balance between computational efficiency and the granularity of the learning updates, since one game can take as long as 1000 episodes. In the context of PPO, after collecting experiences over `n_steps`, these experiences are usually shuffled and then divided into smaller subsets or batches, each of which is used to perform a gradient update. The batch_size determines the size of these subsets, and the `n_epochs` determines how many times we use the collected data in that training session to update the weight of the model. A smaller batch size means the model is updated more frequently with smaller sets of data, while a larger batch size allows for more data to be processed in each learning update. Usually larger batch sizes lead to more stable training, thus our decision to make it as big as possible.

The remaining hyperparameters were kept at their default values as our research and initial trials did not indicate a clear benefit to adjusting them.

2.5 Best Solution Reached

The journey of how we gradually learned and implemented the various changes throughout our project is detailed in Section 3.2.3. As we progressed with the RL training process, we continually evolved our reward policy, culminating in the final set of values presented in Table 2.2. These values represent the culmination of our extensive empirical testing and learning. It's important to highlight that we rescaled these values by dividing them by 10, a common practice in reinforcement learning to maintain numerical stability and efficiency.

Interestingly, the most successful solution, as also discussed in Section 3.2.3, did not rely on the hyperparameters outlined in Section 2.4.4. Instead, this solution was achieved using the original maps from levels 1 and 2 of the game. This approach deviated from our earlier trials with modified environments and hyperparameters, indicating that sometimes simpler, more authentic settings can lead to more effective learning outcomes.

Our experience underscores the dynamic and often unpredictable nature of AI development in reinforcement learning. While empirical tests and adjustments form the backbone of our strategy, unexpected insights and outcomes like these highlight the importance of flexibility and adaptability in our approach.

Table 2.2: Reward Policy Towards the End of the RL Training Process

Category	Description	Value
Positive	Move in the previous direction without a collision	+0.05
	"Fire" action in line with the AI bot "Fire" action	+0.2
	"Direction" action in line with the AI bot "Directino" action	+0.1
	Kill an enemy tank	+5
	Kill an enemy tank close to base	+15
	Kill all enemy tanks to finish the stage	+50
	Get a bonus reward	+1
Negative	Game Over by player dying	–15
	Game Over by castle destroyed	–50
	Get a stupid flag	–0.1
	Destruction of player's tank (lose a life)	–5
	Trying to shoot it's own base	–0.1
	Stay in the same place	up to -1.5

3. System Architecture

3.1 Overview

In Figure 3.1, a scheme of the program architecture is shown. This Figure represents the repository folder structure. The different relevant program files, together with their functionalities and content can be checked.

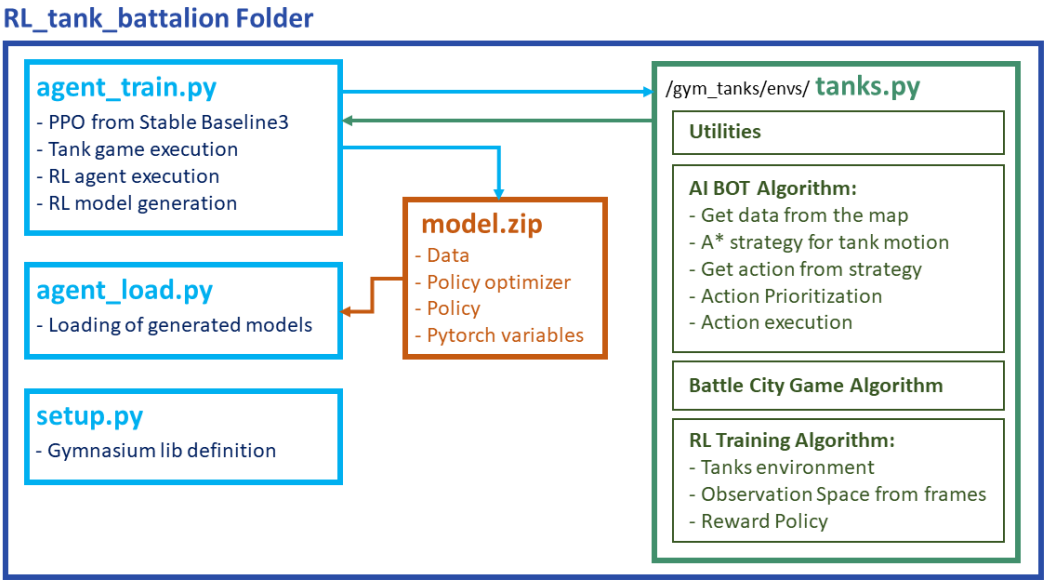

Figure 3.1: Battle City based on RL Architecture.

3.2 How Does the Reinforcement Learning Algorithm Work?

3.2.1 Agent Training: `agent_train.py`

In order to train the RL agent, the user can go to `agent_train.py` and select the test that he wants to run changing the varible `test_name`. This script admits the possibility of starting a training from a file which contains the weight from previous trainings. For this functionality, the user shall specify the number of initial steps of the model that wants to load in `start_steps`.

Finally, to start a training run, the user shall execute the steps that are indicated in detail, together with the installation steps, in the User Guide of the GitHub repository (https://github.com/danisotelo/RL_tank_battalion), briefly run: `python agent_train.py` and then run on a separate terminal `tensorboard --logdir=logs` to visualize the training information.

3.2.2 Agent Loading: `agent_load.py`

More information about the visualizer library is provided in Sections 2.3 and 2.4. In case the user does not want to train the RL agent but just wants to load a model to review its performance, the user shall go to `agent_load.py` file and select the model he wants to load in `models_dir`. Then, the user shall run: `python agent_load.py`.

3.2.3 Tank Game Code: `tanks.py`

The user can modify the RL agent rewards in the file `gym_tanks/envs/tanks.py` (in the final section, in the `step()` function of the environment RL training class).

This script contains the three main parts of the program: non-RL AI agent class, Battle City game algorithm and RL agent algorithm based on rewards policy. Due to the high complexity of Gymnasium environment, it was decided to implement in a combined way with the rest of the Battle City code and also with the AI bot code.

- **Utilities:** from line 1 to 144.
- **Non-RL AI Bot:** from line 145 to 710.
- **Tanks Game Algorithm:** from line 711 to 3027.
- **RL Agent:** from line 3028 to the end.

4. Results

This chapter presents some of the outcomes observed during the training, focusing on both quantitative results such as the rewards obtained, and qualitative observations such as behaviors or strategies exhibited by the AI. Visual data representations provide additional insights into these results. A comprehensive record of the results is available in the logs folder, which contains detailed logs accessed through TensorBoard. This section will primarily discuss the trials labeled as "Serious Try", where a more structured approach to the training process was adopted.

Note: Due to changes in the reward structure between trials, the graphics across different trials are not directly comparable.

First trials (Serious Try 1-5)

The first 5 trials, visualized in the Figure 4.1, were conducted with the set of hyperparameters as described in the section 2.4.4. At this stage we used a simplified observation space that only included the current image, the previous action and a few flags. Our strategy involved sampling multiple agents over around 200k steps and then extensively training the best-performing agent. The first three trials exhibited consistent results among themselves, so we decided to just train two more doubling the training time.

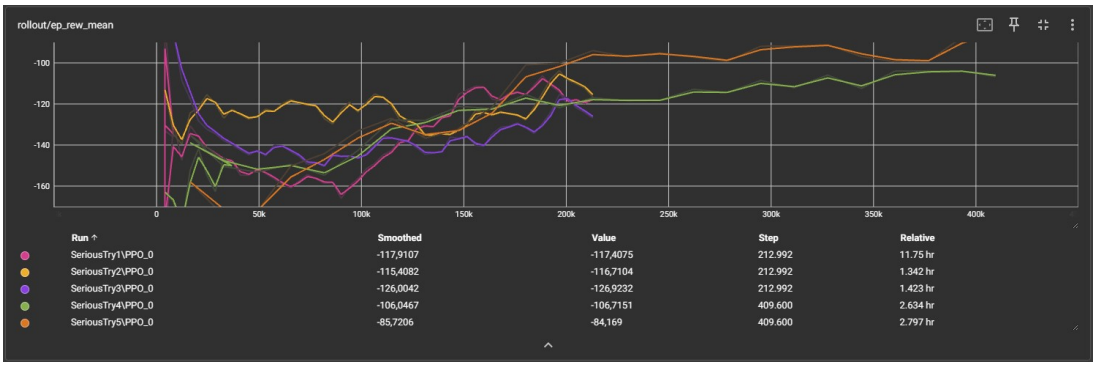

Figure 4.1: ERM of the first 5 trials.

During these trials we observed that the agent was not learning effectively. We noticed that the explained variance value of our train data was 0. The explained variance is a useful metric that represents the proportion of the reward that the agent can attribute to its observation space and actions. A very low or negative value suggests that the observation space may not include all relevant information necessary for effective learning. We later discovered that this was because of the reward associated to the heatmap. This was a crucial insight, as it highlighted the importance of ensuring that the observation space contains sufficient information for the agent to infer the correct reward.

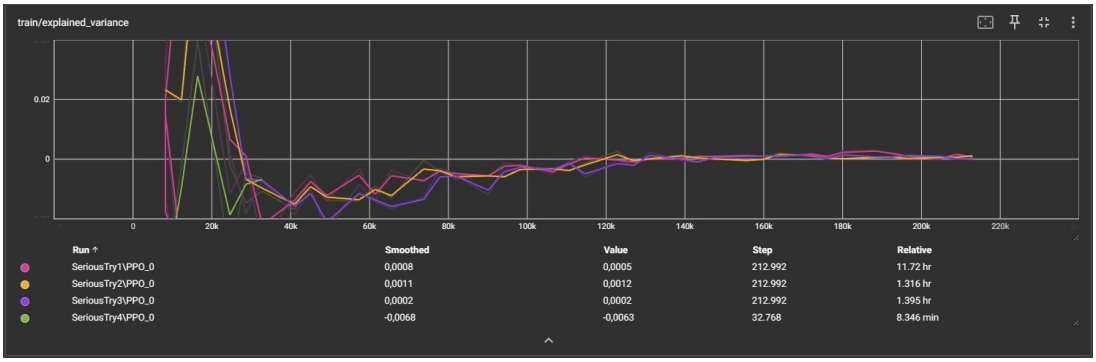

Figure 4.2: Explained variance of the first 5 trials.

Incorporation of Heatmap (Serious Try 7 and 8)

In the seventh trial, we introduced heatmap data into the observation space to address the issue of low explained variance and made adjustments to some rewards. As seen in Figure 4.3, the initial rewards appeared lower, but over time, the agent seemed to learn to avoid staying in the same place, indicating some degree of improvement. However, the agent's movement was erratic, and it failed to effectively defend the base or actively pursue enemies. Moreover, the agent frequently exhibited self-destructive behavior by destroying its own base.

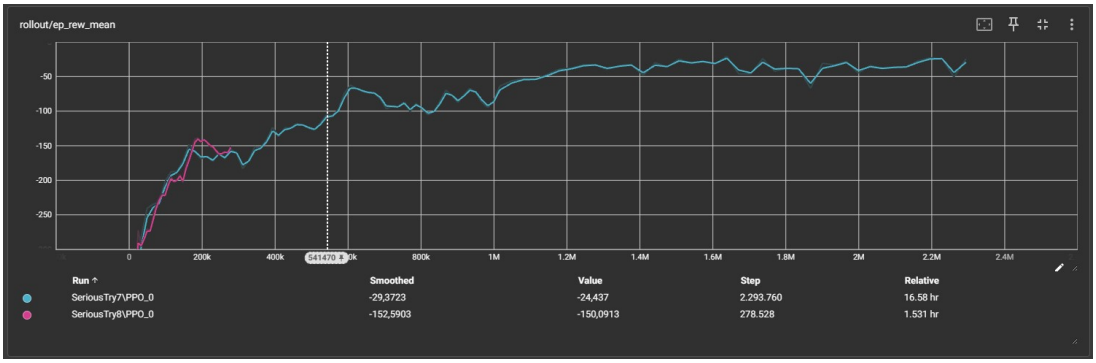

Figure 4.3: ERM of the trial 7 and 8

Figure 4.4 illustrates that the introduction of the heatmap information mitigated the explained variance issue. Please note that due to a technical bug, we were only able to visualize the explained variance for the 8th trial, not the 7th.

Incorporation of Enemy State and Effect of Normalization (Serious Try 9 and 10)

In these trials, we expanded the observation space to include the state of enemy tanks. Initially, we represented this information as a MultiDiscrete space, as defined by Gym. However, as depicted in Figures 4.5 and 4.6, the 9th trial, which extended over 2 million steps, did not show significant learning. This lack of progress led us to a do some more research and finally to a crucial realization: we had not normalized our newly added observations. The absence of normalization in the observation space likely contributed to the underwhelming results in the 9th trial.

After realizing this, we corrected this oversight by converting the enemy state input into a Box space and rescaling all values to the [0, 1] interval (and adjusting some rewards). This modification resulted in a noticeable improvement in the agent's performance, as evidenced in the 10th trial explained variance metrics.

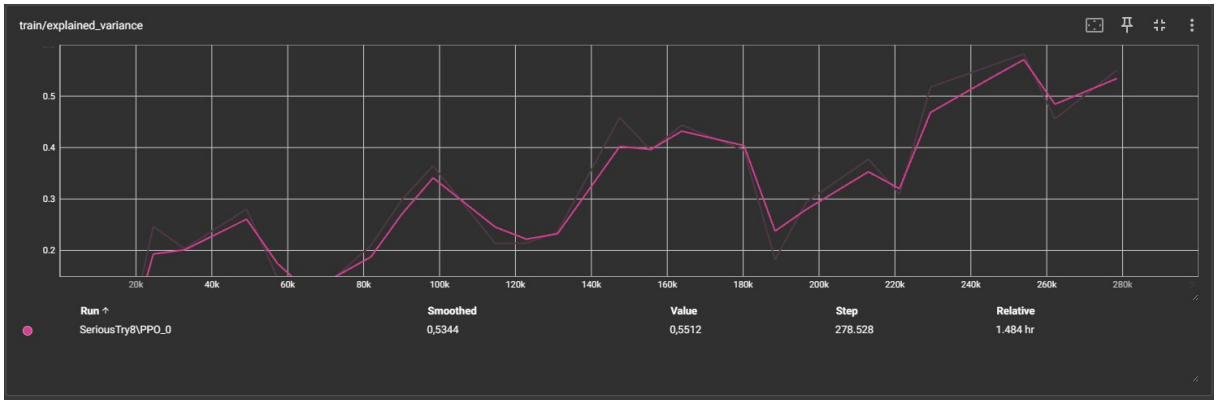

Figure 4.4: Explained variance of the trial 8

Despite the improvements observed in trial 10, it still wasn't learning enough . We decided to conclude this trial early to embark on a new, more refined attempt in Serious Try 11. The experience from these trials underscored the importance of proper normalization in the observation space, significantly impacting the agent's ability to learn and interpret the environment effectively.

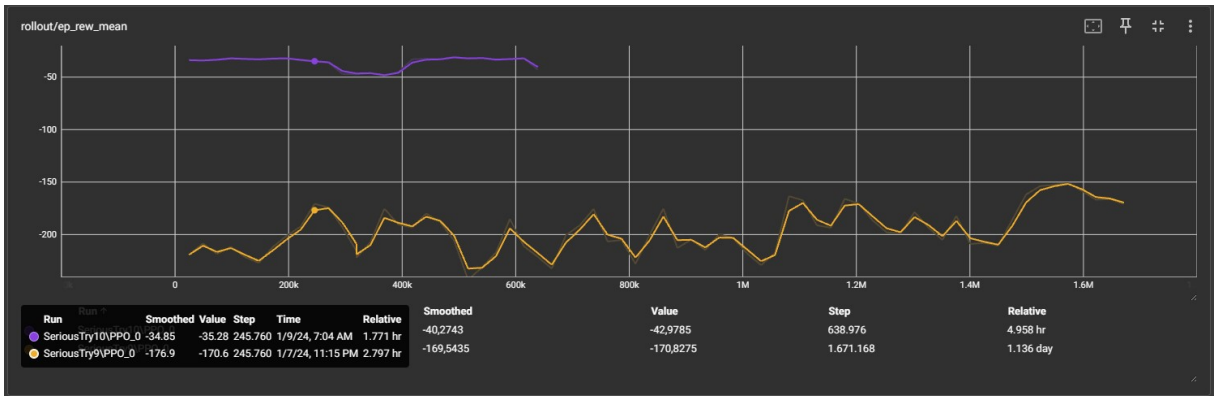

Figure 4.5: ERM of the 9th and 10th trials, illustrating the impact of normalization.

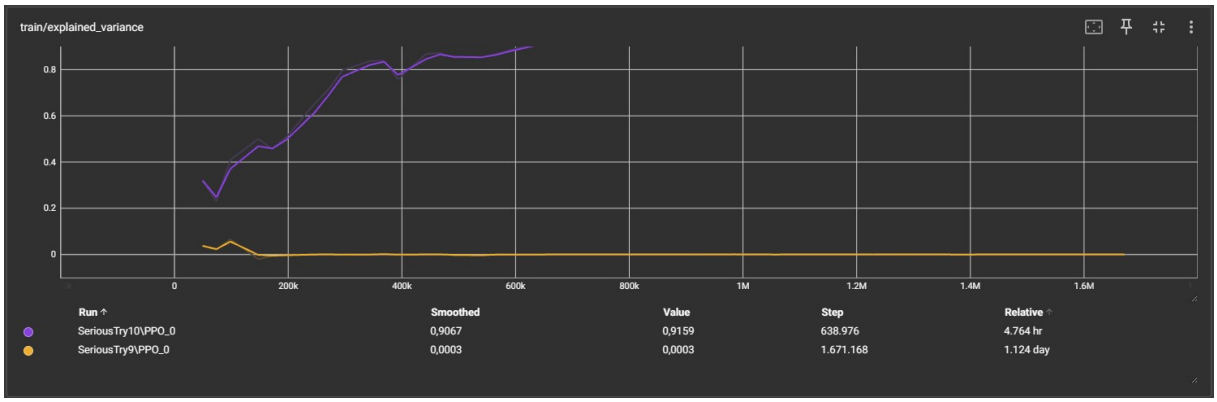

Figure 4.6: Explained variance of the 9th and 10th trials, highlighting the effect of normalization.

Complete Observation Space and Game Simplification (Serious Try 11)

Serious Try 11 marked a significant step forward in our project. We not only implemented a complete observation space as detailed in sec 2.4.2, but also simplified the game as described in chapter 2.4.3 without including the virtual restriction yet. We combined random map generation and player spawning in the hope to force the agent to learn more long term strategies.

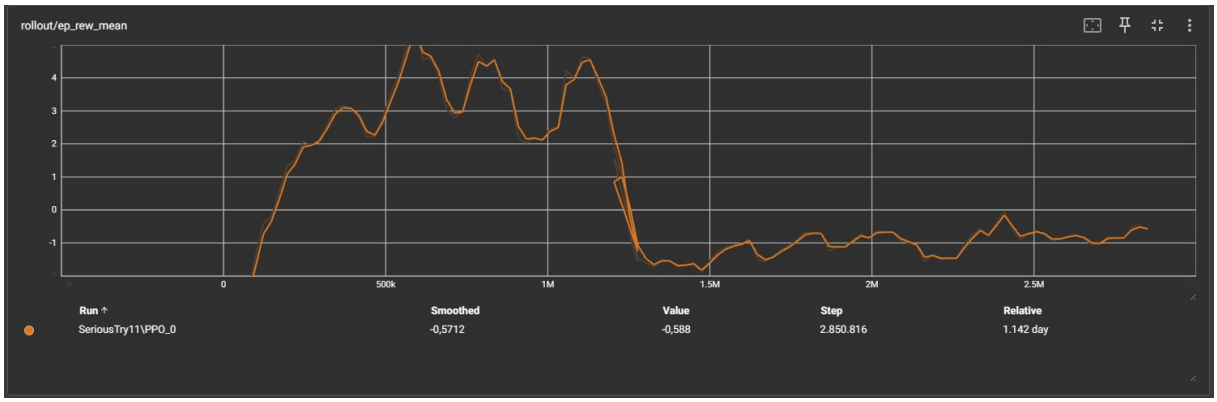

Figure 4.7: Episode Reward Mean (ERM) of the 11th trial, showing impact of changes mid training.

Initially, we set the enemy count at a modest 6, but as the agent started showing promise in tackling these challenges, we raised the stakes by increasing the number to 6-12 after 200,000 steps and 12-20 after 800,000 steps. The impact of these changes are clearly reflected in the ERM metric in Figure 4.7. While the agent got quite adept at taking down enemy tanks, its strategy to defend the base was still a work in progress.

To try to force the tank to defend the base, at 1.179 million steps we adjusted the reward structure to focus only on the outcomes of winning or losing, removing intermediate rewards. We also set the entropy coefficient to zero. These changes proceed a peaked descent in the ERM metric. By the time we reached nearly 3 million steps, our AI had developed. so far, the best strategy for winning, but still didn't exhibit behaviour to defend its base defense. It was able to complete random maps with 20 enemies about half the time, which made us excited to try new things.

Introducing virtual restrictions (Serious Try 13-15)

At Serious Try 13 we restricted player's movement and spawning to an area around the base, as shown in section 2.4.3, with the hope to force it to defend it. We also solved some bugs of the game and completely excluded the bonuses from the map. Additionally, we enabled enemy spawning in the lower part of the map. The number of enemies was kept low, between 2 and 4, since we wanted the agent to sometimes win and receive the huge reward associated with winning. Unfortunately, After 2M steps the agent does not seem to learn any new tactics.

At Serious Try 14 we experimented with changing the entropy to 0.01. After 1.6M steps the agent does not seem to learn anything, with the reward fluctuating wildly.

At Serious Try 15 we lowered the learning rate to 310^{-6}. At 1.47M steps we realize it prioritizes tanks on the top, so we tripled the reward for killing tanks on the lower part of the map. At 2M steps we noticed it made many suicidal moves and destroyed it's own base, so we (finally) decided to implement an antisuicide measure so any actions of shooting at the base would be blocked. A negative reward for trying to shoot was also implemented. After 5M steps, the agent did not successfully learn any better strategies than the Serious Try 11 experiment.

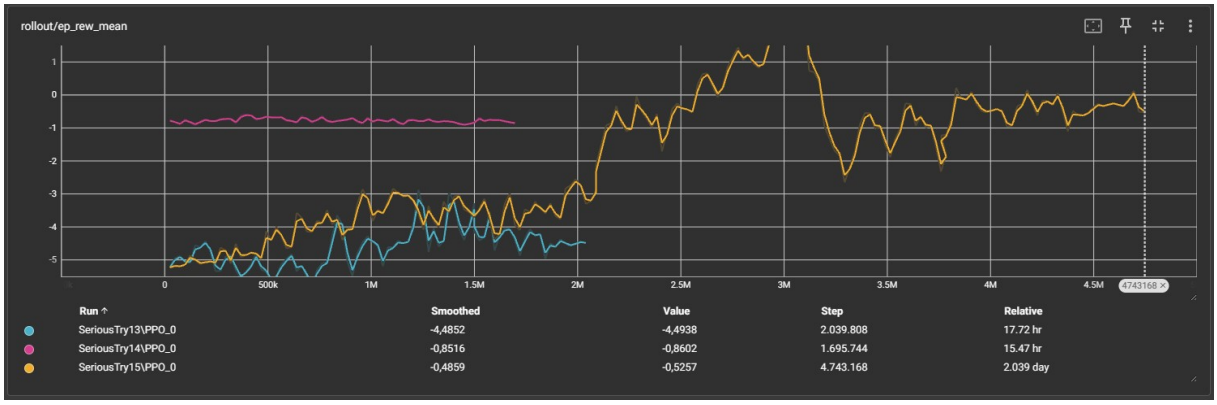

Figure 4.8: Episode Reward Mean (ERM) of trial 13, 14 and 15

Best reached result (Serious Try Final)

In the end, without much time left to try new things, we decided to backtrack on some of our improvements and start a trial with hyperparameters by default, except `TIMESTEPS` and `BATCH_SIZE` that were set as per section 2.4.4. We also decided to train it on the original game maps, focusing on the first two to introduce a bit of variation without too much complexity.

To our surprise, the training was very stable as shown in Figure 4.9. We observed the model once it reached 2M steps and were completely astonished by the result. The tank **could finally make sensible decisions**, moving towards tanks endangering the base and shooting them. It still used some of the camping strategies mentioned previously, but thanks to the heatmap it never stayed on the same place and moved constantly. It combined camping strategies, effective at the start of the level, with a patrolling behaviour in the later stages or when a tank was nearby the base.

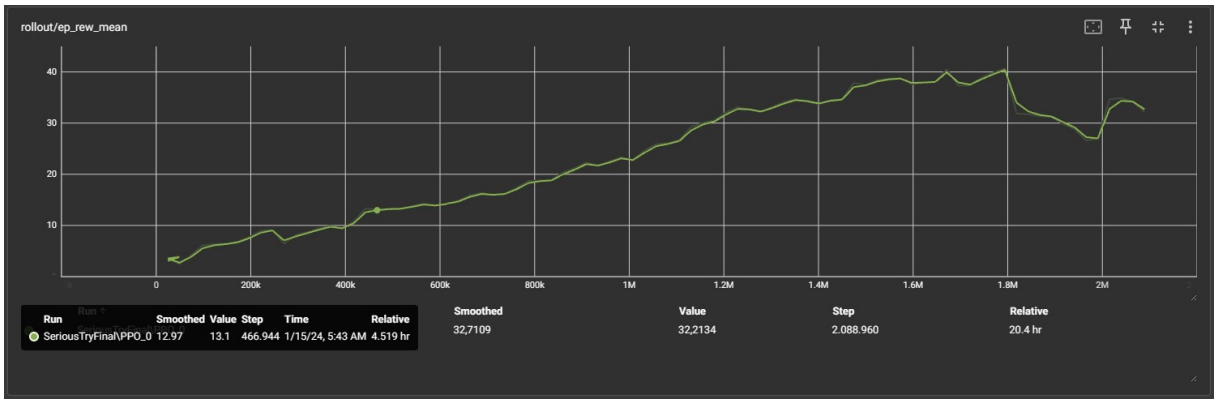

Figure 4.9: Episode Reward Mean (ERM) of the final try

5. User Guide

The steps for getting started to run the project locally are described in detail in the `README.md` file of the project GitHub repository: https://github.com/danisotelo/RL_tank_battalion. As a resume, it is required to clone the repository, install Anaconda, Visual Studio Community 2022 and several dependencies related to Open AI Gymnasium and Stable Baselines 3. Additionally, if it is desired to run the program making use of the GPU instead of the CPU, it is needed to install CUDA Toolkit and additional PyTorch dependencies. All the details are well defined in the `README.md` file. In this same

file the steps for running the program, that is, the user guide, are also described. This will enable a non-expert user to be able to load the agent models and furtherly train the tank agent if desired.

6. Conclusions, Future Lines and Work Contributions

The project aimed to develop an AI agent using RL to proficiently play the Battle City arcade game. The team successfully implemented the RL algorithm, specifically PPO, and adapted it to the game environment, developing tools for model training, loading and visualization. Initially starting with a basic reward policy, the team progressively refined it, aligning it with the game's objectives and complexities. Due to the not-desired observed results, the team continued to explore new alternatives to solve the problem with increasing degree of complexity. At the end, **it was achieved to train an agent to the desired level** to be able to **pass the first game level** and also **different other levels with good chances**. It is observed that in the final version the tank performs quite well, though it does not always win, and its performance is worse than the one of the non-RL agent.

During the progress of the RL improvements, it was achieved to program an improved version of a non-RL agent bot (based on an already cited externally developed code), which is able to reach level 5 in the arcade game, while the external one only could reach level 4. Note that this non-RL agent was not the central development of this project and was simply used in a trial to improve the RL agent reward policy to achieve better performance.

Probably the **performance level** reached at the end with the RL agent **could be furtherly enhanced**. This is because the complexity of the environment requires the agent to train during many **more iterations**. Just as an example, one of the agents that was explored during the academic literature review, the one that was able to play Pokemon, took 50.000 hours to train, while our maximum attempt to train our best reached solution was 36 h (limited by the availability of the computational resources of the team). In addition, the high temporal cost of trying a new alternative, either a new reward policy, or extra item in the observation space, played a key role in the project, even though several implementations were done early on in the project to fasten this training process such as GPU code adapting, frame skipping, frame stacking, multiprocessing, etc.

6.1 Future Lines

Though it is thought the main line to furtherly improve the agent performance is related with the training hours, there are several other factors which could be sources of improvement and that should be studied in further attempts. Some of them are:

- **Different reward system**: Continue to iterate with new different reward policies.
- **More hyperparameter tuning**: Continue to optimize the hyperparameter tuning, specially to avoid local-minima end solutions.
- **Implement more ideas from papers**: Continue to try other RL alternatives (maybe not based on PPO) that resulted to be successful in literature.

Finally, there was no time to do a proper statistical analysis of the percentage of times the RL-agent wins the game, in order to verify the agent performance not in a total reward way, but in a practical way. It could also be interesting to compare this percentage of success with the percentage of success of the improved non-RL AI bot, to determine which of the two performs better.

6.2 Work Contributions

In the present section, the contributions of each of the members of the team are mentioned:

- **Alberto Ibernón Jiménez**: Research and implementation of non-RL AI bot for improving the reinforcement learning solution training, adaptation of tanks game code for non-RL AI bot purposes focusing on useful libraries and tank action commands strategies. Support for RL training activities focusing on preliminary knowledge (learning to protect the own base instead of exploring new regions). Project report development focusing on RL techniques state of the art and AI bot algorithms.
- **Daniel Sotelo Aguirre**: Documentation template design. Preparation and maintenance of GitHub repository and writing of user guide in `README.md` file. Programming of main model automatic saving and loading utils and model training visualization tools. Game code partial adaptation with tank initial position randomization and reward policy optimization. Initial tank agent training trials. Modification of game's action space. Game code adaptation to define reward policy based on non-RL AI bot. Project report partial writing and revision.
- **Jiajun Xu**: Reinforcement learning state-of-the-art research and report writing. Conducted preliminary research on relevant papers, investigated current proposals of RL in games, contributed to reward policies definition, and proposed a reinforcement learning assisted approach. Non-RL AI bot enhancement coding. Code programming of multiple training trials mentioned in the report. Project report partial writing.
- **Vladyslav Korenyak**: Initial game code adaptation to OpenAI's Gymnasium environment. Research on Gymnasium and Stable Baselines 3 for enabling Battle City RL training. Research on other external projects techniques to improve RL agent training. Code programming of multiple training trials implementing these techniques (playing with the observation space to add several additional variables, frame-skipping, etc.). Project report partial writing and revision.

Bibliography

[1] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015. [Online]. Available: https://www.nature.com/articles/nature14236.

[2] Google DeepMind, *Alphago*, https://deepmind.google/technologies/alphago/, Accessed: 01/01/2024, 2016.

[3] O. Vinyals *et al.*, "Grandmaster level in starcraft ii using multi-agent reinforcement learning," *Nature*, vol. 575, pp. 350–354, 2019. DOI: 10.1038/s41586-019-1724-z. [Online]. Available: https://www.nature.com/articles/s41586-019-1724-z.

[4] K. Cai, X. Zhu, and Z. Hu, "Reward poisoning attacks in deep reinforcement learning based on exploration strategies," *Neurocomputing*, 2023. DOI: 10.1016/j.neucom.2023.126578. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231223007014.

[5] Q. Chen, Q. Zhang, and Y. Liu, "Balancing exploration and exploitation in episodic reinforcement learning," *Expert Systems with Applications*, 2023. DOI: 10.1016/j.eswa.2023.120801. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417423013039.

[6] J. Zhu, M. Kuang, W. Zhou, H. Shi, J. Zhu, and X. Han, "Mastering air combat game with deep reinforcement learning," *Defence Technology*, 2023. DOI: 10.1016/j.dt.2023.08.019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2214914723002349.

[7] T. Eimer, M. Lindauer, and R. Raileanu, "Hyperparameters in reinforcement learning and how to tune them," *arXiv preprint arXiv:2306.01324*, 2023. [Online]. Available: https://arxiv.org/pdf/2306.01324.pdf.

[8] Z. Ahmed, N. Le Roux, M. Norouzi, and D. Schuurmans, "Understanding the impact of entropy on policy optimization," in *International conference on machine learning*, PMLR, 2019, pp. 151–160.