Waiting for Godot: Training Agents for Tactical Shooter Games with Godot RL Agents

Donovan Barcelona | db3552@columbia.edu
Michael Chen | yc4131@columbia.edu

ABSTRACT

This paper explores the application of reinforcement learning (RL) in training AI agents within a simplified tactical shooter environment built in the Godot game engine. We investigate strategies for optimizing agent performance and learning efficiency by training an agent to navigate, aim, and engage enemies based on observed states and sparse rewards. While initial successes were achieved, we observed that the agent tended to converge on suboptimal corner-based strategies due to limited exploration. To address these challenges, we implemented resource constraints and parallelized training. Future directions include the development of enhanced collaboration techniques, dynamic environments, and reward shaping to further optimize performance in tactical shooter environments.

I. INTRODUCTION

Reinforcement learning (RL) has gained significant attention in game-playing AI, with notable successes such as Google DeepMind's Agent57 and OpenAI Five.[1,2] While much of this work focuses on general gameplay, there is growing interest in applying RL to more structured, strategic games, such as tactical shooters. These games, exemplified by titles like *Valorant* and *Counter-Strike*, emphasize teamwork, strategy, and careful resource management rather than raw individual skill.

In this work, we focus on training agents within a simplified tactical shooter environment, where the AI must balance both individual actions and collaborative strategies. Drawing inspiration from OpenAI's "hide-and-seek" game, we created a 2D top-down shooter to simulate team-based dynamics in RL, enabling agents to navigate and engage enemies while adhering to tactical objectives.[3]

II. MOTIVATION

At the highest levels of competitive tactical shooters, the skill differences between players are often minimal. Just as in traditional team sports, it is not sufficient to be merely faster, stronger, or more accurate; success also hinges on strategy, coordination, and the ability to outsmart opponents. This idea drives our exploration into simulating AI agents capable of developing and executing complex strategies within the tactical shooter genre.

---

[1] Adrià Puigdomènech, Bilal Piot, Steven Kapturowski, Pablo Sprechmann, Alex Vitvitskyi, Daniel Guo, and Charles Blundell, "Agent57: Outperforming the Human Atari Benchmark," DeepMind Blog, March 31, 2020, https://deepmind.google/discover/blog/agent57-outperforming-the-human-atari-benchmark/.

[2] OpenAI, "OpenAI Five Defeats Dota 2 World Champions," OpenAI Blog, April 15, 2019, https://openai.com/research/openai-five-defeats-dota-2-world-champions.

[3] Bowen Baker, Ingmar Kanitscheider, Todor Markov, Yi Wu, Glenn Powell, Bob McGrew, and Igor Mordatch, "Emergent Tool Use From Multi-Agent Autocurricula," arXiv, September 17, 2019, revised February 11, 2020, https://doi.org/10.48550/arXiv.1909.07528.

Unlike aimbots—which are banned for unfair advantages—our goal is to design agents that can model and discover strategic team plays. By doing so, we provide insights into potential strategies that can be employed by human players to improve their tactical gameplay.

III. BACKGROUND

A. Tactical Shooters

Tactical shooters differ from arcade-style shooters by emphasizing careful planning, strategic execution, and teamwork over fast-paced action. Popular games in this genre, such as Counter-Strike and Valorant, structure matches into rounds where teams must achieve objectives through coordination, rather than by focusing solely on kills. Players assume different roles—such as attackers, defenders, or support—which require distinct strategies for success.

This creates a multi-agent environment where the agent's actions must be optimized not only for individual success but also for team-based objectives. From a game-theoretic perspective, these team dynamics represent a complex environment for RL applications.

B. Markov Decision Process (MDP)

Our environment is modeled as a Markov Decision Process (MDP), where agents interact with the world through states, actions, and rewards:

- **States**: Represent the agent's position, obstacles, and the relative positions of enemies.
- **Actions**: Include movement, aiming, and shooting.
- **Rewards**: Provide feedback for behaviors like hitting enemies or achieving objectives.

Given the large state space of tactical shooters, the agent faces a significant exploration-exploitation trade-off. Balancing the need to explore new strategies versus exploiting known ones is a core challenge in RL.

C. Game Engines

Game engines are essential tools for simulating environments in RL research. They provide the framework for creating, visualizing, and interacting with virtual worlds. OpenAI used MuJoCo, a popular physics simulator, for their hide-and-seek project, though a steep learning curve and outdated tutorials made it infeasible to use for this project.

1) Unity

Unity was our initial choice due to its widespread use in the game development community, comprehensive documentation, and extensive asset store. Unity’s powerful editor and support for C# scripting offered a familiar environment for rapid prototyping. Additionally, Unity provides robust support for 2D game development, which is aligned with our project requirements. However, several challenges emerged during the integration of RL frameworks. While Unity has an open source plugin for ML Agents, Unity’s primary focus on game development often meant that extending its functionality for specialized AI training required significant overhead. Moreover, Unity is also incredibly convoluted and hard to learn, and we decided that spending considerable time and resources on learning Unity itself would take away from our goal of mainly learning and implementing reinforcement learning.

2) Godot

Recognizing these limitations, we broadened our search to include open-source alternatives, leading us to Godot. Godot emerged as a compelling option due to its lightweight architecture, ease of use, and native

support for GDScript, a Python-like scripting language, that facilitated seamless scripting and rapid iteration. We also found GDScript to be much easier for us to learn than C#. Additionally, Godot's flexible scene system allowed for modular game design, which was advantageous for implementing and testing various game mechanics essential for our shooter, as well as the AI agent itself.

Most importantly, Godot now has its own interface for developing RL agents, which was directly inspired by the Unity ML Agents Toolkit.[4] In order to familiarize ourselves with the Godot engine and the RL agent library, we decided to look at multiple projects already created with the library, and create it ourselves from the ground up. Our first game was a simple one: it's pong, but the pitch is a circle and the paddle along it. The objective is to keep the ball bouncing inside the ring. With only two inputs (moving the paddle left and right), a clear reward (keeping the ball from touching the outer circle), this game gives us the ideal starting point, allowing us to focus on learning Godot, GDScript, and the basics of the reinforcement learning library.

IV. GAME SETUP

In addition to Beeching's Godot RL Agents, we were also able to find a simple game design template for the type of shooter we wanted to model.[5] The base game with a few alterations is shown being played with human controls in Appendix A.2.

A. Environment Setup

For our first training task, we wanted to train our agent to identify and shoot enemies. To further simplify the states our agent would observe, we prevented enemies from moving. On initialization, the agent is instantiated in the center of the board, with three enemies spawning at random points around the agent. We chose this setup because it provided a simple environment that we believed the player would be easier to train on.

B. Agent Setup

Implementing an AI agent in the Godot RL Agents package requires four primary methods: **get_obs**, **get_reward**, **get_action_space**, and **set_action**, which mirror the states, rewards, and observations discussed in Section III.B above ("Markov Decision Process"). I will explain our implementation of these four methods and how they relate to reinforcement learning algorithms and markov decision processes.

get_obs returns a dictionary of any environment elements we show to the agent, thus defining its current state. As recommended by Beeching, instead of returning the player's position and enemies' positions as separate observations, each enemy's position is stored relative to the player. In addition to positional data, we also gave the agent information about where they were presently aiming with their angular rotation (in radians). While a more informed agent could theoretically adapt to its environment more precisely, increasing the size of the state space means there are more states to explore and thus longer training times. In addition, given a larger state space, the agent must learn an effective strategy while facing the exploration-exploitation trade-off. In environments like this, there is a risk that agents may over-exploit a small portion of the state space, which might lead to suboptimal behaviors.

get_reward returns the reward at the end of each training session. In our implementation, we give a sparse reward whenever the player successfully hits an enemy. While we messed around with different reward shaping strategies to facilitate faster training and incentivize better strategies, adapting the game setup ultimately proved to be more helpful in this regard. We will discuss more in the next subsection, "Reward Shaping."

---

[4] Beeching, Edward, Jilles Dibangoye, Olivier Simonin, and Christian Wolf. Godot Reinforcement Learning Agents. arXiv preprint arXiv:2112.03636, 2021.

[5] Special thanks to user Unchained112, https://github.com/Unchained112/SimpleTopDownShooterTemplate2D

get_action_space defines the combination of actions a player can take at a given time step, including the name of a given action, how many components it has, and whether it is a discrete or continuous action. Our agent had three actions: move, rotate, and shoot. The "move" action had both an x and y component which could take both negative and positive values. At first we mirrored the WASD control configuration typically used by computer games offering up, down, left, and right movement commands from the user, but simplified the four inputs to just two since up/down and right/left movement effectively cancel each other. The "rotate" action is mirrored similarly to the Ring Pong example given in the tutorial, offering the user the opportunity to spin either clockwise or counterclockwise. Lastly, the "shoot" action determines whether or not the agent is deciding to input a shoot command to the user. This does not necessarily mean the agent will shoot as there is a cooldown between subsequent shots issued by the user, but it seemed unnecessary to penalize the agent for trying to shoot when it could not.

set_action takes the action that the ai_controller wants to take and processes it in a way that Godot can understand it. Here, we also establish bounds on the move and rotate actions to maintain consistency and restrict the size of the state space.

The exact script is given in Appendix B.

C. Reward Shaping

While our submitted code has a sparse rewards system, only rewarding the agent when it defeats an enemy, we tried a number of reward shaping techniques to try incentivizing our agent to perform better. While we initially saw our agent just spinning and shooting, resulting in some level of success, we wished to incentivize more structured gameplay. At first, we penalized shooting, hoping that overall accuracy would increase, as used by Lample and Chaplot.[6] Without sufficient training time, however, we either had a parameter too large causing the agent to stop shooting, or too small such that the agent essentially behaved the same. Next, we tried penalizing rotations, though with similar effects.

Ultimately, the main "reward shaping" we did came in the form of altering the game. We gave the agent a new ammo resource that depleted after every shot, and it only replenished if it cleared a full wave of enemies. With ten bullets and three enemies per wave, we were hoping for a 30% accuracy to continue to the next stage.

For future work, especially in longer simulations, more reward shaping will likely be necessary. For the tactical shooter setting, we may reward attackers with movement towards sites, while defenders might gain a bonus for identifying where the spike is. Depending on how explicitly we want to encourage collaboration, we may give a reward based on whether teammates are moving together or separately.

D. Training Techniques

While we consciously worked on minimizing the size of our state and action space to decrease the amount of exploration that the agent had to do, there are still a number of techniques we used to decrease training time. For instance, we trained on twelve instances of the game at once, replicating a parallelization technique used by many of the examples given in the package. In addition, we utilized the built-in action repeat and speed up parameters given by the package. The action repeat behaves similarly to a "frame skip," a commonly used technique that causes the agent to repeat the same action over multiple timesteps.[7] A higher frame skip helps to expedite training, at the cost of decreased performance since the actions are not as fine-tuned. The speed up ran the game at a faster frame rate, allowing simulations to run

[6] Guillaume Lample and Devendra Singh Chaplot, "Playing FPS Games with Deep Reinforcement Learning," arXiv preprint arXiv:1609.05521, 2018, https://arxiv.org/abs/1609.05521.
[7] Ibid.

faster. Lastly, we ran a number of training sessions without rendering the game itself to reduce the compute needed by our computer.

See Appendix A.1 for a video of us training the model.

V. RESULTS

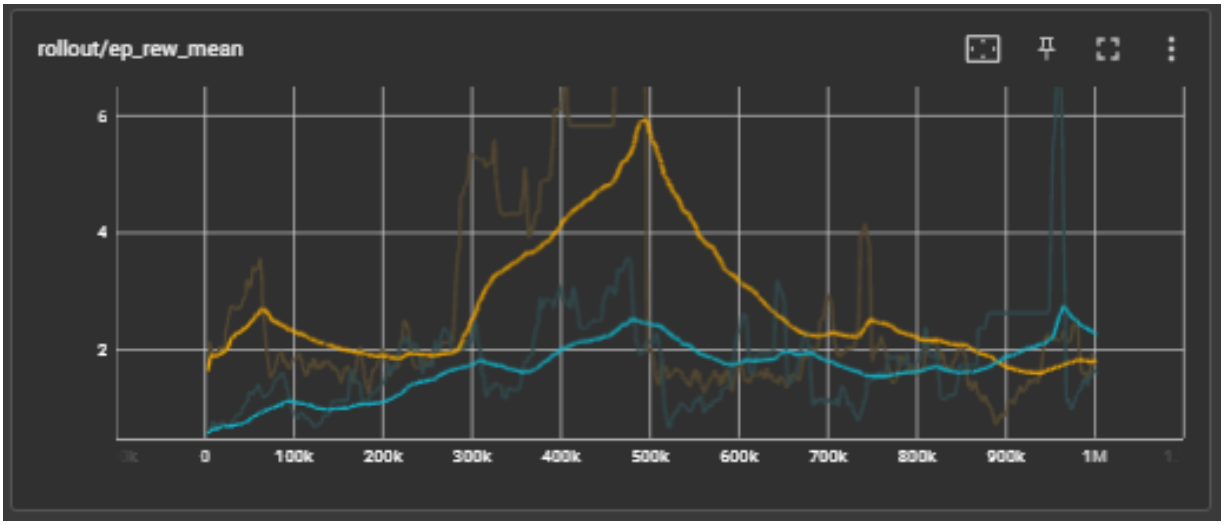


Exhibit 1: Average Training Reward over the first (blue) and second (orange) one million timesteps

As shown in the graph, the agent struggled to demonstrate consistent improvement over the first million timesteps, with reward values fluctuating significantly. This behavior suggests the agent was caught in a phase of suboptimal exploration, where the policy was still developing and adapting to the environment. However, around the 1.4–1.5 million timestep mark, a noticeable increase in the agent's average reward can be observed, stabilizing around a value of 6 or higher. This indicates that the agent was able to refine its policy and start demonstrating more consistent performance.

We logged training approximately every 100,000 timesteps, extracting the model at these intervals to examine the agent's evolving behavior. A series of iterations from one of these intermediate models, shown in *Appendix A.4*, provides insights into the agent's strategy during training. Notably, the agent consistently moved towards the top-left corner of the arena, where it aimed and fired at enemies from a fixed position. This behavior, identified as a **corner-based strategy**, appeared to be a key feature of the agent's learning trajectory.

The corner strategy, in which the agent would gravitate towards one of the four corners of the environment and engage enemies from there, is a significant pattern that emerged during training. Initially, the agent's movement was random, but over time, it began to favor these corner positions. At these corners, the agent could focus on a limited area of the environment, which allowed it to aim and fire more consistently, with less distraction from environmental dynamics. This behavior can be understood in terms of **exploration-exploitation**: the agent discovered that the corner positions offered a relatively safe and predictable location where it could maximize its reward through stable actions (aiming and shooting) without needing to actively explore the environment.

Interestingly, this corner-based strategy may represent a form of **local optimum** within the broader state space. Although the agent was able to exploit this corner strategy for decent rewards, this approach might not be globally optimal. The agent was overly focused on a limited set of states—the four corners—leading to a lack of exploration in other parts of the arena. In environments with a large state

space, this phenomenon is common, as agents often settle on simple, easily exploitable strategies rather than learning more complex, dynamic tactics. The state of the agent in the corner appeared to be better explored than other regions, thus leading to more predictable actions, such as radial shooting (constant firing in random directions). This contrasts with the potentially more rewarding but more complex strategy of exploring the environment and engaging enemies in various locations.

VI. CONCLUSIONS & FUTURE WORKS

In this project, we explored the implementation of reinforcement learning (RL) to train AI agents in a simplified tactical shooter environment. Using the Godot RL framework, we successfully designed an AI that was capable of navigating and engaging with enemies based on its observed states and reward signals. While our agent showed some improvements in performance over training sessions, it often fell into local optima, such as corner-based strategies, rather than developing more dynamic gameplay. This behavior underscores the importance of robust exploration and carefully designed reward structures in RL for complex environments.

Our use of Godot as a game engine provided key advantages in modularity and rapid iteration, though we recognize that further optimizations in state-action representations and training environments could enhance agent performance. For example, the introduction of resource constraints, such as limited ammunition, encouraged more strategic behavior, demonstrating the potential of environmental adjustments in guiding agent learning.

Looking forward, several avenues for improvement and expansion remain:

1. **Enhanced Agent Collaboration:** Exploring multi-agent reinforcement learning (MARL) techniques to model team-based dynamics more effectively.
2. **Reward Shaping:** Developing more nuanced rewards to encourage collaborative strategies and discourage stagnant behaviors.
3. **Dynamic Environments:** Simulating more realistic scenarios, such as moving enemies, variable objectives, and diverse maps, to better replicate the complexity of tactical shooters.
4. **Longer Training Periods and Parameter Tuning:** Allocating additional computational resources to refine training processes and hyperparameters, particularly for fine-tuning agent exploration-exploitation tradeoffs.
5. **Transfer Learning:** Investigating how pre-trained models from simpler environments could accelerate learning in more complex tactical shooter scenarios.

This project highlighted the challenges and opportunities in applying RL to strategic gameplay. By iterating on our methods and leveraging advancements in RL frameworks, we aim to expand upon recent developments in accessible and powerful approaches to game AI.

BIBLIOGRAPHY

Almeida, Pedro, Vitor Carvalho, and Alberto Simões. 2023. "Reinforcement Learning Applied to AI Bots in First-Person Shooters: A Systematic Review" *Algorithms* 16, no. 7: 323. https://doi.org/10.3390/a16070323

Beeching, Edward, Jilles Dibangoye, Olivier Simonin, and Christian Wolf. Godot Reinforcement Learning Agents. arXiv preprint arXiv:2112.03636, 2021.

Dai, Chenyang. "Counter-Strike Self-play AI Agent with Object Detection and Imitation Training." Stanford University, 2021. https://cs230.stanford.edu/projects/fall2021/cs230_final_project_report_dai.pdf.

Lample, Guillaume, and Devendra Singh Chaplot. "Playing FPS Games with Deep Reinforcement Learning." *arXiv*, January 29, 2018. https://arxiv.org/abs/1609.05521.

McPartland, Michelle, and Marcus Gallagher. "Reinforcement Learning in First Person Shooter Games." *IEEE Transactions on Computational Intelligence and AI in Games* 3, no. 1 (March 2011): 43-56. https://doi.org/10.1109/TCIAIG.2010.2050602.

OpenAI, et. al. 2019. *Dota 2 with Large Scale Deep Reinforcement Learning*. December 13, 2019. https://openai.com/research/dota-2-with-large-scale-deep-reinforcement-learning.

OpenAI, et. al. 2020. "Emergent Tool Use from Multi-Agent Autocurricula." *arXiv*, February 11, 2020. https://arxiv.org/abs/1909.07528.

VII. APPENDIX

A. Gameplay

1) Training Demo: https://drive.google.com/file/d/1Cn2qshq9264_yCBIb8VaABDLjvd3omYQ/view?usp=sharing
2) 1 Player Human-Controlled: https://drive.google.com/file/d/1MtsSRAye8BBfN7LxfJ5cD3PqyNhvy8Oq/view?usp=sharing
3) 2 Player Human-Controlled: https://drive.google.com/file/d/168qEwB-wKkZTIC1ASbGhmGe6Z8-clXN3/view?usp=sharing
4) Best Model: https://drive.google.com/file/d/1ia-9mfmmyhCFB2j4m9U0-eoNdYYmpOL7/view?usp=sharing

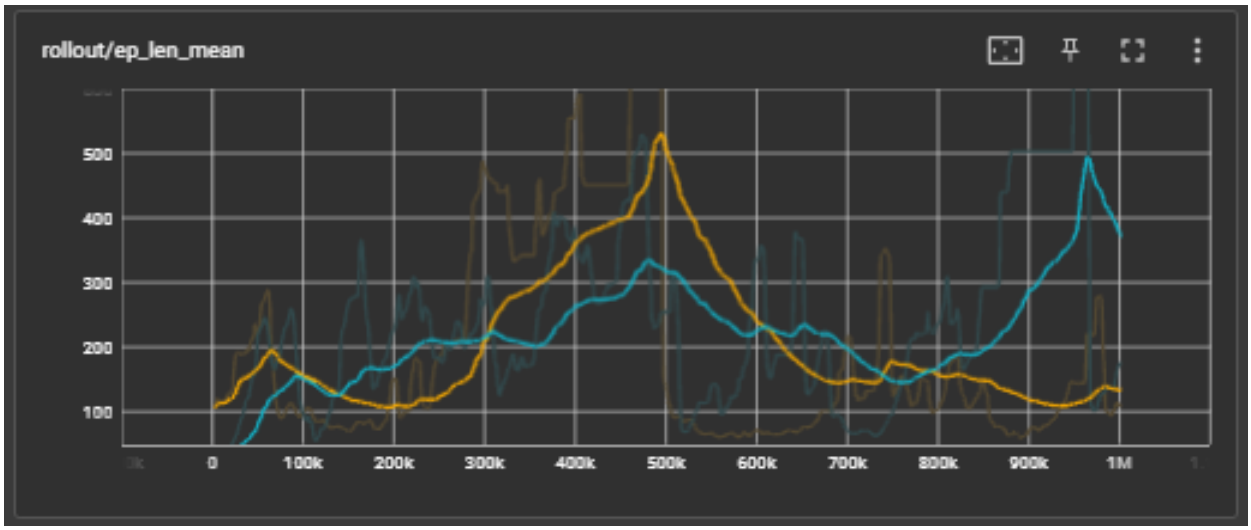
Google Drive Links are accessible by CUID emails

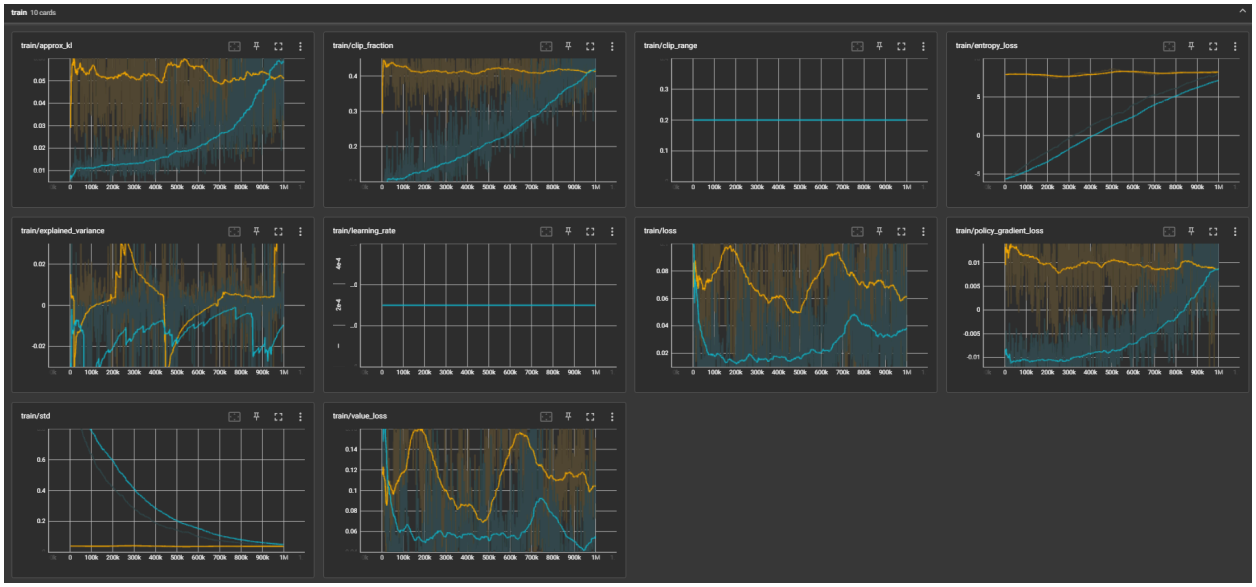
B. AI Agent Script

```
func get_obs() -> Dictionary:
    # Add player's position and rotation to observations
    #var player_pos = to_local(_player.global_position)
    var obs = []

    var player_aim = _player.body_rotate.rotation
    obs.append(player_aim)

    ## Add enemy positions to observations
    var enemies = _player.get_enemies()

    for enemy in enemies:
        var enemy_position: Vector2 = _player.to_local(enemy.global_position)
        obs.append(enemy_position.x)
        obs.append(enemy_position.y)

    return {"obs": obs}

func get_reward() -> float:
    return reward

func get_action_space() -> Dictionary:
    return {
        "move": {
            "size": 2,
            "action_type": "continuous"
        },
        "rotate": {
            "size": 1,
            "action_type": "continuous",
        },
        "shoot": {
            "size": 1,
            "action_type": "discrete",
        }
    }
```

```
func set_action(action) -> void:
    move.x = clamp(action["move"][0], -1.0, 1.0)
    move.y = clamp(action["move"][1], -1.0, 1.0)
    amt_rotate = clamp(action["rotate"][0], -1.0, 1.0)
    shoot = action["shoot"]
```

C. Training Statistics





Row-wise, from the top left: approx_kl, clip_fraction, clip_range, entropy_loss, explained_variance, learning_rate, loss, policy_gradient_loss, std, value_loss

Across the board, the graphs show the model getting “stuck” on a policy, suggesting some local optimum.