Improving Proximal Policy Optimization for Goal-reaching Simulation in Unity with ML-Agents

Qin Lu

Keywords: Proximal Policy Optimization, Reinforcement Learning, Goal-reaching Simulation, Unity, ML-Agents, Customized Step-wise Training Strategy, Hyperparameter Tuning

Summary

This paper presents a comprehensive overview of building a simulation environment in Unity and applying the Proximal Policy Optimization (PPO) algorithm from Unity's built-in ML-Agents toolkit. We designed and implemented a goal-reaching simulation game in which the PPO-based chasing agent is trained to locate a target agent as quickly and accurately as possible. A key contribution of our work is the introduction of a creative and customized step-wise training strategy, specifically designed to address the limitations of CPU-only training environments. This progressive training method enables effective learning under constrained computational resources. Additionally, the paper offers practical suggestions for future optimization and reflects on the broader potential applications of reinforcement learning-based simulation games.

Contribution(s)

1. We hope this paper contributes meaningfully to three key objectives.

   First and foremost, the most significant contribution of our paper is the introduction of a creative and customized step-wise training strategy, specifically tailored for Unity-based reinforcement learning simulation projects. This progressive approach not only improves training precision but also enhances training efficiency. While it was developed for our specific simulation, the strategy is broadly applicable to similar RL projects and can even be extended to Machine Learning and Deep Learning workflows within Anaconda-based or PyCharm-based environments.

   Secondly, this project serves as an excellent hands-on entry point for beginners in reinforcement learning. It offers both theoretical insights into the Proximal Policy Optimization (PPO) algorithm and practical exposure to how PPO operates in real-world simulation scenarios—helping users better understand and appreciate the power of RL.

   Finally, our paper provides a comprehensive guide—from building the simulation environment in Unity to deploying the reinforcement learning algorithm and fine-tuning hyperparameters. As such, it can also serve as a valuable introductory project for anyone interested in developing reinforcement learning-based games or applications using Unity.
   Context: None

Improving Proximal Policy Optimization for Goal-reaching Simulation in Unity with ML-Agents

Qin Lu
qin.lu.2025@mumail.ie

Department of Electronic Engineering, Maynooth University, Ireland

Abstract

Reinforcement learning (RL) has demonstrated significant success in training autonomous agents to perform complex tasks in simulated environments. Among various RL algorithms, Proximal Policy Optimization (PPO) has gained widespread adoption due to its robustness and sample efficiency. However, PPO's performance can vary significantly depending on task complexity and environment dynamics, especially in goal-reaching tasks where precise navigation and decision-making are essential.

This paper introduces a goal-reaching simulation game in Unity using the PPO algorithm from the ML-Agents toolkit, where a chasing agent learns to locate a target efficiently. A key contribution is a creative, step-wise training strategy designed for CPU-only environments, enabling effective training under limited resources and offering insights for future improvements on the similar projects and potential reinforcement learning applications.

1 INTRODUCTION

Reinforcement Learning (RL) is a type of machine learning where an agent learns to make effective decisions by interacting with its environment. The agent tries different actions, observes the outcomes, and receives rewards or penalties based on how well those actions achieve a specific goal. Over time, the agent uses this feedback to improve its decisions, aiming to maximize its total cumulative reward.

This process is inspired to learn through trial and error—exploring, experimenting, and adapting to achieve better outcomes. One of the most famous examples of RL is AlphaGo, developed by DeepMind, which made headlines by defeating a world champion Go player—a game considered more complex than chess. RL's ability to adapt and optimize strategies makes it ideal for developing intelligent, dynamic game characters and simulations.

Unity has become one of the most popular platforms for game development, offering powerful tools for creating interactive 3D environments. Its flexibility and extensive asset ecosystem make it a preferred choice for developers worldwide.

Unity's ML-Agents toolkit, an open source toolkit from unity, integrates state-of-the-art RL algorithms into the engine, providing an accessible framework for training intelligent agents within Unity environments, specifically reinforcement learning. This toolkit helps you to develop complex behaviors enabling your game entities or characters to dynamically adapt and learn from their environment. Among these algorithms, Proximal Policy Optimization (PPO) is widely used due to its balance of sample efficiency and stability during training.

2 Proximal Policy Optimization in ML-Agents

Proximal Policy Optimization (PPO) is a reinforcement learning algorithm designed to maintain training stability by limiting the size of policy updates. It optimizes a clipped objective function to prevent drastic changes to the policy during updates. PPO achieves a good balance between sample efficiency and training stability, making it suitable for complex environments with continuous action spaces. The PPO implementation built into Unity ML-Agents allows developers to quickly train agents for various tasks. This algorithm performs well across many Unity simulation environments and is widely used in game AI and robotic control. The algorithm processes are as follows:

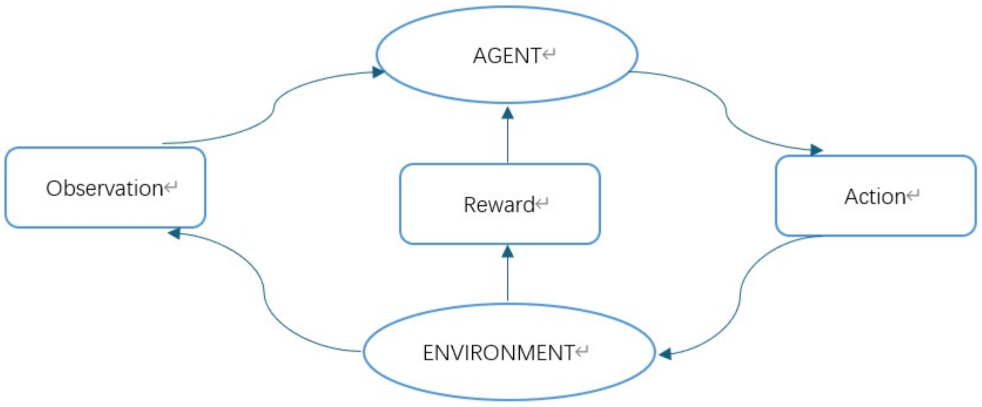


Figure 1: PPO Process in Unity

3 Simulation Design

3.1 Tools and Versions Used

ML-Agents toolkit: release-22 = v3.0.0

Unity Hub: 3.11.1

Unity Editor: 6000.0.25f1

Python: 3.10.12

PyTorch: 2.2.1

Anaconda3: 2024.10-1

3.2 Simulation Overview

Goal-reaching simulation ganme in Unity by combining to use ML-Agents toolkit and Anaconda involves training an agent to navigate and interact with environments to reach predefined goal target. This task tests the agent's ability to learn efficient policies in continuous and limited spaces. Building on this, our work focuses on using PPO algorithm and improving it by optimizing hyperparameters with a creative and customized step-wise training strategy to enhance agent performance and learning speed.

3.3 Simulation Implementation Steps

- **Install Anaconda** : Please follow online tutorials to complete the installation yourself.
- **Set Up Python Environment**:

(1) to open anaconda powershell prompt

(2) setup a virtual environment named "mlagents": conda create -n mlagents python=3.10.12

(3) activate it: conda activate mlagents

(4) install dependency numpy library: conda install numpy=1.23.5

(5) install CPU PyTorch: pip3 install torch =2.2.1 –index-url https://downlad.pytorch org/whl/cpu

(6) test whether installation has been successfully:

> Python

>>>import numpy

>>>print(numpy.__version__)

>>>import torch

>>>print(torch.__version__)

- **Download ML-Agents package**:

  (1) open browser with below address: `https://github.com/Unity-Technologies/ml-agents/tree/release_22`

  (2) download ZIP and extract to your pc, and rename it as “ml-agents”

- **Install ML-Agents in Anaconda** :

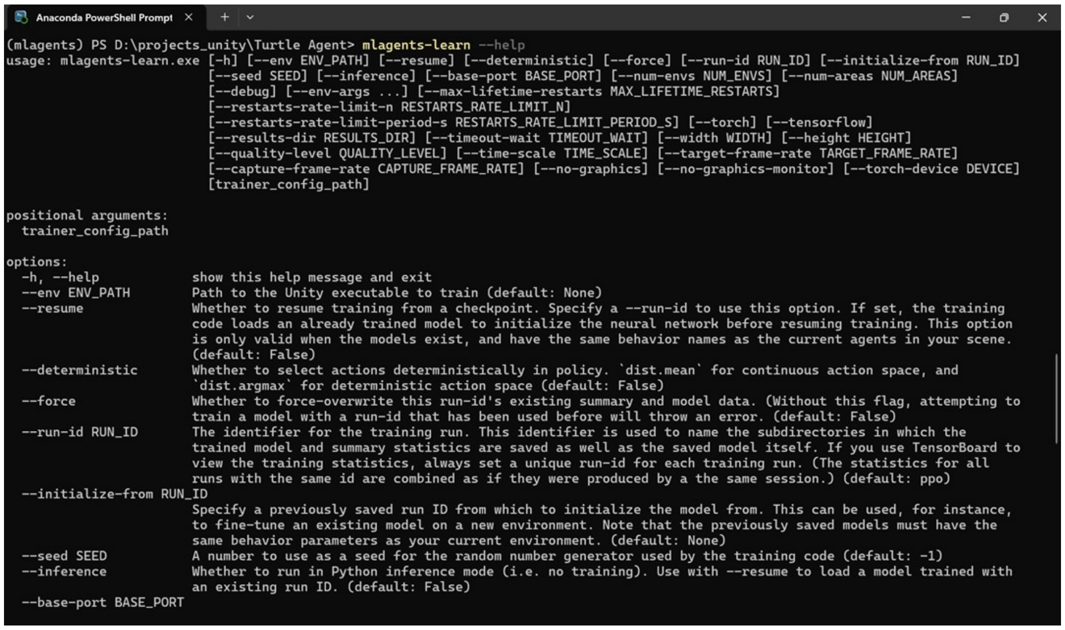
  (1) open Anaconda PowerShell prompt, and navigate to your file saved ml-agents, for example, you extract in D:, then you could input in terminal: cd D:-agents

  (2) install ml-agents envs by below command: python -m pip install ./ml-agents-envs

  (3) and then install ml-agents: python -m pip install ./ml-agents

  (4) test it: mlagents-learn −−help

  then success if you see the guidance as below:



```
(mlagents) PS D:\projects_unity\Turtle Agent> mlagents-learn --help
usage: mlagents-learn.exe [-h] [--env ENV_PATH] [--resume] [--deterministic] [--force] [--run-id RUN_ID] [--initialize-from RUN_ID]
                          [--seed SEED] [--inference] [--base-port BASE_PORT] [--num-envs NUM_ENVS] [--num-areas NUM_AREAS]
                          [--debug] [--env-args ...] [--max-lifetime-restarts MAX_LIFETIME_RESTARTS]
                          [--restarts-rate-limit-n RESTARTS_RATE_LIMIT_N]
                          [--restarts-rate-limit-period-s RESTARTS_RATE_LIMIT_PERIOD_S] [--torch] [--tensorflow]
                          [--results-dir RESULTS_DIR] [--timeout-wait TIMEOUT_WAIT] [--width WIDTH] [--height HEIGHT]
                          [--quality-level QUALITY_LEVEL] [--time-scale TIME_SCALE] [--target-frame-rate TARGET_FRAME_RATE]
                          [--capture-frame-rate CAPTURE_FRAME_RATE] [--no-graphics] [--no-graphics-monitor] [--torch-device DEVICE]
                          [trainer_config_path]

positional arguments:
  trainer_config_path

options:
  -h, --help            show this help message and exit
  --env ENV_PATH        Path to the Unity executable to train (default: None)
  --resume              Whether to resume training from a checkpoint. Specify a --run-id to use this option. If set, the training
                        code loads an already trained model to initialize the neural network before resuming training. This option
                        is only valid when the models exist, and have the same behavior names as the current agents in your scene.
                        (default: False)
  --deterministic       Whether to select actions deterministically in policy. `dist.mean` for continuous action space, and
                        `dist.argmax` for deterministic action space (default: False)
  --force               Whether to force-overwrite this run-id's existing summary and model data. (Without this flag, attempting to
                        train a model with a run-id that has been used before will throw an error. (default: False)
  --run-id RUN_ID       The identifier for the training run. This identifier is used to name the subdirectories in which the
                        trained model and summary statistics are saved as well as the saved model itself. If you use TensorBoard to
                        view the training statistics, always set a unique run-id for each training run. (The statistics for all
                        runs with the same id are combined as if they were produced by a the same session.) (default: ppo)
  --initialize-from RUN_ID
                        Specify a previously saved run ID from which to initialize the model from. This can be used, for instance,
                        to fine-tune an existing model on a new environment. Note that the previously saved models must have the
                        same behavior parameters as your current environment. (default: None)
  --seed SEED           A number to use as a seed for the random number generator used by the training code (default: -1)
  --inference           Whether to run in Python inference mode (i.e. no training). Use with --resume to load a model trained with
                        an existing run ID. (default: False)
  --base-port BASE_PORT
```

Figure 2: Success if show ML-Agents guidance

- **Install Unity Editor**:

  (1) Open Unity Hub

  (2) and then open browser and navigate to below address: https://unity.com/cn/releases/editor/archive

  (3) Choose 6000.0.25f1 and then choose "install"

  (4) wait for finishing installation

- **Start Project**:

  (1) open Unity Hub

  (2) choose "New Project" in the right upper button

  (3) make sure choose editor version is 6000.0.25f1

  (4) choose "Universal 3D (Core) "

  (5) put "Turtle Agent" in the Project Name

  (6) click "Create Project" button

- **Set Up Simulation Environment** :

  (1) after open the project, click "Windows" -> "Package Manager" -> "Unity Registry" -> scroll down -> "ML Agents" -> install it -> close it after finishing installion

  (2) in Hierarchy Tab -> right-click -> "create empty" -> rename it as "Environment" -> keep the transform values as default

  (3) right-click on the "Environment" -> "create" -> "3D Object" -> "Plane" -> Rename it as "Ground"

  (4) right-click on the "Ground" -> "create" -> "3D Object" -> "Cube" -> rename it as "Wall" -> adjust the values of Transform in Inspector Tab -> Postion: X=0,Y=0.25,Z=5.25 -> Scale:X=10.Y=0.5,Z=0.5

  (5) left-click on the "Wall" -> press Ctrl+D to duplicate it -> "Wall(1)" appear -> adjust the values of Transform in Inspector Tab -> Postion: X=0,Y=0.25,Z=-5.25 -> Scale:X=10.Y=0.5,Z=0.5

  (6) left-click on the "Wall(1)" -> press Ctrl+D to duplicate it -> "Wall(2)" appear -> adjust the values of Transform in Inspector Tab -> Postion: X=-5.25,Y=0.25,Z=0 -> Rotation:X=0,Y=90,Z=0 -> Scale:X=11,Y=0.5,Z=0.5->it will make sure it is flush with the other two walls

  (7) left-click on the "Wall(2)" -> press Ctrl+D to duplicate it -> "Wall(3)" appear -> adjust the values of Transform in Inspector Tab -> Postion: X=5.25,Y=0.25,Z=0 -> Rotation:X=0,Y=90,Z=0 -> Scale:X=11,Y=0.5,Z=0.5 -> it will make sure it is flush with the other two walls

  (8) left-click on "Environment" -> "create empty" -> rename it as "Walls" -> drag all walls into the "Walls"

  (9) right-click on "Assets" in the Project Tab -> "Create" -> "folder" -> Name it as "Materials"

  (10) open the file "Materials" in the Assets Tab -> right-click -> "Create" -> "Material" -> In Inspector Tab -> choose "Transparent" for "Surface Type" -> choose the bar besides "Base Map" in "Surface Inputs" -> set R=255,G=0,B=0,A=150 -> Rename it as "Wall Material" -> drag this wall material to every wall objects in the Scene

  (11) in Hierarchy Tab -> right-click -> "create" -> "3D Object" -> "Cylinder" -> rename it as "Turtle" -> In Inspector Tab -> adjust the values of Transform as below -> Position: X=0,Y=0.15,Z=0 -> Rotation:not change -> Scale:X=0.7,Y=0.15,Z=0.7 -> open the file "Materials" in the Assets Tab -> right-click -> "Create" -> "material" -> in Inspector Tab -> choose the bar besides "Base

Map" in "Surface Inputs" -> set R=0,G=0,B=255,A=255 -> Rename it as "Turtle Material" -> drag this turtle material to Turtle object in the Scene

(12) in Hierarchy Tab -> right-click -> "create" -> "3D Object" -> "Sphere" -> rename it as "Left Eye" -> in Inspector Tab -> adjust the values of Transform as below -> Position: X=-0.07,Y=0.306,Z=0.217 -> Rotation:not change -> Scale:X=0.14,Y=0.14,Z=0.14 -> press ctrl+D to duplicate it -> rename it as "Right Eye" -> In Inspector Tab -> adjust the values of Transform as below -> Position: X=0.1,Y=0.306,Z=0.217 -> Rotation:not change -> Scale:X=0.14,Y=0.14,Z=0.14 -> open the file "Materials" in the Assets Tab -> right-click -> "Create" -> "material" -> In Inspector Tab -> choose the bar besides "Base Map" in "Surface Inputs" -> set R=255,G=150,B=0,A=255 -> Rename it as "Eye Material" -> drag this eye material to two eye objects in the Scene -> then parent them to the turtle game object

(13) right-click on "Environment" -> "create" -> "3D Object" -> "Sphere" -> rename it as "Goal" -> In Inspector Tab -> adjust the values of Transform as below -> Position: X=2.5,Y=0.25,Z=0 -> Rotation:not change -> Scale:X=0.5,Y=0.5,Z=0.5

(14) and open the file "Materials" in the Assets Tab -> right-click -> "Create" -> "material" -> In Inspector Tab -> choose the bar besides "Base Map" in "Surface Inputs" -> set R=0,G=255,B=0,A=255 -> Rename it as "Goal Material" -> drag this goal material to goal object in the Scene

- **Adjust Turtle Object to be Trained** :

(1) select "Turtle" in Hierarchy -> go to the Inspector -> scroll down -> "Add Component" -> search "Rigidbody" -> choose it -> in the rigidbody component -> "Freeze Rotation" X and Z in "Contraints" section

(2) secect both left and right eye in Hierarchy -> go to the Inspector -> find "Sphere Collider" -> choose "remove Component"

(3) select the Goal in Hierarchy -> go to the Inspector -> find "Sphere Collider" -> check "Is Trigger"

(4) click "Edit" in the upper bar -> select "Project Settings" -> choose "Tags and Layers" -> click "+" to add button, named it as "Wall" -> continue to add "Goal" -> close the window

(5) select goal in Hierarchy -> go to the Inspector -> find "Tag" -> choose "Goal" -> set this object as goal

(6) span the Walls object in Hierarchy -> select all wall objects -> find "Tag" -> choose "Wall" -> set these objects as wall

(7) right-click Assets Tab -> "create" -> "folder" -> name it as "Scripts" -> we will create control script for our turtle agent -> open "Scripts" folder -> right-click -> "Create" -> "MonoBehaviour Script" -> Name it "TurtleAgent" -> double-click it and open it -> then we should import unity.MLAgents, and its Actuators and Sensors -> see attached code "TurtleAgents.cs" -> left-click Turtle in Hierarchy -> drag TurtleAgent to Inspector -> then we will see "Turtle Agent (Script)" as a new component -> then select "Goal" in Hierarchy -> drag it to "Goal" bar in the new component -> set "Max Step" as 5000 show how long an episode could last before it automatically resets

(8) still in Turtle Inspector bar -> find "Behavior Parameters" -> Rename "Behavior Name" as "Turtle" -> Set "Space size" from 1 to 5 because we have: goalPosX, goalPosZ, turtlePosX, turtlePosZ, turtleRotation -> set "Stacked Vectors" from 1 to 2, which will make past 2 steps actions or info as memory could be taken into account -> set "Branch 0 Size" from 1 to 4 because we have 4 actions: do nothing=0, move forward=1, rotate left=2, rotate right=3

(9) still in Turtle Inspector bar -> search "Decision Requester" -> choose it -> set "Decision Period" keep 5 as default because it represents that how many actions should be updated after -> set "Decision Step" as 0 default since we have only one agent

(10) still in Turtle Inspector bar -> Find "Capsule Collider" -> adjust values of "Center" as below -> X=0,Y=1.34,Z=0

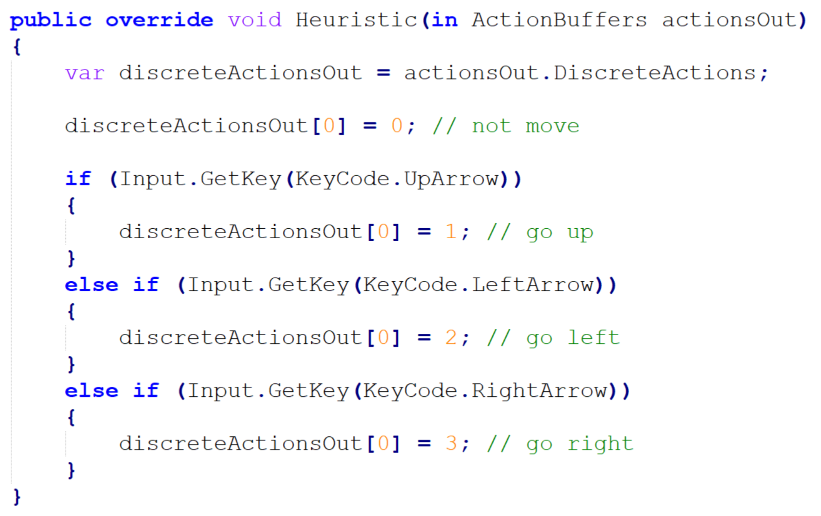
(11) still in Turtle Inspector bar -> find “Ground Renderer -> drag “Ground” in Hierarchy to “Ground Renderer” field of Inspector bar

(12) make good view, choose "Main Camera" in Hierarchy -> adjust the values of Transform as below -> Postion:X=0,Y=6.5,Z-6.5 -> Rotation:X=54.5,Y=0,Z=0 -> Scale:X=Y=Z=1

- **Use Heuristic Function**:

  In this part, you could manually control the chasing agent to reach out the target goal in the simulation scenery.

  (1) select "Turtle" in Hierarchy -> find "Behavior Parameters" -> find "Behavior Type" -> choose "Heuristic Only" -> after that, press the run button -> use your upper, down, left, right keyboard in your pc to control the turtle agent to chase the goal target -> the code is as attached for reference



```
public override void Heuristic(in ActionBuffers actionsOut)
{
    var discreteActionsOut = actionsOut.DiscreteActions;

    discreteActionsOut[0] = 0; // not move

    if (Input.GetKey(KeyCode.UpArrow))
    {
        discreteActionsOut[0] = 1; // go up
    }
    else if (Input.GetKey(KeyCode.LeftArrow))
    {
        discreteActionsOut[0] = 2; // go left
    }
    else if (Input.GetKey(KeyCode.RightArrow))
    {
        discreteActionsOut[0] = 3; // go right
    }
}
```

Figure 3: Code of Heuristic function

  (2) this not only help you to test the environment whether successful or not, but also could help our agent to collect more actions to help to improve training policy

  (3) after simulate manually, do not forget to set "Heuristic Only" back to "Default"

- **Set Up Graphical User Interface (GUI)**:

  In Unity, GUIStyle is a class that defines the appearance and behavior of GUI elements (like buttons, labels, text fields) when using the IMGUI (Immediate Mode GUI) system. It makes the simulation processes visually.

  (1) in Hierarchy Tab -> right-click -> "create empty" -> rename it as "GUITurtleAgent"

  (2) in Assets Tab -> go to the "Scripts" folder -> "Create" -> "MonoBehaviorScript" -> name it "GUITurtleAgent" -> open it -> pls see attached "GUITurtleAgent.cs" -> select "GUITurtleAgent" in Hierarchy Tab -> drag the script to Inspector Tab -> drag the Turtle object in Hierarchy Tab to the "Turtle Agent" field in the Inspector Tab

  (3)click "File" -> click "Save" button to save our scene

- **Create Prefabs**:

  (1) right-click "Assets" Tab -> "Create" -> "Folder" -> name it as "Prefabs"

(2) open the file -> select "Environment" panel in Hierarchy to the Prefabs file -> usually we could easily duplicate it by Prefabs

(3) usually, Prefabs is used to duplicate the simulation environment, especially parallel training in GPU resouces

- **Inference Train without Python Backend Required**:

  (1) right-click "Assets" in Project Tab -> "Create" -> "Folder" -> name it as "AI Models" -> drag ONNX model in your unity project results folder to this "AI Models" folder

  (2) open "Prefabs" folder in Project Tab -> double-click "Environment" -> select "Turtle" in the Environment -> go to the Inspector -> scroll down -> find "Behavior Parameters" -> find "Model" -> drag ONNX model to the "Model" field -> choose "Behavior Type" from "Default" to "Inference Only", which means we do not train it by using Anaconda, no Python backend required

  (3) run it

  (4) if you want to look the training process, could use Anaconda tensorboard to check it by below command: tensorboard –logdir results->open local link by "`http://localhost:6006/`"

- **Train by Using Customed yaml File**:

  (1) open "Prefabs" folder in Project Tab -> double-click "Environment" -> select "Turtle" in the Environment -> go to the Inspector -> scroll down -> find "Behavior Parameters" -> choose "Behavior Type" from "Inference Only" to "Default", which could make you train in Anaconda

  (2) save your scene by pressing ctrl+s

  (3) open the Turtle Agent folder in your pc -> create folder named as "config" -> put "Turtle.yaml" into the config folder -> adjust the parameters in the file

  (4) open Anaconda PowerShell Prompt, activate your environment and navigate to your Turtle Agent file -> input below comman to train by the config: mlagents-learn config/Turtle.yaml –run-id=turtle

- **Code and Video Link** :

  GitHub: `https://github.com/AliceCQ-dev/Improving-Proximal-Policy-Optimization-for-Goa`

  YouTube: `https://youtu.be/cEUxypOZtsM`

  The Unity Scene is as follow Figure:

4 Creative and Customized Step-wise Train Strategy

- **Design Principle**: This creative and customized step-wise incremental training approach is intentionally designed to separate training into progressively more complex phases. This modular design facilitates better convergence and more efficient learning, compared to attempting all optimizations simultaneously. Gradual refinement—starting from a baseline and scaling up in complexity—makes the training process more interpretable, controllable, and ultimately, more effective and better performance.

- **Step 1: Warm-up Phase – Environment Initialization**:

  Begin by initializing and stabilizing the training environment. This phase allows the agent to familiarize itself with the basic dynamics of the environment and develop preliminary policies. It is essential for establishing a baseline before any complex tuning or performance optimization.

- **Step 2: Hyperparameter Fine-Tuning Phase**:

  Once the agent exhibits relatively stable learning behavior, proceed to fine-tune key hyperparameters to improve performance and training efficiency. This includes:

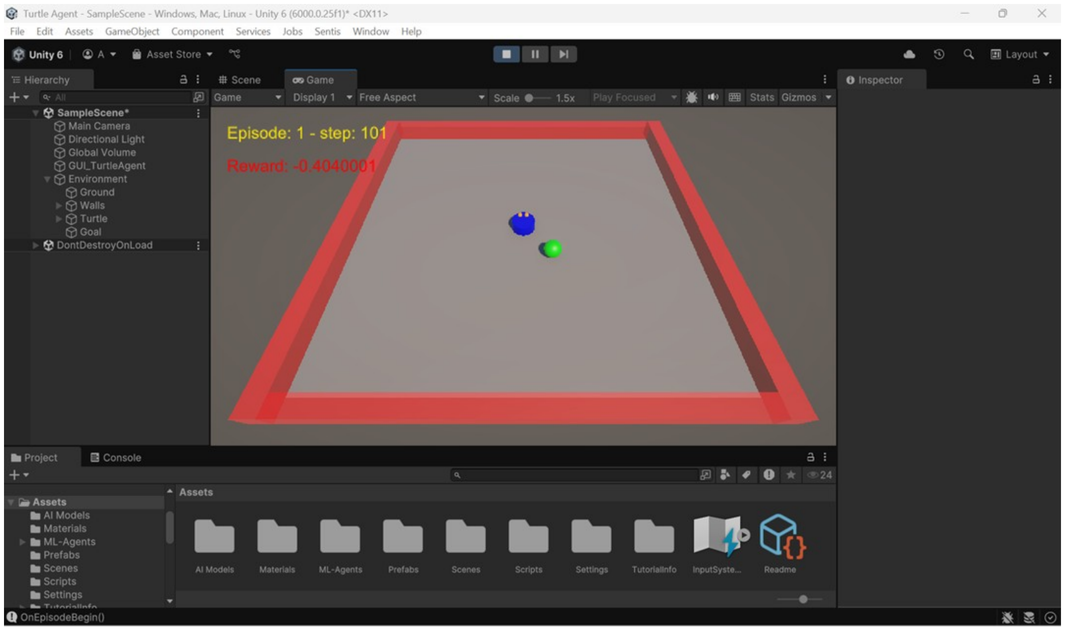


Figure 4: Simulation Scene in Unity

(1) Learning Rate Adjustment: Reduce the learning rate to enable more precise and stable gradient updates as training progresses.

(2) Buffer Size Expansion: Increase buffer size to improve sample diversity and reduce overfitting.

(3) Reward Signal Configuration: Use reward signals: extrinsic: strength: 1.0 to fine tune the influence of the environment's extrinsic reward on learning.

(4) Early Stopping Strategy: Combine max steps with a reward threshold to control training duration, enabling early stopping when performance plateaus.

(5) Normalization Techniques: Enable normalization for observations and rewards to enhance training stability and precision, although this may introduce additional computational cost.

These adjustments aim to fine-balance learning stability with model expressiveness and are crucial for refining the policy after initial convergence.

- **Step 3: Resume Training for Efficient Resource Use**:

  Due to the training being conducted on a CPU, continuous manual monitoring is not feasible. To address this, the best ONNX model is periodically saved during training. Subsequent training sessions leverage the –resume option, allowing training to continue from the last checkpoint without starting over:

  mlagents-learn config/Turtle.yaml –run-id=turtle –resume This strategy helps manage training over longer periods without wasting computational resources.

- **Step 4: Model Selection and ONNX Export**:

  The objective is to export the best-performing model rather than merely the most recent one. Recommended actions include:

  (1) Run training in –train mode with a suitable ——checkpoint-interval (e.g., every 20,000 steps) to generate periodic model snapshots.

  (2) Monitor training performance using TensorBoard, focusing on the cumulative reward curve.

(3) When the reward reaches a peak and stabilizes, manually stop the training. At this point, the latest .onnx file typically reflects the model's optimal performance.

(4) Alternatively, set a reward threshold to automatically halt training once satisfactory performance is achieved, ensuring the exported model is of high quality.

This step ensures the final exported ONNX model is not just recent, but also generalizes well.

- **Step 5: Advanced Model Comparison and Network Expansion**:

  To further enhance model precision, increase the network complexity—such as raising the number of layers or hidden units. While this adjustment increases training time, it allows for a deeper representation of the agent's policy. Evaluate the impact of these architectural changes by comparing model performance metrics across different configurations to determine whether the accuracy improvement justifies the additional computational cost.

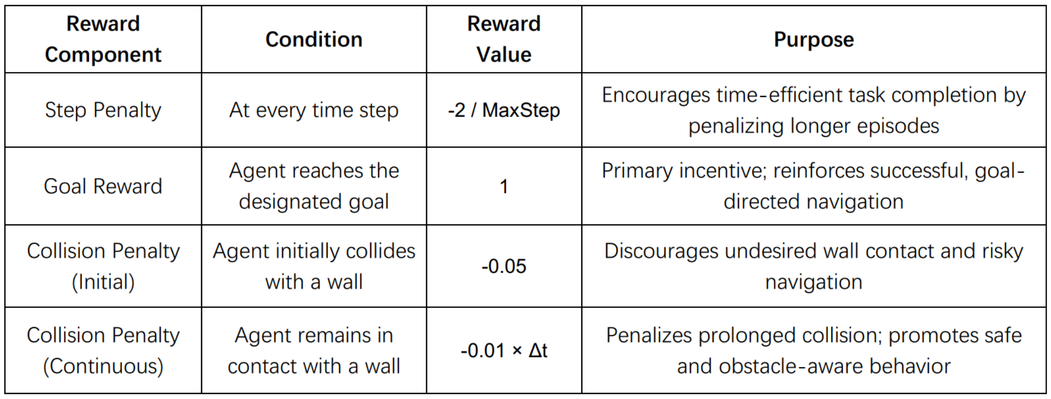
- **Step 6: Real-time simulation by Inference Only method**:

  After training and exporting the optimal ONNX model, the final step is to deploy the agent in real-time using Unity's Inference Only mode. In this mode, the agent relies solely on the pre-trained policy encoded in the ONNX file, with no further learning or parameter updates. This ensures deterministic behavior and is well-suited for testing, demonstration, and integration into broader simulation systems. Activation is straightforward—simply set the agent's Behavior Type to Inference Only in the Unity Editor or via the .yaml configuration. This approach facilitates consistent performance benchmarking, behavioral validation in dynamic scenarios, and effective assessment of the model's generalization capability for deployment.

5 Reward Function Design and Evaluation

5.1 Reward Function Design

The details of our reward function is designed as shown in the figure.



Reward Component	Condition	Reward Value	Purpose
Step Penalty	At every time step	-2 / MaxStep	Encourages time-efficient task completion by penalizing longer episodes
Goal Reward	Agent reaches the designated goal	1	Primary incentive; reinforces successful, goal-directed navigation
Collision Penalty (Initial)	Agent initially collides with a wall	-0.05	Discourages undesired wall contact and risky navigation
Collision Penalty (Continuous)	Agent remains in contact with a wall	-0.01 × Δt	Penalizes prolonged collision; promotes safe and obstacle-aware behavior

Figure 5: Rewar Function Details

This structured reward scheme supports the development of a PPO agent that learns to act efficiently, safely, and purposefully within a constrained training environment.

Together, these reward components form a balanced and task-specific reinforcement learning strategy that promotes fast, safe, and goal-driven behavior. The overall design ensures that the agent learns to navigate efficiently, avoid obstacles, and minimize time and error during task completion.

5.2 Evaluation

In our current implementation, the primary metric used to evaluate agent performance is the Episode Reward, which represents the cumulative reward an agent receives over the course of a single episode. This metric provides a direct and intuitive measure of how well the agent performs the intended task under the current policy.

To gain a more stable and informative view of training progress, we further computed the mean and variance of the episode rewards over multiple episodes. The mean episode reward serves as an indicator of the agent's average performance, while the variance reflects the consistency and stability of the learned behavior. A rising mean reward combined with decreasing variance typically signals effective learning and policy convergence.

6 Constraints, Future Work and Prospects

6.1 Constraints

This study faced two primary limitations. Firstly, time constraints significantly impacted the scope and depth of experimentation. Due to the heavy academic workload and overlapping project commitments, there was insufficient time to perform exhaustive and precise testing.

Secondly, limited computational resources hindered progress. The training was conducted entirely on a CPU, which resulted in long training times and reduced efficiency compared to GPU-based training, making it impractical to perform extended experiments or multiple training iterations.

6.2 Future Work

Although the final performance metrics are not fully presented, the approach is built upon a well-performing baseline, and our enhancements are designed to further improve training efficiency and accuracy. Preliminary results indicate the potential for superior outcomes with continued training.

Future research will prioritize access to GPU resources, which are essential for significantly accelerating training and enabling more complex experiments. Building on the performance improvements discussed in this paper, further optimization should include:

• Reward Function Design: PPO is highly sensitive to the shape and structure of the reward signal. Therefore, carefully designing a reward function tailored to the specific goal-reaching task in Unity is crucial for stable and efficient learning.

• Enhanced Evaluation Metrics: In addition to episode rewards, we design to add two supplementary metrics—Episode Length and Entropy—to obtain a more comprehensive understanding of the agent's learning dynamics. Episode Length reflects task efficiency by indicating how many steps the agent takes to complete an episode, helping to distinguish between strategic improvement and prolonged exploration. Entropy, on the other hand, measures the randomness in action selection and provides insight into the exploration–exploitation balance. High entropy suggests active exploration, while a decline over time signals policy convergence.

6.3 Prospects

This type of reinforcement learning simulation project holds substantial potential for a wide range of exciting future applications. For instance, the simulation environment can be adapted into a more complex and immersive setting, such as an Amazon rainforest adventure. Instead of being limited to two agents, multiple agents can be introduced to enrich the dynamics of interaction. The chasing agent could be player-controlled, while the target agent may become a dynamic, policy-driven entity trained using PPO algorithms. With further development, this framework could be transformed into a 3D augmented reality (AR) game, and potentially into a mixed reality (MR) experience as immersive technologies continue to evolve. Moreover, similar gameplay mechanics and agent behaviors can be extended to diverse environments and thematic scenarios. Such advancements would not only enhance user engagement but also open up innovative avenues for applying reinforcement learning in interactive entertainment. For gaming enthusiasts, this presents a highly promising and captivating direction.

Acknowledgments

During the development and documentation process of this project, I utilized the most powerful GenAI tool, OpenAI's ChatGPT, as a supplementary tool to enhance efficiency and quality. Specifically, ChatGPT was employed to assist in:

• Finding relevant research materials and technical references

• Translating technical terms and concepts into fluent and professional English

• Refining the language, formatting, and structure of written content

All technical decisions, implementations, and perspective were conducted and validated independently, with AI assistance serving only as a supportive resource.

References

[1] A. Kumar, J. Fu, O. Nachum, G. Tucker, and S. Levine, "Stabilizing Off-Policy Q-Learning via Bootstrapping Error Reduction," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, pp. 11794–11805, 2020.

[2] A. Juliani, V.-P. Berges, E. Teng, A. Cohen, J. Harper, C. Elion, Y. Gao, H. Henry, M. Mattar, and D. Lange, "Unity: A General Platform for Intelligent Agents," *arXiv preprint arXiv:1809.02627*, 2019.

[3] D. Burda, H. Edwards, A. Storkey, and O. Klimov, "Exploration by Random Network Distillation," *arXiv preprint arXiv:1810.12894*, 2019.

[4] Unity Technologies, "ML-Agents Toolkit," https://github.com/Unity-Technologies/ml-agents. [Accessed: May 18, 2025].

[5] Unity Technologies, "ML-Agents Documentation," https://unity-technologies.github.io/ml-agents/. [Accessed: May 18, 2025].

[6] Unity Technologies, "ML-Agents Release 22 Branch," https://github.com/Unity-Technologies/ml-agents/tree/release_22. [Accessed: May 18, 2025].

[7] Unity Technologies, "Get Started with ML-Agents in Unity - Part 1: Setup & Installation," YouTube video, 2023. https://www.youtube.com/watch?v=bT3SV1SLqHA&list=PLWcLPdrF6kOnovZnG7VT86ffqdpOUcIUV&index=2. [Accessed: May 18, 2025].

[8] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," *Journal of Machine Learning Research*, vol. 20, no. 1, pp. 1–52, 2019.

[9] A. Juliani, V.-P. Berges, E. Teng, A. Cohen, J. Harper, C. Elion, Y. Gao, H. Henry, M. Mattar, and D. Lange, "Unity: A General Platform for Intelligent Agents," *arXiv preprint arXiv:1809.02627*, updated 2020.

[10] S. Khadka and K. Tumer, "Evolution-Guided Policy Gradient in Reinforcement Learning," in *Proc. AAAI Conference on Artificial Intelligence*, 2020, pp. 1–8.

[11] J. Ibarz, M. Andrychowicz, M. Zhu, P. Wawrzyński, and W. Zaremba, "Reward Shaping for Reinforcement Learning: A Case Study of a 3D Environment," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 5, pp. 2058–2069, 2021.