Tuning Proximal Policy Optimization Algorithm in Maze Solving with ML-Agents

Mac Duy Dan Truong

Phan Thanh Hung

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

Bachelor of Computer Science

Hoa Lac campus - FPT University

20 April 2022

This thesis is dedicated to our hard-working Assoc. Prof. Hung, whose much-needed assistance, continuous support and funding made this work reach its fullest potential as of today.

DECLARATION

We declare that the work in this dissertation titled "Tuning Proximal Policy Optimization Algorithm in Maze Solving with ML-Agents" has been carried out by our research with FPT University Computer Science Department. The information derived from the literature has been acknowledged daily in the text, and a list of references is provided. No part of this dissertation was submitted to any university or institution for any degree, diploma, or other qualification.

Signed: [illegible] [illegible]

Date: 04/20/2022

Mac Duy Dan Truong, Phan Thanh Hung and full qualifications
FPT University

ABSTRACT

The proximal Policy Optimization algorithm is the ML-Agents toolkit's default reinforcement algorithm. This approach can switch between sampling data via interaction with the environment and utilizing stochastic gradient descent to optimize a "surrogate" cost function. Although when creating a new machine learning model, it is tough to know the optimal model architecture for a given project immediately. In most cases, We can either utilize the algorithm's default values or we may use the machine to undertake this exploration and automatically select the best model architecture. Hyperparameters define the model architecture; thus, searching for the best model is called hyperparameter tuning. We focus on comparing four hyperparameters: Beta, Epsilon, Lambd, Num_epoch of PPO algorithm in solving a maze. The results obtained in the training process show the difference in the selection of hyperparameters. The modification of hyperparameters will depend on the maze's complexity and the complexity of the Agent's actions. This thesis will help to make appropriate choices at hyperparameters in concrete and practical projects. Code is available at hungpt17102k/Maze-Solving-ML-Agent (github.com).

Contents

List of Tables

List of Figures

1. INTRODUCTION

1.1. Objective

Today, automated software is used in more projects as a substitution for humans. One of them is a puzzling maze made up of several branches of tunnels in which the solver must locate the most effective route to the destination in the lowest amount of time[1]. Artificial Intelligence is critical in determining the most efficient technique to solve any maze. That is when Reinforcement Learning[2] came in handy.

The sensitivity to hyperparameters is a critical factor in the usefulness of an RL approach[3]. Complex tasks might take hours or days to learn, and fine-tuning hyperparameters is time-consuming. Thus, this research focuses on changing the hyperparameters in configuration for a well-trained PPO algorithm[4] in maze solving with ML-Agent library and Unity[5]. This research also provides a helpful guideline for tuning hyperparameters when redeployment algorithms on novel environments in the future.

1.2. Background

In recent years, several techniques to RL using neural network function approximators have been proposed. The paper published by John Schulman[4] has proposed enhancing the current situation by proposing a method that achieves data efficiency and consistent performance, called Proximal Policy Optimization (PPO). The research team alternates between sampling data from the policy and executing many epochs of optimization upon that sampled data to optimize policies. They came to the conclusion that these methods offer the same level of stability and reliability as trust-region solutions, but are significantly easier to apply. They simply take a few lines of code change to a standard policy gradient implementation, are more generalizable, and perform better overall. However, the algorithm was successful on various problems without tuning hyperparameter values, meaning that the results still did not achieve the best possible outcome.

PPO is increasingly automating the assessment of game material in a casual mobile puzzle game, which has traditionally been a labor-intensive process, with a particular attention on improving its dependability in training and generalization during game play[6]. By adopting a reset approach in which the environment is reset after a specified number of steps, but without addressing hyperparameter tuning, this study was able to successfully transform the popular RL method PPO to a production-grade puzzle platformer game for training play-testing agents.

Different rule-based systems for machine learning methods have been created by other autonomous game-playing entities. Many efforts have also been made to create Agents that behave in a manner that is as near to that of a human player as possible. To estimate the player completion rate of several levels in Lily’s Garden by Tactile Games[7], A set of PPO-based reinforcement learning agents was constructed by the study team. It looked at how the Agent's number of steps for finishing the levels correlated with the behavior of 900,000 players. The results demonstrate that the two-step training scenario generates the most skilled Agent, based on 60% of the game mechanics[7]. In contrast, the Agent attains the most significant correlation to real players’ completion rates with the one-step curriculum. The work is just for a small number of levels with default hyperparameter values, so the given results may not be the best outcome possible.

All the above studies showed that PPO Is an efficient technique in various problems, but the main focus is testing video games or mimicking how humans play. Although most research still uses the default hyperparameter or just a little tuning, proper hyperparameter initialization and search can improve results.

1.3. Design

We design the model to compare the efficiency of different hyperparameters of the PPO algorithm in solving mazes. Use ML-Agent to build models and use Unity to design interfaces to visualize. The maze model is built-in, and algorithms design the model. Agent's goal is to move and find the final finish line in the maze. Agents will be rewarded and distributed during the move. We also design config files for training and can evaluate the influence of hyperparameters during training. Furthermore, this is also the purpose of doing this research. Detailed information such as the rule or the scene design will be discussed in the later Implementation section.

2. METHODOLOGY

2.1. Introduction to Reinforcement Learning

Artificial Intelligence has moved one step closer to its aim of imitating the human brain, thanks to advances in computing technology and the development of new sophisticated algorithms.

In that area, a branch that is becoming more and more important is reinforcement learning (RL)[8]. RL is a sort of learning that has a specific goal in mind. It can be viewed as an approach between supervised and unsupervised learning. It is not strictly supervised as it does not rely only on a set of labeled training data but is not unsupervised learning because RL has a reward that agent aims to maximize.

In order to optimize a reward, an agent learns by interacting with an unknown environment, usually through trial and error. The agent receives input in the form of a reward (or punishment) from the environment, which it then uses to train itself and gain experience and information about the environment[9]. This is the most typical approach for a child to learn: by doing something and watching what happens. In any given situation, the agent must choose between utilising its current knowledge of the environment (performing an action that has already been attempted in that condition) and investigating actions that have never been tried before in that context.

2.1.1. Elements of Reinforcement Learning System

A reinforcement learning system, in addition to the agent and the environment, has four core components: a policy, a reward, a value function, and, potentially, an environment model [9]. A policy is a description of how an agent behaves at a certain point in time. A policy is a mapping between environmental conditions, actions, and the activities that an agent does in the environment in general. In the most precise instances, the policy can be as basic as a function or lookup table, but it can also include complex function computations. The agent's knowledge is built on the foundation of the policy. The goal of a reinforcement learning problem is defined by a reward. At each time step, the agent's activities result in a reward. The agent's ultimate goal is to maximize the total amount of mean earned. As a result, the reward distinguishes between the agent's positive and negative action outcomes. We may think of rewards as pleasure and pain experienced in a natural system.

The reward is the most common way to affect policy; If a policy-determined action produces a poor reward, the policy can be altered to choose an alternative action in the same situation. A value function describes what is desirable in the long run, whereas the reward signal shows positive activities in an instant sense: each action results in an immediate reward. The total quantity of rewards that an agent can expect in the future if it starts from that state is the value of a state. The values show the long-term attractiveness of a set of states, taking into account the most likely future states as well as the benefits obtained from them. Even if a state provides a modest immediate reward, it might still be valuable since it is frequently followed by states that provide more significant benefits.

For beginners, the interaction between incentives and values might be perplexing because one is a sum of the other. Values are secondary projections of rewards, whereas rewards are primary and instantaneous. Without rewards, there are no values, and the primary purpose of calculating values is to gain greater rewards. Nonetheless, values are taken into account when making and assessing decisions. Value judgments are ultimately used to guide action decisions. The agent will seek activities that bring the highest value states, not the highest reward, because these states will lead to acts that earn the most reward in the long term.

2.1.2. Application of Reinforcement Learning

As learning for an unknown environment, reinforcement learning algorithms have been proposed. The basis of search problems that utilize robots or agents consists of maze learning. The most common application of the RL technique is to solve issues by applying a feedback system (rewards and penalties) on an agent that operates in an environment and must proceed through a series of phases to reach a pre-defined final state. In a maze, a rat (agent) is attempting to discover the shortest path from a beginning cell to a destination cheese cell (environment). To reach its purpose, the agent is exploring and utilizing previous experiences (episodes). It may fail repeatedly, but with enough trial and error (rewards and penalties), it should be able to solve the problem.

Take a look at a gameing frontier, namely AlphaGo Zero. [10]. AlphaGo Zero taught the game of Go from the ground up using reinforcement learning. It figured out how to play against itself. Alpha Go Zero was able to exceed the version of Alpha Go known as Master, which had defeated world number one Ke Jie, after 40 days of self-training. It just had a single neural network and only used black and white stones as input characteristics. Without employing Monte Carlo rollouts, a simple tree search based on a single neural network is utilized to evaluate location and sample moves. RL has also been used to train artificial intelligence to play games such as Dota2 with OpenAI Five (2017), Chess and Go (2018), and StarCraft (2019)[11].

2.1.3. Introduction to Deep Reinforcement Learning (DRL)

Deep reinforcement learning has been one of the most controversial areas in artificial intelligence in recent years. It blends deep learning's perceptual ability with reinforcement learning's decision-making ability to control agents' behavior directly through high-dimensional perceptual input learning. Generally speaking, it applies the neural network structure to the process of reinforcement learning. Deep Q Network, Deep Deterministic Policy Gradient, Asynchronous Advantage Actor-Critic, and Proximal Policy Optimization are some of the most popular deep reinforcement learning techniques today (PPO). We used the PPO as our primary AI algorithm in our group project, so we combed the PPO principle in the next section.

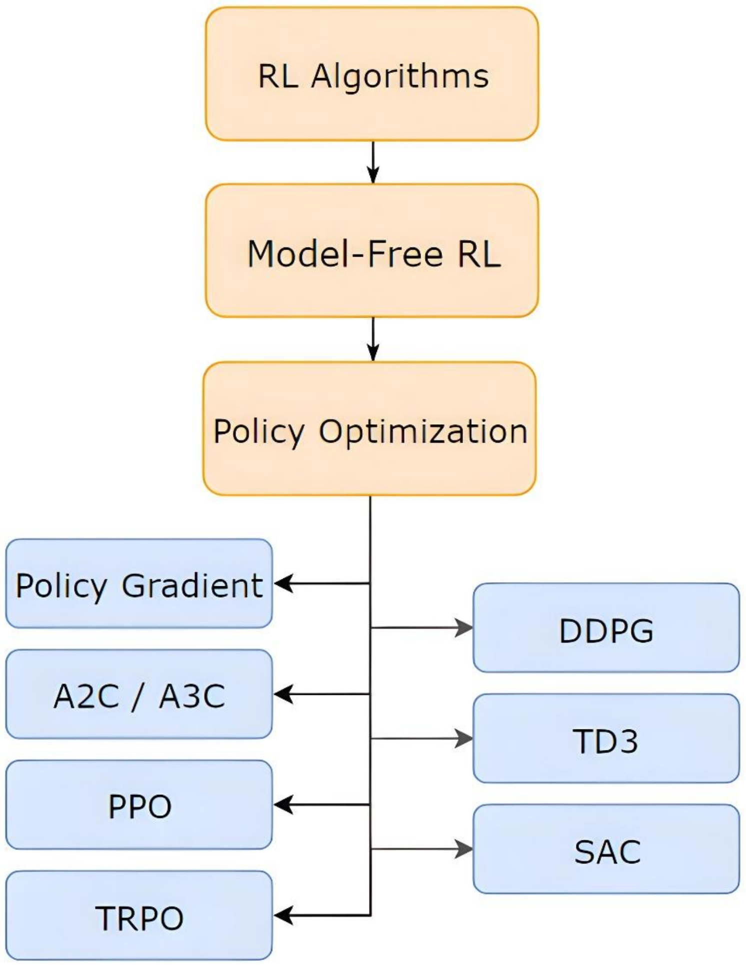


Fig. 1. Main Algorithms of Deep Reinforcement Learning

2.2. Policy Gradient (PG)

Policy Gradient (PG)[12] are frequently used algorithms in reinforcement learning. In PG, the agents observe the state of the environment then take actions based on its policy on the state. After the actions, the agent will enter a new environment state. Like this, the agent constantly observes the environment and takes actions correspondingly. After a trajectory of motions, the agent adjusts his instinct based on the total rewards received. Here are some essential expressions of PG:

In reinforcement learning, the policy π is described as:

$$\pi_\theta(u|s)$$

Our purpose is to find a policy θ that create a trajectory τ; the trajectory consists of the continuous states s and actions u:

$$(s_1, u_1, s_2, u_s, \dots, s_H, u_H)$$

The sum of the probability of a trajectory τ and its corresponding rewards is the expected rewards:

$$J(\theta) = E\left[\sum_{t=0}^{H} R(s_t, u_t)\,; \pi_\theta\right] = \sum_{\tau} P(\tau;\theta)R(\tau)$$

R(τ) means the rewards of the trajectory.

And the PG use this policy to update the θ:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta log\pi_\theta(a_{i,t}|s_{i,t}))(\sum_{t=1}^{T} r(s_{i,t}, a_{i,t})\right)$$

$$\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$$

The advantage functions A and rewrites the policy gradient:

$$A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t)$$

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta log\pi_\theta(a_{i,t}|s_{i,t})A^\pi(s_{i,t}, a_{i,t})$$

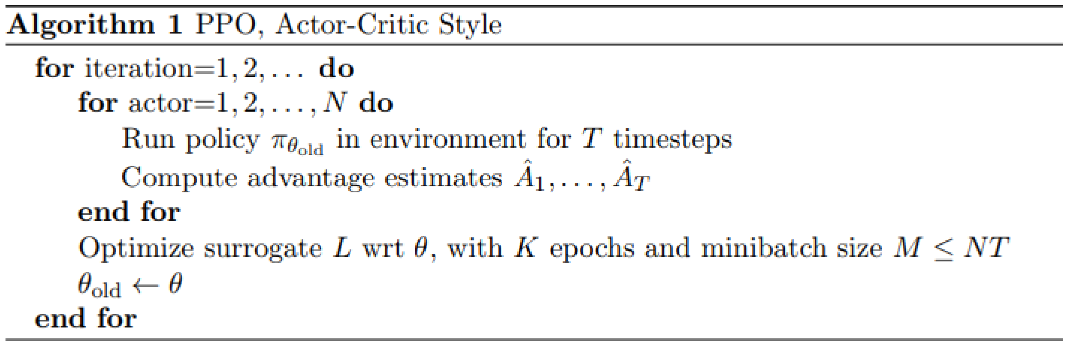
πθ is the policy related to action a and states. The advantage function A includes total rewards Q and the value which is not related to total rewards V.

2.3. Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO)[13] is an optimization approach that uses solely first-order optimization to improve the data efficiency and reliability of Trust Region Policy Optimization (TRPO). PPO is an optimized version based on Policy Gradient and TPRO. Even if it uses the same way to perform multiple optimization steps, Policy Gradient Method may often lead to destructively significant policy updates. TPRO uses a hard constraint instead of a penalty since choosing a fixed penalty coefficient is difficult. In TPRO, the KL penalty coefficient needs to be adjusted to improve the algorithm's performance. The primary objective function of PPO is:

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

In the L^CLIP (θ) in PPO, if the agent has too large of a policy update, it will be punished, which is different from L^CLIP (θ) (The objective function in TPRO). The clip() function can be used to keep the incentive factors from moving rt outside of the range [1-ε, 1+ε]. The steps of the PPO algorithm are:



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

Fig. 2. PPO Algorithm

According to Fig. 2, in each iteration of the PPO algorithm, each parallel (or maybe not parallel) actors collect the T timesteps from the environment then computes the advantages of each T correspondence. At the end of one iterratio, PPO will optimize the surrogate objective function for K epochs.

2.4. ML-Agents

The Unity Machine Learning Agents Toolkit (ML-Agents Toolkit) is an open-source project that allows intelligent agents to be trained in games and simulations. Agents are trained using reinforcement learning, imitation learning, neuroevolution, or other machine learning approaches utilizing a simple Python API[11]. ML-Agents uses a socket to communicate processes during the training phase; it uses python to create a socket server and uses C# (Unity environment) to create a socket client. The TensorFlow trained model saves the bytes file generated after the training in its format. ML-Agents uses TensorFlow Sharp to read the trained model and use it as a Brain in the Unity Environment to accurately guide the Agent's interaction with the environment during the Inference phase.

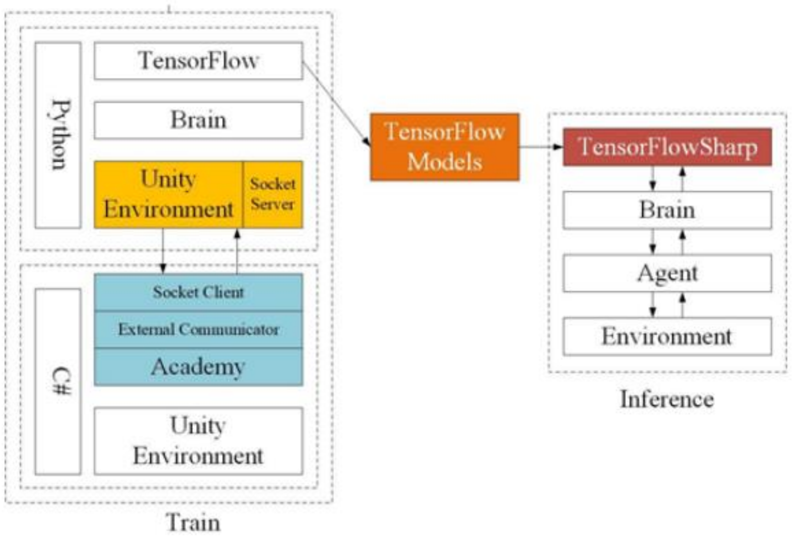


Fig. 3. Unity ML-Agents Structure

There are three key components of Unity ML-Agents: Learning Environment, Python API, and External Communicator[14]. Three more components in the Learning Environment aid in the organization of Unity situations.

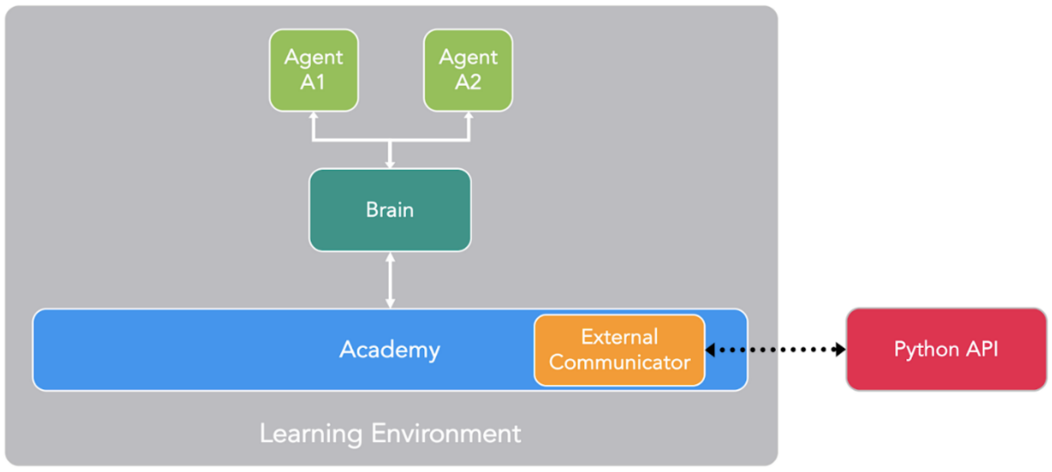


Fig. 4. An example of Learning Environment in Unity ML-Agents

Key Components:

- **Learning Environment**: This is where you'll find the Unity scene as well as all of the game's characters. The Unity scene creates a setting in which agents can watch, act, and learn.
- **Python Low-Level API**: This package includes a Python interface for interacting with and changing a learning environment at a basic level.
- **External Communicator**: This connects the Python Low-Level API to the Learning Environment. It is a component of the Learning Environment.
- **Python Trainers**: This is where you'll find all of the machine learning algorithms for training agents.
- **Gym Wrapper**: A Gym wrapper supplied by OpenAI is a standard approach for machine learning researchers to engage with simulation environments.
- **Agents**: Agents are tied to a Unity GameObject (any actor in a scene) and are in charge of generating observations, conducting actions, and awarding a reward (positive or negative) as needed.
- **Behavior**: specifies the agent's specific characteristics, such as the amount of actions it can perform. Learning, Heuristic, and Inference are the three forms of behavior.

2.5. Hunt and Kill Algorithm

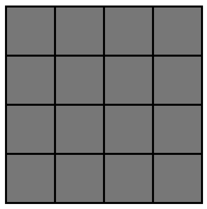
The Hunt and Kill Algorithm[15] work very similarly to the Recursive Backtracker. The algorithm picks a random location and starts a random walk. It continues to walk until it hits a dead end. At this point, the Recursive Backtracker would take a step back, but the Hunt and Kill Algorithm does something different. Instead of backtracking, it will scan the maze for an uncut cell at restart the walking process at that location. It continues this process until all cells have been cut.

Pseudo Algorithm

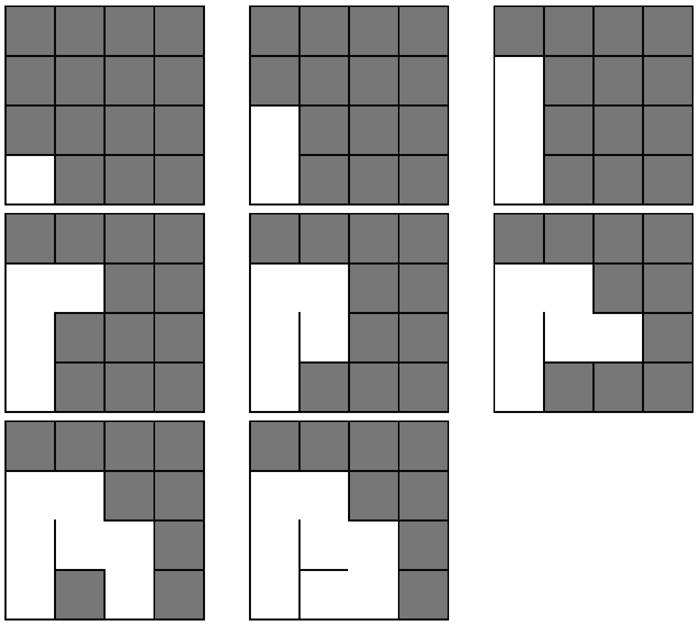
```
1. Pick a cell at random. The current cell is this one. Add it to your list of places it have
   been.
2. Select a cell in the visited list that is adjacent to the present one at random. This is
   now the active cell.
3. Remove the line that connects the previous and current cells. Add the present cell to
   the list of cells that have been visited.
4. Repeat steps 2 and 3 until no more travel is feasible.
5. Scanning the grid from top to bottom, left to right
       If a non-visited cell is found
           The cell is converted into the current cell
           Go to 2
       Else
           The algorithm is complete
```

An example

This is a basic 4×4 grid:

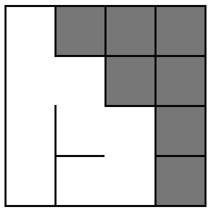


The stroll phase is just a series of frames here; it's not really intriguing until it comes to a halt.

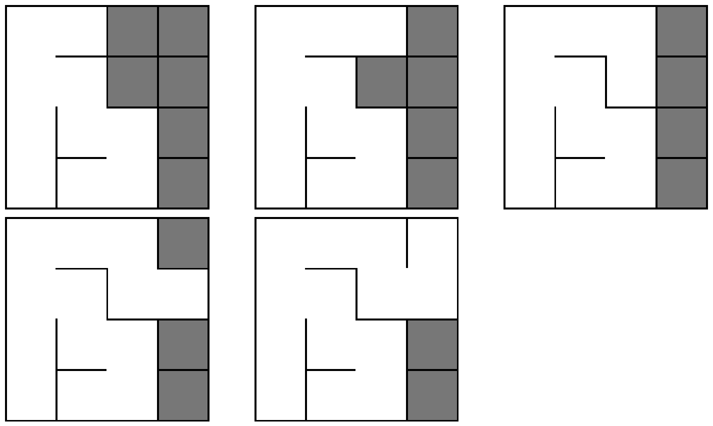


A leisurely drunken stroll also comes to an end. All roads either lead out of limits or into a neighbor who has already been visited. At this point, the recursive backtracker will begin looking for an unvisited neighbor in a stack cell that has already been visited.

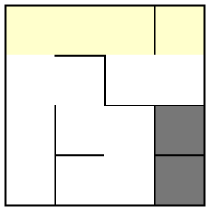
Starting with the first row, we scan each row for an unvisited cell with a visited neighbor in the first row. Today turns out to be our lucky day: our very first cell is a match: unvisited, with a previous occupant. We connect the two of them:



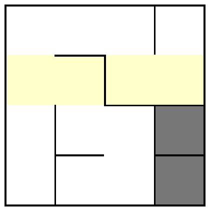
And then start a random walk from the new starting point:



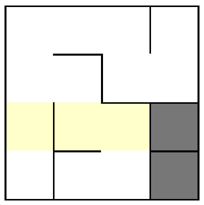
Stuck again, so we go hunting. There are no cells in the first row that match:



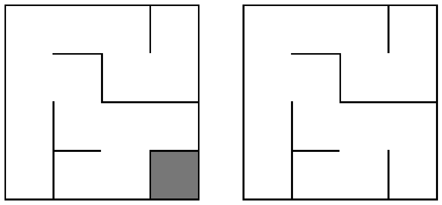
And no matches in the second row, either. (Remember, we are looking for *unvisited* cells with *visited* neighbors)



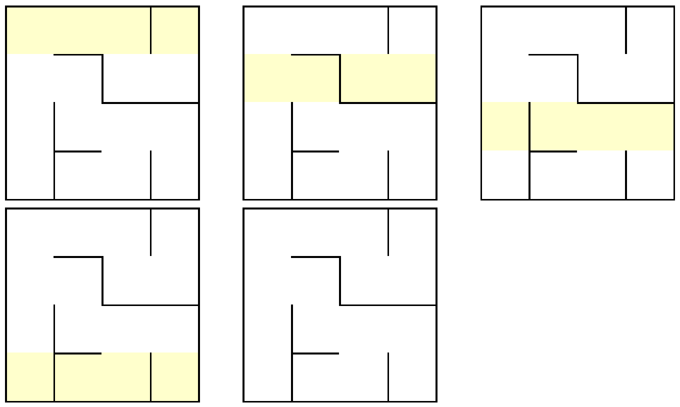
The third row, on the other hand, has a match in the final cell:



As a result, we connect that unvisited cell to any of its visited neighbors (at random) and proceed with our random walk.:



In addition, we've stubbed our digital toes on yet another dead-end. We're stuck, so we go seeking again, row by row, for a cell that hasn't been visited yet.



When the scan is finished and no unvisited cells are found, the algorithm stops and we're left with our maze.

3. IMPLEMENTATION

3.1. Unity Setup

Step 1: Install Python

Python for Windows is available for download and installation. We may create a different management environment for different Python distributions using Pycharm. Python 3.7 or 3.8 is required for this project. Version 3.8 is used in our project.

Step 2: Setup and Activate Environment

Open cmd in the project location

Create a virtual environment:

```
python -m venv venv
```

We must activate this environment in order to utilise it. In cmd, navigate to venv/Scripts/activate.

Install Pytorch next. Pip, a package management system for installing Python packages, is used to install this package. Type the following command in the same CMD:

```
pip install torch==1.10.1+cu102 torchvision==0.11.2+cu102
torchaudio===0.10.1+cu102 -f https://download.pytorch.org/whl/cu102/torch_stable.html
```

Step 3: Install ML-Agent package

We install the following command:

```
pip install mlagents
```

Once it has been done, check it by:

```
mlagents-learn --help
```

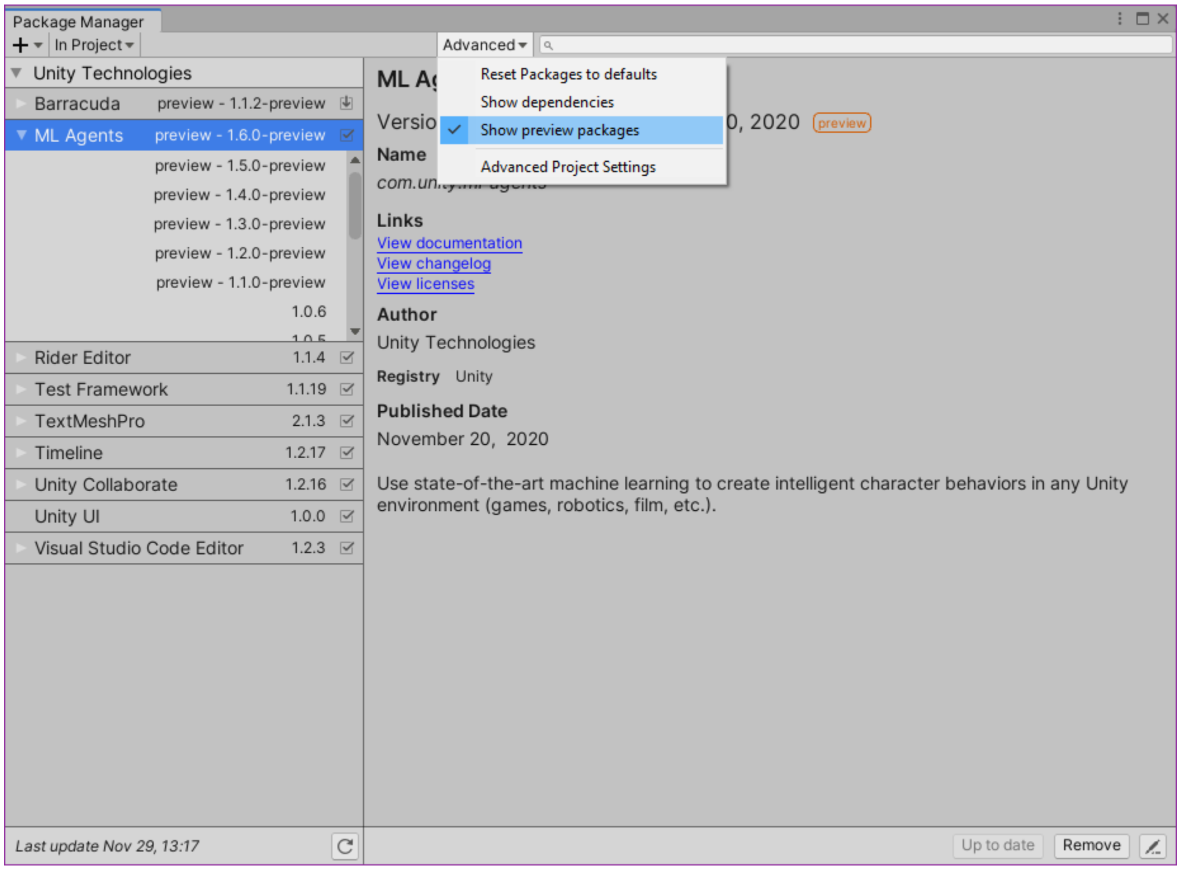
Go to Unity:

ML-Agents is installed via the Unity Package Manager.

To open the Package Manager in Unity, go to **Window > Package Manager**.

Within the Package Manager dialog box:

- Click on **Advanced** and enable **Show preview packages**
- Make sure the **Unity Registry** option is selected above the list of packages
- Search for "ML-Agents" and click on it
- Click **See all versions**
- Choose the version that matches the release downloaded from GitHub
- Click the **Install** button and allow the package to install



Unity Technologies

Barracuda	preview - 1.1.2-preview
ML Agents	preview - 1.6.0-preview
	preview - 1.5.0-preview
	preview - 1.4.0-preview
	preview - 1.3.0-preview
	preview - 1.2.0-preview
	preview - 1.1.0-preview
	1.0.6
	1.0.5
Rider Editor	1.1.4
Test Framework	1.1.19
TextMeshPro	2.1.3
Timeline	1.2.17
Unity Collaborate	1.2.16
Unity UI	1.0.0
Visual Studio Code Editor	1.2.3

Last update Nov 29, 13:17

ML Ag

Versio 0, 2020 preview

Name

com.unity.ml-agents

Links

View documentation

View changelog

View licenses

Author

Unity Technologies

Registry Unity

Published Date

November 20, 2020

Use state-of-the-art machine learning to create intelligent character behaviors in any Unity environment (games, robotics, film, etc.).

Fig. 5. Unity Package Manager

We are using version 2.0.1. Click install and ML-Agent package auto-install to Unity project.

3.2. Maze Design

We have two types of design:

- Maze with fixed design
- Maze with design by Hunt and Kill algorithm

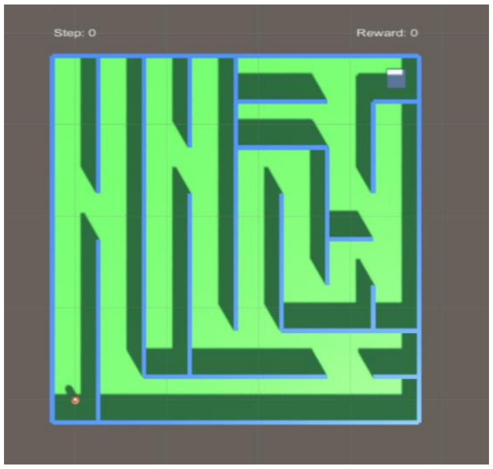


Fig. 6. Fixed Maze 8x8

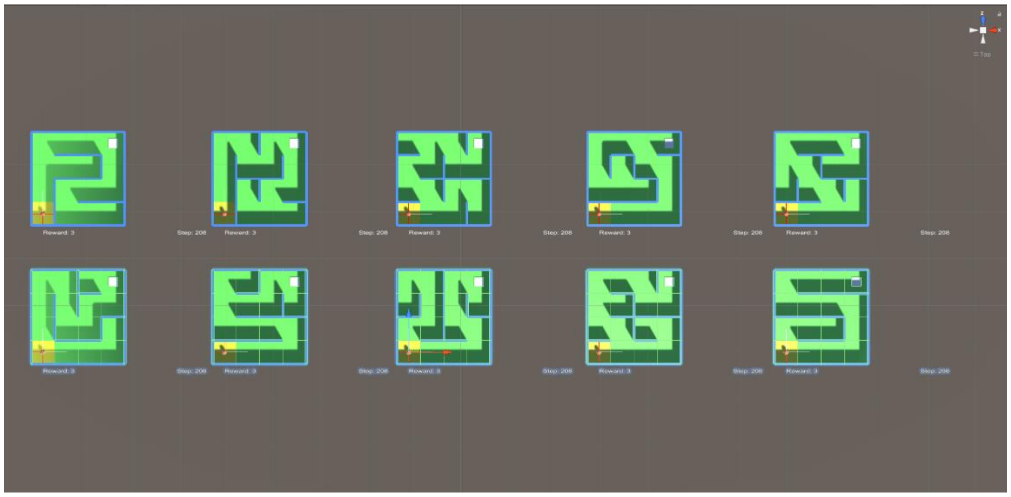


Fig. 7. Random Maze 4x4

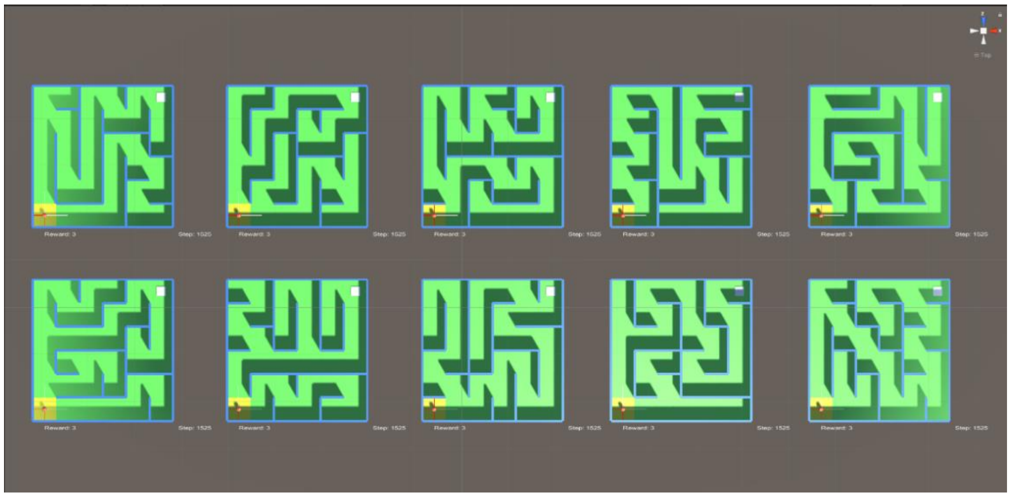


Fig. 8. Random Maze 6x6

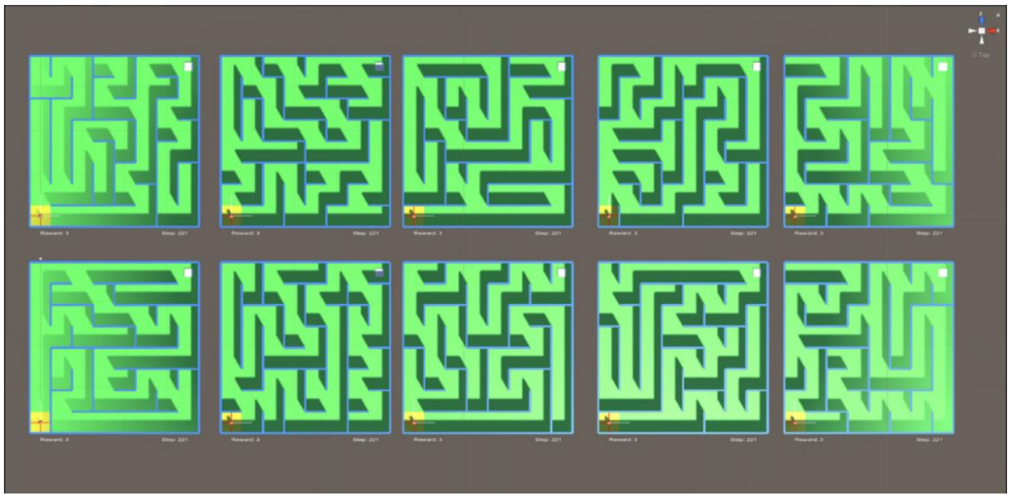


Fig. 9. Random Maze 8x8

3.3. Application Structure

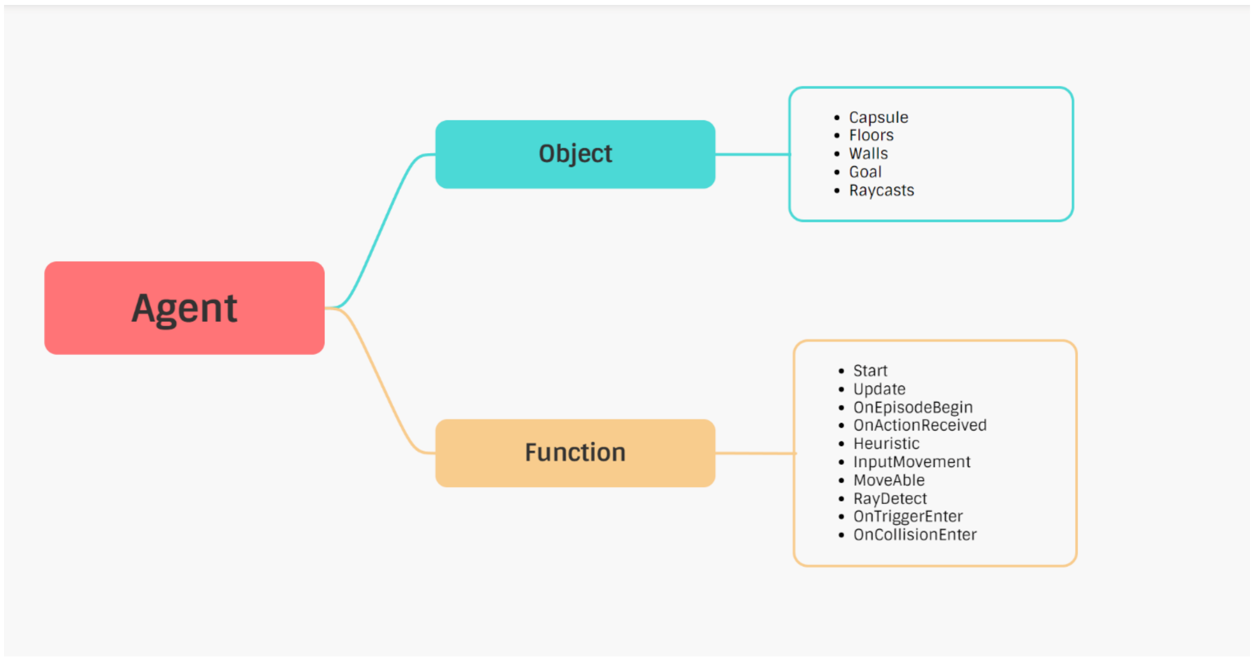
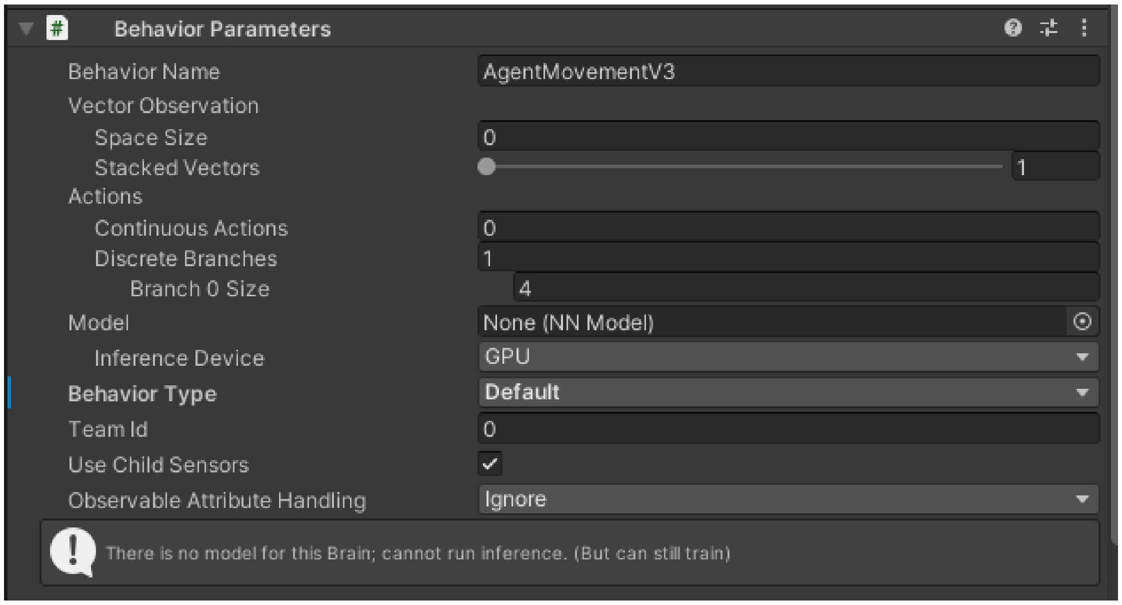


Fig. 10. Agent Structure

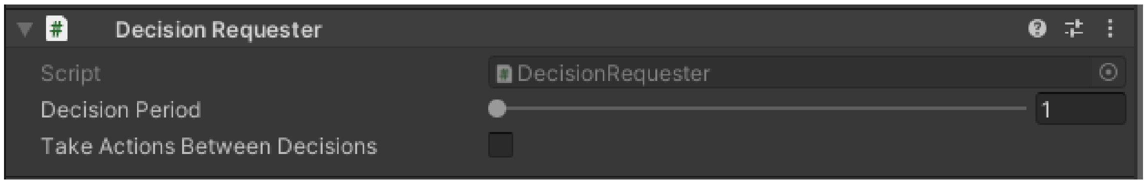


Behavior Parameters

Behavior Name	AgentMovementV3
Vector Observation	
Space Size	0
Stacked Vectors	1
Actions	
Continuous Actions	0
Discrete Branches	1
Branch 0 Size	4
Model	None (NN Model)
Inference Device	GPU
Behavior Type	Default
Team Id	0
Use Child Sensors	✓
Observable Attribute Handling	Ignore

There is no model for this Brain; cannot run inference. (But can still train)

Fig. 11. Behavior Parameter of Agent



Decision Requester

Script	DecisionRequester
Decision Period	1
Take Actions Between Decisions	☐

Fig. 12. Decision Requester of Agent

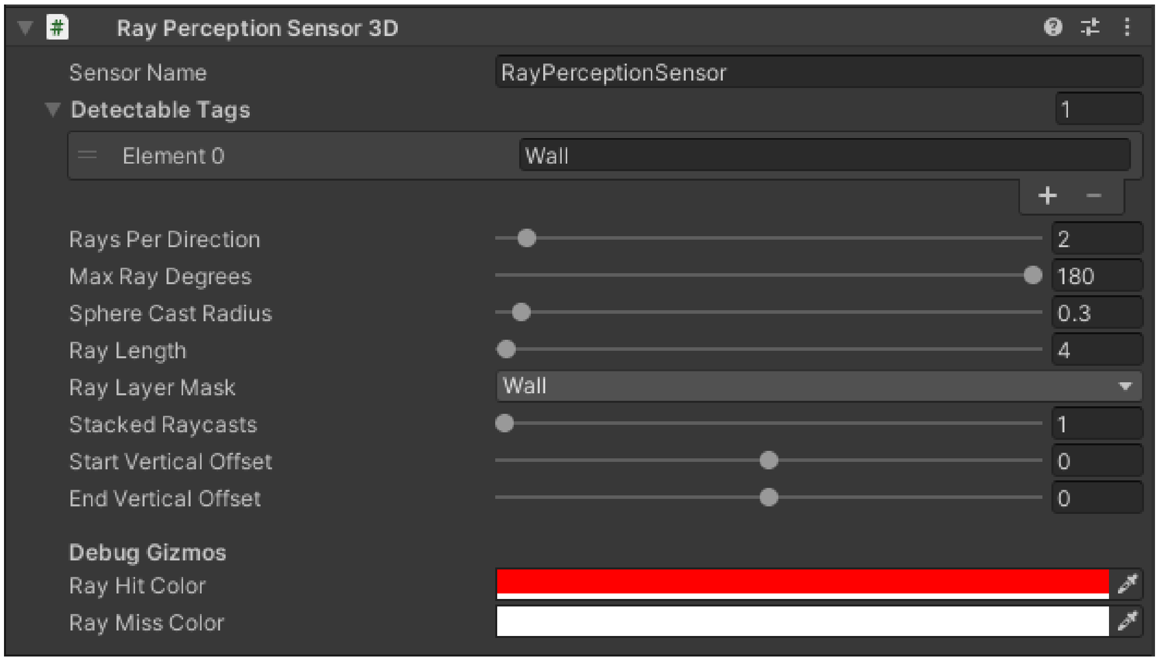
3.4. Environment Logic

Maze design for training Agent:

- We created a fixed maze has 8x8 cells and an algorithm named Hunt and Kill to generate a maze. A cell includes four walls and one floor.
- There is a destination for the Agent to complete the maze. Collision with it will end an episode.

Agent actions behavior:

- An agent with four discrete actions: go up, down, left, and right. Each action is to move into a cell. Furthermore, there is a destination for the agent to complete the maze.
- The agent has four raycasts (to detect collisions with the maze walls) on four sides around the agent. The length of the raycast is one cell.
- Moreover, the agent has four 3D Ray Perception Sensors[16] - the agent's observations, arranged according to the other four raycasts.
- The total number of observations created is: (Observation Stacks) * (1 + 2 * Rays Per Direction) * (Num Detectable Tags + 2) = 1 * (1 + 2 * 2) * (1 + 2) = 15.



Ray Perception Sensor 3D

Sensor Name	RayPerceptionSensor
Detectable Tags	1
Element 0	Wall
Rays Per Direction	2
Max Ray Degrees	180
Sphere Cast Radius	0.3
Ray Length	4
Ray Layer Mask	Wall
Stacked Raycasts	1
Start Vertical Offset	0
End Vertical Offset	0

Debug Gizmos

Ray Hit Color	🟥
Ray Miss Color	⬜

Fig. 13. Ray Perception Sensor of Agent

Agent when entering a cell will be awarded or be punished:

- Entering for the first time, Agent gets 3 points, and that background box turns yellow.
- Entering the second time, the Agent deducts 0.5, and the background cell turns orange.
- The Agent deducts 1 point the third time, and the background box turns purple.
- Entering from the fourth time onwards, the Agent has deducted 2 points, and the background is still purple. Purple is the final penalty level when entering.
- When colliding with the end of the maze, Agent will be awarded 100 points and finish solving the maze.
- When the Agent moves in a specific direction and that side's raycast detects the wall, but the Agent still decides to go in that direction, 1 point will be deducted.

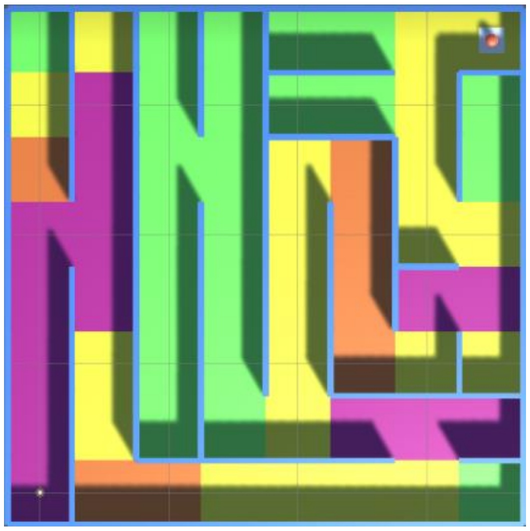


Fig. 14. Simple 8x8 Maze.

3.5. Hyperparameters Configuration

These are hyperparameters that are important in the context of PPO and are not included in the conventional training parameters[17].

- **Beta**: This regulates the entropy regularization's strength, allowing the agent to explore spaces throughout training. The value of beta is usually between 1e-4 and 1e-2.
- **Epsilon**: This determines how quickly the policy can depart from previous policies. The policy updates are more stable when the value is lower. The value of Epsilon is usually between 0.1 and 0.3.
- **Lambd**: When calculating GAE, the regularization factor is employed. Typically, a low value resembles utilizing the current benefit value, while a high value resembles using the actual environmental advantages (high variance). The value of Lambd is usually between 0.9 and 0.95.
- **Num_epoch**: The number of times the buffer is passed through before the gradient descent step is applied. Slower and more stable updates will result if this is reduced. The value of Num epcho is usually between 3 and 10.

➢ To perform tuning, the hyperparameters are taken to default values and then changed each of its values to observe and evaluate the results.

```
hyperparameters:
    batch_size: 128
    buffer_size: 2048
    learning_rate: 0.0003
    beta: 0.005
    epsilon: 0.2
    lambd: 0.95
    num_epoch: 3
    learning_rate_schedule: linear
```

Fig. 15. Default configuration hyperparameters.

4. TRAINING RESULTS

4.1. Results of Fixed Maze 8x8

Beta	Reward	Time Cost
0.01	122.8	49m 45s
0.001	134.1	43m 10s
0.005	143.1	43m 53s
0.0001	143.2	43m 30s

Table 1. Compare the results when changing the hyperparameter Beta of Fixed Maze 8x8.

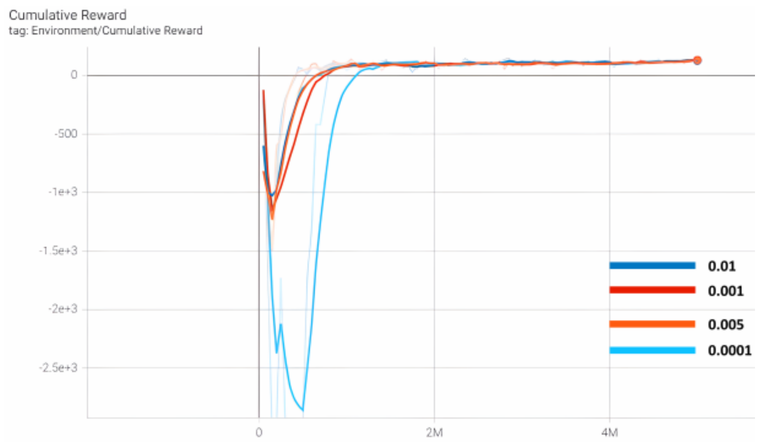


Fig. 16. Graphs with different Beta values in Fixed Maze 8x8.

Epsilon	Reward	Time Cost
0.1	111	43m 12s
0.2	143.1	43m 53s
0.3	132.3	43m 37s

Table 2. Compare the results when changing the hyperparameter Epsilon of Fixed Maze 8x8.

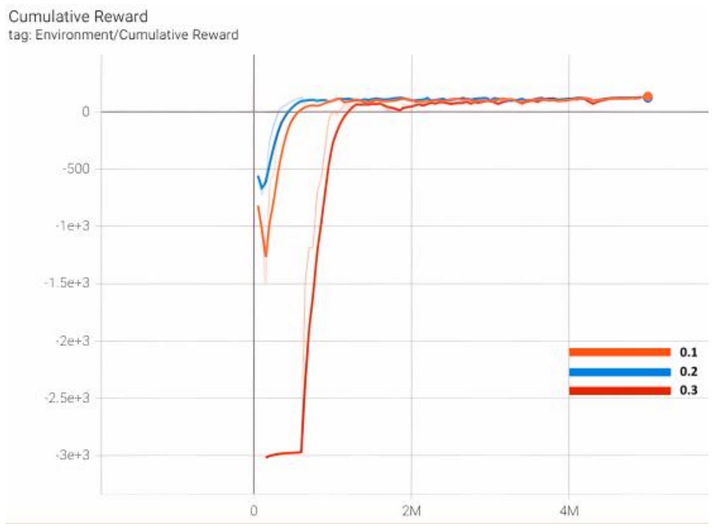


Fig. 17. Graphs with different Epsilon values in Fixed Maze 8x8.

Lambd	Reward	Time Cost
0.9	34.16	49m 5s
0.95	130.8	43m 53s
0.99	140.2	43m 4s

Table 3. Compare the results when changing the hyperparameter Lambd of Fixed Maze 8x8.

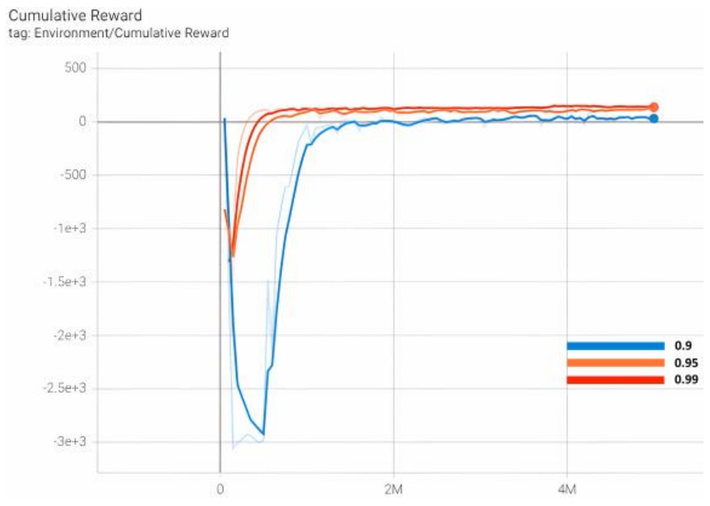


Fig. 18. Graphs with different Lambd values in Fixed Maze 8x8.

Num_epcho	Reward	Time Cost
1	117.6	25m 52s
3	134.5	43m 53s
8	125.2	1h 17m 0s

Table 4. Compare the results when changing the hyperparameter Num_epcho of Fixed Maze 8x8.

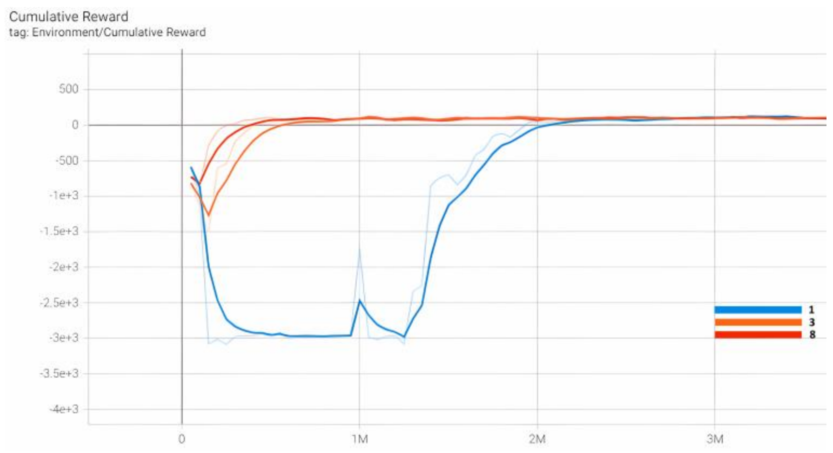


Fig. 19. Graphs with different Num_epoch values in Fixed Maze 8x8.

4.2. Results of Random Maze 4x4

4.2.1. Results of Random Maze 4x4

Beta	Reward	Time Cost
0.01	91.27	1h 3m 20s
0.001	90.83	1h 29m 20s
0.005	92.49	1h 20m 34s
0.0001	76.26	2h 6m 18s

Table 5. Compare the results when changing the hyperparameter Beta of Random Maze 4x4.

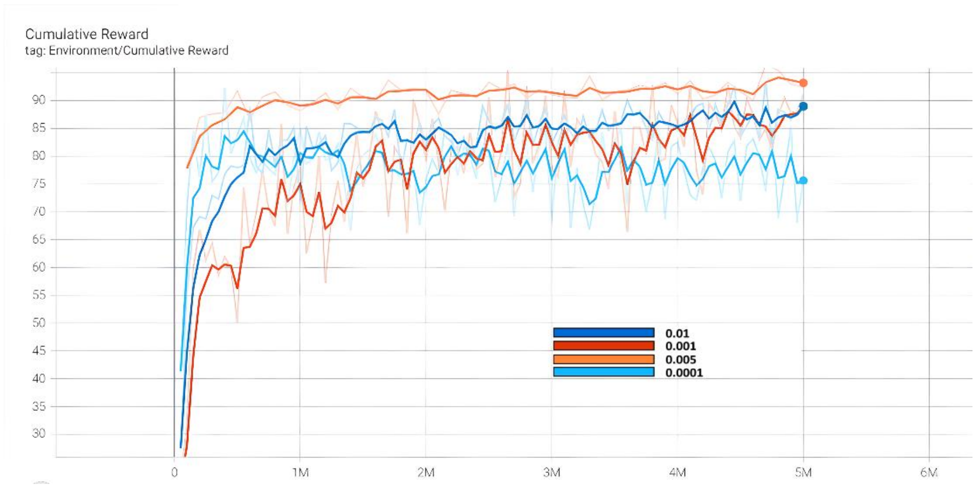


Fig. 20. Graphs with different Beta values in Random Maze 4x4.

Epsilon	Reward	Time Cost
0.1	104.9	1h 14m 28s
0.2	92.49	1h 20m 34s
0.3	112.7	2h 6m 18s

Table 6. Compare the results when changing the hyperparameter Epsilon of Random Maze 4x4.

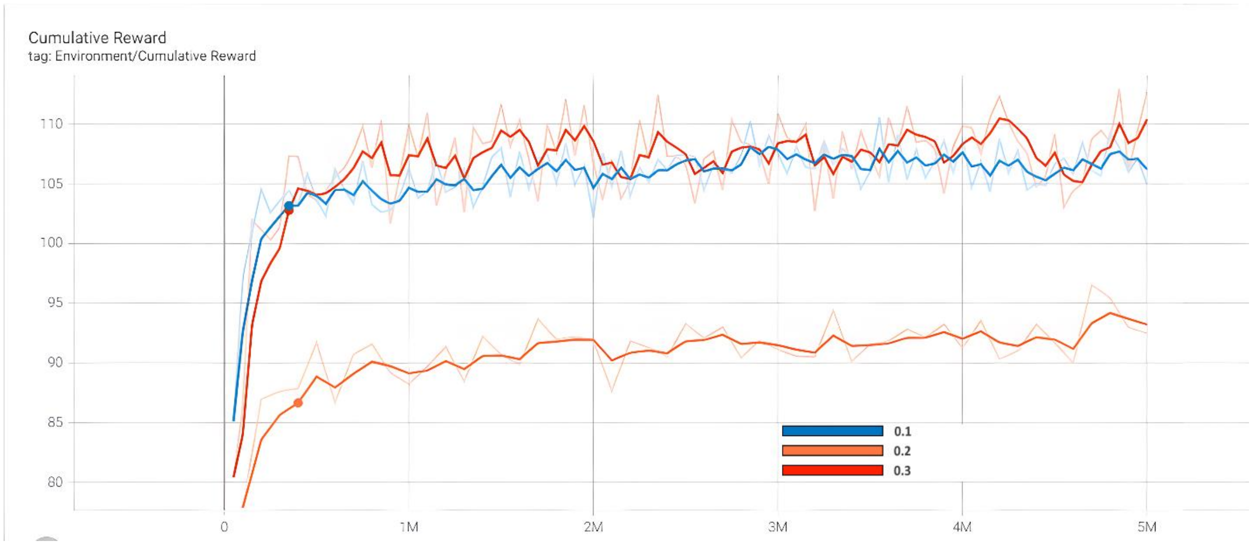


Fig. 21. Graphs with different Epsilon values in Random Maze 4x4.

Lambd	Reward	Time Cost
0.9	87.54	1h 47m 8s
0.95	106.6	1h 15m 16s
0.99	92.49	1h 20m 34s

Table 7. Compare the results when changing the hyperparameter Lambd of Random Maze 4x4.

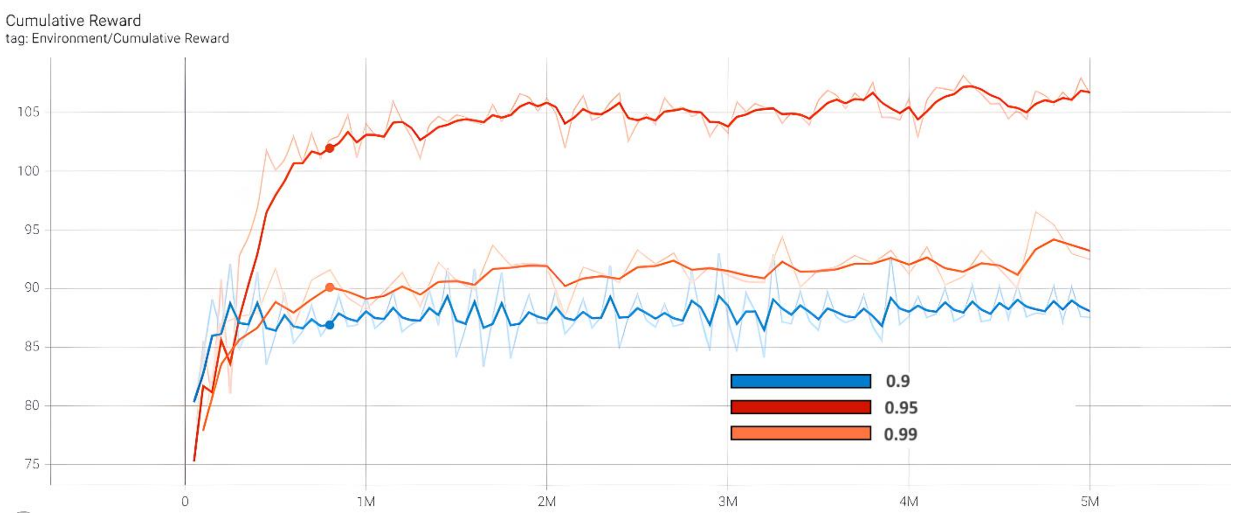


Fig. 22. Graphs with different Lambd values in Random Maze 4x4.

Num_epcho	Reward	Time Cost
1	103.6	57m 32s
3	92.49	1h 20m 34s
8	95.91	2h 15m 26s

Table 8. Compare the results when changing the hyperparameter Num_epcho of Random Maze 4x4.

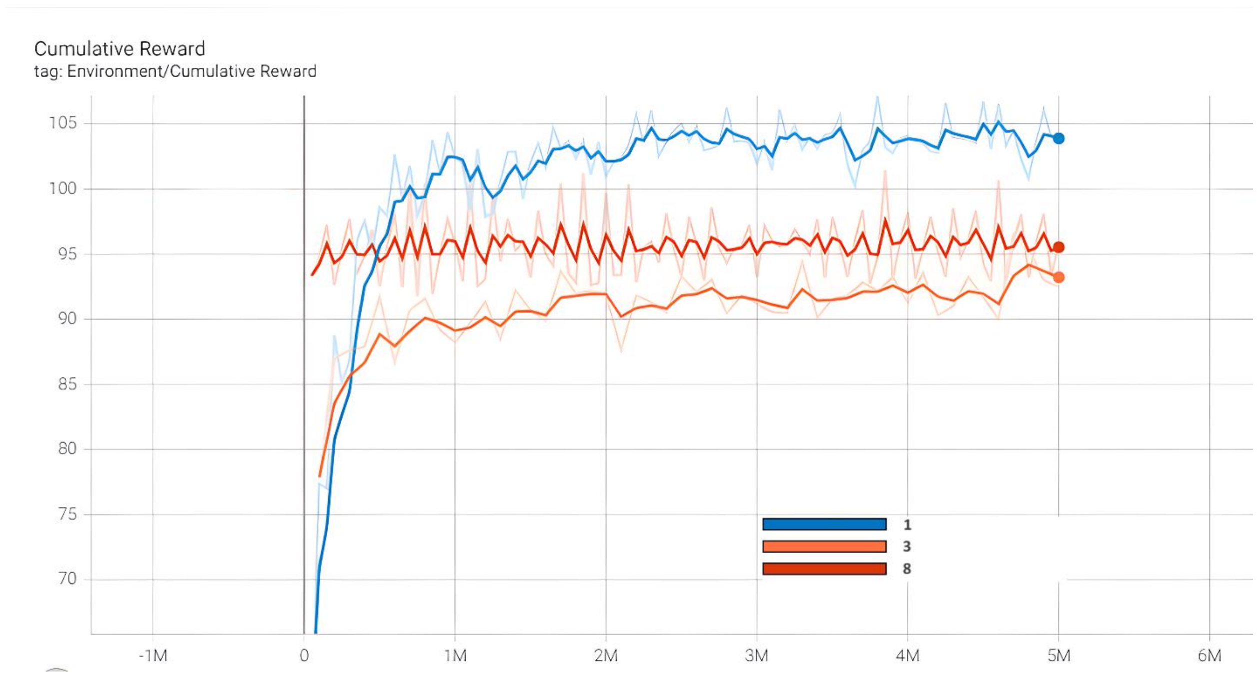


Fig. 23. Graphs with different Num_epoch values in Random Maze 4x4.

4.2.2. Results of Random Maze 6x6

Beta	Reward	Time Cost
0.01	-1949	1h 4m 13s
0.001	-17.02	1h 5m 26s
0.005	-227.4	1h 9m 22s
0.0001	43.44	1h 12m 37s

Table 9. Compare the results when changing the hyperparameter Beta of Random Maze 6x6.

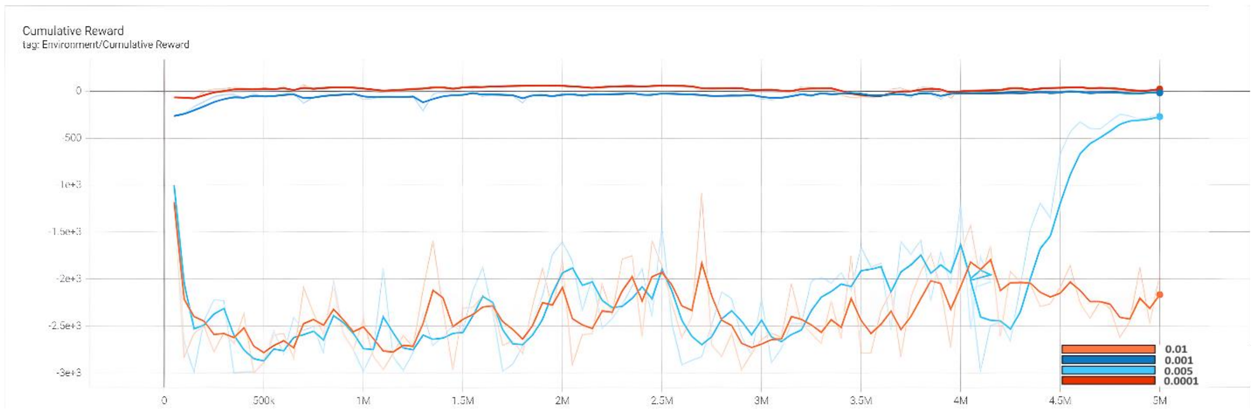


Fig. 24. Graphs with different Beta values in Random Maze 6x6.

Epsilon	Reward	Time Cost
0.1	42.2	1h 6m 50s
0.2	-268.7	1h 9m 22s
0.3	-2056	1h 26m 5s

Table 10. Compare the results when changing the hyperparameter Epsilon of Random Maze 6x6.

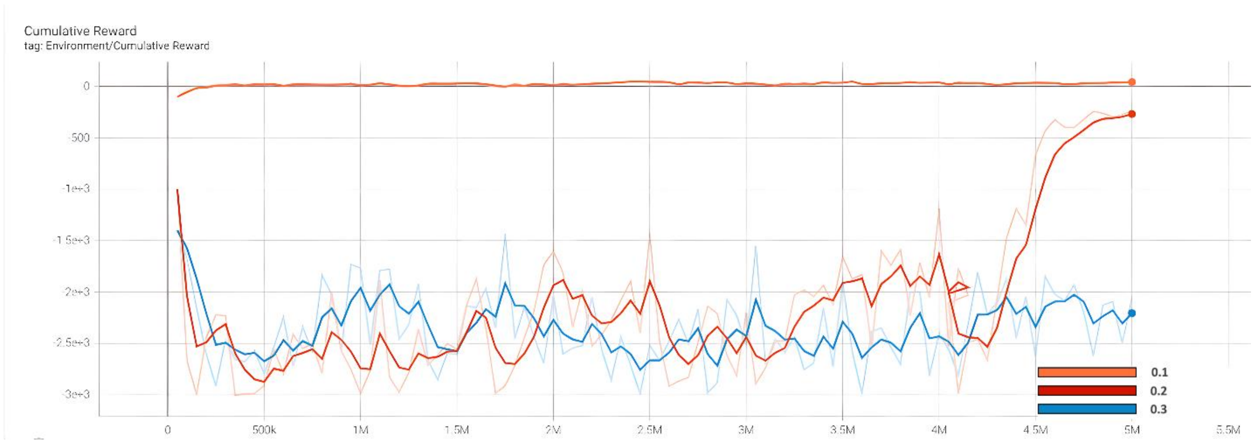


Fig. 25. Graphs with different Epsilon values in Random Maze 6x6.

Lambd	Reward	Time Cost
0.9	-2740	1h 1m 45s
0.95	-227.4	1h 9m 22s

0.99	4.037	1h 36m 0s

Table 11. Compare the results when changing the hyperparameter Lambd of Random Maze 6x6.

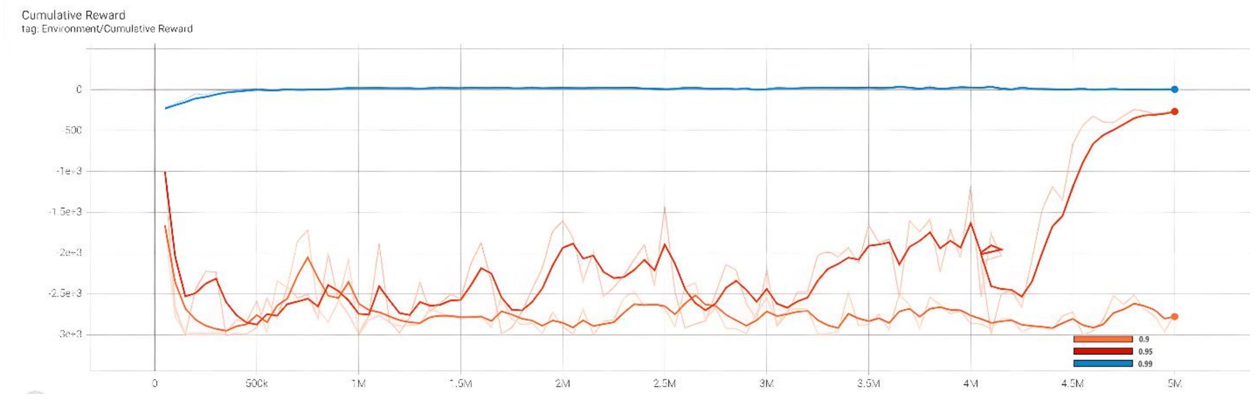


Fig. 26. Graphs with different Lambd values in Random Maze 6x6.

Num_epcho	Reward	Time Cost
1	-121.3	56m 52s
3	-227.4	1h 9m 22s
8	-64.08	1h 48m 21s

Table 12. Compare the results when changing the hyperparameter Num_epcho of Random Maze 6x6.

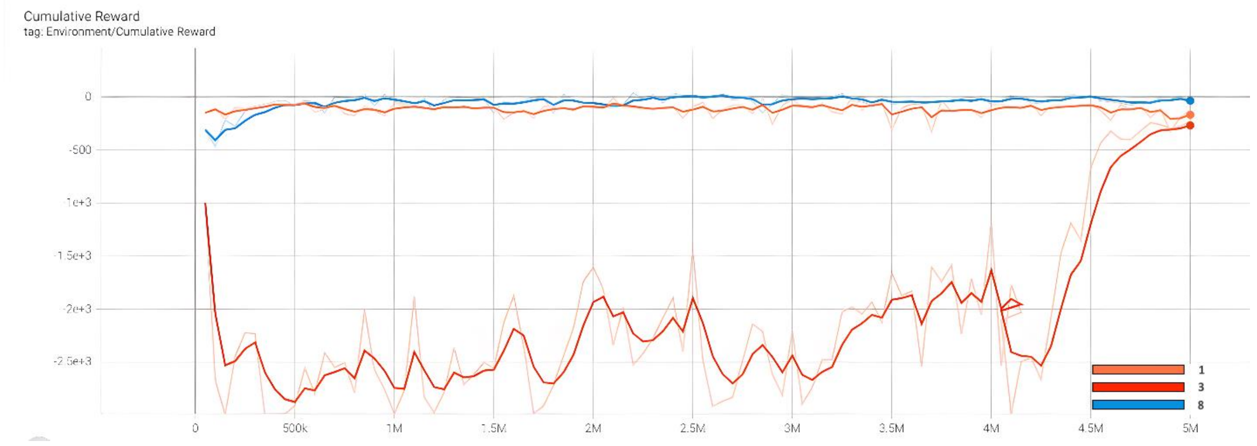


Fig. 27. Graphs with different Num_epoch values in Random Maze 6x6.

4.2.3. Results of Random Maze 8x8

Beta	Reward	Time Cost
0.01	-2785	1h 4m 47s
0.001	-2687	1h 35m 10s
0.005	-1370	1h 39m 53s
0.0001	-2690	1h 13m 11s

Table 13. Compare the results when changing the hyperparameter Beta of Random Maze 8x8.

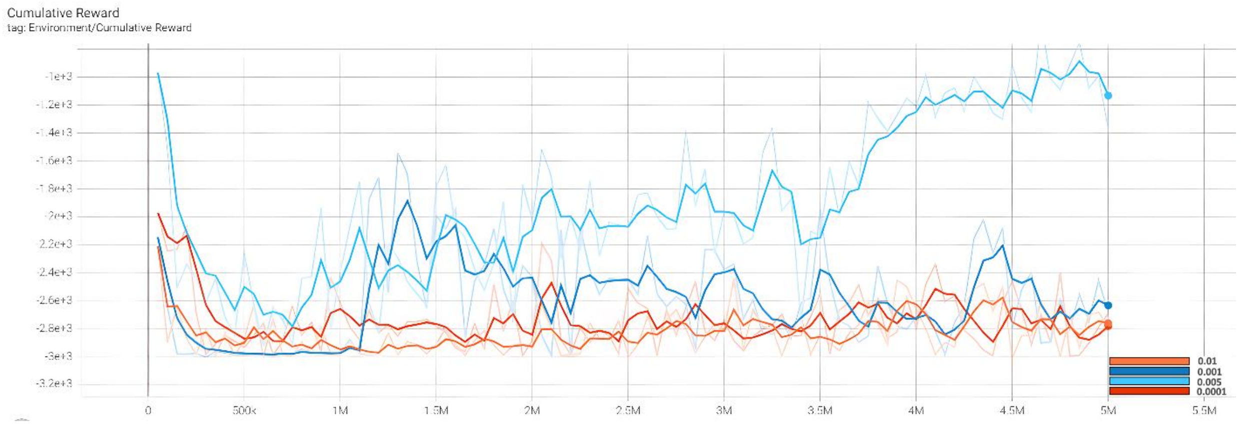


Fig. 28. Graphs with different Beta values in Random Maze 8x8.

Epsilon	Reward	Time Cost
0.1	-114.1	1h 20m 35s
0.2	-1370	1h 39m 53s
0.3	-2540	1h 32m 50s

Table 14. Compare the results when changing the hyperparameter Epsilon of Random Maze 8x8.

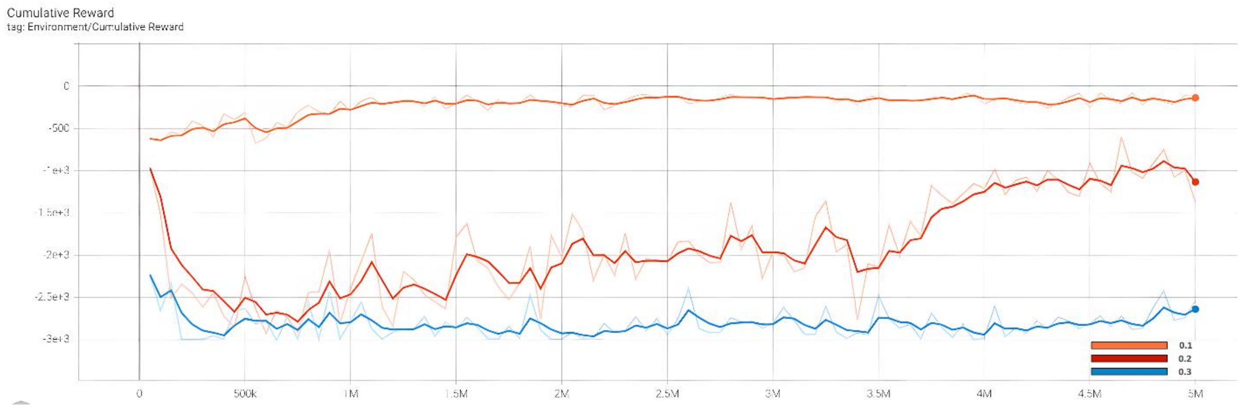


Fig. 29. Graphs with different Epsilon values in Random Maze 8x8.

Lambd	Reward	Time Cost
0.9	-2510	2h 30m 6s
0.95	-1370	1h 39m 53s
0.99	-2661	1h 0m 33s

Table 15. Compare the results when changing the hyperparameter Lambd of Random Maze 8x8.

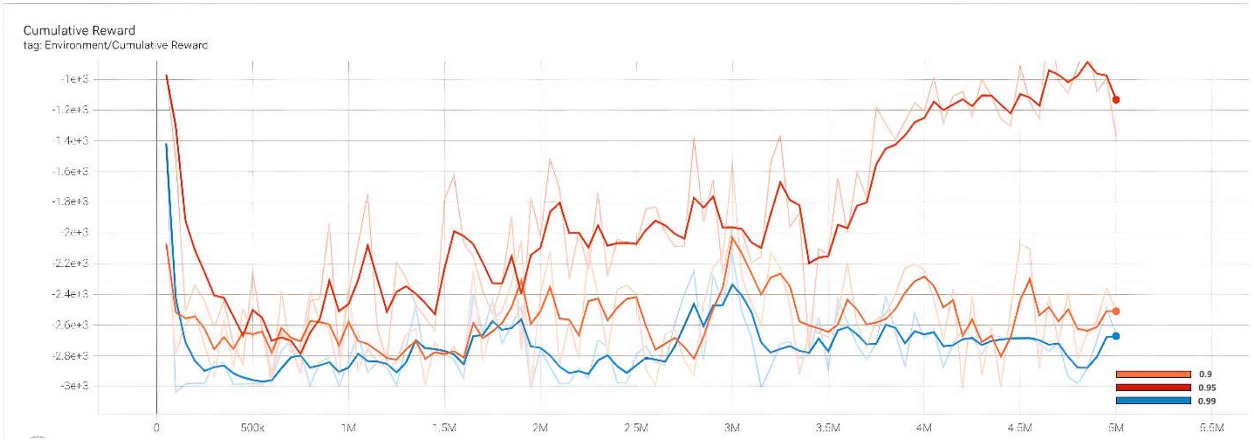


Fig. 30. Graphs with different Lambd values in Random Maze 8x8.

Num_epcho	Reward	Time Cost
1	-2818	41m 11s
3	-1370	1h 39m 53s

8	-2990	3h 10m 29s

Table 16. Compare the results when changing the hyperparameter Num_epcho of Random Maze 8x8.

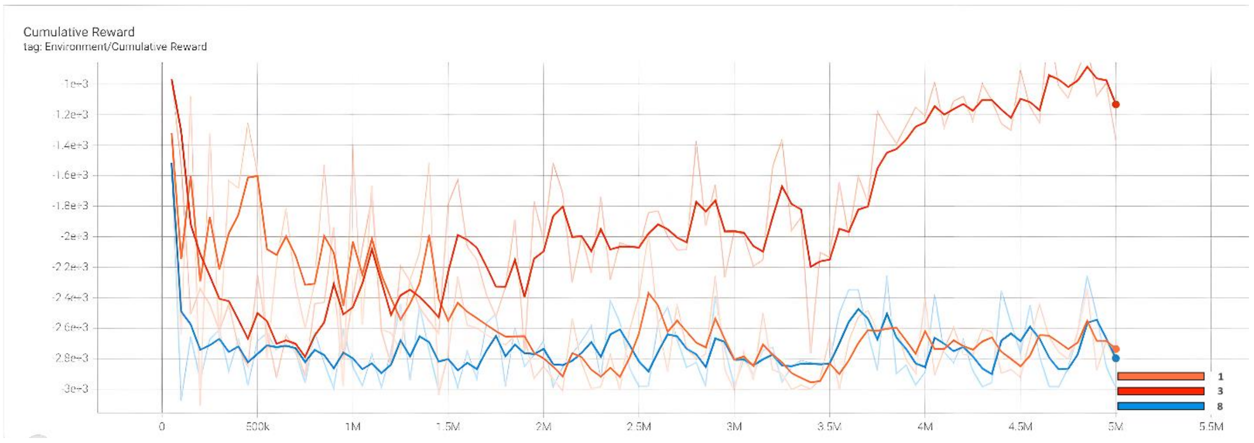


Fig. 31. Graphs with different Num_epoch values in Random Maze 8x8.

4.3. Training Conclusion

Beta: Beta corresponds to the strength of the entropy regularization, which makes the policy "more random." Beta ensures that agents properly explore the action space during training. Table 1 shows that when the BetaBeta decreases to 0.0001, the Agent explores the maze lesser and keeps moving at a certain distance. Increase the Beta. Agent will take more random action to explore the maze faster. However, keep training for a long time; the smallest BetaBeta gets the most reward out of 4 tests.

Epsilon: It can be seen that the reward of epsilon with a value of 0.3 in the first steps substracts significantly from the other two values , and it takes longer to reach the destination. The value 0.2 has the best training result of the three tests from solving the maze and getting the most points.

Lambd: With a Lamda value of 0.9, the training is inferior. The reward is much less than the other two values and the training time is also a bit more. Agent solves the maze about 1 million steps slower. Lambd values from 0.95 - 0.99 give good results, and Agent learns faster.

Num_epoch: Changing this value will make the model train fast or slow and significantly affect the model's performance quality. Num_epoch has a small value (equal to 1) that makes the training unstable, even taking 2 million steps to solve the maze, much worse than the other two values. Increasing this value makes the Agent learn faster and update more consistently. However, the training time will extend; because of the number of passes made through the buffer before the gradient descent step is applied.

5. CONCLUSION AND PERSPECTIVE

This thesis gives the tuning for the PPO algorithm through hyperparameters Beta, Epsilon, Lambd, and Num_epoch. These values are changed, and RL learning results evaluate with the maze solving problem. The results show that the most minor Beta gets the most reward the longer the training. Meanwhile, Epsilon, with a value of 0.2, has the best training result of the three tests from solving the maze and getting the most points. Lambd values from 0.95 - 0.99 give good results, and Agent learns faster. Num_epoch may need to configure carefully, this value makes the Agent learn faster and update more consistently, but in return, the amount of training time will extend.

This research also provides a helpful reference for tuning hyperparameters when redeployment PPO algorithms on novel environments in the future.

6. REFERENCES

[1] A. M. J. Sadik, M. A. Dhali, H. M. A. B. Farid, T. U. Rashid, and A. Syeed, "A comprehensive and comparative study of maze-solving techniques by implementing graph theory," *Proc. - Int. Conf. Artif. Intell. Comput. Intell. AICI 2010*, vol. 1, pp. 52–56, 2010, doi: 10.1109/AICI.2010.18.

[2] R. R. Torrado, P. Bontrager, J. Togelius, J. Liu, and D. Perez-Liebana, "Deep Reinforcement Learning for General Video Game AI," *IEEE Conf. Comput. Intell. Games, CIG,* vol. 2018-August, Oct. 2018, doi: 10.1109/CIG.2018.8490422.

[3] P. Hamalainen, A. Babadi, X. Ma, and J. Lehtinen, "PPO-CMA: Proximal policy optimization with covariance matrix adaptation," *IEEE Int. Work. Mach. Learn. Signal Process. MLSP*, vol. 2020-September, Sep. 2020, doi:

10.1109/MLSP49062.2020.9231618.

[4] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal Policy Optimization Algorithms,” Jul. 2017, Accessed: Feb. 13, 2022. [Online]. Available: http://arxiv.org/abs/1707.06347.

[5] A. Juliani *et al.*, “Unity: A General Platform for Intelligent Agents,” pp. 1–28, 2018, [Online]. Available: http://arxiv.org/abs/1809.02627.

[6] J. T. Kristensen and P. Burelli, “Strategies for Using Proximal Policy Optimization in Mobile Puzzle Games,” *ACM Int. Conf. Proceeding Ser.*, 2020, doi: 10.1145/3402942.3402944.

[7] J. T. Kristensen, A. Valdivia, and P. Burelli, “Estimating Player Completion Rate in Mobile Puzzle Games Using Reinforcement Learning,” *IEEE Conf. Comput. Intell. Games, CIG*, vol. 2020-Augus, pp. 636–639, 2020, doi: 10.1109/CoG47356.2020.9231581.

[8] E. Elgeldawi, A. Sayed, A. R. Galal, and A. M. Zaki, “Hyperparameter tuning for machine learning algorithms used for arabic sentiment analysis,” *Informatics*, vol. 8, no. 4, pp. 1–21, 2021, doi: 10.3390/informatics8040079.

[9] E. F. Morales and J. H. Zaragoza, “An introduction to reinforcement learning,” *Decis. Theory Model. Appl. Artif. Intell. Concepts Solut.*, pp. 63–80, 2011, doi: 10.4018/978-1-60960-165-2.ch004.

[10] T. Nakayashiki and T. Kaneko, “Learning of Evaluation Functions via Self-Play Enhanced by Checkmate Search,” *Proc. - 2018 Conf. Technol. Appl. Artif. Intell. TAAI 2018*, pp. 126–131, Dec. 2018, doi: 10.1109/TAAI.2018.00036.

[11] OpenAI *et al.*, “Dota 2 with Large Scale Deep Reinforcement Learning,” Dec. 2019, Accessed: Feb. 13, 2022. [Online]. Available: http://arxiv.org/abs/1912.06680.

[12] T. Chen, K. Zhang, G. B. Giannakis, and T. Basar, “Communication-Efficient Policy Gradient Methods for Distributed Reinforcement Learning,” *IEEE Trans. Control Netw. Syst.*, 2021, doi: 10.1109/TCNS.2021.3078100.

[13] W. Zhu and A. Rosendo, “A Functional Clipping Approach for Policy Optimization Algorithms,” *IEEE Access*, vol. 9, pp. 96056–96063, 2021, doi: 10.1109/ACCESS.2021.3094566.

[14] X. Arriaga and J. A. Ruiz, “The Unity Toolkit used to train agents,” 2021.

[15] V. Bellot *et al.*, “How to Generate Perfect Mazes ? To cite this version : HAL Id : hal-03174952 Preprint submitted to Elsevier,” 2021.

[16] R. Jafri, R. L. Campos, S. A. Ali, and H. R. Arabnia, “Visual and Infrared Sensor Data-Based Obstacle Detection for the Visually Impaired Using the Google Project Tango Tablet Development Kit and the Unity Engine,” *IEEE Access*, vol. 6, pp. 443–454, 2017, doi: 10.1109/ACCESS.2017.2766579.

[17] T. Kim and J. H. Lee, “Effects of Hyper-Parameters for Deep Reinforcement Learning in Robotic Motion Mimicry: A Preliminary Study,” *2019 16th Int. Conf. Ubiquitous Robot. UR 2019*, pp. 228–235, Jun. 2019, doi: 10.1109/URAI.2019.8768564.

Phan Thanh Hung graduated from FPT University, Hanoi, Vietnam, with a degree in Computer Science in 2022.
His current research interests include developing 2D, and 3D video games with Unity, specializing in Artificial Artificial and Code Optimization.

Mac Duy Dan Truong graduated from FPT University, Hanoi, Vietnam, with a degree in Computer Science in 2022.
His current research interests include Artificial Intelligence, Deep Learning, and Computer Vision.

Phan Duy Hung has worked as a Lecturer at FPT University in Hanoi, Vietnam, since 2009 and has also served as the Head of Department and Director of the Master Program in Software Engineering.
In 2008, Phan Duy Hung obtained his Ph.D. from the INP Grenoble in France.
His current research interests include Digital signal and image processing, IoT, BigData, Artificial Intelligence, Measurement and Control Systems, and industrial networking.