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

**MDD Truonga PT Hunga PD Hunga**

*a FPT University, Hoa Lac High Tech Park, Hanoi, 10000, Vietnam*

truongmddhe141711@fpt.edu.vn, hunghe140966@fpt.edu.vn, hungpd2@fe.edu.vn

Received: xx, 2022

**Table of Contents**

[1. INTRODUCTION 3](#_Toc98064845)

[1.1. Objective 3](#_Toc98064846)

[1.2. Background 3](#_Toc98064847)

[1.3. Design 4](#_Toc98064848)

[2. METHODOLOGY 5](#_Toc98064849)

[2.1. Introduction to machine learning 5](#_Toc98064850)

[2.2. Elements of Reinforcement Learning System 5](#_Toc98064851)

[2.3. Application of Reinforcement Learning 6](#_Toc98064852)

[2.4. Introduction to Deep Reinforcement Learning (DRL) 7](#_Toc98064853)

[2.5. Policy Gradient (PG) 8](#_Toc98064854)

[2.6. Proximal Policy Optimization (PPO) 9](#_Toc98064855)

[2.7. ML-Agents 10](#_Toc98064856)

[2.8. Hunt and Kill Algorithm 12](#_Toc98064857)

[3. IMPLEMENTATION 15](#_Toc98064858)

[3.1. Unity Setup 15](#_Toc98064859)

[3.2. Maze Design 17](#_Toc98064860)

[3.3. Application Structure 19](#_Toc98064861)

[3.4. Environment Logic 19](#_Toc98064862)

[3.5. Hyperparameters Configuration 21](#_Toc98064863)

[4. TRAINING RESULTS 21](#_Toc98064864)

[4.1. Results of Fixed Maze 8x8 21](#_Toc98064865)

[4.2. Results of Random Maze 4x4 24](#_Toc98064866)

[4.3. Results of Random Maze 6x6 27](#_Toc98064867)

[4.4. Results of Random Maze 8x8 29](#_Toc98064868)

[4.5. Training Conclusion 32](#_Toc98064869)

[5. CONCLUSION AND PERSPECTIVE 32](#_Toc98064870)

[6. REFERENCES 33](#_Toc98064871)

**Abstract**— The proximal Policy Optimization algorithm is the default reinforcement algorithm used in the ML-Agents toolkit. This algorithm can alternate between sampling data through interaction with the environment and optimizing a "surrogate" cost function using stochastic gradient descent. 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 use the default given values from the algorithm's creator or use the machine to perform this exploration and automatically select the optimal 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 paper will help to make appropriate choices at hyperparameters in concrete and practical projects.

*Keywords:* Reinforcement Learning, Proximal Policy Optimization, ML-Agents, Maze-Solving.

# 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 that consists of a different branch of passages where the solver aims to reach the destination by finding the most efficient route within the shortest possible time[1]. Artificial Intelligence plays a vital role in defining the best possible way of solving any maze effectively. That is when Reinforcement Learning[2] came in handy.

A vital issue in the usability of an RL method is sensitivity to hyperparameters[3]. Learning complex tasks can take hours or days, fine-tuning hyperparameters is tedious. 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 paper also provides a helpful guideline for tuning hyperparameters when redeployment algorithms on novel environments in the future.

## 1.2. Background

Several approaches have been proposed for RL with neural network function approximators in recent years. The paper published by John Schulman has proposed improving the current state of affairs by introducing an algorithm that attains data efficiency and reliable performance, called Proximal Policy Optimization (PPO)[4]. To optimize policies, the research team alternate between sampling data from the policy and performing several epochs of optimization on the sampled data. They concluded that these methods have the stability and reliability of trust-region strategies but are much simpler to implement. They require only a few lines of code change to a vanilla policy gradient implementation, are applicable in more general settings, and have better overall performance. However, the algorithm was successful on various problems without tuning hyperparameter values, meaning that the results still did not achieve the best possible outcome.

While traditionally a labor-intensive task, the testing of game content is progressively becoming more automated by PPO in a casual mobile puzzle game with a specific focus on improving its reliability in training and generalization during game playing[6]. This study has successfully adapted the popular RL method PPO to a production-grade puzzle game for training play-testing agents by introducing a reset strategy where the environment is reset after a fixed number of steps, but not considering hyperparameter tuning. This works well in specific games but may not always be feasible in other types. To finetune even further, we also tinker with anchor box generation in this study. Because anchor boxes theoretically cannot be generalized well for every dataset, nevertheless having a significant impact on the performance of one-stage detectors, we empirically choose the best possible aspect ratios and scales for the driving dataset BDD100K[7], where objects vary from large upfront trucks to tiny further cars.

Other autonomous game-playing agents have developed different rule-based systems for machine learning methods. Many efforts have also been put into creating Agents that behave in a way that resembles as closely as possible the way a human player would play. To estimate the player completion rate of several levels in Lily’s Garden by Tactile Games[8], the research team developed a set of PPO-based reinforcement learning agents. It evaluated how the number of steps taken by the Agent for completing the levels relates to the behavior of a sample of ∼900,000 players. The results based on ∼60% of the game mechanics – demonstrate that the two-step training scenario leads to the most proficient Agent[8]. In contrast, the Agent attains the most significant correlation to real players’ completion rates with the one-step curriculum. The work is only for a limited subset of levels with default values hyperparameter, 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 machine learning

With the development in computing technology and the inception of new intelligent algorithms, the goal of Artificial Intelligence (AI) has become a step closer to the goal of mimicking the human brain. In that area, a branch that is becoming more and more important is reinforcement learning (RL)[9]. RL is the type of learning guided by a specific objective. 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.

An agent learns by interacting with an unknown environment to maximize a reward, typically in a try-and-error way. The agent receives feedback in terms of a reward (or punishment) from the environment; then, it uses this feedback to train itself and collect experience and knowledge about the environment[10]. This is the most common way of learning for a child, who does something and observes what happens. Another relevant characteristic of an RL problem is that in any situation, the agent has to choose between exploiting its current knowledge of the environment (perform an action already tried previously in that situation) or exploring actions never tried before in that situation.

## 2.2. Elements of Reinforcement Learning System

A reinforcement learning system has four primary components, in addition to the agent and the environment: a policy, a reward, a value function, and, optionally, a model of the environment[10]. A policy describes how an agent acts at a specific point in time. A policy, in general, is a mapping between environmental conditions to actions to the activities the agent makes in the environment. The policy can be a simple function or lookup table in the most specific circumstances, but it can also entail complicated function computations. The policy is the foundation of the agent's knowledge. A reward defines the goal of a reinforcement learning problem. The agent's actions result in a reward at each time step. The agent's ultimate goal is to maximize the overall amount of money it receives. 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 primary approach to influence the policy is through the reward; if a policy-selected action results in a low reward, the policy can be altered to select a different 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 entire aggregated quantity of rewards that the agent can expect in the future if it starts from that state is the value of a state. Values represent the long-term attractiveness of a set of states, taking into consideration the most likely future states as well as the rewards derived 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. Rewards are primary and immediate; values, on the other hand, are secondary projections of rewards. There are no values without rewards, and the sole point of calculating values is to obtain additional 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.

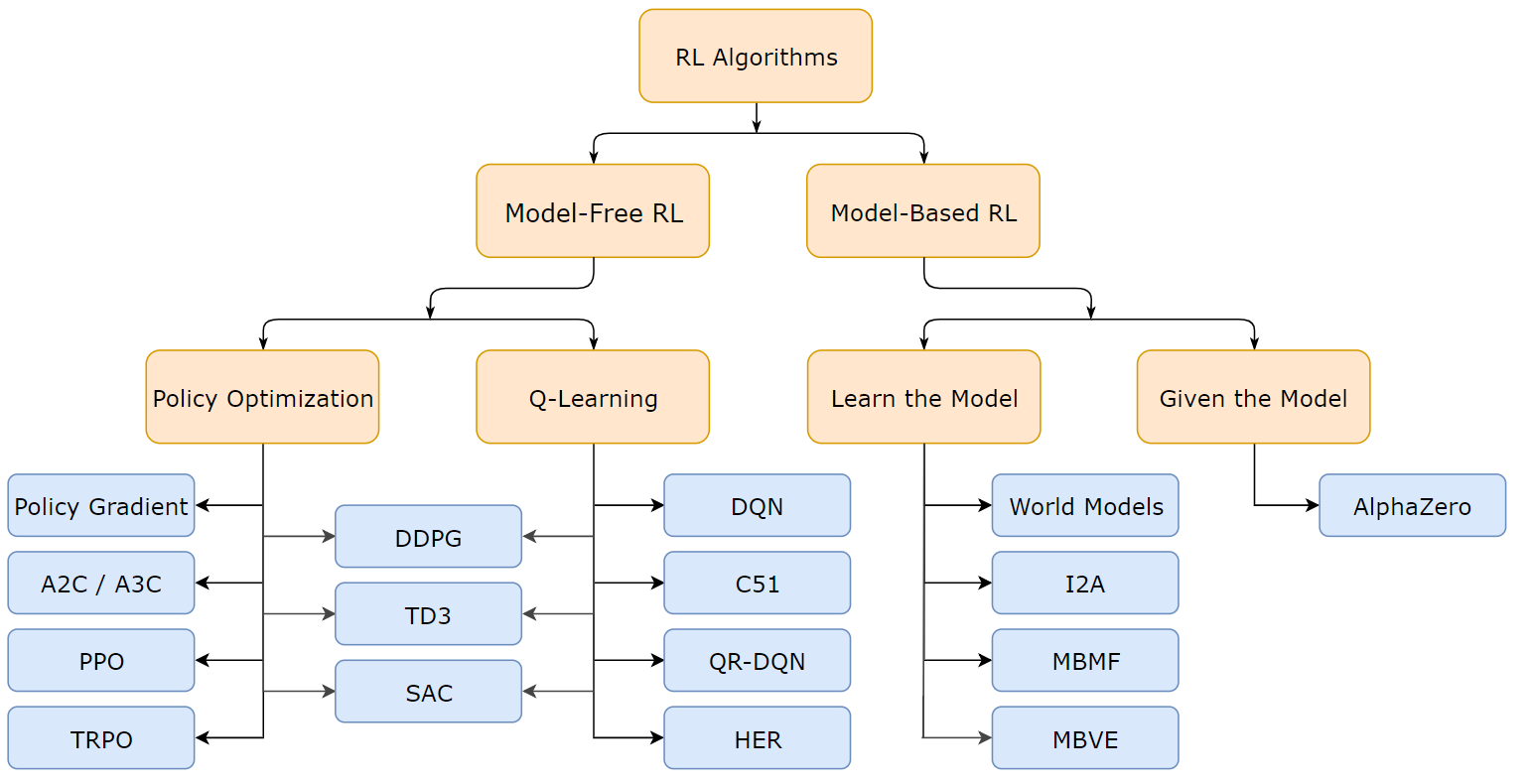
## 2.3. Application of Reinforcement Learning

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 main application of the RL technique is for solving problems by a feedback system (rewards and penalties) applied on an agent that operates in an environment and needs to move through a series of states to reach a pre-defined final state. A typical example is a rat (agent) trying to find the shortest route from a starting cell to a target cheese cell in a maze (environment). The agent is experimenting and exploiting past experiences (episodes) to achieve its goal. It may fail again and again, but hopefully, after lots of trial and error (rewards and penalties), it will solve the problem.

Look at an application in the gaming frontier, specifically AlphaGo Zero[11]. Using reinforcement learning, AlphaGo Zero learned the game of Go from scratch. It learned by playing against itself. After 40 days of self-training, Alpha Go Zero was able to outperform the version of Alpha Go known as Master that has defeated world number one Ke Jie. It only used black and white stones as input features and a single neural network. A simple tree search that relies on the single neural network is used to evaluate positions moves and sample moves without using any Monte Carlo rollouts. 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)[12].

## 2.4. Introduction to Deep Reinforcement Learning (DRL)

 In recent years, deep reinforcement learning has been one of the most concerned directions in artificial intelligence. It combines the perceptual ability of deep learning with the decision-making ability of reinforcement learning and directly controls agents' behavior through high-dimensional perceptual input learning. Generally speaking, it applies the neural network structure to the process of reinforcement learning. Nowadays, the major deep reinforcement learning algorithms include Deep Q Network, Deep Deterministic Policy Gradient, Asynchronous Advantage Actor-Critic, Proximal Policy Optimization (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

## 2.5. Policy Gradient (PG)

Policy Gradient (PG)[13] 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:

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

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

R(τ) means the rewards of the trajectory.

And the PG use this policy to update the θ:

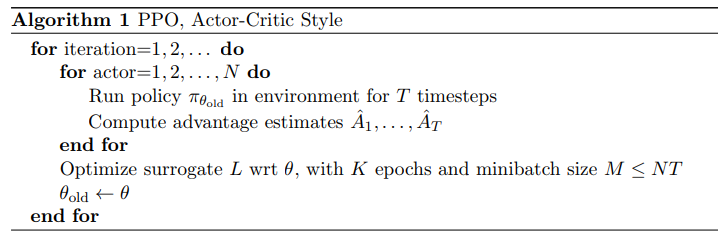
The advantage functions A and rewrites the policy gradient:

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.

## 2.6. Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO)[14] is an optimization algorithm that can improve the data efficiency and reliable performance of Trust Region Policy Optimization (TRPO) while only using the first-order optimization. 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:

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 prevent the incentive factors from moving rt outside the interval [1-ε, 1+ε]. The steps of the PPO algorithm are:

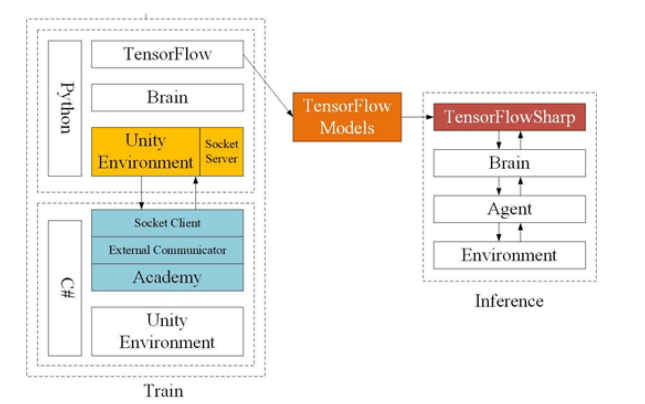


**Fig. 2.** PPO Algorithm

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. PPO will optimize the surrogate objective function for K epochs at the end of one iteratio.

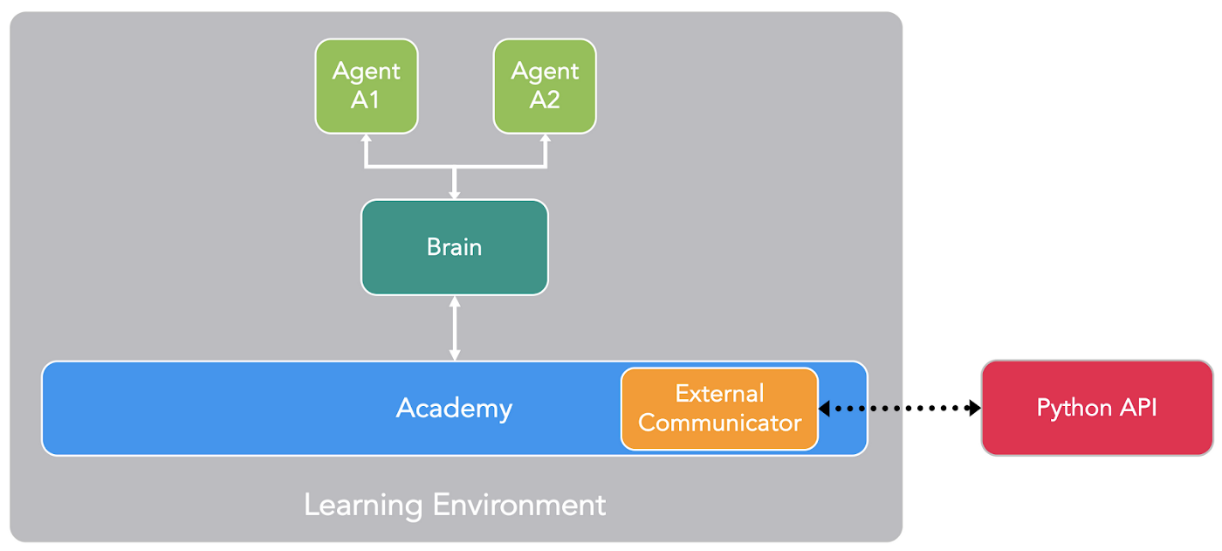
## 2.7. ML-Agents

The Unity Machine Learning Agents Toolkit (ML-Agents Toolkit) is an open-source project that enables games and simulations to serve as environments for training intelligent agents. Using a simple-to-use Python API[12], agents are trained using reinforcement learning, imitation learning, neuroevolution, or other machine learning methods. 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. In the Inference phase, ML-Agents uses TensorFlow Sharp to read the trained model and use it as a Brain in Unity Environment to precisely guide the Agent to interact with the environment.



**Fig. 3.** Unity ML-Agents Structure

There are three key components of Unity ML-Agents: Learning Environment, Python API, and External Communicator[15]. The Learning Environment includes three additional components that help organize Unity scenarios.



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

Key Components:

**• Learning Environment**: This contains the Unity scene and all the game characters. The Unity scene provides the environment where agents observe, act, and learn.

**• Python Low-Level API**: This contains a low-level Python interface for interacting and manipulating a learning environment.

**• External Communicator**: This connects the Learning Environment with the Python Low-Level API. It lives within the Learning Environment.

**• Python Trainers**: This contains all the machine learning algorithms that enable training agents.

**• Gym Wrapper**: A common way machine learning researchers interact with simulation environments is via a Gym wrapper provided by OpenAI.

**• Agents**: are attached to a Unity GameObject (any character within a scene) and handle generating its observations, performing the actions it receives, and assigning a reward (positive/negative) when appropriate.

**• Behavior**: defines specific attributes of the agent, such as the number of actions that agent can take. A Behavior can be of one of three types: Learning, Heuristic, or Inference.

## 2.8. Hunt and Kill Algorithm

The Hunt and Kill Algorithm[16] 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. Select a random cell. This is the current cell. Add it to the visited list

2. Randomly pick a cell adjacent to the current one in the visited list. This becomes the current cell.

3. Remove the edge between the previous cell and the current cell. Add the current cell to the visited list.

4. Repeat 2 and 3 until travel is no longer possible

5. Scan the grid top->bottom, left->right

o If a non-visited cell is found

The cell becomes the current cell

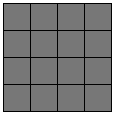
Go to 2

o Else

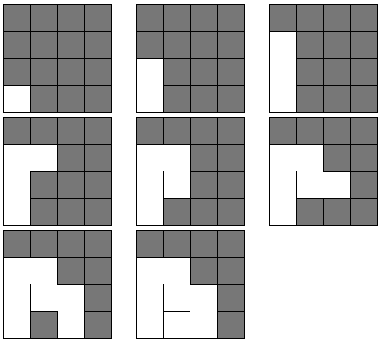
The algorithm is complete

**An example**

This is a basic 4×4 grid:

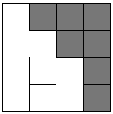


The walk phase is a sequence of frames here; it is not that interesting until it reaches a dead-end.

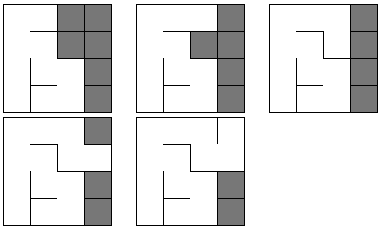


Moreover, a leisurely inebriated stroll comes to a screeching halt. All possible directions lead either out of bounds or into an already-visited neighbor. At this point, the recursive backtracker would begin backtracking, looking for a previously visited cell in the stack that had unvisited neighbors.

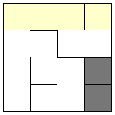
Beginning at the first row, we begin scanning each row for an unvisited cell with a visited neighbor at the first row. It turns out to be our lucky day: our very first cell is a match: unvisited, with a visited neighbor. We connect the two:



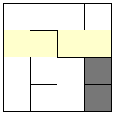
And then start a random walk from the new starting point:



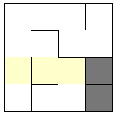
Stuck again, so we go hunting. There are no cells in the first row that match:



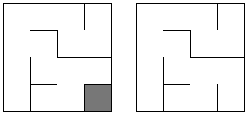
And no matches in the second row, either. (Remember, we are looking for *unvisited* cells with *visited* neighbors)



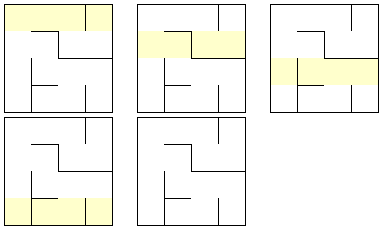
The third row, however, has a match in its last cell:



So, we connect that unvisited cell to any one of its visited neighbors (at random) and do our random walk:



Furthermore, we again stub our digital toes on another dead-end. We are stuck, so we go hunting again, looking row-by-row for an unvisited cell.



The scan is completed without finding any unvisited cells, so the algorithm terminates and leaves us with our maze.

# 3. IMPLEMENTATION

## 3.1. Unity Setup

**Step 1: Install Python**

Download and install Python for windows. Using Pycharm, we can build a different mange environment for different distributions of Python. Python 3.7 or 3.8 is required. Our project use version 3.8.

**Step 2: Setup and Activate Environment**

Open cmd in the project location

Create a virtual environment:

python -m venv venv

To use this environment, we must activate it. Go to venv/Scripts/activate in cmd.

Next, install Pytorch. Install this package using pip, a package management system used to install Python packages. In the same CMD, type in the following command:

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

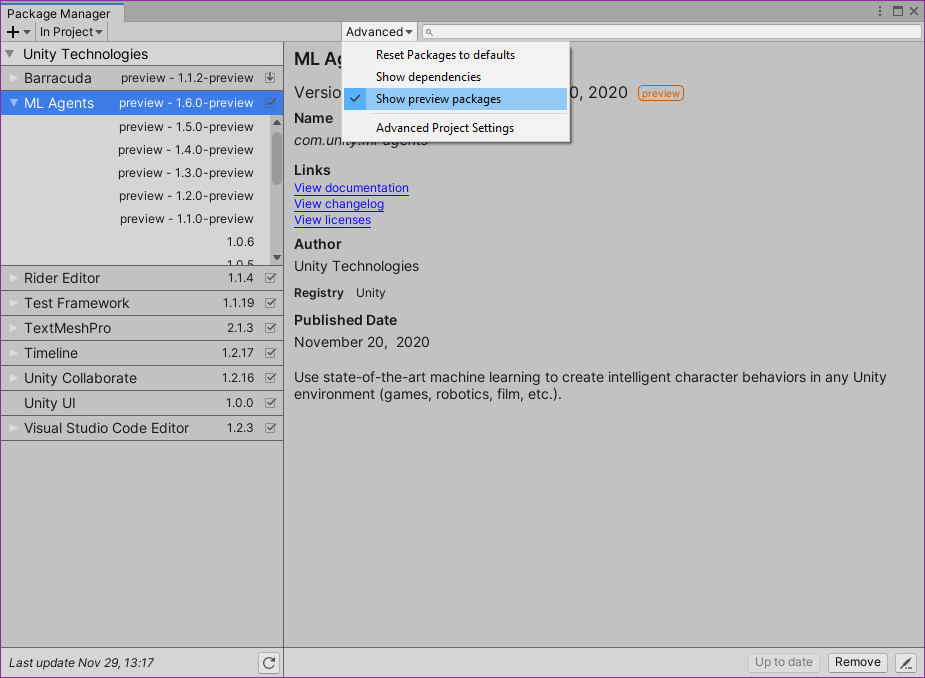
Go to Unity:

ML-Agents is installed via the Unity Package Manager.

In Unity, click **Window > Package Manager** to open the Package Manager.

Within the Package Manager window:

* 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



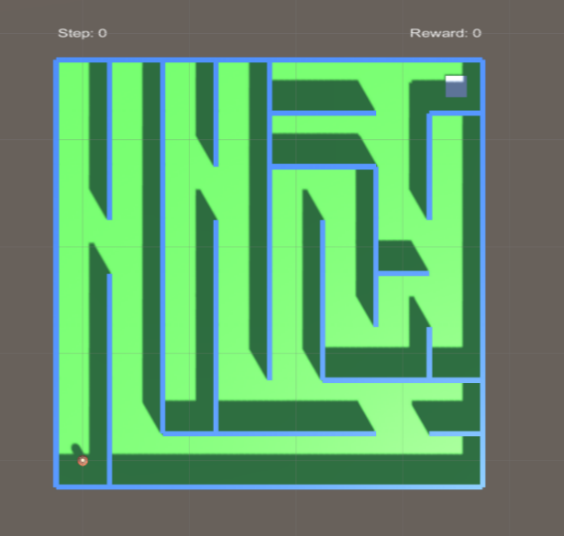
**Fig. 5.** Unity Package Manager

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



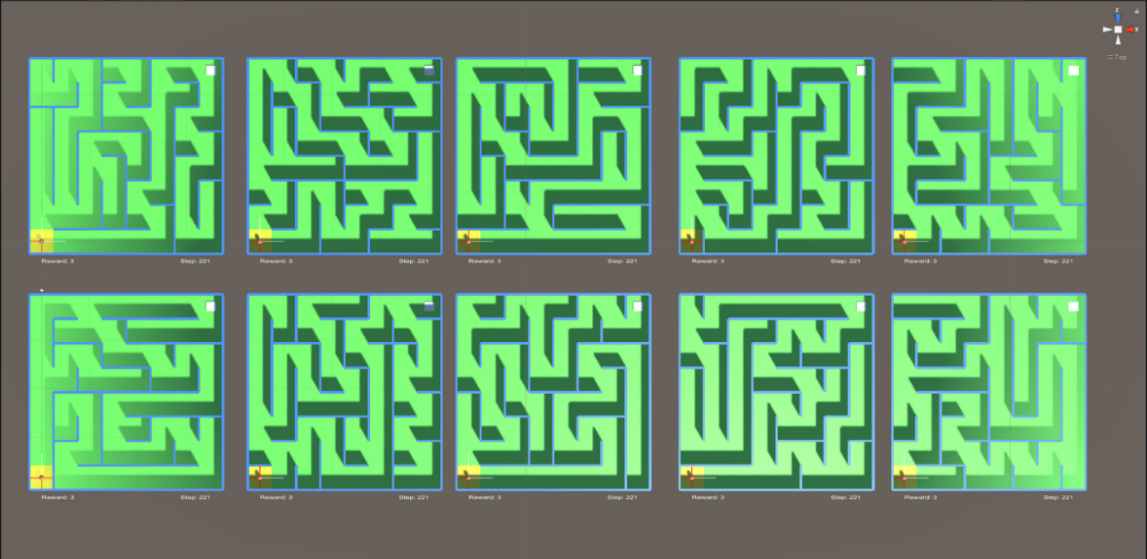
**Fig. 6.** Fixed Maze 8x8



**Fig. 7.** Random Maze 4x4

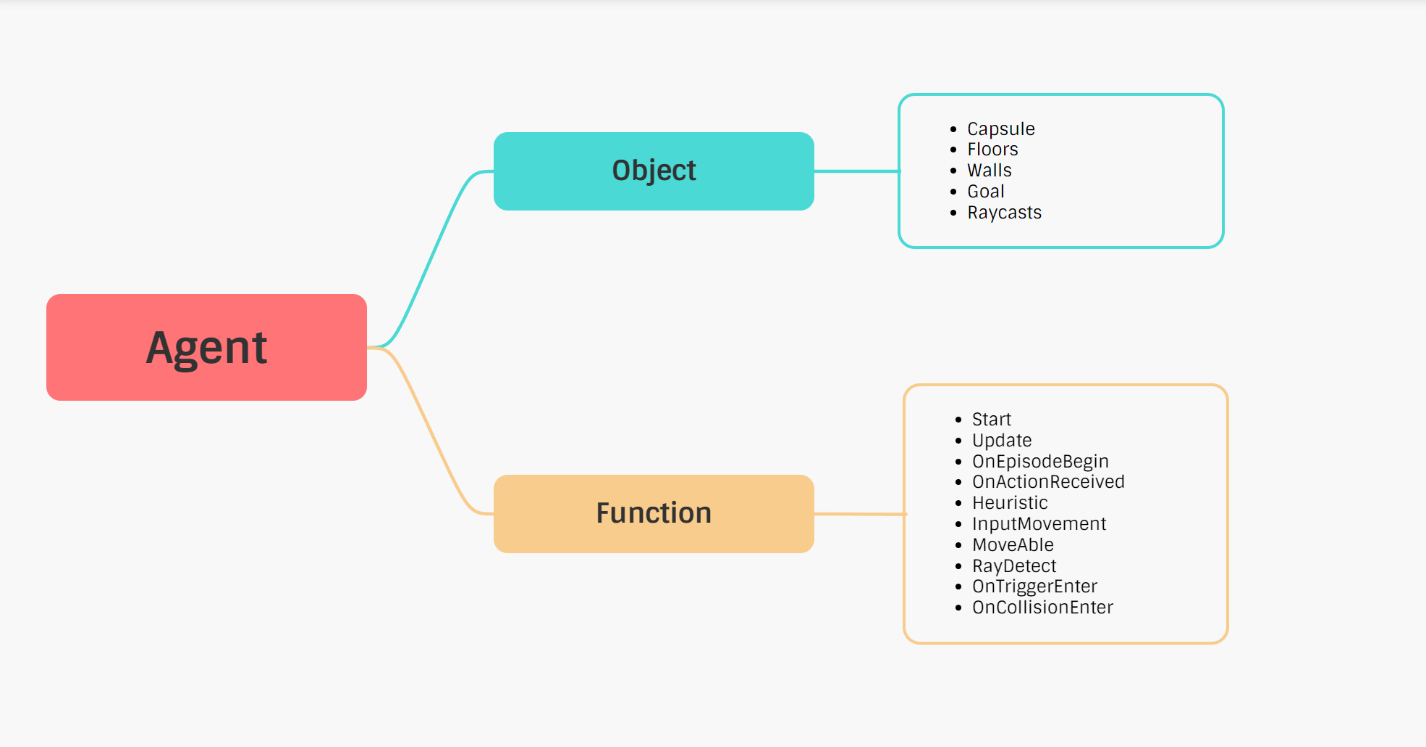


**Fig. 8.** Random Maze 6x6

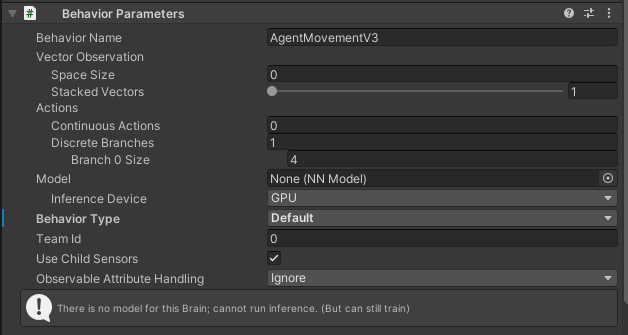


**Fig. 9.** Random Maze 8x8

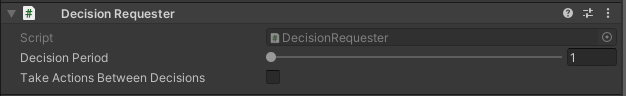
## 3.3. Application Structure



**Fig. 10.** Agent Structure



**Fig. 11.** Behavior Parameter of Agent

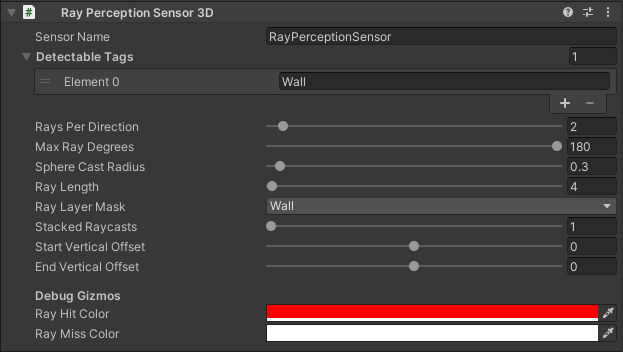


**Fig. 12.** Decision Requester of Agent

## 3.4. Environment Logic

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[17] - the agent's observations, arranged according to the other four raycasts.
* The total size of the created observations is: (Observation Stacks) \* (1 + 2 \* Rays Per Direction) \* (Num Detectable Tags + 2) = 1 \* (1 + 2 \* 2) \* (1 + 2) = 15.



**Fig. 13.** Ray Perception Sensor of Agent

Agent moves in 4 directions (up and down, left, right). 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.

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.



**Fig. 14.** Simple 8x8 Maze.

## 3.5. Hyperparameters Configuration

These are specific hyperparameters that are important in the context of PPO and the standard training parameters[18].

* **Beta**: This controls the strength of the entropy regularization so that the agent can explore spaces during training. Beta typically has a value between 1e-4 and 1e-2.
* **Epsilon**: This controls how swiftly the policy can diverge from an older policy. A smaller value has stable updates on the policy. Epsilon typically has a value between 0.1 and 0.3.
* **Lambd**: The regularization factor used in calculating GAE. Typically a low value resembles using the current advantage value, and a high value signifies using the actual advantages received from the environment (high variance). Lambd typically has a value between 0.9 and 0.95.
* **Num\_epoch**: The number of passes made through the buffer before the gradient descent step is applied. Decreasing this will lead to slower and stable updates. Num\_epcho typically has a value 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 |

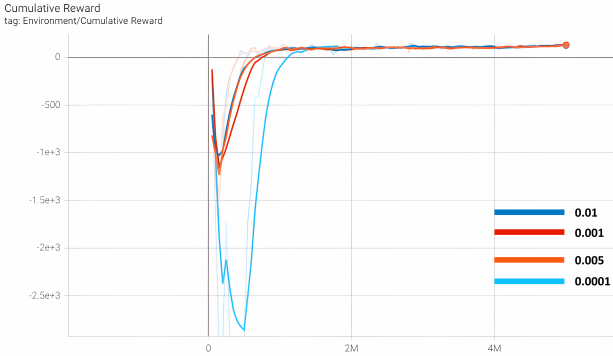
**Fig. 15.** Default configuration hyperparameters.

# 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 |

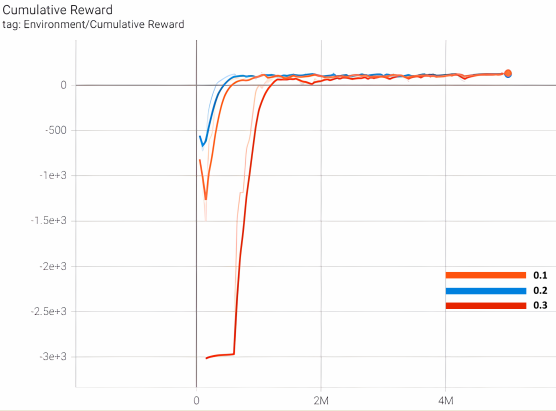
**Table 1.** Compare the results when changing the hyperparameter Beta.



**Fig. 16.** Graphs with different Beta values.

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

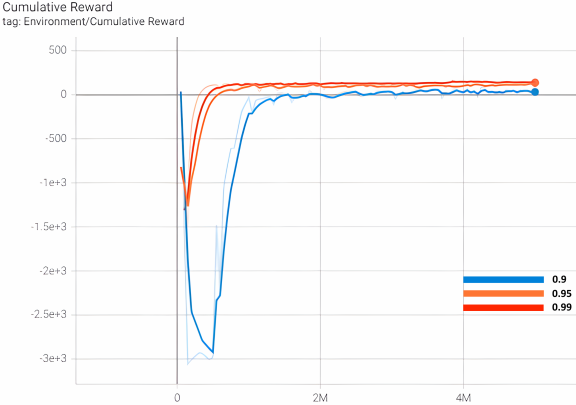
**Table 2.** Compare the results when changing the hyperparameter Epsilon.



**Fig. 17.** Graphs with different Epsilon values.

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

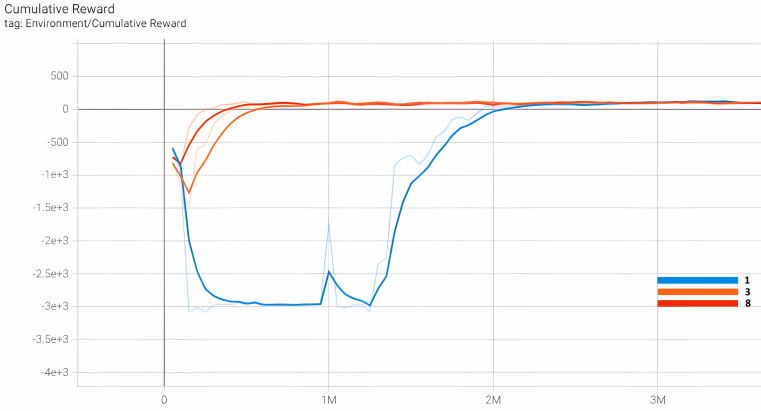
**Table 3.** Compare the results when changing the hyperparameter Lambd.



**Fig. 18.** Graphs with different Lambd values.

|  |  |  |
| --- | --- | --- |
| 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.

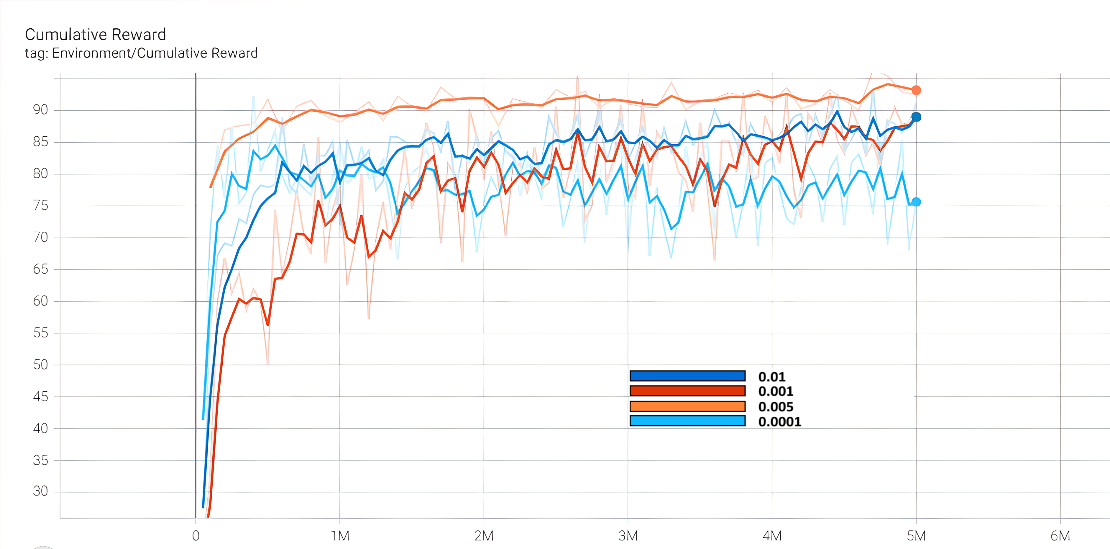


**Fig. 19.** Graphs with different Num\_epoch values.

## 4.2. 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 |

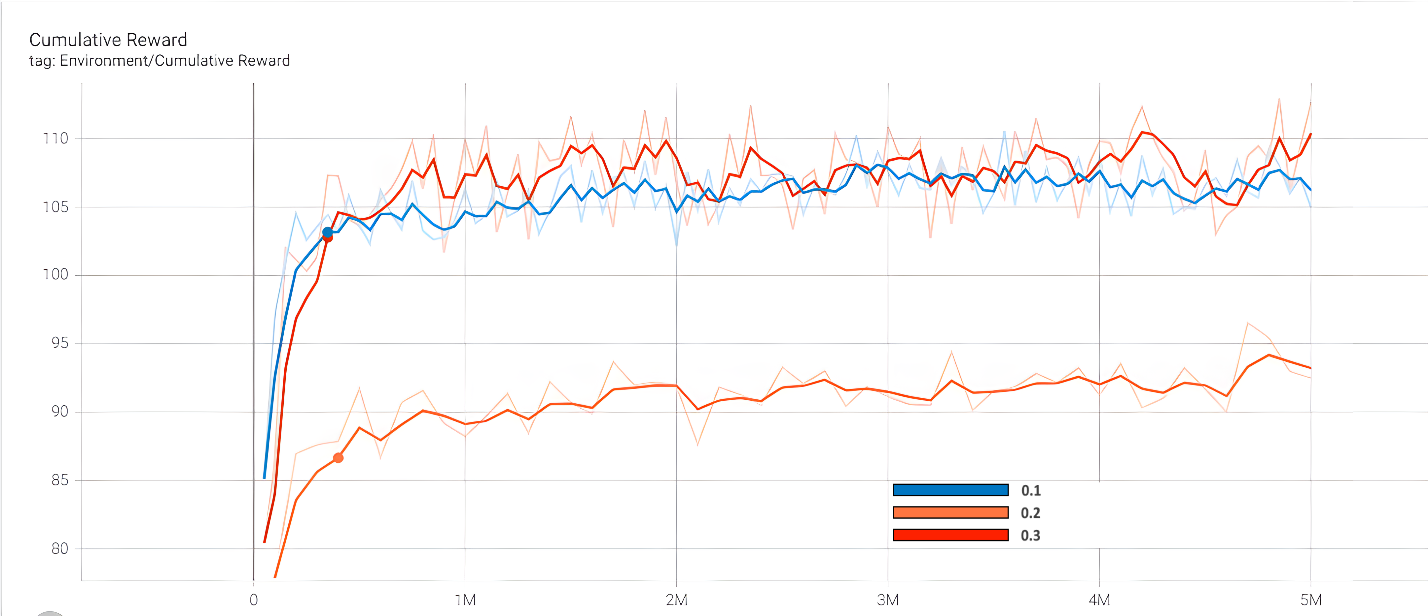
**Table 5.** Compare the results when changing the hyperparameter Beta.

****

**Fig. 20.** Graphs with different Beta values.

|  |  |  |
| --- | --- | --- |
| 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 |

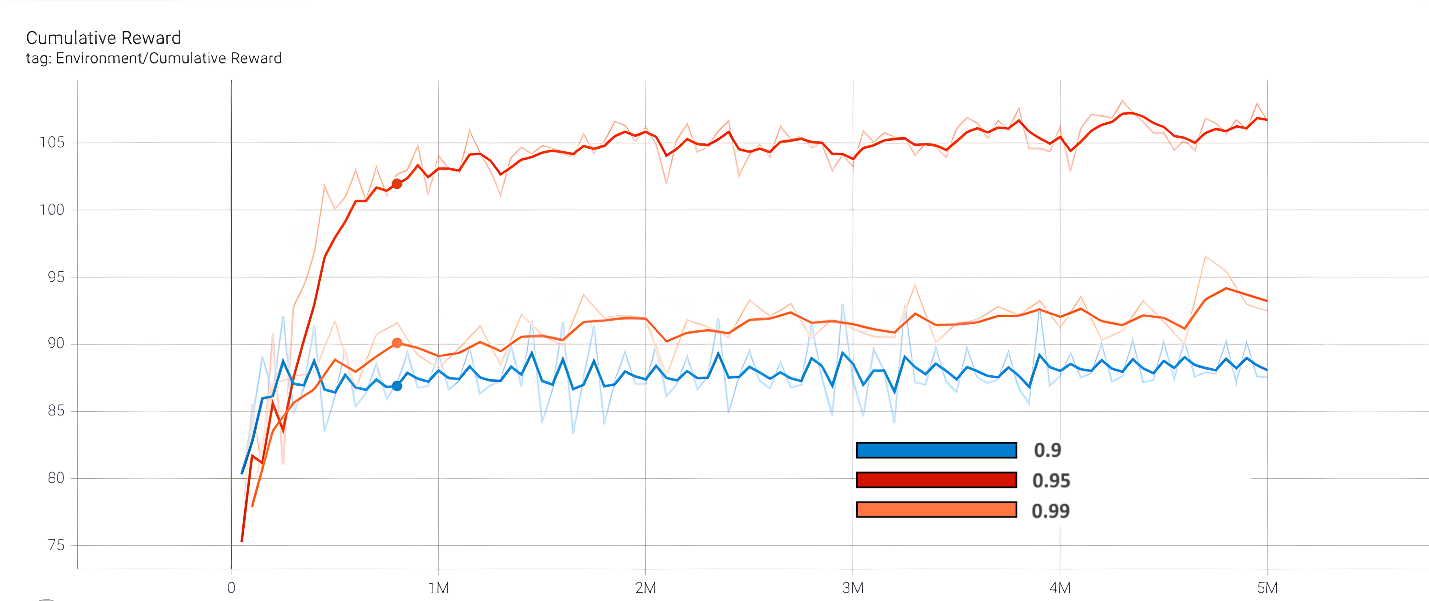
**Table 6.** Compare the results when changing the hyperparameter Epsilon.



**Fig. 21.** Graphs with different Epsilon values.

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

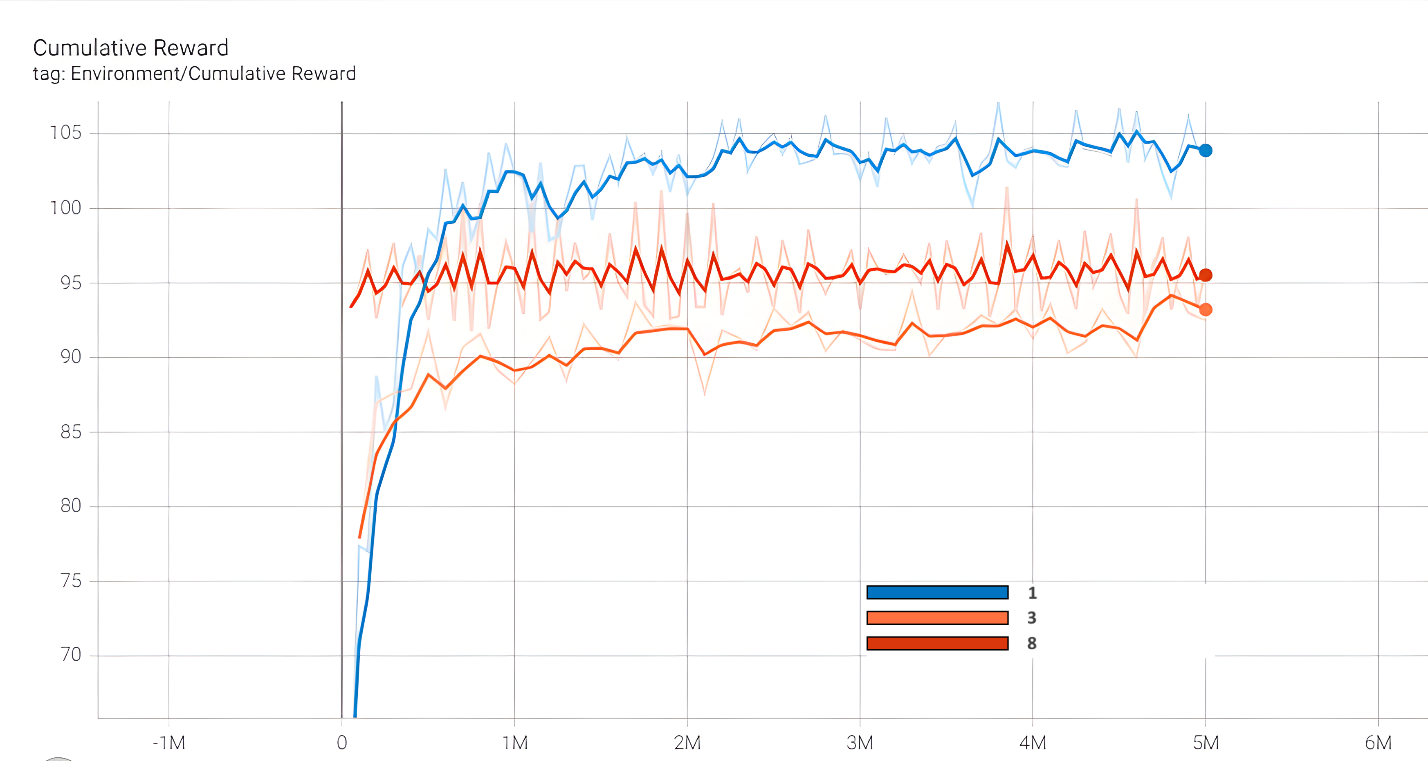
**Table 7.** Compare the results when changing the hyperparameter Lambd.



**Fig. 22.** Graphs with different Lambd values.

|  |  |  |
| --- | --- | --- |
| 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.

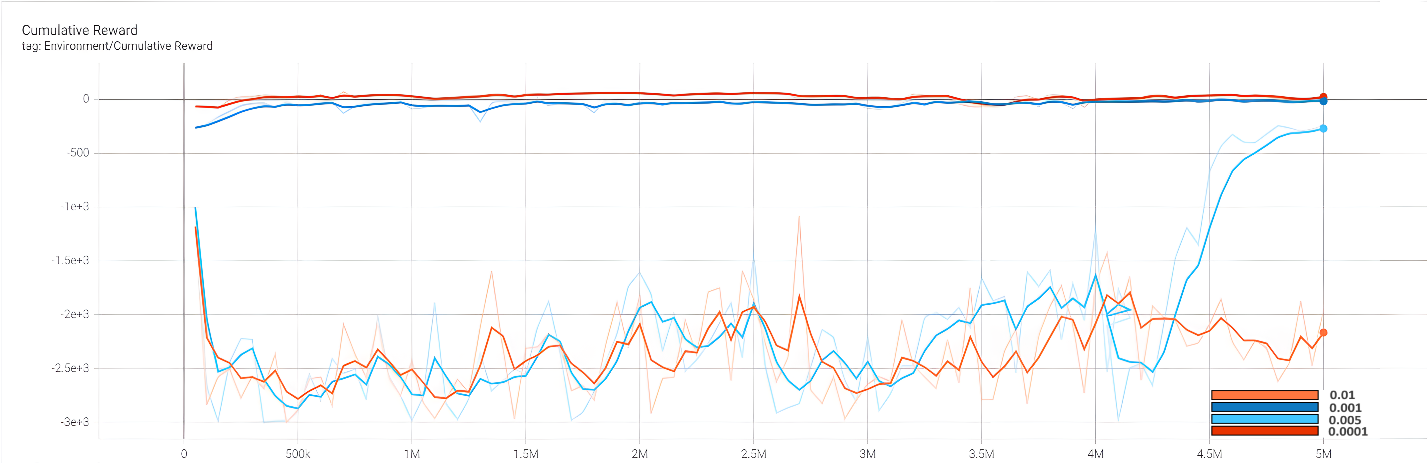


**Fig. 23.** Graphs with different Num\_epoch values.

## 4.3. 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 |

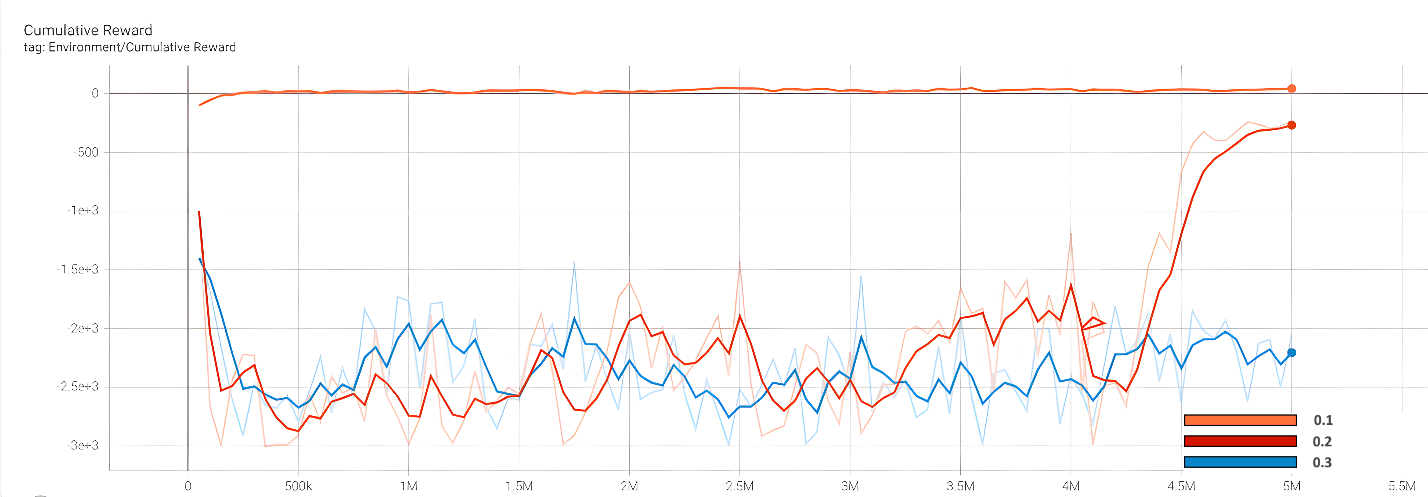
**Table 9.** Compare the results when changing the hyperparameter Beta.



**Fig. 24.** Graphs with different Beta values.

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

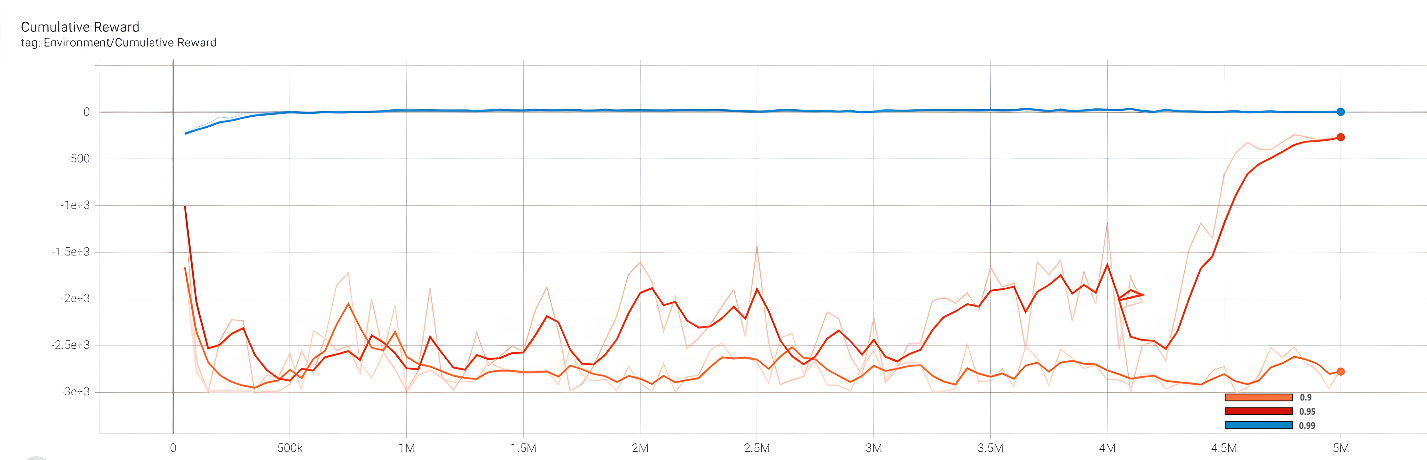
**Table 10.** Compare the results when changing the hyperparameter Epsilon.



**Fig. 25.** Graphs with different Epsilon values.

|  |  |  |
| --- | --- | --- |
| Lambd | Reward | Time Cost |
| 0.9 | -2740 | 1h 1m 45s |
| 0.95 | -227.4 | 1h 9m 22s |
| 0.99 | 4.037 | 1h 36m 0s |

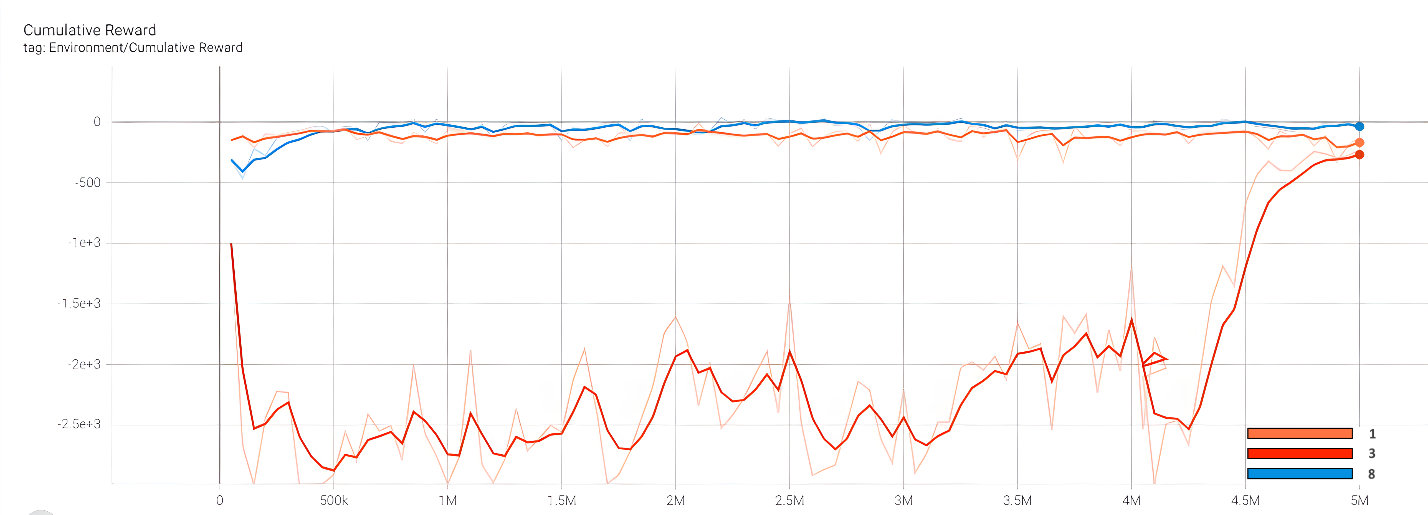
**Table 11.** Compare the results when changing the hyperparameter Lambd.



**Fig. 26.** Graphs with different Lambd values.

|  |  |  |
| --- | --- | --- |
| 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.



**Fig. 27.** Graphs with different Num\_epoch values.

## 4.4. 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 |

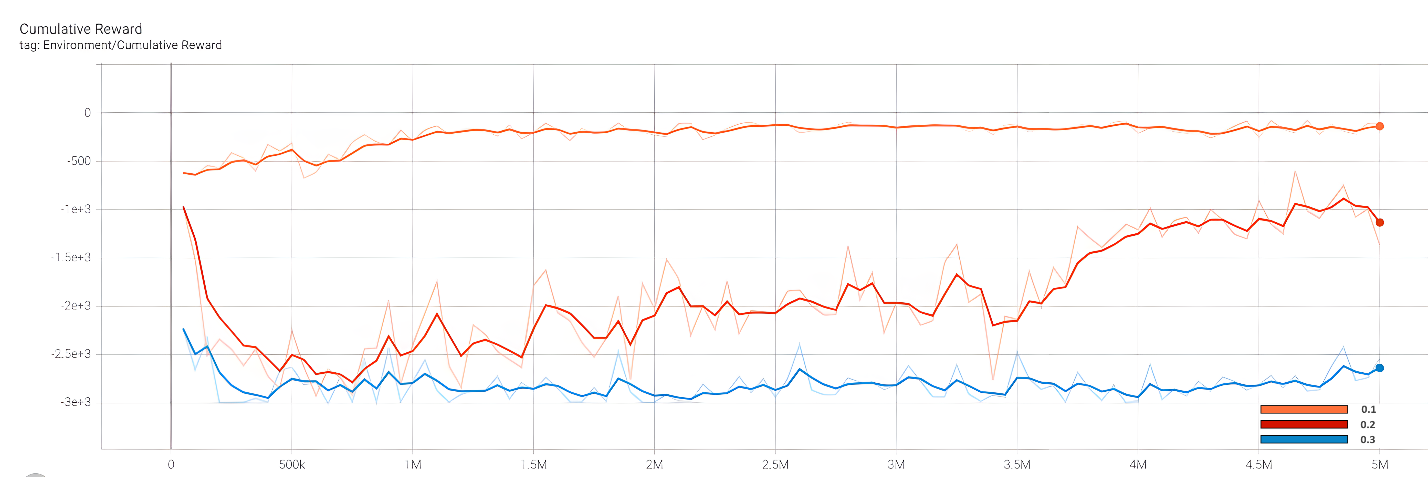
**Table 13.** Compare the results when changing the hyperparameter Beta.



**Fig. 28.** Graphs with different Beta values.

|  |  |  |
| --- | --- | --- |
| 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.



**Fig. 29.** Graphs with different Epsilon values.

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

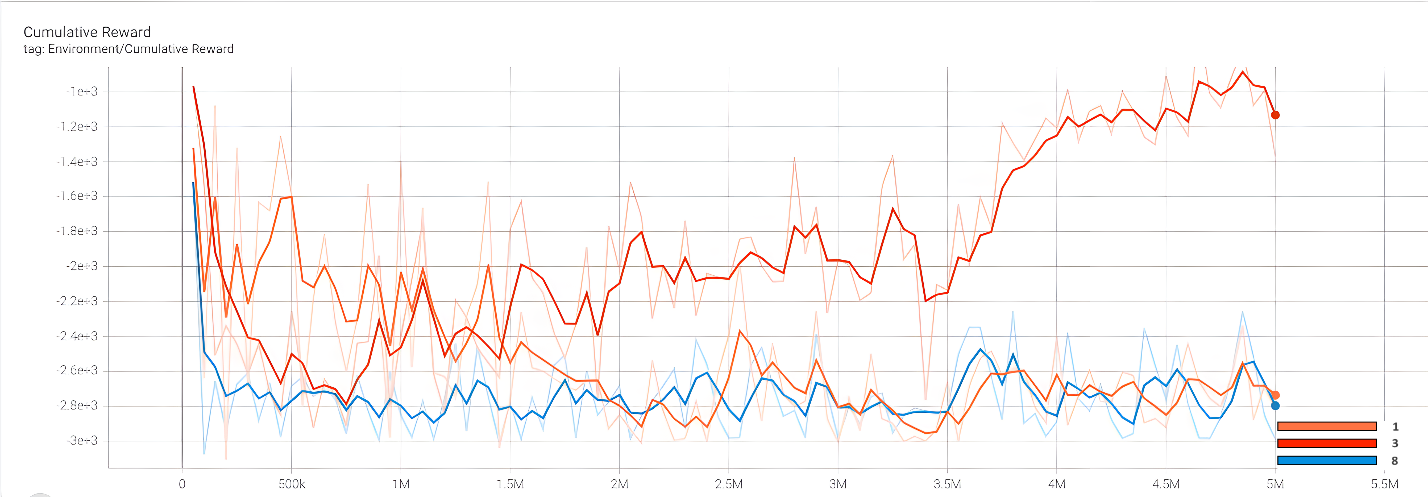
**Table 15.** Compare the results when changing the hyperparameter Lambd.



**Fig. 30.** Graphs with different Lambd values.

|  |  |  |
| --- | --- | --- |
| 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.



**Fig. 31.** Graphs with different Num\_epoch values.

## 4.5. 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 paper shows how to tune PPO algorithm through Beta, Epsilon, Lambd and Num\_epoch hyperparameters. These values are changed, and the RL learning results will be evaluated by the maze problem. From the results that in the above research process, we can see the clear difference and the hyperparameters affecting the algorithm and the training process. The change is also based on different cases according to the complexity of the maze. Therefore, it is necessary to choose the reasonable hyperparameters to set the best training results.

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] F. Yu *et al.*, “BDD100K: A Diverse Driving Dataset for Heterogeneous Multitask Learning,” *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, pp. 2633–2642, 2020, doi: 10.1109/CVPR42600.2020.00271.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

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

[17] 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.

[18] 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.