THE HONG KONG POLYTECHNIC UNIVERSITY
香港理工大學

COMP5511 Artificial Intelligence Concepts
Group Project: Maze Game

Student Name	Student ID
Chen Qi	18083945G
Gao Huahui	18071699G
Wang Di	18075629G
Su Gengmin	18003394G
Zhao Haiqi	18087348G

Content

1 Objective 3
2 Introduction 3
    2.1 Background 3
    2.2 Design..3
3 Game Play 4
    3.1 Human-Computer Competition 4
    3.2 Rule 4
    3.3 Scene..6
4 Algorithm and its Application 6
    4.1 Reinforcement Learning 6
        4.1.1 Introduction of Reinforcement Learning..6
        4.1.2 Elements of Reinforcement Learning System..7
        4.1.3 A Reinforcement Learning Model..7
    4.2 Deep Reinforcement Learning (DRL) 8
        4.2.1 Introduction of DRL..8
        4.2.2 Policy Gradient (PG) 9
        4.2.3 Proximal Policy Optimization (PPO)..11
    4.3 ML-Agents..12
    4.4 Suitability 14
5 Implementation 14
    5.1 Gameplay implementation 14
        5.1.1 Environment Setup 15
        5.1.2 Maze Design..16
        5.1.3 Game Logic..17
        5.1.4 User Interface Design..18
    5.2 ML-agent Implementation..19
        5.2.1 Application Structure 19
        5.2.2 Key Functions for Machine Learning 21
        5.2.3 Parameters..22
        5.2.4 Training Conclusion..26
6 Conclusion..27
Reference..28
USER MANUAL 29
Role and contributions of group members..31

1 Objective

This project is aiming at realizing a game AI with reinforcement learning concepts learned in class and comprising it in the game designed by us. Unity is used in this project as a platform to support our game while the basic ml-agents package was obtained from GitHub [1].

In this project, the reinforcement learning algorithm comprised in a typical game should be realized. The game design that suits AI algorithm is going to be studied and the deeper understanding about the AI will be gained by all of the group members.

2 Introduction

2.1 Background

Game industry mentioned AI a long time ago, but the previous AI mainly refers to something like automatic pathfinding of NPC in the game. However, the combination of games and AI today refers more to the application of various artificial intelligence technologies in the game industry such as deep neural networks. Actually, except for AlphaGo, more and more game projects are using machine learning.

For example, in 2017, a team from Carnegie Mellon University released a completely autonomous agent which is able to play Doom (a First-Person Shooter Games) trained only by pixel data. The team used deep reinforcement learning with a recent Action-Navigation architecture to train the agent, and this architecture uses two different deep neural networks for exploring the map and fighting enemies. Besides, it also applies lots of techniques such as augmenting high-level game features, reward shaping and sequential updates for efficient training and effective performance.

2.2 Design

We would like to design a game that suits comprising an AI player. A maze game is created in this project since there are only two results, namely, win and lose. Only two final results can be easily simplified to rewards in the reinforcement learning algorithm. The detail about the suitability of our game adapting AI will be illustrated in the later section.

A maze game is designed based on a prototype “JumpWall” game got from GitHub [1]. The final goal of player is that moving the cube, which represents the player character, to the escape point (destination). The player will have a first-person perspective, this will increase the difficulty of escaping from the maze.

Besides, in order to increase the player’s feeling of tension, we design a competition mode. The player will compete with an AI player. Once there is a party reaching the destination, the whole game ends. The one who reaches the destination win the whole game.

To give player more entertainment and make one turn of the game has a reasonable time period, we designed 15 rounds in total per turn. The player has to continuously finish 15 rounds (reach the destination 15 times) before they win the game.

Detailed information such as the rule or the scene design will be discussed in the later Game Play section.

3 Game Play

3.1 Human-Computer Competition

The Human-Computer Competition mode allows the user to play against our AI. The user will initiate the game at the same time as the AI player. They will have the same spawn point in each round. For example, the spawn point in the 8th round of user is the same as that of AI player. However, the spawn point in each round will be randomly picked from the default spawn points shown in Figure 2.

The user will compete for the speed with AI player. The one who finishes the 15 rounds firstly will be the winner. After one player won, the game will stop automatically and show the winner information on the screen.

3.2 Rule

The goal of this maze game is reaching the destination in the shortest time in order to get the highest mark. The player will become a blue cube who trapped inside the Maze. The only way to escape from this maze is that find the unknown escape point (destination) which exists inside the maze. The human player and AI player both have the first-person perspective like the picture shown in Figure 3.1.

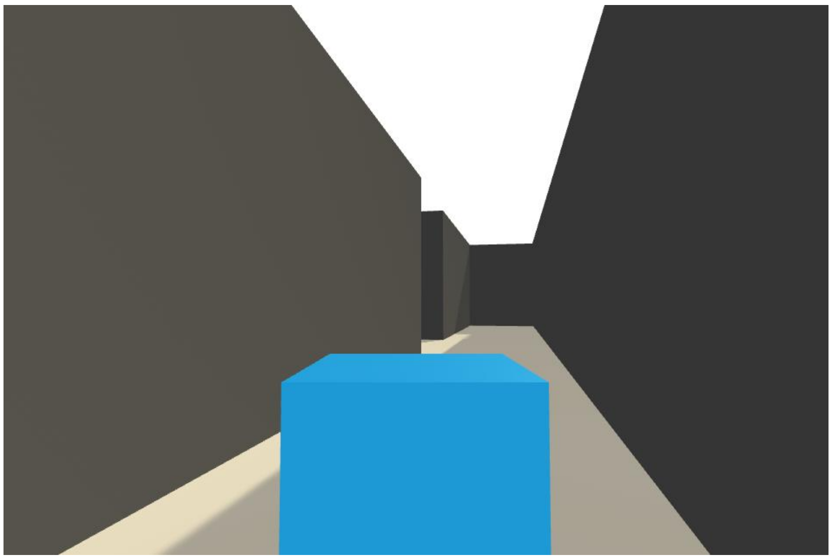


Figure 3.1 Player viewport

In total there are 15 rounds per turn. The player who successfully passes 15 rounds will be regarded as the winner. In the competition mode, the one who finishes 15 rounds firstly would be the final winner. The 15 rounds will start from different spawn points randomly picked from 15 default spawn points shown in Figure 3.2.

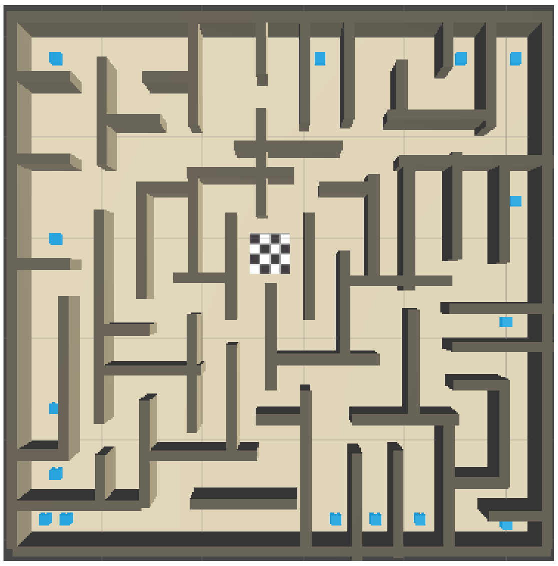


Figure 3.2 Fifteen Random spawn points

3.3 Scene

There are two identical mazes for the human player and AI player. The blue cube represents the player. The chequered with the black and white flag in the midpoint is the escape point (destination). The maze is constituted of two objects, namely, floor and wall. The floor is a brown solid flat object which supports every other object while the wall is a black squared solid object used to separate the floor into different paths. The blue cube is unable to pass through the floor and wall and look through them either. The perspective of the player can be seen from Figure 3.1.

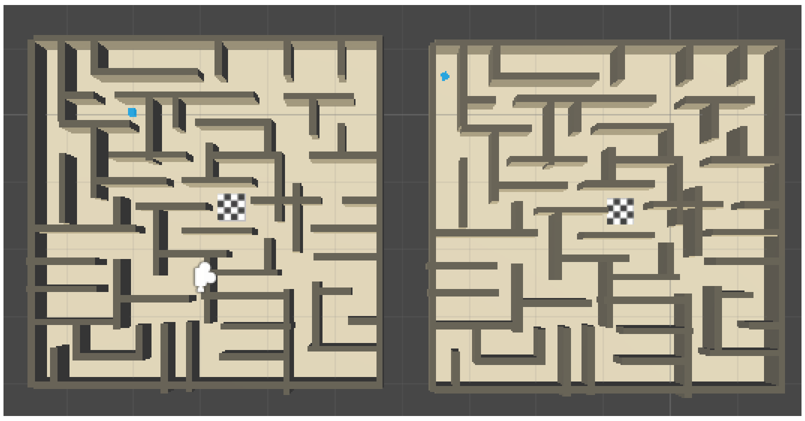


Figure 3.3 Play Filed Capture

4 Algorithm and its Application

4.1 Reinforcement Learning

4.1.1 Introduction of Reinforcement Learning

Reinforcement learning is a branch of machine learning, which can solve problems with an agent that perceives the information of the environment. Instead of teaching the agent which acts to take under a certain environment in advance, the agent must constantly try various actions to get the optimal policy. Specifically, the agent will receive rewards (positive or negative) as feedback from the environment after executing an action or more, and then adjust strategies according to those rewards. In the end, the reinforcement learning agent will find the optimal policy to solve a specific problem.

Reinforcement learning is different from supervised learning which learns from training data set based on input-output pairs (label) and unsupervised learning which learns from training data set that without a label. On the one hand, the information generated from direct interaction between agent and environment is more reliable when there is noise interference in the label information or the labels obtained by some active learning in supervised learning. On the other hand, instead of trying to find hidden structure like unsupervised learning, reinforcement learning is trying to maximize the rewards. Therefore, according to Richard S. Sutton, reinforcement learning is a third learning paradigm alongside supervised learning and unsupervised learning or maybe other learning paradigms.

4.1.2 Elements of Reinforcement Learning System

According to Richard S. Sutton and Andrew G. Barto, there are four more main elements in reinforcement learning except for agent and environment: a policy, a reward signal, a value function, and environment model. The policy is the mapping from states which agent perceive to actions which agent take, meaning that policy is the core of reinforcement learning agent because it determines how the agent behaves in a specific situation. A reward signal is a single number that sent to reinforcement learning agent from environment at each time step, whose objective is to maximize the total reward, which defines the reinforcement learning problem's target.

However, a reward signal only describes what is good for reinforcement learning agent in a short term, but a value function represents what's the best in a long term. For example, in Go, the player’s one step may get a lot of points, but this step does not necessarily have a positive impact on the final victory. Therefore, we can say that value function is predictions of rewards. Environment model is something that mimics a real environment’s behavior, in other words, environment model allows reasoning to be made on how the real environment will behave. With the assistance of the environment model, the agent is able to consider the future situation when planning the strategy.

4.1.3 A Reinforcement Learning Model

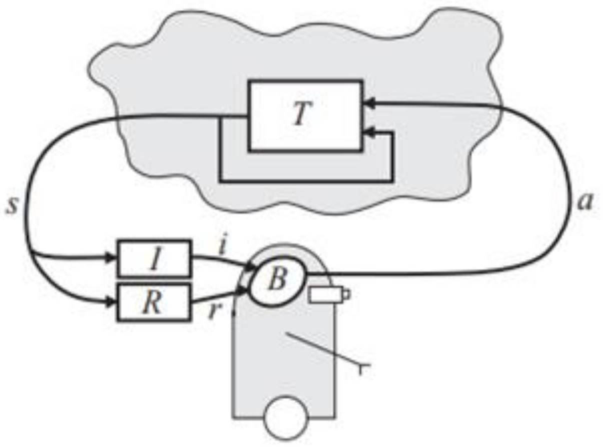


Figure 4.1 Reinforcement Learning Model

A reinforcement learning model, like Figure 4.1, consists of [2]:

(i) Agent: The lower part of Figure 4.1. The agent can perceive the environment and execute some actions.

(ii) Environment: The upper part of Figure 4.1. The environment is a world where the agent stays.

(iii) State: The specific part of the environment sensed by the agents, which is “s” part in Figure 4.1.

(iv) Behavior: The “B” in Figure 4.1. It determines what action the agent should do so that it can get the long-run sum of rewards.

(v) Action: The “a” in Figure 4.1, which is some operation executed by an agent in the environment.

(vi) Reward Function: The “R” in Figure 4.1, which determines when the agent can get the reward or punishment, and the quantity that it should get.

(vii) Reward: The reinforcement signal “r”, which is the feedback after executing the action.

(viii) Input Function: The “I” in Figure 4.1, which determines how the agent observes the environment state.

(ix) Inputs: The “i” in Figure 4.1, which is some indication of the current state that perceived.

4.2 Deep Reinforcement Learning (DRL)

4.2.1 Introduction of DRL

Deep reinforcement learning is one of the most concerned directions in the field of artificial intelligence in recent years. It combines the perceptual ability of deep learning with the decision-making ability of reinforcement learning and directly controls the behavior of agents through high-dimensional perceptual input learning. General speaking, it's applying 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), and so on. In our group project, we use the PPO as our major AI algorithm, so we combed the PPO principle in the next section.

Table 4.1 Main Algorithms of Deep Reinforcement Learning

Algorithms	**Year**	**Based on**
Deep Q Network (DQN)	2013	Value
Nature DQN	2015	Value
Deep Deterministic Policy Gradient (DDPG)	2015	Policy
Proximal Policy Optimization (TRPO)	2015	Policy
Asynchronous Advantage Actor-Critic (A3C)	2016	Policy
Unsupervised Reinforcement and Auxiliary Learning (UNREAL)	2016	Policy
Distributed Proximal Policy Optimization (DPPO)	2017	Policy

4.2.2 Policy Gradient (PG)

Policy Gradient (PG) are frequently used algorithms in reinforcement learning. In PG, the agents observe the state of the environment, then takes actions based on its policy on the state. After the actions, the agent will enter to a new state of the environment. 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 important expressions of PG [3]:
In reinforcement learning, the policy π 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:

$$(s_1, u_1, s_2, u_2, \ldots, s_H, u_H)$$

The sum of the probability of a trajectory τ and it's corresponding rewards is the expected rewards:

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

R(τ) means the rewards of the trajectory.

And the PG use this policy to update the θ:

$$\boxed{\begin{array}{l} \nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t})\right)\left(\sum_{t=1}^{T} r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t})\right) \\ \\ \theta \leftarrow \theta + \alpha \nabla_\theta J(\theta) \end{array}}$$

The advantage functions A and rewrite the policy gradient:

$$A^\pi(\mathbf{s}_t, \mathbf{a}_t) = Q^\pi(\mathbf{s}_t, \mathbf{a}_t) - V^\pi(\mathbf{s}_t)$$

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

π_θ 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.

4.2.3 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) is an optimization algorithm, which can improve the data efficiency and reliable performance of Trust Region Policy Optimization (TRPO) while only using the first-order optimization. In fact, PPO is an optimized version based on Policy Gradient and TPRO. For Policy Gradient Method, even if it uses the same way to perform multiple steps of optimization, it may often lead to destructively large policy updates. For TPRO, it uses a hard constraint instead of a penalty since it is difficult to choose a fixed penalty coefficient. In TPRO, the KL penalty coefficient needs to be adjusted to improve the performance of the algorithm.

The main objective function of PPO is:

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

In this function, θ is the policy parameter, E_t is the expectation over timestamps, A_t is the advantage function at time t, ε is a hyperparameter, which is usually 0.1 to 0.3, it is the ratio of the probability under the new and old policies.

In the $L^{CLIP}(\theta)$ in PPO, if the agent has too large of a policy update, it will be punished, which is different from $L^{CPI}(\theta)$ (The objective function in TPRO). The clip() function can prevent the incentive factors from moving r_t outside the interval [1-ε, 1+ε]. Figure 4.2 shows the comparison of objective functions between PPO and TRPO [4]:

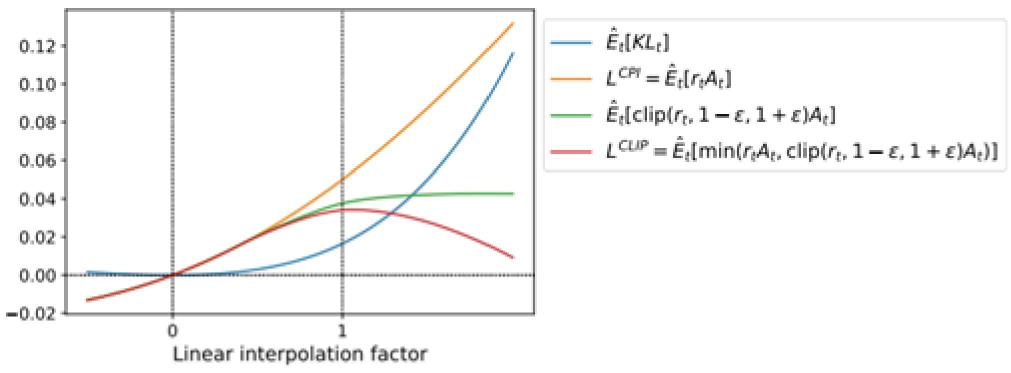


Figure 4.2 Comparison of objective functions between TPRO and PPO

The steps of the PPO algorithm are [4]:

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
```

Figure 4.3 PPO Algorithm

According to the Figure 4.3, in each iteration of the PPO algorithm, each parallel (or maybe not parallel) actors collect the T timesteps from the environment, then compute the advantages of each T correspondence. At the end of one iteration, PPO will optimize surrogate objective function for K epochs.

4.3 ML-Agents

ML-Agents Unity Machine Learning Agents Toolkit [5], is an open-source Unity plugin which provides lots of Python API with the implementation of game AI. ML-Agents uses a socket to achieve communication between 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 is saved in its own format, which is the bytes file generated after the training. In the Inference phase, ML-Agents uses TensorFlow Sharp to read the trained model and use it as a Brain in Unity Environment to specifically guide the Agent to interact with the environment.

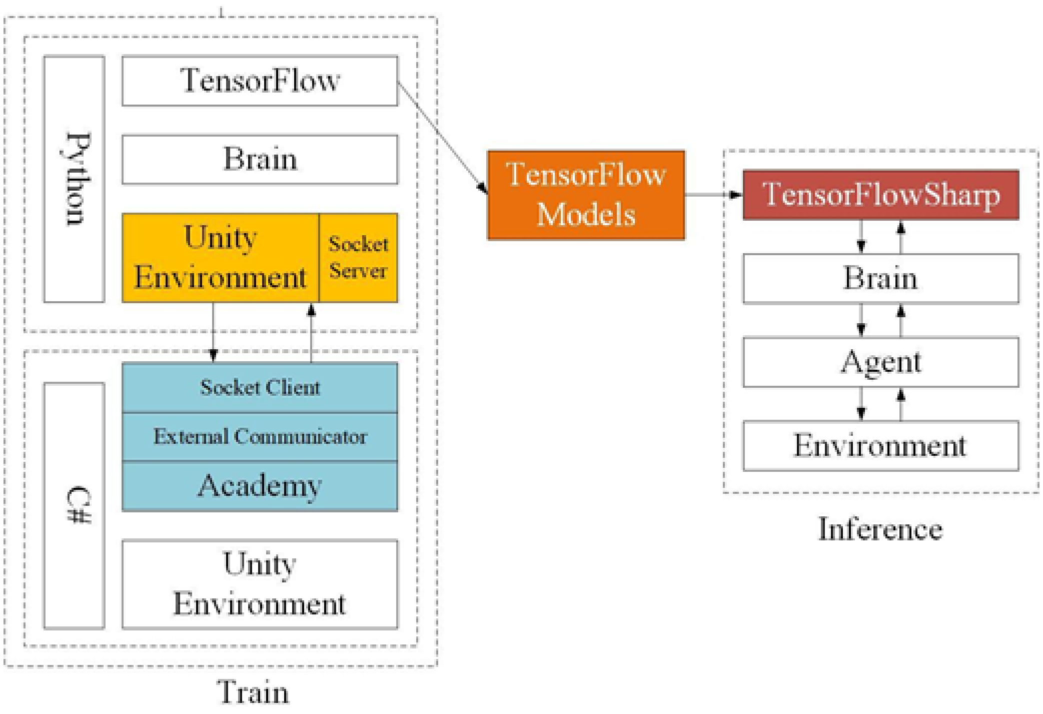


Figure 4.4 Unity ML-Agents Structure

There are three key components of Unity ML-Agents: Learning Environment, Python API and External Communicator. Among them, the Learning Environment includes three additional components that help organize Unity scenarios. Figure 4.5 shows an example of the relationship between each component in Unity ML-Agents [5].

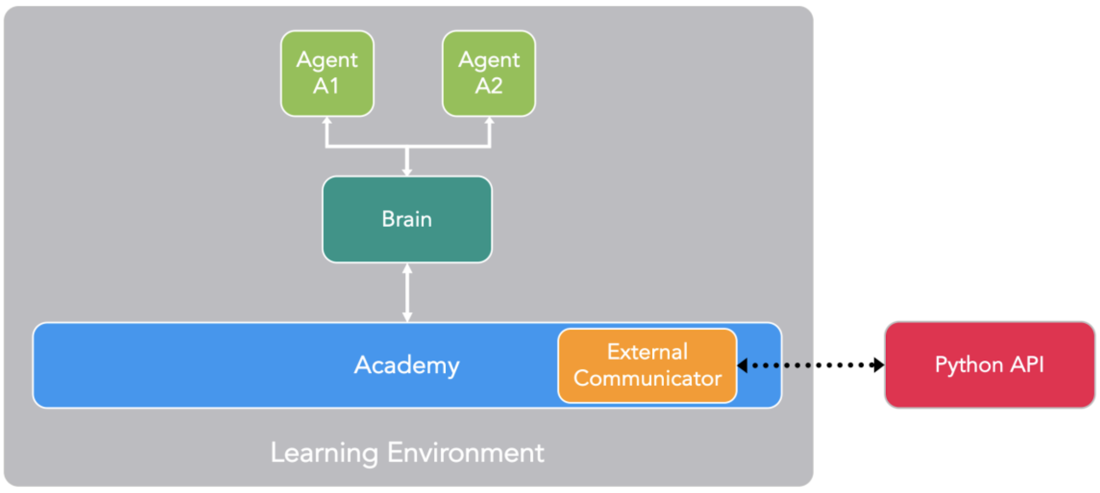


Figure 4.5 An Example of Learning Environment in Unity ML-Agents)

- Learning Environment
  - Agents: The NPC in Unity games, which can observe the environment and execute the actions.
    - Observations: The partly environment or state which is

        viewed or sensed by the agents. Observations can be some figures or pictures.
        - Actions: In Unity games, the action can be movements or some specific steps which are executed by agents.
        - Reward Signals: It contains a positive and negative value, which provides feedback on every successful or failed action for agents.
    - Brains: Provide the decision logic for the agents.
        - External: Make decisions by Python API (ML-Agents Tools).
        - Internal: Make decisions by embedded Tensorflow Models, usually using the models after training.
        - Player: Make decisions by real inputs (keyboard).
        - Heuristic: Make decisions by hard-coded behaviors, which are written in C# scripts.
    - Academy: Academy can coordinate the observations and decisions.
- Python API: Encapsulated Unity machine learning algorithms.
- External Communicator: Provide the message communication between Python API and the Learning Environment.

4.4 Suitability

The algorithm is suitable for our project. There are some reasons for it. First, our game is a maze game, which is a simple game with a clear game logic. Besides, we can define the state of the game, the action of the game, the reward of the game clearly. It matches the requirement of the usage of ML-Agent. The complexity of our game is low that the AI can learn our games faster than others.

For example, the rule of our game is reaching the destination in the maze meaning that we can give a positive reward when the player reaches the destination. And give a negative reward (punishment) base on the time the player spent. The longer time they spend, the more punishment they get. Therefore, reinforcement learning is suitable for our project.

5 Implementation

5.1 Gameplay implementation

5.1.1 Environment Setup

Step 1: Install Python via Anaconda

Download and install Anaconda for Windows. By using Anaconda, we can manage separate environments for different distributions of Python. Python 3.5 or 3.6 is required

Step 2: Setup and Activate a New Conda Environment

Creating a new Conda environment to be used with the ML-Agents toolkit. This means that all the packages that we install are localized to just this environment. It will not affect any other installation of Python or other environments. Whenever we want to run ML-Agents, we will activate this Conda environment.

To create a new Conda environment, open a new Anaconda Prompt (Anaconda Prompt in the search bar) and type in the following command:

```
conda create -n ml-agents python=3.6
```

To use this environment, we must activate it. (To use this environment In the future, we can run the same command). In the same Anaconda Prompt, type in the following command:

```
activate ml-agents
```

Next, install Tensorflow. Install this package using pip - which is a package management system used to install Python packages. Latest versions of TensorFlow won't work, so we install version 1.7.1. In the same Anaconda Prompt, type in the following command:

```
pip install tensorflow==1.7.1
```

Step 3: Install Required Python Packages

Cloning the ML-Agents Toolkit Github repository to our local computer. We do this by using Git (download here) and running the following commands in an Anaconda Prompt:

```
git clone https://github.com/Unity-Technologies/ml-agents.git
```

change to the python directory inside the ml-agents directory:

```
cd C:\Downloads\ml-agents\ml-agents
```

Installing the packet in the path

```
pip install .
```

5.1.2 Maze Design

Inside the game “Maze”, we need a goal field, when a player or AI arrives the goal field, player or AI will move to the next round and born in the next spawn point. The goal field is located in the center of “Maze”. Please refer to Figure 5.1 about the goal field.

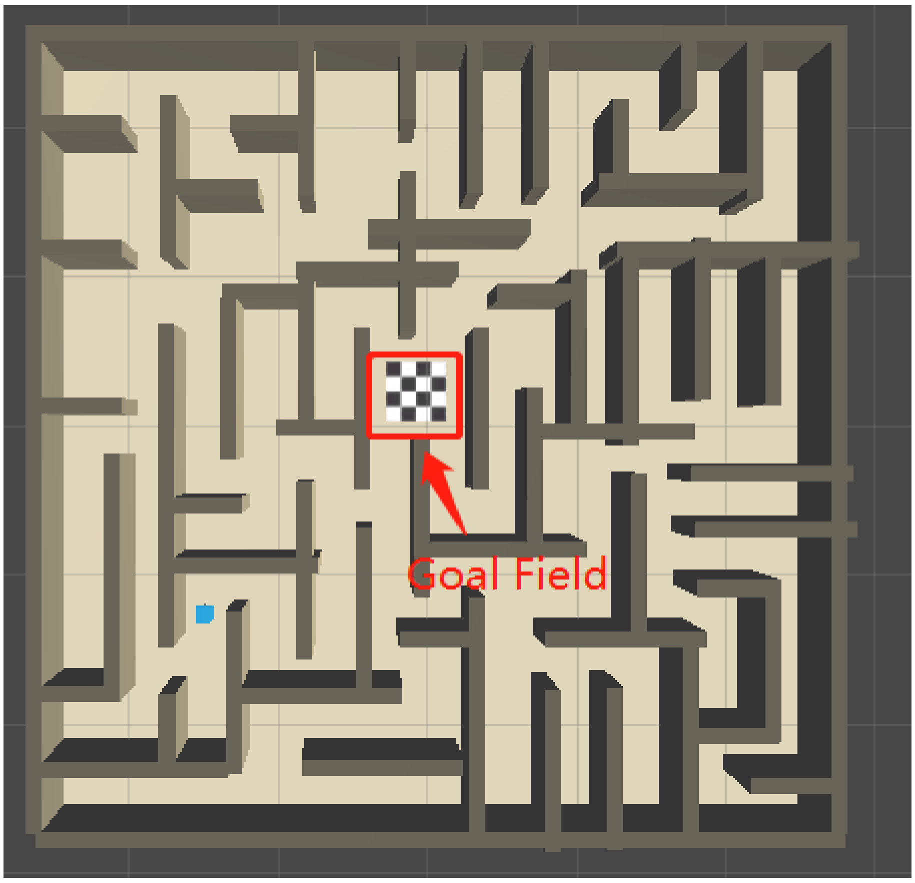
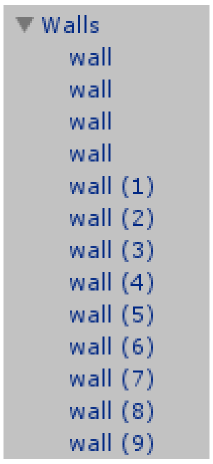


Figure 5.1 Goal Field

Of course, inside the “Maze”, there are walls to hinder player or AI. We use lots of wall object to create the “Maze”. Please refer to Figure-5.2 for more details.



▼ Walls
wall
wall
wall
wall
wall (1)
wall (2)
wall (3)
wall (4)
wall (5)
wall (6)
wall (7)
wall (8)
wall (9)

Figure 5.2 Walls

By using these walls, the "Maze" is created. For the layout of "Maze", please refer to Figure 5.3

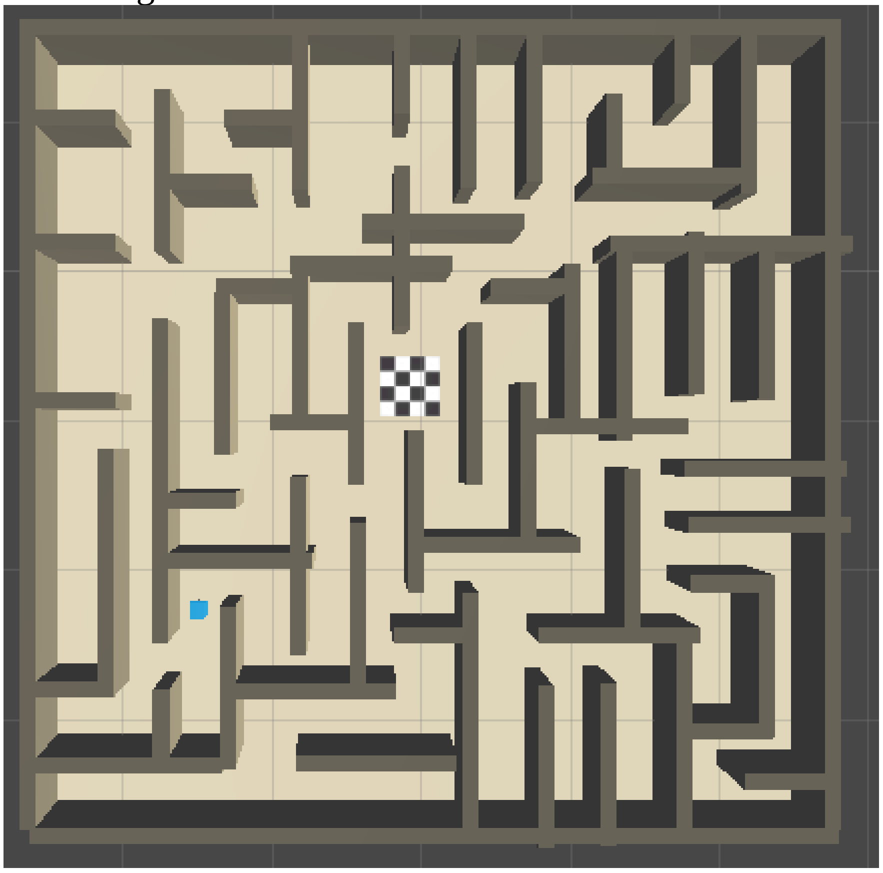


Figure 5.3 Maze Layout

5.1.3 Game Logic

The game has 15 fixed spawn points, which are located in the border of "Maze". There spawn points are stored in an array. When the game starts(Round-1), both player and AI will be located in the spawn point-1 (point number is regarding the Round number). Both AI and player will try to reach goal field. Once they arrive the goal field, the cube will be relocated to next spawn point. By repeating the procedures, if AI or player arrived goal field in Round-15, game over. For more details, please refer to the flowchart -- Figure 5.4.

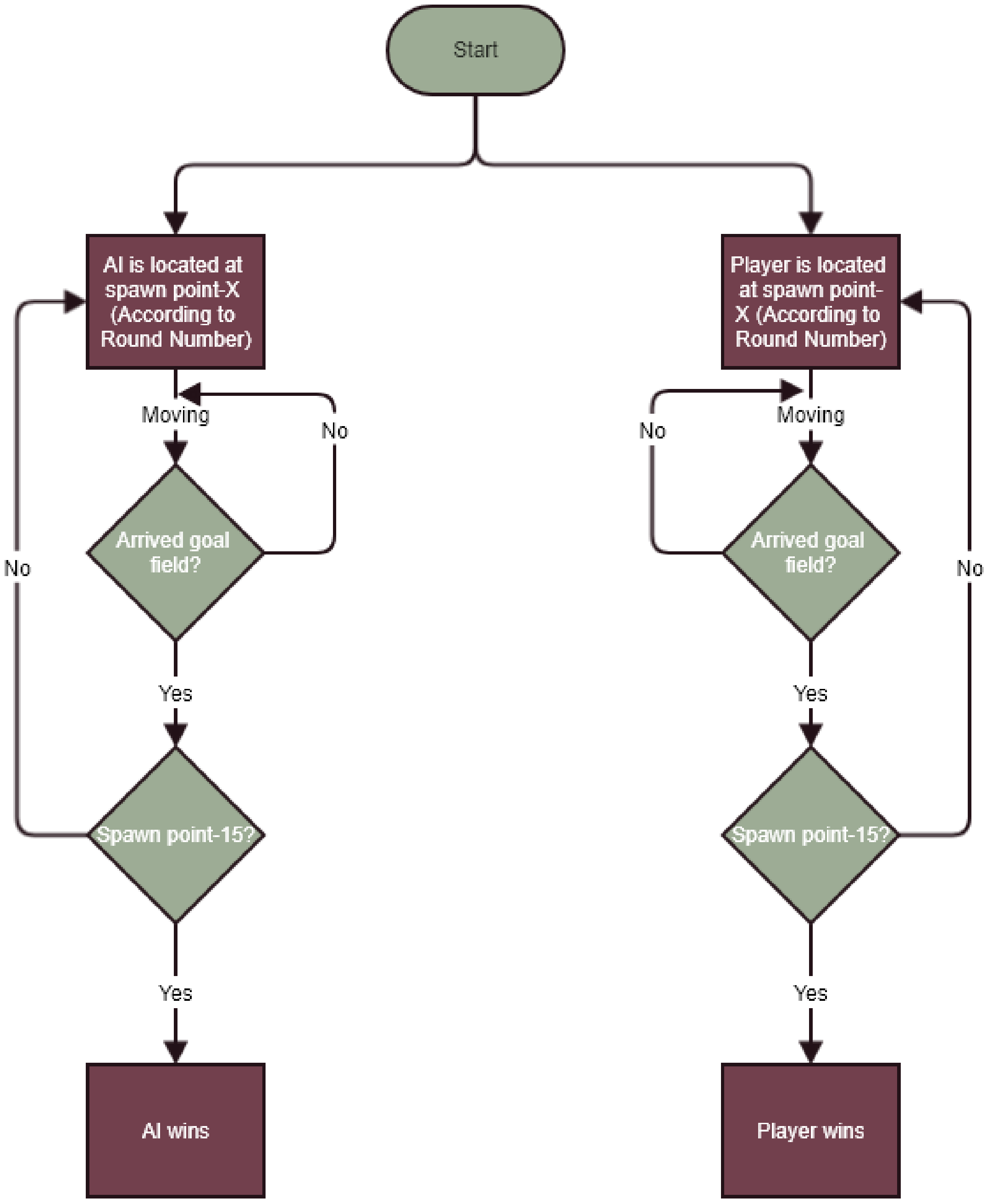

Start
AI is located at spawn point-X (According to Round Number)
Player is located at spawn point-X (According to Round Number)
Moving
No
Arrived goal field?
Yes
Spawn point-15?
No
Yes
AI wins
Player wins

Figure 5.4 Game Flow

5.1.4 User Interface Design

The user interface contains below parts:

- AI Round Text
- Player Round Text: Indicating the round number of player's.
- Restart Button: Restarting game from round-1.
- Control Hints Text: Giving some control hints.
- Game Over Text: When a player wins, it will show "Player Win!!". When AI wins, it will show "AI Win!!".
- Game View: The Game view is divided into 2 parts. The left part is AI control view. And right part is player control view.

For more details, please refer to Figure 5.5.

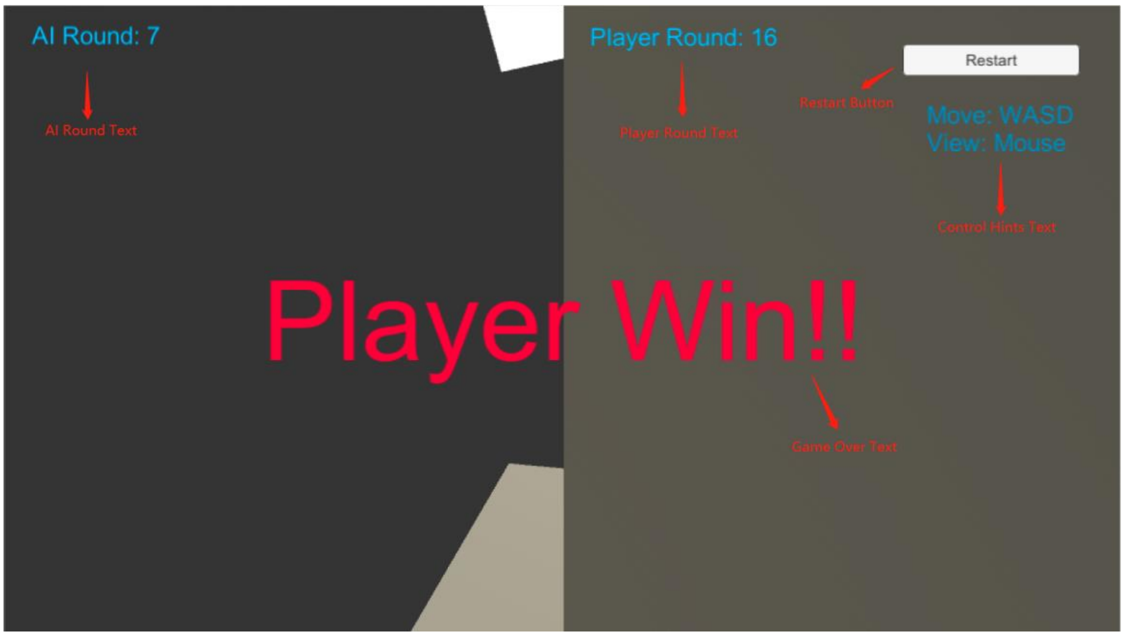


Figure 5.5 User Interface

5.2 ML-agent Implementation

5.2.1 Application Structure

Inside the game, there are two parts, "Player Area" and "AI Area". Please refer to Figure 5.6 and Figure 5.7 for details.

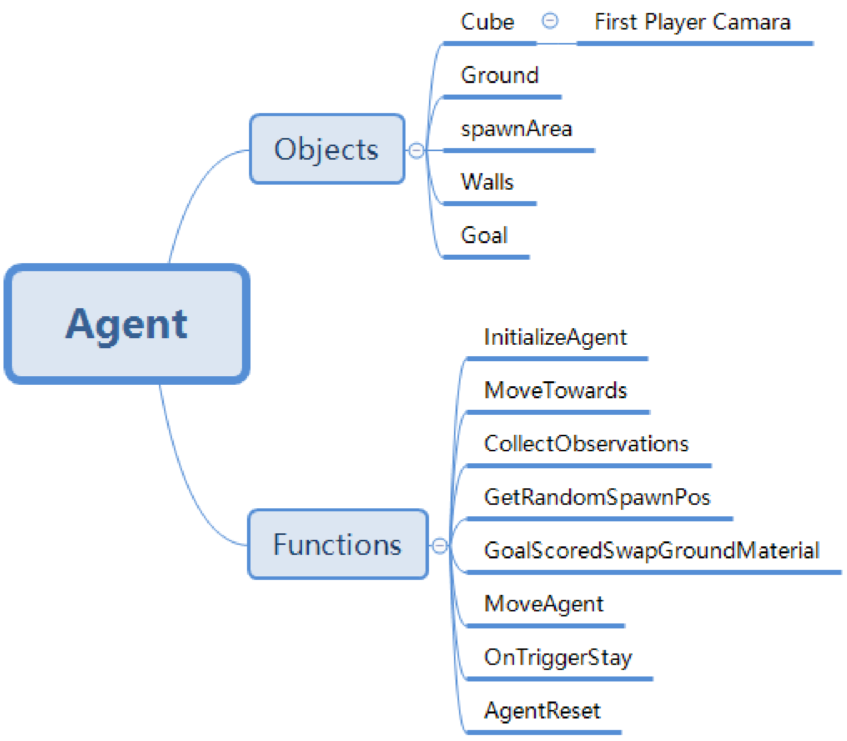


Figure 5.6 AI Area Structure

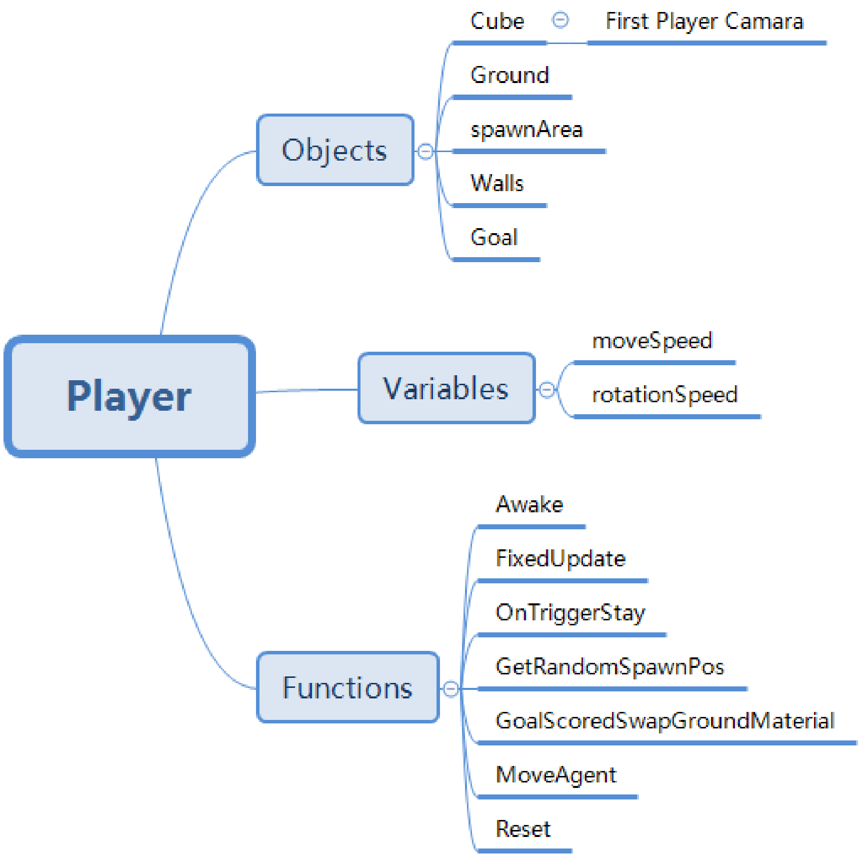


Figure 5.7 Player Area Structure

5.2.2 Key Functions for Machine Learning

```
OnTriggerStay:
 void OnTriggerStay(Collider col)
    {
       if (col.gameObject.CompareTag("goal") && DoGroundCheck(true))
       {
          SetReward(1f);
          Done();
          StartCoroutine(
             GoalScoredSwapGroundMaterial(academy.goalScoredMaterial, 2));
       }
    }
```

When agent(cube) collides with other objects, this function will be called. This function is used to check whether the agent(cube).If agent arrived goal filed, this function will call the ML-Agents API SetReward(float REWARD). We can adjust the goal reward for better training result.

```
MoveAgent:

public void MoveAgent(float[] act)
    {
       AddReward(-0.005f);
       bool smallGrounded = DoGroundCheck(true);
       bool largeGrounded = DoGroundCheck(false);

       Vector3 dirToGo = Vector3.zero;
       Vector3 rotateDir = Vector3.zero;
       int dirToGoForwardAction = (int) act[0];
       int rotateDirAction = (int) act[1];
       int dirToGoSideAction = (int) act[2];
       int jumpAction = (int) act[3];

       if (dirToGoForwardAction==1)
          dirToGo = transform.forward * 1f * (largeGrounded ? 1f : 0.5f);
       else if (dirToGoForwardAction==2)
          dirToGo = transform.forward * -1f * (largeGrounded ? 1f : 0.5f);
       if (rotateDirAction==1)
          rotateDir = transform.up * -1f;
       else if (rotateDirAction==2)
          rotateDir = transform.up * 1f;
       if (dirToGoSideAction==1)
          dirToGo = transform.right * -0.6f * (largeGrounded ? 1f : 0.5f);
       else if (dirToGoSideAction==2)
          dirToGo = transform.right * 0.6f * (largeGrounded ? 1f : 0.5f);
       if (jumpAction == 1)
          if ((jumpingTime <= 0f) && smallGrounded)
          {
            Jump();
        }

    transform.Rotate(rotateDir, Time.fixedDeltaTime * 300f);
    agentRB.AddForce(dirToGo * academy.agentRunSpeed,
                ForceMode.VelocityChange);

    if (jumpingTime > 0f)
    {
        jumpTargetPos =
        new Vector3(agentRB.position.x,
                jumpStartingPos.y + academy.agentJumpHeight,
                agentRB.position.z) + dirToGo;
        MoveTowards(jumpTargetPos, agentRB, academy.agentJumpVelocity,
                academy.agentJumpVelocityMaxChange);

    }

    if (!(jumpingTime > 0f) && !largeGrounded)
    {
        agentRB.AddForce(
        Vector3.down * fallingForce, ForceMode.Acceleration);
    }
    jumpingTime -= Time.fixedDeltaTime;
}
```

This function is used to handle the decisions from "brain". Agent will move according to different decisions from "brain".
Besides, each move action of the agent will deduct for a certain amount of reward by using ML-Agents API **AddReward(float REWARD).** For each moving step, we will punish the agent, therefore, the agent will try to arrive goal field with fewer steps. By adjusting the punishment of each step, we can train a smarter brain which can use fewer time to achieve the goal.

5.2.3 Parameters

Unity ML-Agents use Proximal Policy Optimization (PPO) or Behavior Cloning (BC) as the major reinforcement learning algorithm. In our project, we choose PPO algorithm because we don't have. There are some parameters in PPO which called "Hyperparameters". In addition, there are some optional parameters in Unity ML-Agents, which can provide several additional functions for training.

Here are the parameters information [6] and our setting strategies:

- Gamma: The discount factor for future reward.
  - Meaning of high value: The agent will focus more on the future.
  - Meaning of low value: The reward will be more direct to agent's actions.
  - Typical Range: 0.8 - 0.995
  - Our choice: 0.99
  - Reason: In our maze game, the agent must take many steps (actions) to win and get rewards, which means that agencies needed to care more about possible reward in the future. Therefore, the value of Gamma should be large.

- Lambda: Lambda is the parameter when using in calculating GAE (Generalized Advantage Estimate for use in updating policy).
  - Meaning of high value: When updating policy, the agent will depend more on the observation of the environment.
  - Meaning of low value: When updating policy, the agent will depend more on the current estimate.
  - Typical Range: 0.9 - 0.95
  - Our choice: 0.95
  - Reason: Compared to current estimates, our agent should rely more on actual rewards received from the environment. Therefore, the value of Lambda should be large.

- Buffer Size: The experience the agent takes before updating the model.
  - Typical Range: 2048 - 409600
  - Our choice: 10240
  - Reason: In general, buffer_size is a multiple of batch size and a larger buffer_size corresponds to more stable training update. Due to a lack of understanding, we follow the default setting of Unity’s example.

- Batch Size: For one iteration of a gradient descent update, the size of experience used.
  - Meaning of high value: Continuous action space.
  - Meaning of low value: Discrete action space.
  - Typical Range (Continuous): 512 - 5120
  - Typical Range (Discrete): 32 - 512
  - Our choice: 1024
  - Reason: We are using a discrete action space (turn left, turn right, move forward, move ward), so this value should be smaller.

- Number of Epochs: Number of times the gradient descended buffer.
    - Typical Range: 3 - 10
    - Our choice: 3
    - Reason: Lowering the number of epochs ensures a more stable update at the cost of taking more time to train.

- Learning Rate: The strength of each gradient descent update.
    - Typical Range: 1e-5 - 1e-3
    - Our choice: 3.0e-4
    - Reason: Due to a lack of understanding of code logic, we follow the default setting of Unity's example.

- Time Horizon: How many experiences are collected before each agent adds to the experience buffer.
    - Meaning of high value: If the trainer needs to capture enough actions in the buffer.
    - Meaning of low value: If the rewards are frequent, or the episodes are very large.
    - Typical Range: 32 - 2048
    - Our choice: 64
    - Reason: In the initial training process, we found that the frequency of getting a reward is not high. Therefore, we believe that increasing the value of this parameter can speed up the efficiency of learning.

- Max Steps: The maximum value of the agent action during the training.
    - Typical Range: 5e5 - 1e7
    - Our choice: 1e7
    - Reason: Through testing, we found that the agent needs to take a huge amount of actions to complete the training better. Therefore, we use the maximum value of the typical range.

- Beta: Intensity of entropy regularization.
    - Meaning of high value: The policy will be more randomly.
    - Meaning of low value: The policy will not be more randomly.
    - Typical Range: 1e-4 - 1e-2
    - Our choice: 5.0e-3
    - Reason: Due to a lack of understanding of code logic, we follow the default setting of Unity's example.

- Epsilon: The acceptable threshold of divergence between the old

and new policies during gradient descent.
    - Meaning of high value: The training procedure becomes faster, but it will cause an unstable update.
    - Meaning of low value: More stable update but slow down the procedure of training.
    - Typical Range: 0.1 - 0.3
    - Our choice: 0.2
    - Reason: In order to get a relatively stable update, the value of this attribute will be relatively small.

- Normalize:
    - Meaning of true: In some complex continuous control problems, it may be useful.
    - Meaning of false: In discrete control problems, do not use normalization.
    - Our choice: false
    - Reason: As mentioned, we are using a discrete action space, so the value of this parameter should be false.

Intrinsic Curiosity Module Hyperparameters:
In many scenarios, external rewards are limited because some goals are difficult to achieve for agents [7]. In reinforcement learning, intrinsic motivation is extremely important because the agents need to be driven to execute more actions aimlessly. In our maze game, there are lots of obstruction so It's hard for the agent to move to the target place. In this case, we need to use curiosity module to add the intrinsic rewards to the agents.

- Curiosity Encoding Size: The size of the hidden layer used to encode observations within the intrinsic curiosity module.
    - Typical Range: 64 - 256
    - Our choice: 128
    - Reason: In the initial training process, we found that the agent is often trapped inside the dead end of the starting point. In that case, we choose the smaller value of this property to encourage agents to explore places that they have never been to.

- Curiosity Strength: The volume of the intrinsic reward generated by the intrinsic curiosity module.
    - Typical Range: 0.1 - 0.001
    - Our choice: 0.01

- Reason: Due to a lack of understanding of code logic, we follow the default setting of Unity's example.

5.2.4 Training Conclusion

For Training record, please take a reference to attachment file --"training-recod.doc".

In the experiments, we found that the parameter "Curiosity" [8] working well in our "Maze" game. We use two environments with same parameters setting apart from "Curiosity". After 100000 steps training, we found that the reward of training is better when we open "Curiosity". Please check Figure 5.8 for the testing result.

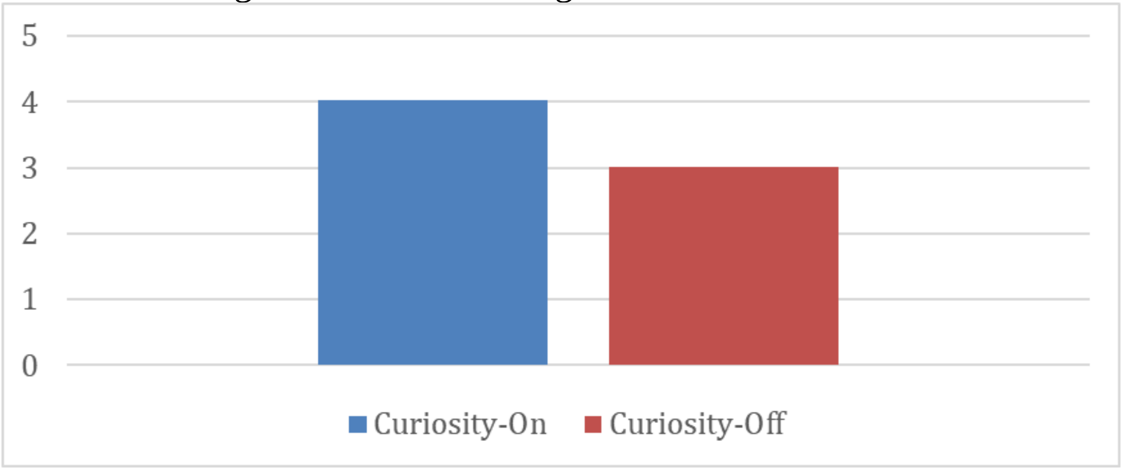


Figure 5.8 Curiosity On/ Off Testing Result

According to above research result, we decide to open the "Curiosity" for getting a better result of training.

In the next stage, we try to modify the punishment of each step. We choose to reduce 0.0005 and 0.005 respectively in two experiment scenarios. After training for hours, we compare different training result. Please refer to the Table 5.1 for result of the experiments which record the time cost of AI to arrive goal field.

Table 5.1 Time Cost of AI to Arrive Goal Field

	TEST-1	TEST-2	TEST-3	TEST-4	TEST-5	MEAN
-0.005 72H	206.18s	345.76s	211.38s	268.16s	216.77s	249.65s
-0.0005 72H	158.99s	142.79s	132.00s	150.79s	118.99	140.712s
-0.005 84H	191.58s	155.19s	149.96s	152.01s	258.35s	181.418s
-0.0005 84H	160.39s	272.46s	180.93s	225.37s	246.97s	217.224s

-0.0005 108H	142.66s	202.58s	217.97s	195.98s	236.77	199.192

The final model(brain) based on the above experiment. We choose the smartest one which is set “Step Reward = -0.005”, and “Training Hours = 72”.

In conclusion, open the “Curiosity” setting can help to improve the result of training. And setting bigger “Step Punishment” cannot help the result of training.

6 Conclusion

Our project has realized a maze game with an AI player and human player competition. This project has shown us the strength and weakness of the game AI. The reinforcement learning algorithm is adapted in this project. The details of the parameter setting in the ml-agents, which is the package used to train this AI, has been discussed in the previous section. Although we have experienced a sort of problems in this project, the completeness is guaranteed. Some improvement might also be made after this project to perfect our Game AI. In addition, we learned more about reinforcement learning and deep reinforcement learning by reading some literature and official documents.

Reference

[1]. Github, “Unity ML-Agents Toolkit”, https://github.com/Unity-Technologies/ml-agents

[2]. Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of artificial intelligence research*, *4*, 237-285.

[3]. Medium, “RL—Policy Gradient”, https://medium.com/@jonathan_hui/rl-policy-gradients-explained-9b13b688b146

[4]. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

[5]. Github, “ML-Agents Toolkit Overview”, https://github.com/Unity-Technologies/ml-agents/blob/master/docs/ML-Agents-Overview.md

[6]. Github, “Training with Proximal Policy Optimization”, https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-PPO.md

[7]. Pathak, D., Agrawal, P., Efros, A. A., & Darrell, T. (2017, May). Curiosity-driven exploration by self-supervised prediction. In *International Conference on Machine Learning (ICML)* (Vol. 2017).

[8]. Arthur Juliani, “Solving sparse-reward tasks with Curiosity”, https://blogs.unity3d.com/cn/2018/06/26/solving-sparse-reward-tasks-with-curiosity/

[9]. Richard S. Sutton & Andrew G. Barto 2018. Reinforcement Learning: An introduction. Cambridge, Massachusetts

[10]. Zhao xingyu & Ding shifei. (2018). Research on Deep Reinforcement Learning. Computer Science, 45(7), 1-6

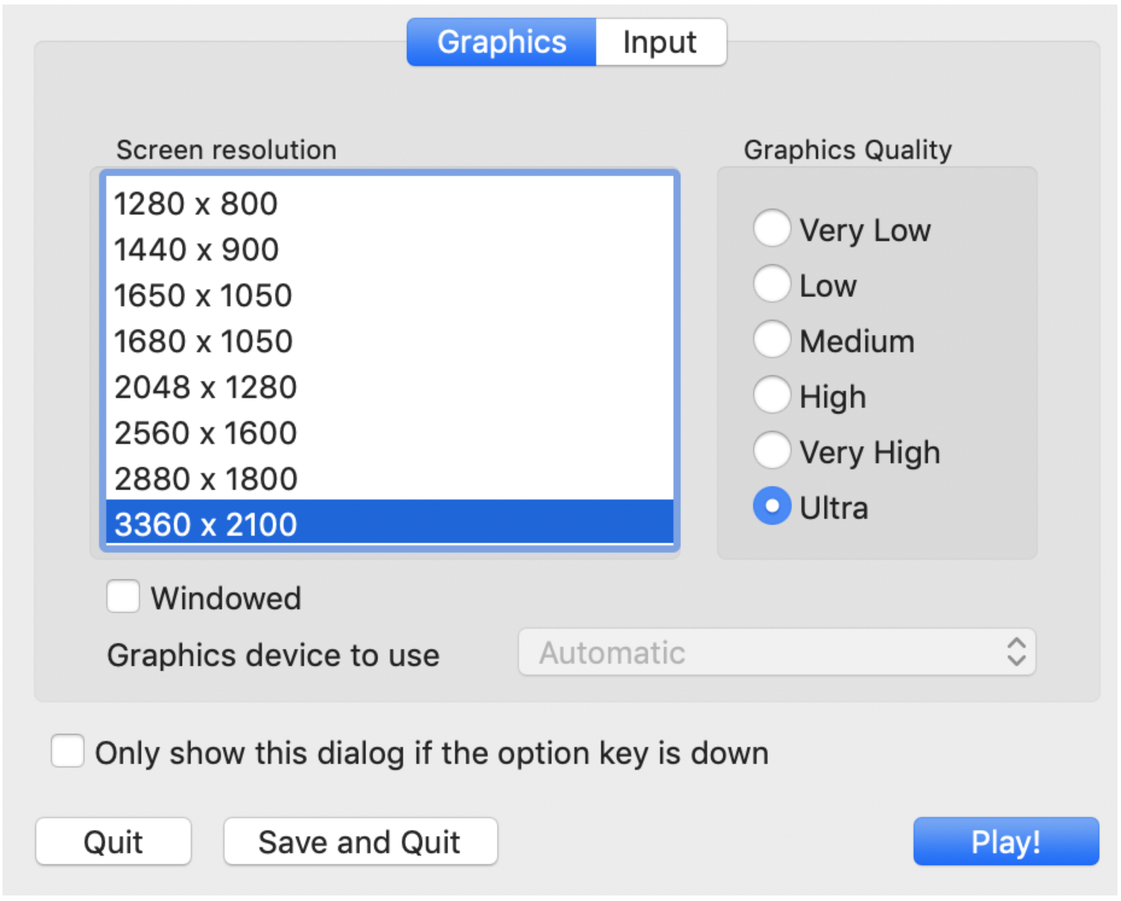
USER MANUAL

Requirement: MacOS

Step 1:
Download the Program

Step 2:
Run the Application MazeGame_v1.0.xx.app on the build directory.

Step 3:
Select the Screen resolution and click the play button



Screen resolution

1280 x 800
1440 x 900
1650 x 1050
1680 x 1050
2048 x 1280
2560 x 1600
2880 x 1800
[✓ 3360 x 2100]

Graphics Quality

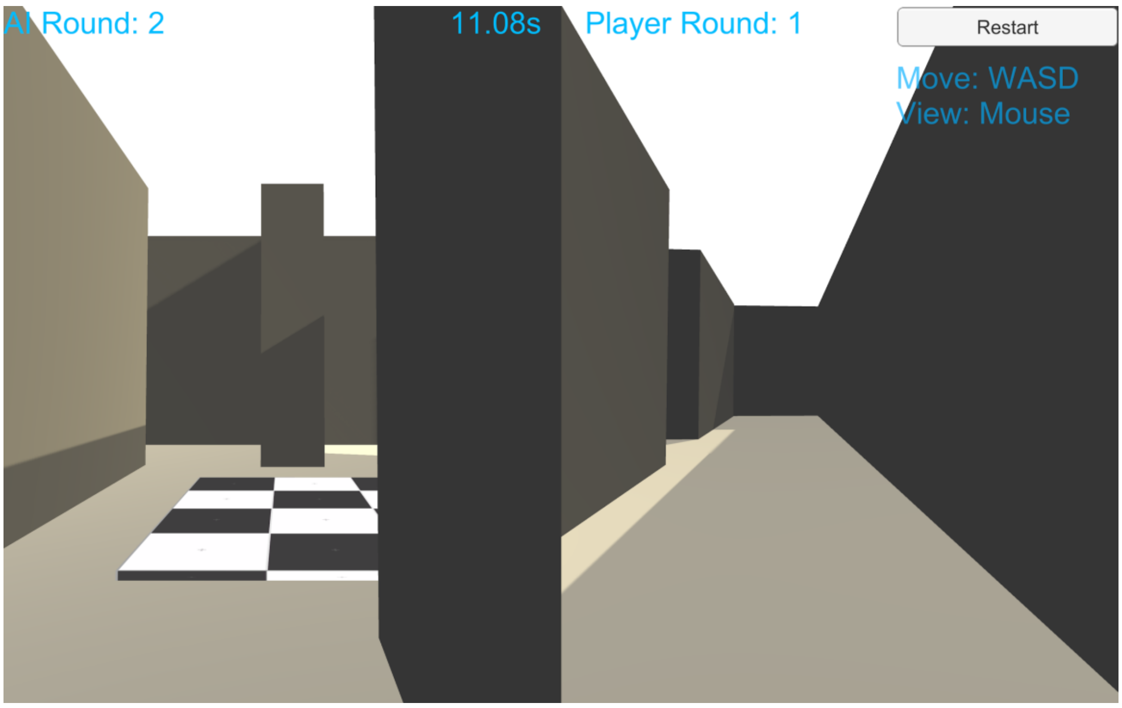
Very Low
Low
Medium
High
Very High
[✓ Ultra]

☐ Windowed

Graphics device to use: Automatic

☐ Only show this dialog if the option key is down

Step 4:
WASD for Movement, Mouse for Rotate view. Restart button for the restart.



Installation guide

For the installation guide, please refer our Github Repo
https://github.com/tavik000/MazeGameAI#author

Role and contributions of group members

Name	**Contribution**
Chen Qi	20%
Gao Huahui	20%
Wang Di	20%
Su Gengmin	20%
Zhao Haiqi	20%