Project No: A3070-201

# Multiuser Online Game with AI (B)

**Submitted by: Siew Jun Leong**
**Matriculation Number: U1821383D**

**Supervisor: Associate Professor Chua Hock Chuan**

School of Electrical & Electronic Engineering

A final year project report presented to the Nanyang Technological University
in partial fulfilment of the requirements of the degree of
Bachelor of Engineering

**2021**

# Table of Contents

# Abstract

Games are played for many reasons. It can be a platform for social interaction, a way to challenge oneself, or just to escape reality [1]. Many games have a learning curve and logical reasoning is usually required. Any opponents or enemies in a game are usually hardcoded, thus they will be unable to match a human player since they operating within limited and specific parameters.

With the introduction of newer deep reinforcement learning (RL) algorithms, Artificial Intelligence (AI) will now do more than ever before, whether it was playing a dynamic game like Dota or auto-generation of coherent images from partial images, AI succeeded in all of these.

Without any experience in doing AI, it will be a great learning experience to be able to train an AI for a strategic game without hardcoding it. Thus the objective of the project is to build a imperfect information, multiuser online game integrated with AI. Through this project, an AI was trained with a promising level of proficiency using the Unity ML agent package, which simplifies the process of RL for a person who does not have much knowledge about AI and its training, but yet able to use an algorithm to train an RL AI for their purposes. If AI can be trained to human-like proficiency with such a simplified interface, it will open up a whole avenue for the future of games.

# Acronyms

| | |
|---|---|
| AI | Artificial Intelligence |
| ML | Machine Learning |
| NN | Neural Network |
| PPO | Proximal Policy Optimisation |
| UI | User Interface |
| RL | Reinforcement Learning |

# List of Figures

# List of Tables

# Chapter 1 Introduction

## 1.1 Background

Games are played for many reasons. It can be a platform for social interaction, a way to challenge oneself or just to escape reality [1]. Many games have a learning curve and logical reasoning are usually required. Any opponents or enemies in a game are usually hardcoded, thus they will be unable to match a human player since they operating within limited and specific parameters.

Artificial Intelligence (AI) is a term used on any computer system with the ability to perform tasks that requires human intelligence to operate. To prove that an AI capable to learn and reason just like a human being, this gave rise to the effort put into programming Artificial Intelligence to be able to compete with human players. There are quite a number of ways an AI can learn, however the focus (of this project) will only be on two of them, namely deep learning and reinforcement learning.

Deep Learning is a subset of Machine Learning (ML), which utilise artificial neural networks (NN) to improve the AI's ability to analyse large amount of unstructured and unlabelled data, with the purpose to find patterns within the data.

Meanwhile, in Reinforcement Learning, the AI learns by trial and error. This is achieved by setting a reward system to nudge the AI in the desired direction, reducing its training time. The AI will take in inputs from the environment, identify what move it is allowed to make, and produce an output based on those inputs. A reward will then be given out based on the action that is taken. The AI will try to obtain the highest reward from training.

With the rise of deep reinforcement learning algorithms, the training time for the AI is shortened. One example of an AI using deep reinforcement learning algorithm is Google's AlphaGo, which defeated a Go master in the game [2].

Nevertheless, games such as Go and chess are games with perfect information, which means everything can be seen. However, situations where no one is 100% sure are very common [3]. This is known as imperfect information, in which there are unknowns or variables that come with the situation. In recent years, studies on the implementation of AI in imperfect information games had shown results, such as Microsoft's Suphx AI for Mahjong [4] and OpenAI for Dota2 [5], in which both AI excelled in their respective fields.

## 1.2 Motivation

With the introduction of newer deep reinforcement learning algorithms, Artificial Intelligence (AI) will now do more than ever before, whether it was playing a dynamic game like Dota or auto-generation of coherent images from partial images, AI succeeded in all of these. Without any experience in doing AI, it will a great learning experience to be able to train an AI for a strategic game without hardcoding it.

## 1.3 Objectives and Scope

The aim of the project is to build a imperfect information, multiuser online game integrated with AI. The AI will be trained using a deep reinforcement learning algorithm, Proximal Policy Optimisation (PPO) [6] through the usage of Unity's own ML agent toolkit [7]. The game will be developed using C# on Unity.

As this project is a 2 part project, I will be focusing on the AI aspects of the game while my partner Sheryl Teo Swe Zen will be in charge of developing the User Interface (UI) and logic of the game [8].

## 1.4 Accomplishment

Through this project, an AI was trained with a promising level of proficiency. Also, understanding about Reinforcement Learning (RL), training rewards and scenarios, Machine Learning (ML) agents is deepened. This acts as a stepping stone for any future projects regarding these areas.

## 1.5 Organisation of thesis

Chapter 1 covers the summary, motivation, objectives and what was accomplished for the project.
Chapter 2 contains all the literature review done based on what was required for this project.
Chapter 3 talks about what was done for the project, its implementation and process.
Chapter 4 consists of the conclusion of the project and recommendations for future works.

# Chapter 2

# Literature Review

## 2.1 Proximal Policy Optimisation (PPO) [6]

PPO was an algorithm created by OpenAI to be better than its predecessors by finding a balance between complexities of samples, tuning ease, ease of implementation, and per-step update computations that minimises the cost function while making sure that there is a minimal deviation from the previous policy. The outcome of this is an algorithm that has equal performance or even outperforms other popular algorithms while having ease in both implementation and tuning.

## 2.2 Unity Learning Environment Examples [7]

Unity provided various example learning environments for reference on how to use the Machine Learning agent toolkit. The examples include pre-trained NN models and pre-written trainer file, which allows the user to see the agents in action, and train their own agents using the trainer files provided. Analysis of a few of the examples was done in order to understand how to use the toolkit.

### 2.2.1 3DBall

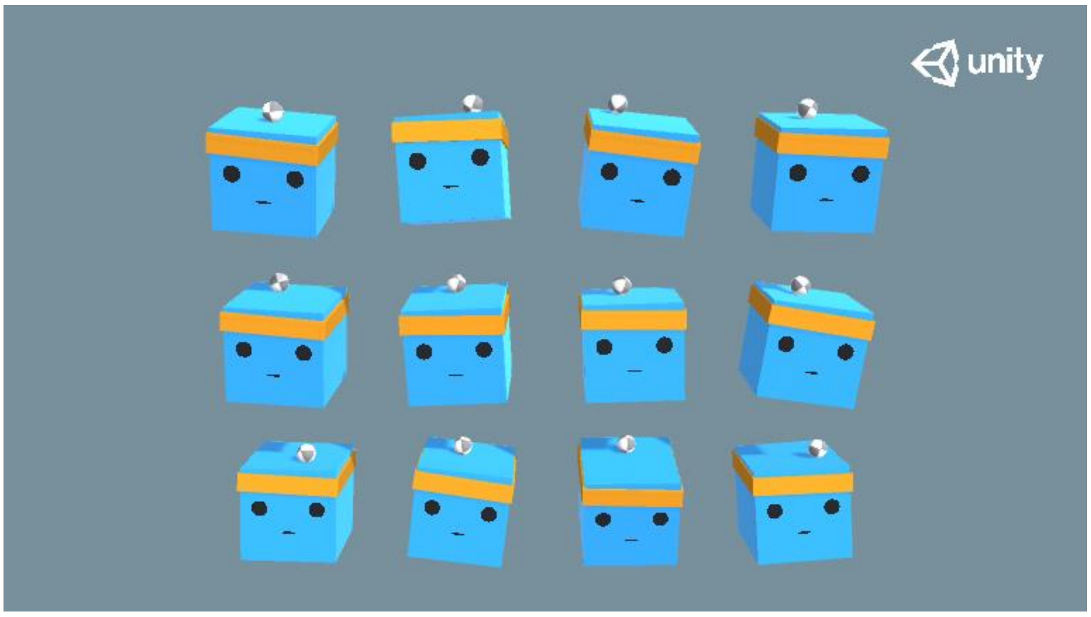


Figure 2.1: Snapshot of 3DBall from Unity

This environment requires the agent to balance the balls on the platform for as long as possible. The episode will end when the agent drops the ball, which will start another episode and reset the state to its initial stage as seen from Figure.

During training, the agents were given +0.1f every step if the ball is still on the platform, and -1f if the ball drops from the platform.

In order to balance the ball, the agents were given access to information such as the balancing platform x and z rotation, the ball vector position from the centre of the balancing platform, and the velocity of the ball. Through this information, the agent will make its decision and rotate the platform accordingly to balance the ball.

### 2.2.2 Worm

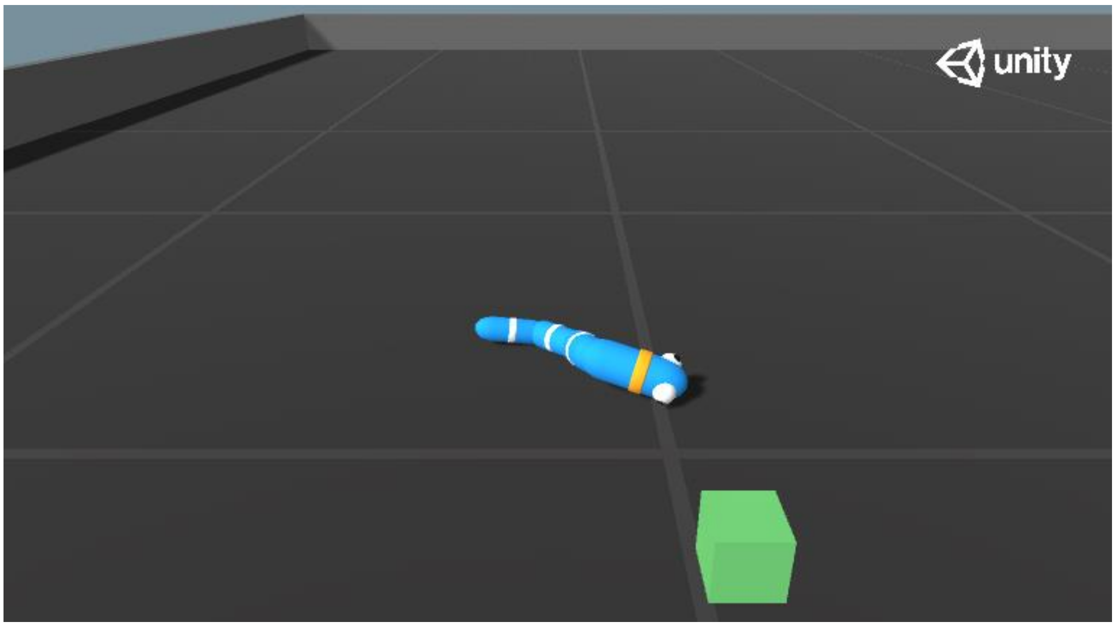


Figure 2.2: Snapshot of Worm from Unity

In this environment shown in Figure, the worm is spawned in the middle of the area and has to move towards the target (green cube). The target will respawn and its position will be random after the first one is reached.

To train the agent, the agent is rewarded +1f if the food is eaten, +0.01f * dot product between 2 normalised vectors (1 for the same direction, 0 for perpendicular direction, and -1 for opposite direction) for when the agent is moving in the direction of the food, +0.01f * dot product of 2 normalised vectors in order to train the agent to face the target, and -0.001f every step to prevent the agent from finding workarounds for maximising rewards without reaching the target.

The agent can see information such as the position of the body and the raycast position of the target for it to decide on its next step. To move, the agent is given the ability to rotate the 4 individual parts that make up the worm. This means that the agent makes to only rotate the individual parts to move towards the target.

### 2.2.3 GridWorld

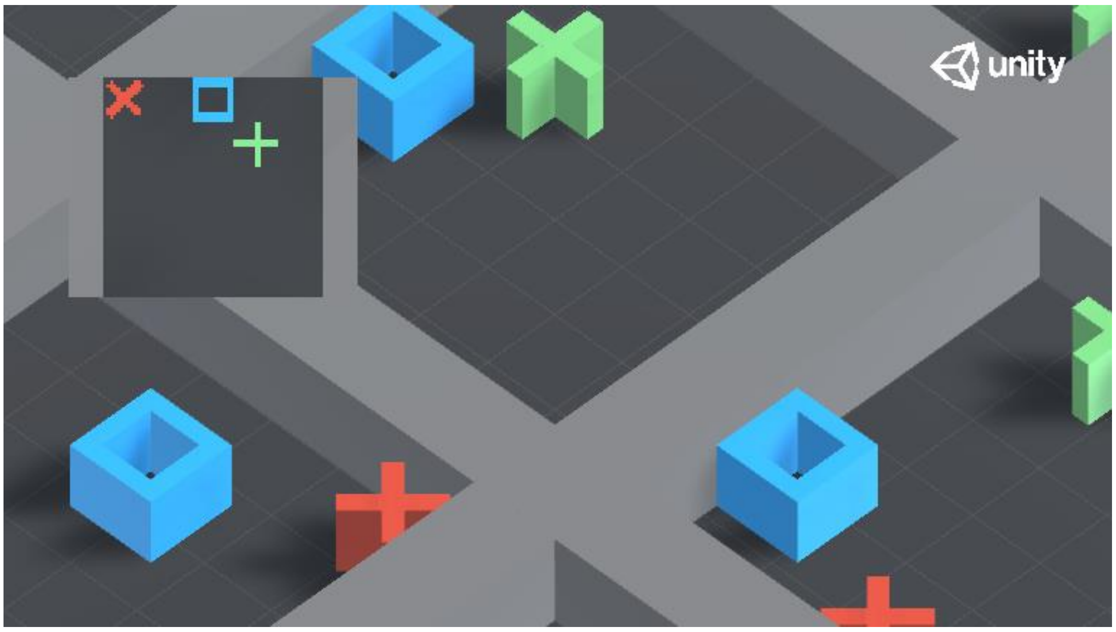


Figure 2.3: Snapshot of GridWorld from Unity

In this environment shown in Figure, the objective is to get to the goal while avoiding the obstacle. The player, goal and obstacle are spawned randomly in the 5x5 grid.

A +1f reward is given if the agent hits the goal, a -1f reward when it hits the obstacle, and -0.01f reward per step taken by the agent to reach the goal to make the agent take the shortest route to the goal.

For this example, the agent takes in the top-down render of the area as the observation instead of direct inputs from the environment like the previous examples. To achieve the objective, the agent can move in the 4 basic directions, and invalid actions, such as attempting to move left when the wall is on the left, will be masked.

## 2.3 Tic Tac Toe Agent [9]

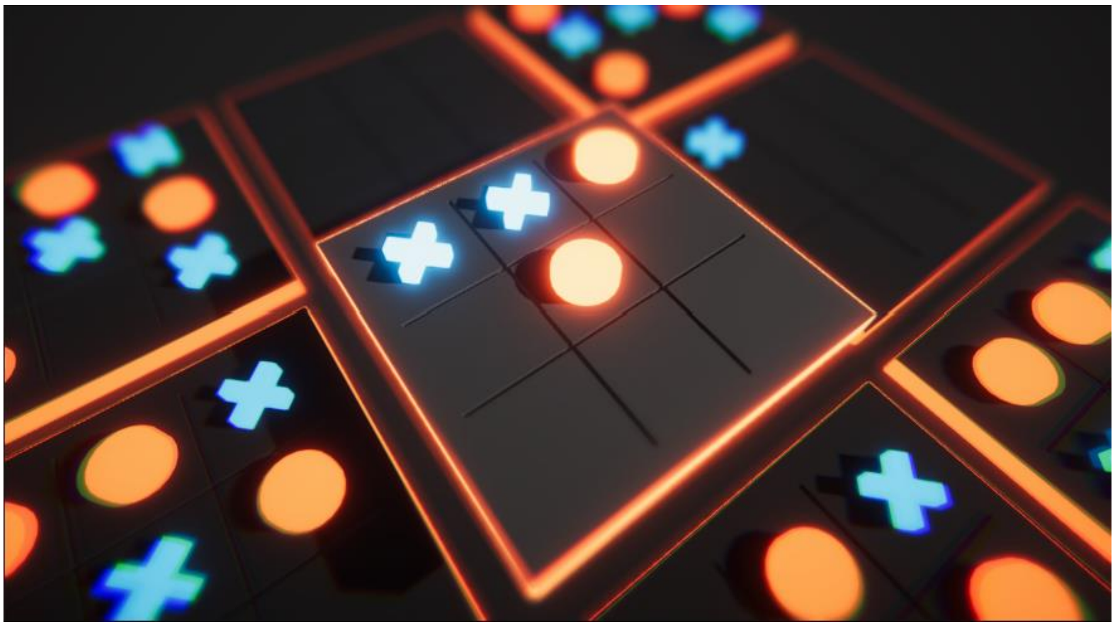


Figure 2.4: Snapshot of Tic Tac Toe Game from Unity

In order to further understand how ML agents can be applied to turn-based games since most of the examples provided by Unity happens in real-time, more research was required for reference purposes. Furthermore, the scripts used to code the scenes in the Unity examples could not be found, and thus there were no reference materials on how the code's implementation for the data going into the agent-specific functions.

### 2.3.1 Training of the Agent

This project found on Github is doing a Tic Tac Toe game using Unity ML agents. This is different from the other examples seen because this one is adversarial, and that means the agent learns by playing against itself. Thus, for the non-adversarial types of game, the cumulative reward tends towards 100 as it is the maximum reward the agent can get, as seen from Figure 2.5. However, its adversarial nature produced a cumulative reward graph as shown in Figure 2.6. This can be explained using Figure 2.7.

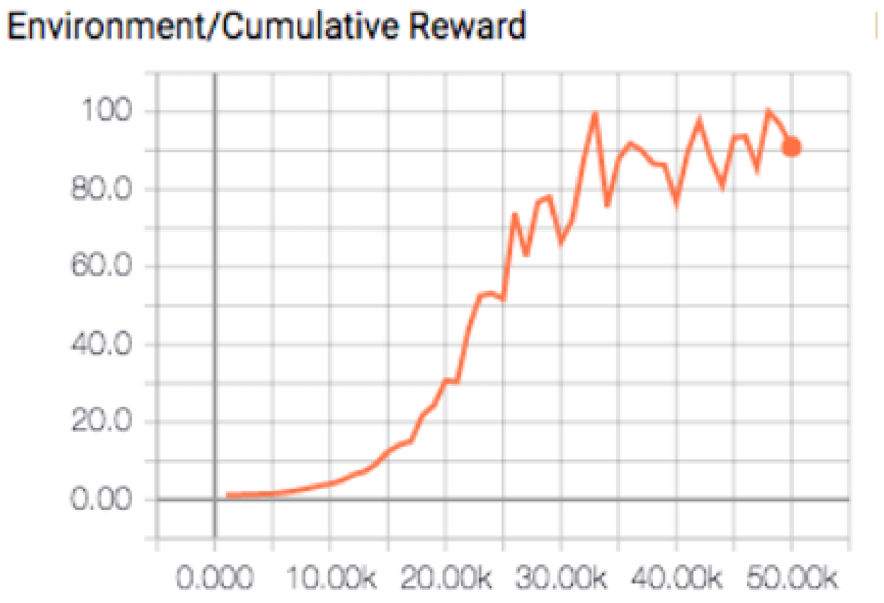


Figure 2.5: 3DBall Cumulative Reward Graph

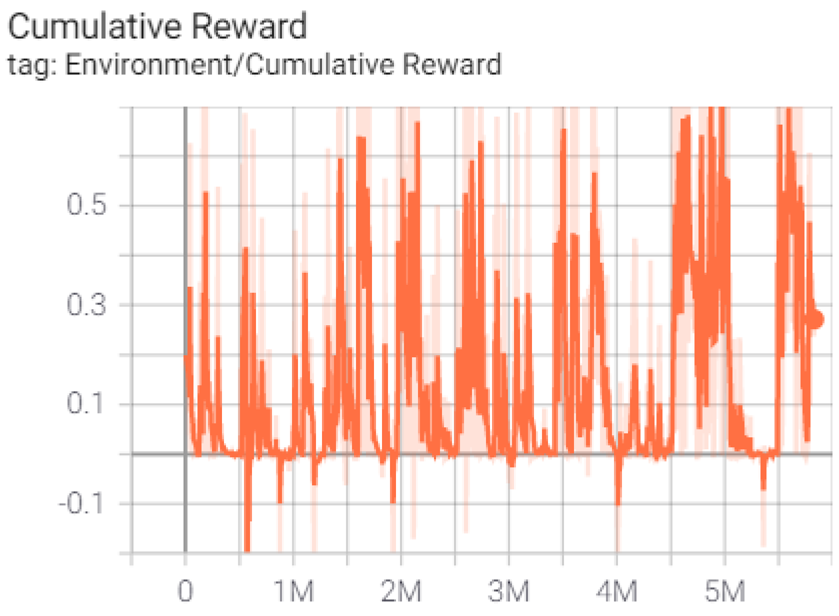


Figure 2.6: Tic Tac Toe Cumulative Reward Graph

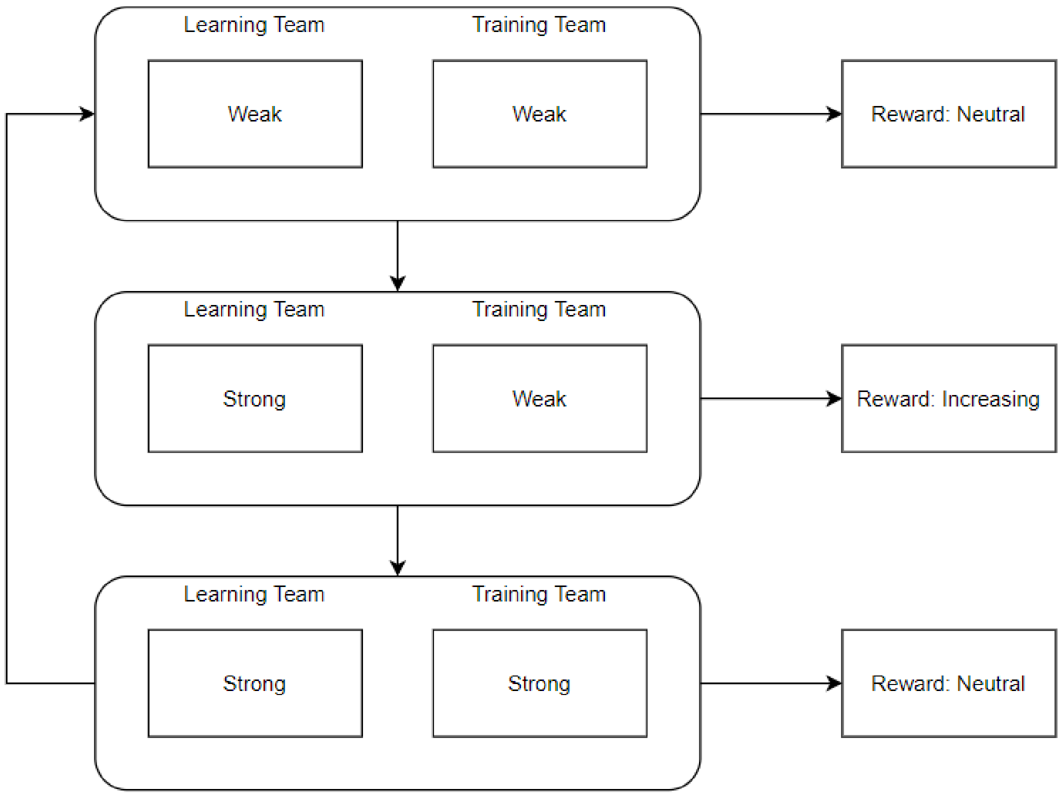


Figure 2.7: Illustration for the varied Cumulative Reward Graph

The learning team is the team figuring out new ways to win while the training team is using previous snapshots of the NN to play, thereby training the learning team. Since all teams begin with equivalent proficiency, the cumulative reward remains neutral. Since the learning team is learning, it will eventually defeat the training team, and that allows it to win more often, which results in the increase of cumulative rewards. Once the snapshot buffer is updated, the training team will be able to get the most recent snapshot, leading to the increase of proficiency of the training team, and proficiency of both teams becomes equivalent again.

As a result, proficiency cannot be determined solely by cumulative rewards. Other metrics, such as episode duration and entropy, can be viewed on Tensorboard. The agent gameplay should also be watched occasionally to gain an understanding of the agent’s proficiency.

## 2.3.2 Implementation of the Code

This environment consists of multiple 3x3 Tic Tac Toe grids as the playing area, with the purpose of not losing to the other. The agents are assigned different team ID, the one which has 0 will be X, and the other, O. When a winner is found or a draw occurs, the episode for the agent will end and the board will be cleared to start a new game. For the agent to learn, the following rewards are implemented:

- If there is a winner
    - Winner Reward, +1f
    - Loser Reward, -1f
- If it’s a draw
    - The reward for the agent which have the first move, -0.75f
    - The reward for the other agent, +0.25f

If a reward is only given when there is a winner, there will be a scarcity of rewards, thus making it harder for the agent to learn. Therefore the rewards when the game ends in a draw are implemented. Since in Tic Tac Toe, the agent who has the first move is more advantageous as compared to the other, if a draw occurs, that agent will be penalised and the other agent will be rewarded. This is to tune the agent towards getting a draw, but win if that is possible.

The agent is given access to the state of each box of the grid. Since this game is adversarial, despite being opponents, the agents are essentially solving the same problem, thus the state of the grid needs to be adjusted with respect to which side the agent belongs to. Therefore the value 1 is assigned for the agent’s side while the value 2 for the opponent’s side. This is shown in Figure 2.8 below.

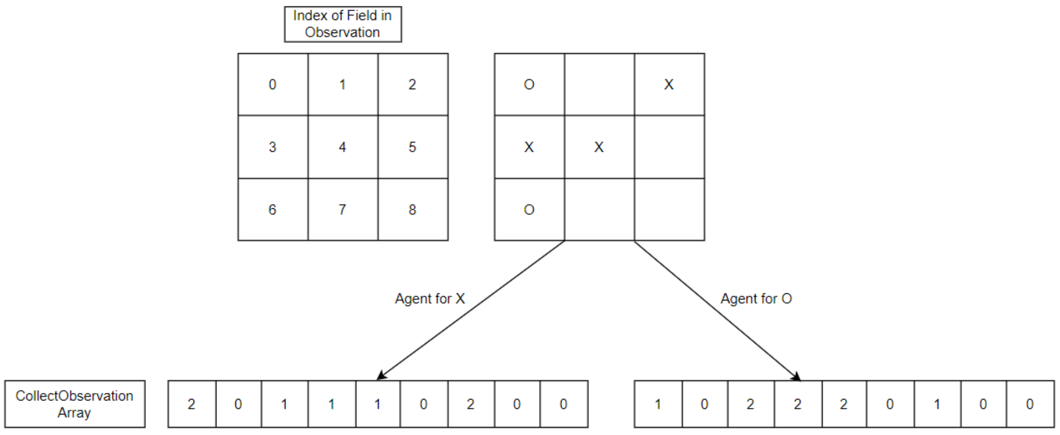


Figure 2.8: Observation Array in the Tic Tac Toe Game

Since the agent should be allowed to choose 1 of the 9 fields, the action it can take is from 0 to 8 based on the index of the field. However, the agent should not be able to choose the index that was already occupied, thus this is where the action mask comes in.

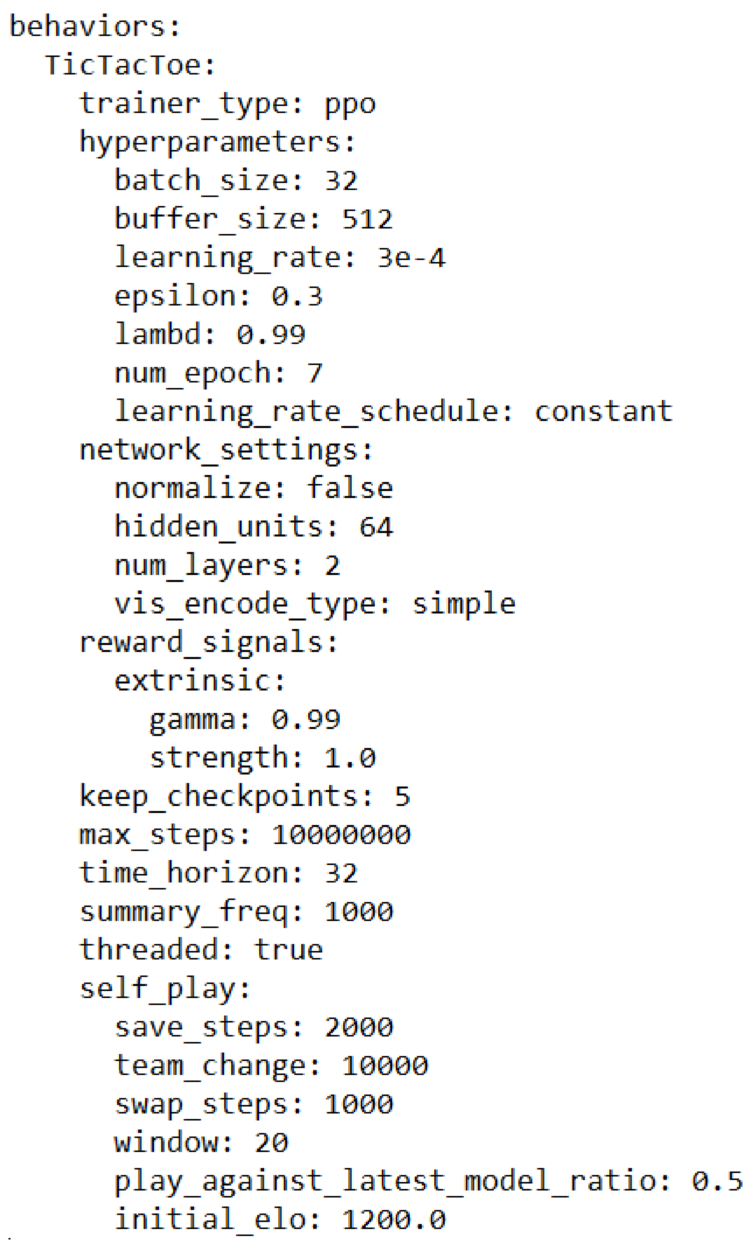
A function is required to find out which fields are occupied, then fed into the action mask buffer. With that in place, the legal action buffer for agent O, using the example in Figure will be shown in Figure 2.9.

| | | | | | |
|---|---|---|---|---|---|
| Action Mask Buffer | 0 | 2 | 3 | 4 | 6 |
| Legal Actions | 1 | 5 | 7 | 8 | |

Figure 2.9: Action Mask Buffer and Legal Actions

## 2.4 Trainer File Configuration [7]

In order for the agent to be able to train, a trainer file with the parameters for the specific scenario is required. An example for the file will be shown in figure 2.10.



```
behaviors:
  TicTacToe:
    trainer_type: ppo
    hyperparameters:
      batch_size: 32
      buffer_size: 512
      learning_rate: 3e-4
      epsilon: 0.3
      lambd: 0.99
      num_epoch: 7
      learning_rate_schedule: constant
    network_settings:
      normalize: false
      hidden_units: 64
      num_layers: 2
      vis_encode_type: simple
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
    keep_checkpoints: 5
    max_steps: 10000000
    time_horizon: 32
    summary_freq: 1000
    threaded: true
    self_play:
      save_steps: 2000
      team_change: 10000
      swap_steps: 1000
      window: 20
      play_against_latest_model_ratio: 0.5
      initial_elo: 1200.0
```

Figure 2.10: Tic Tac Toe Trainer Configuration File

This trainer file is used in the Tic Tac Toe example [9]. The purpose of each parameter that can be changed can be found on [7] under /docs/Training-Configuration-File.md.

## 2.5 Agent Class and Sequence of Actions

The agent script will inherit from the Agent class provided by Unity, which allows interfacing the game with the AI script.

The important portions of the code are the ones in the loop as shown in Figure 2.11 below: CollectObservations() and OnActionReceived(). In this case, where we do not want the agent to override any of the slots that were already occupied in the case of the TicTacToe example, CollectDiscreteActionMasks() is important as well.

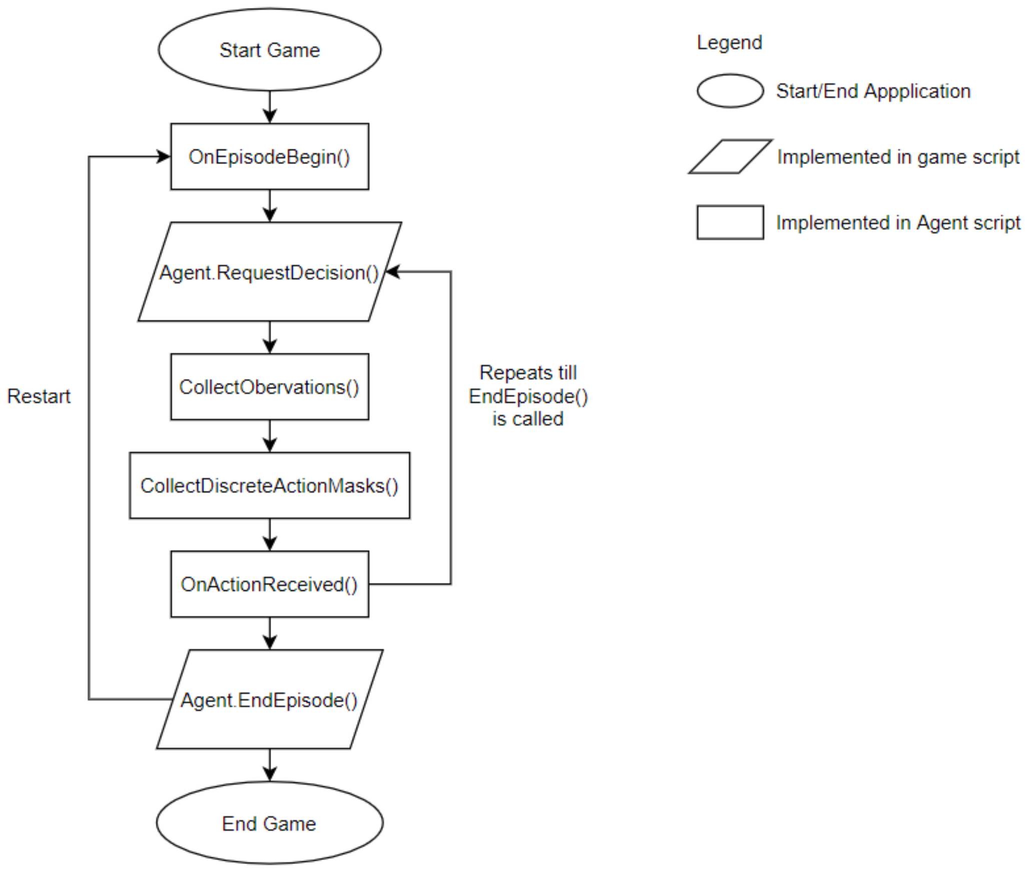


Figure 2.11: Agent Class Decision Flow

### 2.5.1 OnEpisodeBegin()

This function is called at the beginning of the agent episode, which includes the start of the simulation. This means that any function or variable that should be called or assigned at the start should be coded here.

### 2.5.2 CollectObservations()

This function will be called automatically whenever RequestDecision() is called in the program script. Any information the agent is allowed to access can be added here using AddObservation(input). Possible input types include integer, Boolean, Vector2, Vector3 and Quartenion. The number of observations made must be defined in Unity Editor under Behaviour Parameters > Vector Observations > Space Size. The observation buffer will be fed into the neural network, and an output will be given based on the input.

### 2.5.3 [Observable] attribute

This attribute can be used instead of CollectObservations(). By appending this attribute above the variable, the agent will be able to access the information as stated.

### 2.5.4 OnActionReceived()

This function will be called when an action is received from the neural network. Actions that the agent can take will be placed in here and rewards are usually assigned here based on the outcome of the action taken (eg. the agent got closer/further away from the target, time-based penalty). In the case of the TicTacToe example, rewards are going given based on the outcome of the game, such as win, lose or draw.

### 2.5.5 CollectDiscreteAgentMasks(DiscreteActionMasker actionMasker)

This function allows access to the class DiscreteActionMasker, and it is used to prevent the agent from taking actions that are deemed impossible, such as choosing the direction towards the wall. This function is only possible to use when the agent is controlled by a neural network and the action space is discrete, such as directional keys.

### 2.5.6 Rewards

Rewards are integral to a Reinforcement Learning AI. This allows them to know what actions are preferred over another based on the rewards given. Therefore, positive rewards should be given when they are doing the right thing, and vice versa.

## 2.6 Mahjong

Mahjong is a tile-based game commonly played by 4 players. It contains 144 tiles and the tile suits are shown in Table 2.1.

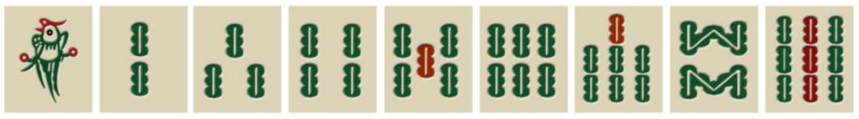
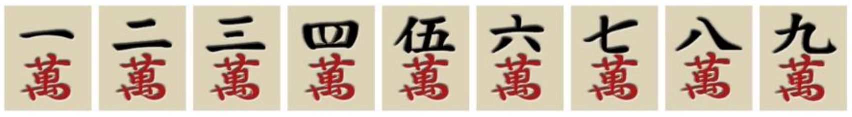
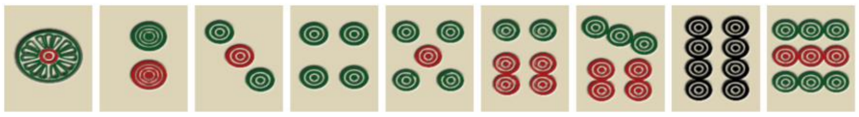
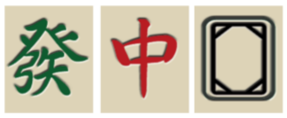
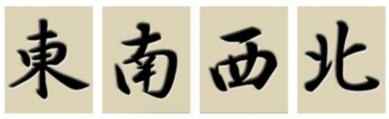
| Suits | Tiles |
|---|---|
| Bamboo |  |
| Characters |  |
| Dots |  |
| Dragons |  |
| Wind |  |

Table 2.1: Mahjong Tiles and their Suits

Each player starts the game off with 13 tiles. The players are all allocated a wind, and in the sequence of action, East > South > West > North. During the turn for each player, they can choose to either chow, pong or kong by taking the previously discarded tile if it is possible, and discard a tile from their hands. If not, the player can draw a tile from the wall, and proceed to discard a tile from their hand. This is shown in Figure 2.12.

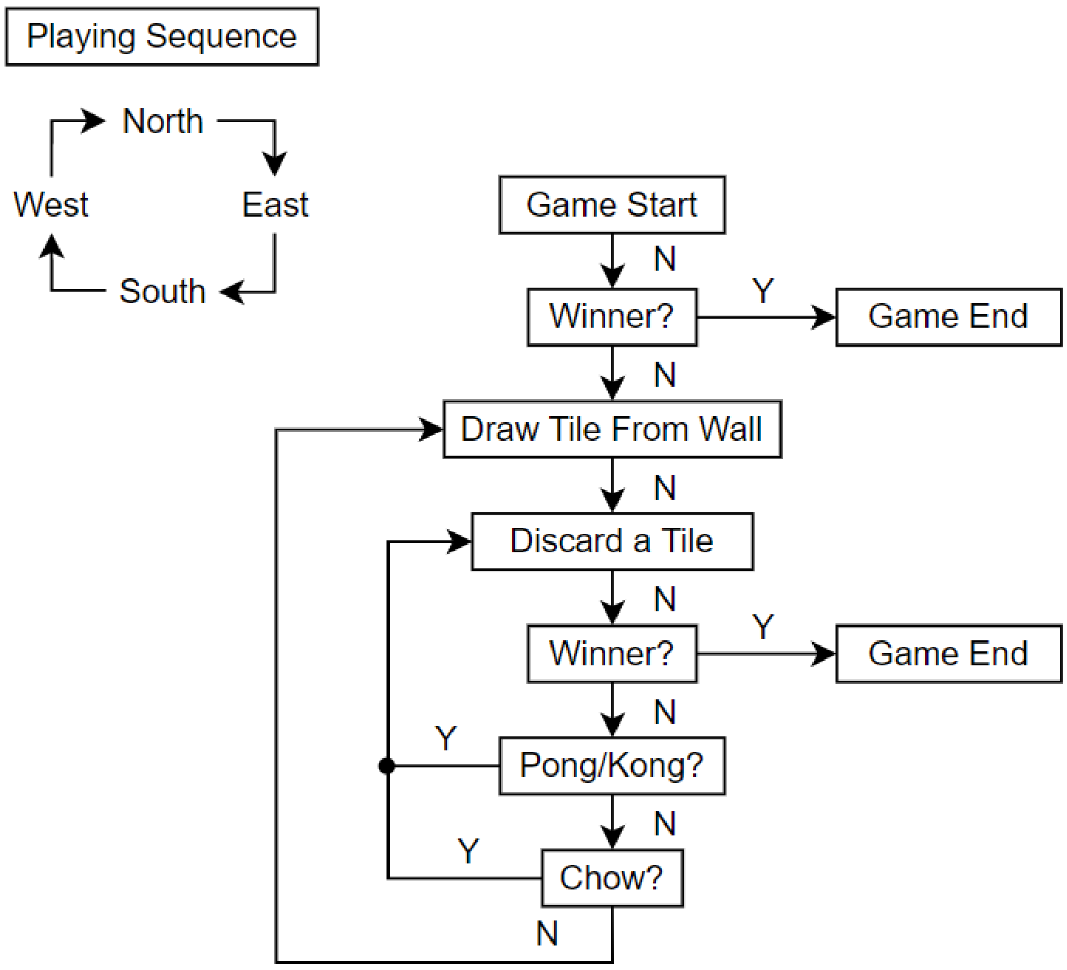


Figure 2.12: Game Flow for Mahjong

In order to win the game, the players must have 4 melds and 1 eye. An eye is a pair of tiles with the same suit and number, and melds will be discussed in detail in the sections below.

### 2.6.1 Melds

The basis of melds means that the tiles must be the same suit, and fulfil 1 of the 2 conditions: the tiles should be either in running numbers or 3 of a kind. Figure 2.13 show an example of the running number meld, and Figure 2.14 shows an example of 3 of a kind meld.

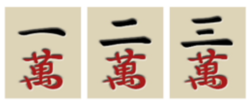


Figure 2.13: Meld with Running Numbers

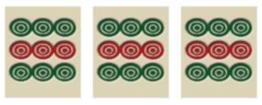


Figure 2.14: Meld with 3 of a Kind

Any melds formed without the use of discarded tiles do not have to be revealed to the other players. Otherwise, the melds formed with the use of discarded tiles have to be revealed to the other players.

## 2.6.2 Chow

Chow, the lowest priority in getting the discarded tile, is used to complete running number melds, and the source of the discarded tile can only come from the previous player. This can be visualised using Figure 2.15.

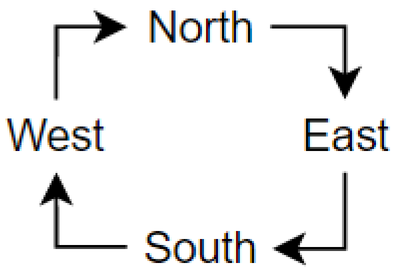


Figure 2.15: Player Turns

Since it is based on the playing sequence, the East player can take the discarded tile by the North player, the West player can take the discarded tile from the South player etc.

## 2.6.3 Pong/Kong

Pong/Kong has a lower priority to get the discarded tile as compared to winning, and it is used for completing 3 or 4 of a kind. This action allows the player to take the tile from any other players and skips the turn of any other player in between. An example is shown in Figure 2.16.

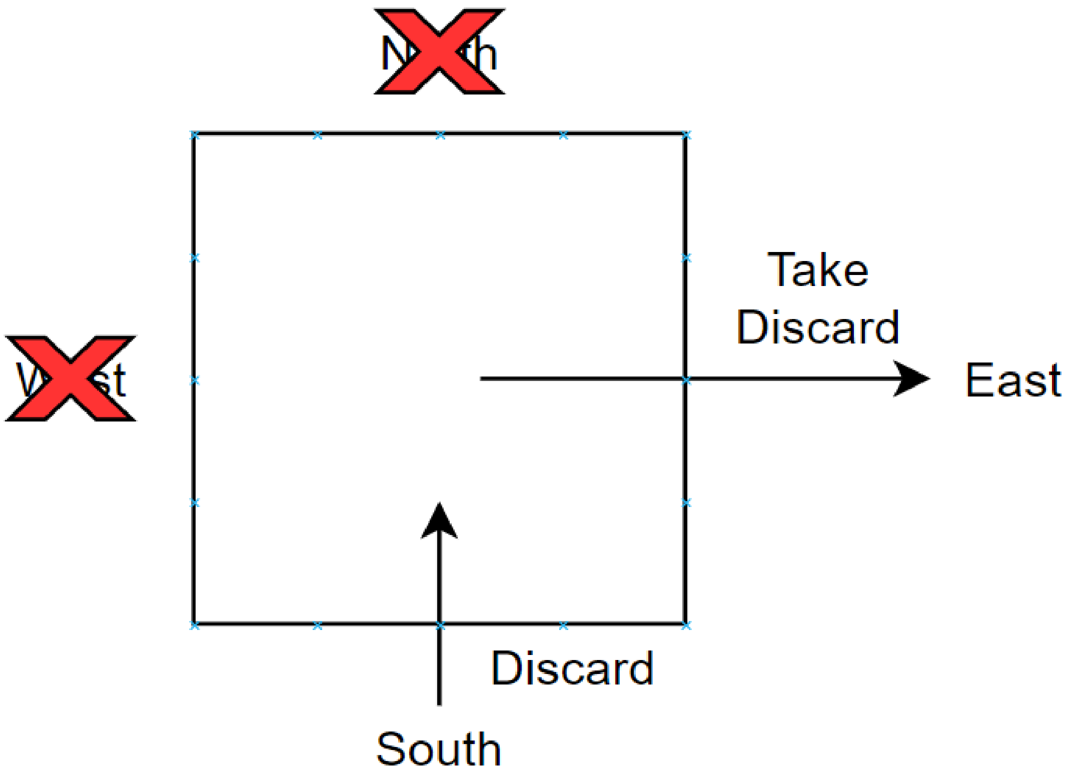


Figure 2.16: Example of Pong Skipping Turns

If the South player discards and the East player can pong or kong, the turns of the West and North players are skipped.

Chow and pong are discard tile actions that only happens if the player takes the discard tile. However, kong works for both discarded tiles and self-draw. A player who drew a tile from the wall that completes a kong has a choice to reveal the kong or not. If the kong is revealed, the player will draw a tile from the back of the wall instead of the front. In many rules, the player who kong are usually entitled to instant monetary gains.

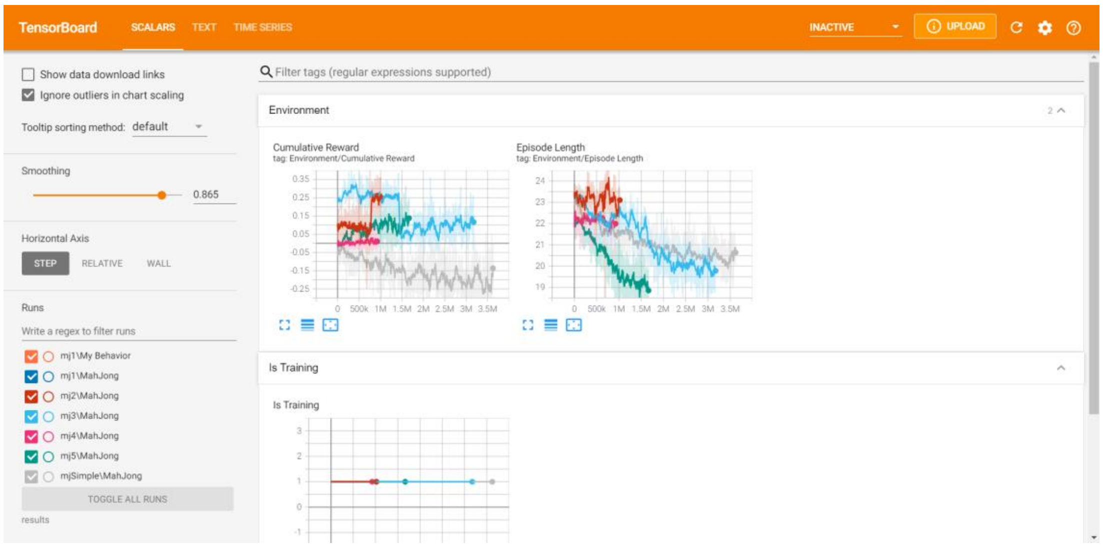
## 2.7 Tensorboard

To view the training progress, Tensorboard can be used to do that. Open the command terminal and changing the directory to the location of the project, and use the command shown in Figure 2.17 to access Tensorboard.

```
tensorboard --logdir results
```

Figure 2.17: Tensorboard command

Where results is the folder name for the training results which contains the NN file. The command will allow the user to use the browser to view the results on localhost. The page for Tensorboard is shown in Figure 2.18.



☐ Show data download links
☑ Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing 0.865
Horizontal Axis: [✓ STEP] RELATIVE WALL
Runs
Write a regex to filter runs
☑ mj1\My Behavior
☑ mj1\MahJong
☑ mj2\MahJong
☑ mj3\MahJong
☑ mj4\MahJong
☑ mj5\MahJong
☑ mjSimple\MahJong

Figure 2.18: Tensorboard

There are various graphs on Tensorboard such as cumulative reward, entropy, policy loss and many more. The details can be found in Table 2.2.

| Graph | Definition |
|---|---|
| Cumulative Reward | The average episode reward from all agents. |
| Episode Length | The average episode length for all agents. |
| Policy Loss | The average magnitude of the policy loss function. A large change in the policy results in a larger loss and vice versa. |
| Value Loss | The average loss of the value function update. This refer to the model ability to predict the value of each state. |
| Entropy | The randomness of the model’s decision. This should decrease gradually as the model learns succesfully. |

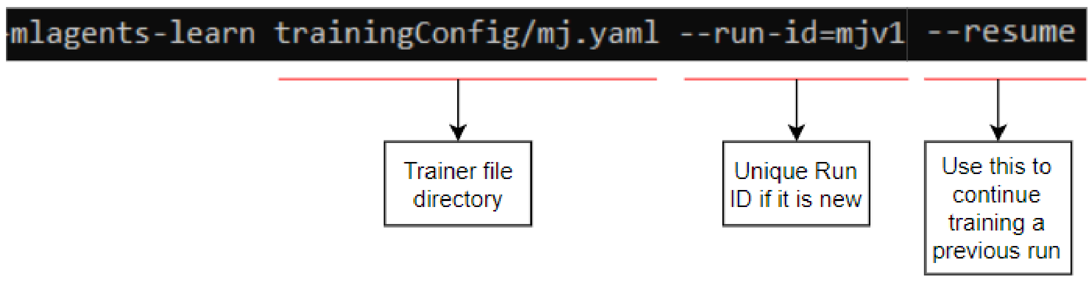
Table 2.2: Graphs and its Purposes

## 2.8 Training Process

With the agent class configurations, reward functions and trainer file ready, the agent can start to train. There are 2 ways to train the agent: training straight using the Unity editor or building the application from the Unity editor such that we get an application file (.exe). The former allows debugging to take place in case an error was found, but the latter allows faster training since more instances of the program to be running concurrently. Both options have very similar steps to start the training.

### 2.8.1 Training Using the Unity Editor

To train using the Unity editor, the Unity editor must first be open. Next, open the command terminal and change the directory to where the project is located. Once that is done, use the command shown in Figure 2.19.

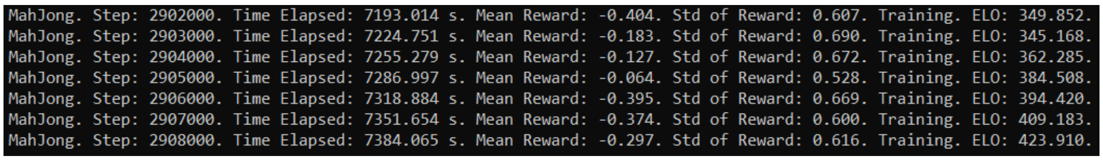


```
mlagents-learn trainingConfig/mj.yaml --run-id=mjv1 --resume
```

Trainer file directory
Unique Run ID if it is new
Use this to continue training a previous run

Figure 2.19: Command for Training

In the example given, the run ID is mjv1, and this ID can be anything if it is a new run. If the continuation of the training for the previous run is required, the –resume tag is required to be used and the run ID must be the one used for the previous run. If the tag is not used for existing runs, the error that the run ID is used will appear.
In most cases, if it is successful, the Unity logo will appear in the command terminal, and the play button in the Unity editor is required to be pressed. Once the play button is pressed, the training will commence and data will be displayed every 1000 steps. An example of this is shown in Figure 2.20.



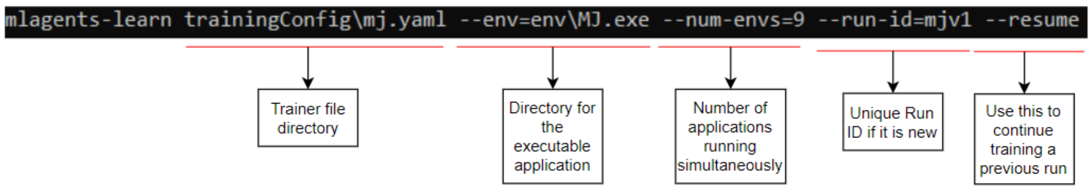
```
MahJong. Step: 2902000. Time Elapsed: 7193.014 s. Mean Reward: -0.404. Std of Reward: 0.607. Training. ELO: 349.852.
MahJong. Step: 2903000. Time Elapsed: 7224.751 s. Mean Reward: -0.183. Std of Reward: 0.690. Training. ELO: 345.168.
MahJong. Step: 2904000. Time Elapsed: 7255.279 s. Mean Reward: -0.127. Std of Reward: 0.672. Training. ELO: 362.285.
MahJong. Step: 2905000. Time Elapsed: 7286.997 s. Mean Reward: -0.064. Std of Reward: 0.528. Training. ELO: 384.508.
MahJong. Step: 2906000. Time Elapsed: 7318.884 s. Mean Reward: -0.395. Std of Reward: 0.669. Training. ELO: 394.420.
MahJong. Step: 2907000. Time Elapsed: 7351.654 s. Mean Reward: -0.374. Std of Reward: 0.600. Training. ELO: 409.183.
MahJong. Step: 2908000. Time Elapsed: 7384.065 s. Mean Reward: -0.297. Std of Reward: 0.616. Training. ELO: 423.910.
```

Figure 2.20: Step and Reward Details

### 2.8.2 Training Using the Program Executable

The program must first be built. At the moment of building, all variables values at that current point of time will be saved. Thus if there are any settings that allow the human to play instead of the agent, it will affect the training of the agent and result in a timeout error if no agent response is detected. To rectify that, the program has to be rebuilt.

The steps are the same for training using Unity editor and the program executable, with its difference at the command to initiate training. This command and its explanation is shown in Figure 2.21.



```
mlagents-learn trainingConfig\mj.yaml --env=env\MJ.exe --num-envs=9 --run-id=mjv1 --resume
```

Trainer file directory
Directory for the executable application
Number of applications running simultaneously
Unique Run ID if it is new
Use this to continue training a previous run

Figure 2.21: Training Command using Executable

Once the command is entered, the application will automatically load and if everything is booted successfully, the agent will begin training.

# Chapter 3 Mahjong Application and AI Training

## 3.1 Overview of Mahjong Game

### 3.1.1 Game Flow

The flow of the game is required to understand where the agent is required to respond and what parameters are required to be implemented in order for the agent to do so. This is depicted in the flowchart in Figure 3.1.

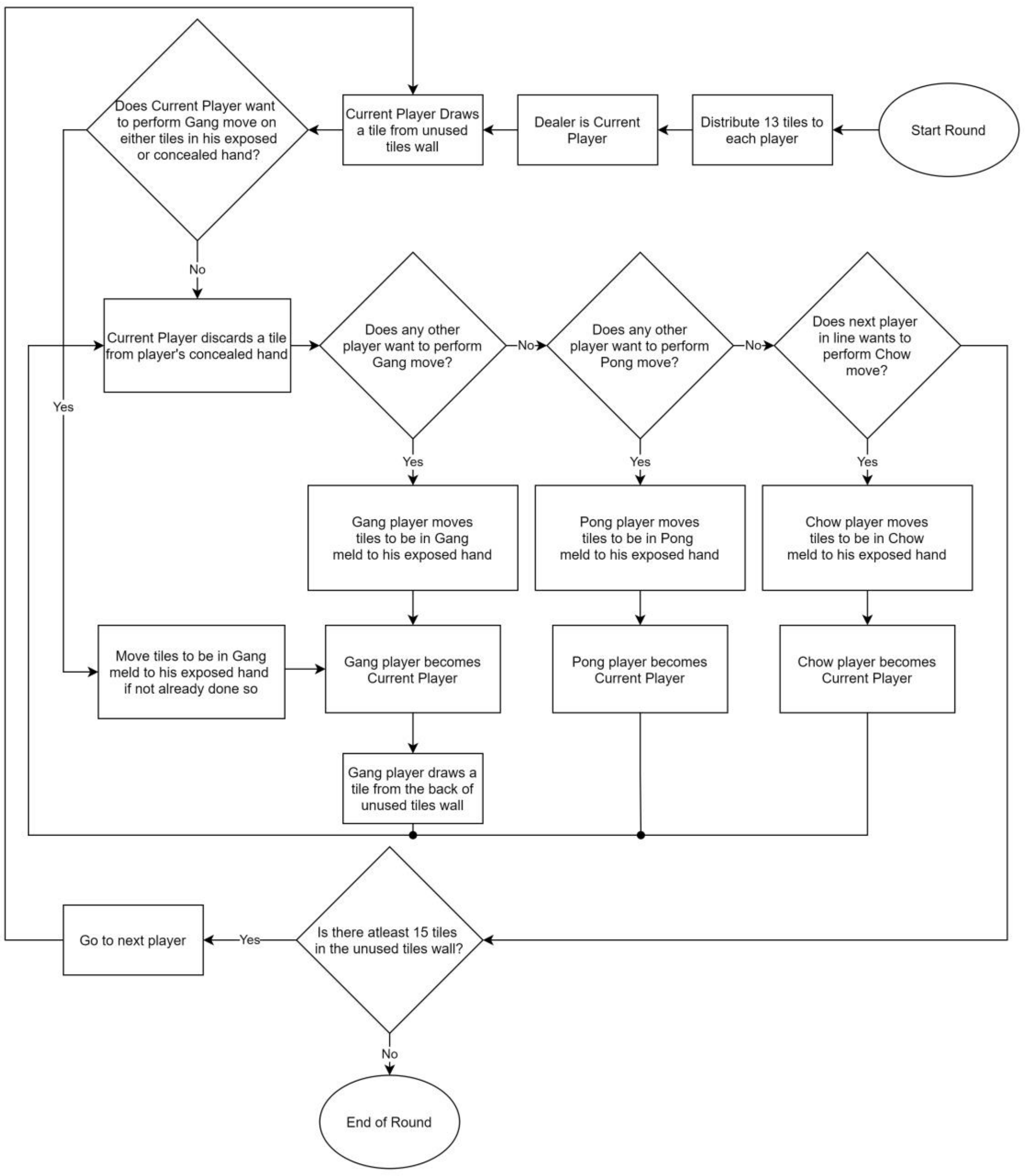

Start Round
Distribute 13 tiles to each player
Dealer is Current Player
Current Player Draws a tile from unused tiles wall
Does Current Player want to perform Gang move on either tiles in his exposed or concealed hand?
No
Yes
Current Player discards a tile from player's concealed hand
Does any other player want to perform Gang move?
No
Does any other player want to perform Pong move?
No
Does next player in line wants to perform Chow move?
Yes
Yes
Yes
Gang player moves tiles to be in Gang meld to his exposed hand
Pong player moves tiles to be in Pong meld to his exposed hand
Chow player moves tiles to be in Chow meld to his exposed hand
Move tiles to be in Gang meld to his exposed hand if not already done so
Gang player becomes Current Player
Pong player becomes Current Player
Chow player becomes Current Player
Gang player draws a tile from the back of unused tiles wall
Go to next player
Yes
Is there atleast 15 tiles in the unused tiles wall?
No
End of Round

Figure 3.1 Game Flow Diagram [8]

## 3.1.2 Numbering of Tiles

It is important for the tile to have a naming convention to link the tile face to the code. The tiles are given a unique number each, from 00 to 33. Since the name is stored as a string, the usage of double digits is to make it easier for its retrieval from a string to an integer. The numbering is shown in Figure 3.2.

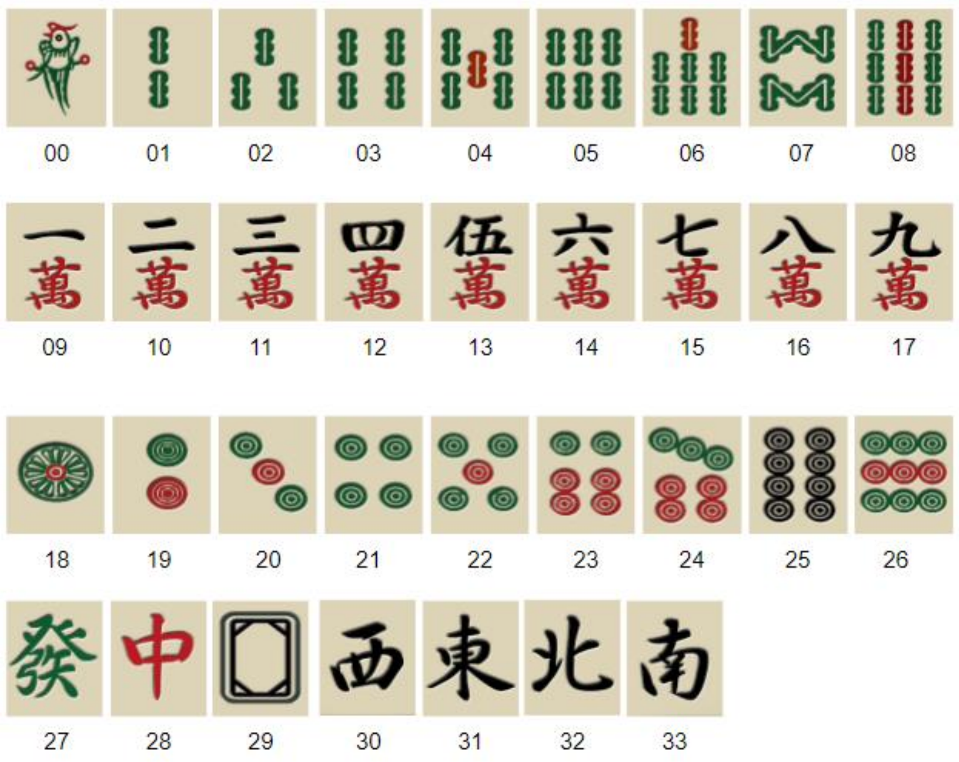


Figure 3.2 Tile Numbering

### 3.1.3 Storage of Tile Set

The format used for storing the tileset for each player is in a dynamic list. A player has 2 lists: 1 for the tileset for the player's hand, 1 for the exposed sets. An image for its respective sizes is shown in Figure 3.3.

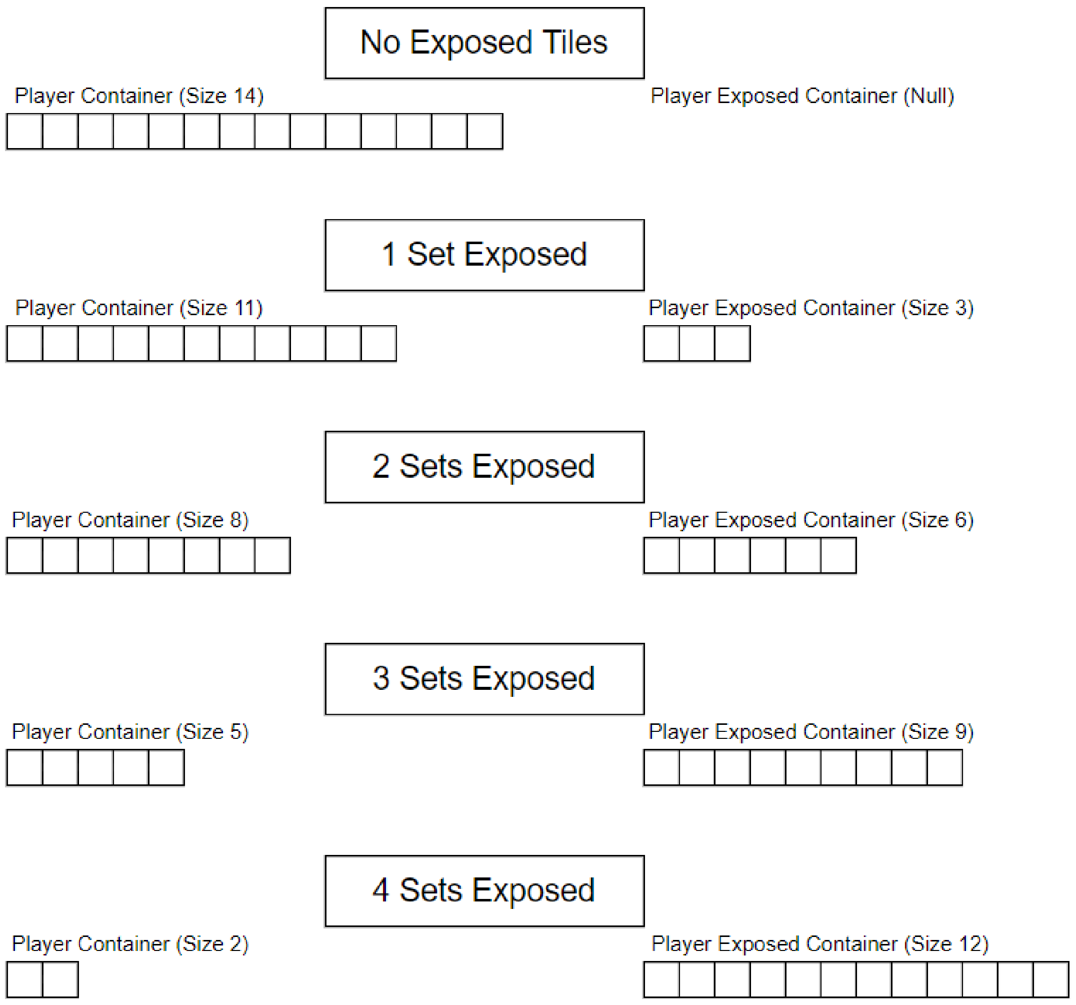

No Exposed Tiles
Player Container (Size 14)
Player Exposed Container (Null)
1 Set Exposed
Player Container (Size 11)
Player Exposed Container (Size 3)
2 Sets Exposed
Player Container (Size 8)
Player Exposed Container (Size 6)
3 Sets Exposed
Player Container (Size 5)
Player Exposed Container (Size 9)
4 Sets Exposed
Player Container (Size 2)
Player Exposed Container (Size 12)

Figure 3.3: Player Container and Player Exposed Container Size

The player container size holds true when it's currently the player's turn. If it is currently not the player's turn, the container size will be 1 less due to no card draw. Each of the element in the list will hold the value of the tile.

## 3.2 Conversion of data for AI training

The agent needs to collect observations before making a decision, thus formatting the data is required in order to use it for the agent's observation. The initial idea was to make use of the code structure which was already in place for this purpose. So, the maximum size of the player container is 14 while the maximum size of the exposed container is 16 due to the possibilities of 4 kongs. Figure 3.4 shows what would it be like if the existing code structure is used for the agent's training.

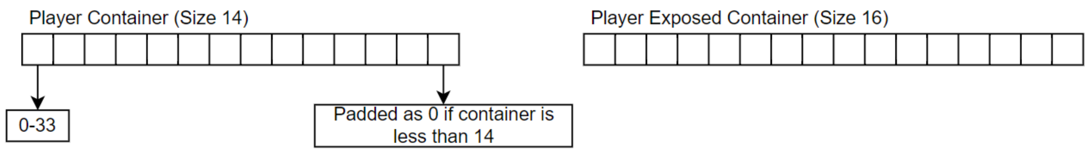

Player Container (Size 14)
Player Exposed Container (Size 16)
0-33
Padded as 0 if container is less than 14

Figure 3.4: Container Sizes

This is a problem as when the player has no exposed tileset, the player exposed container will not exist, thus throwing a null reference exception. Besides, each element of the container can have 34 possibilities, thus increasing the complexity of the agent's training. Also, padding with 0 may cause problems since 0 is used as 1 bamboo. With that, a new structure has to be created, which is shown in Figure 3.5.

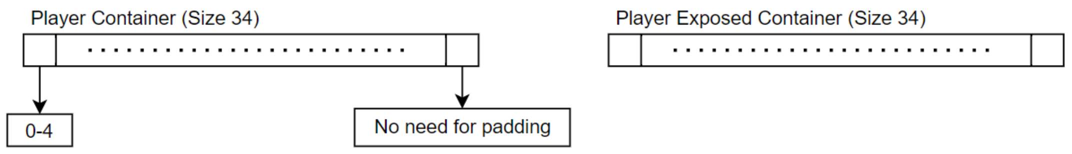
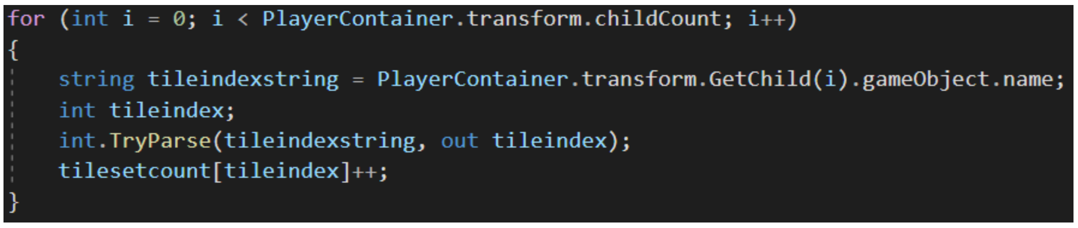

Player Container (Size 34)
Player Exposed Container (Size 34)
0-4
No need for padding

Figure 3.5: Container Sizes

The index of the array represents all the different tiles from 0 to 33. Thus the container will be able to hold all possible tiles, and there is no need for padding. It also means that this structure can be applied for both the player containers and the player exposed containers, thus there is no need to edit the conversion code.

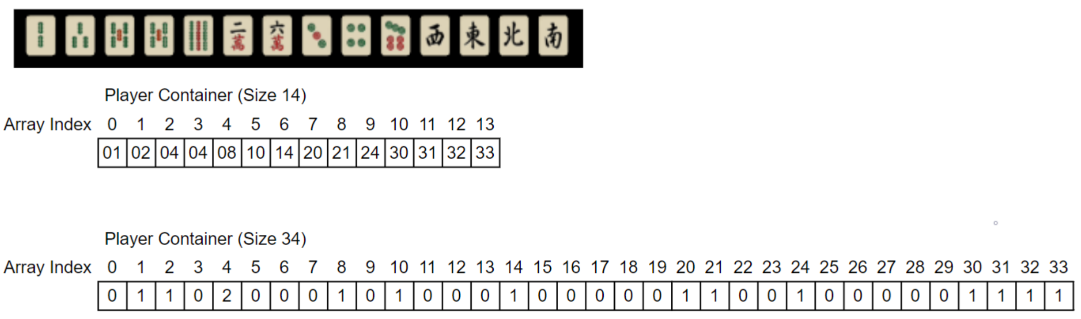
For the conversion, the player's number is fed into the function. Using the number, the player's container can be obtained using a simple switch case, and the conversion is done by the code shown in Figure 3.6.



```
for (int i = 0; i < PlayerContainer.transform.childCount; i++)
{
    string tileindexstring = PlayerContainer.transform.GetChild(i).gameObject.name;
    int tileindex;
    int.TryParse(tileindexstring, out tileindex);
    tilesetcount[tileindex]++;
}
```

Figure 3.6: Snapshot of Code

The code will look through the whole container and associate the name of the tile to its respective array index. An example of the conversion is shown in Figure 3.7.



Player Container (Size 14)

| Array Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 01 | 02 | 04 | 04 | 08 | 10 | 14 | 20 | 21 | 24 | 30 | 31 | 32 | 33 |

Player Container (Size 34)

| Array Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 1 | 0 | 2 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

Figure 3.7: Containers with Tiles Example

The type of tile is represented by the array index of the bottom array, and the number of tiles is represented by the value of the array element. This conversion code is present in ComputeHand() and ComputeOthers().

## 3.3 Agent Class Configurations

With the details shown in section 2.5, the agent has to be fitted with the required configuration for the Mahjong game.

### 3.3.1 OnEpisodeBegin()

Since Mahjong is a 4 player game, each agent needs to be allocated a number to identify them. To do this, each agent is given a team ID, and this allows each agent to be identified as which player they are by checking the current player number and the agent’s player number. The team ID is also important during training to identify the learning team and training team.

### 3.3.2 CollectObservations()

There are a few things the agent should be able to observe:

- Their respective concealed hand tiles
- Their own exposed tiles
- Others exposed tiles
- Discarded tiles

Knowing their own tiles is important to be able to plan to win, and in order not to create confusion from combining the concealed tiles and exposed tiles, both are kept separated. The exposed tiles of others and discarded tiles are important at a higher level, such as preventing other people from winning by not discarding the similar type of tiles. In this project, the point system is not implemented: it is only a base game to try to win. Thus the exposed tiles of others and the discarded tiles are combined during the conversion. The illustration of the CollectObservations() array is shown in Figure 3.8.

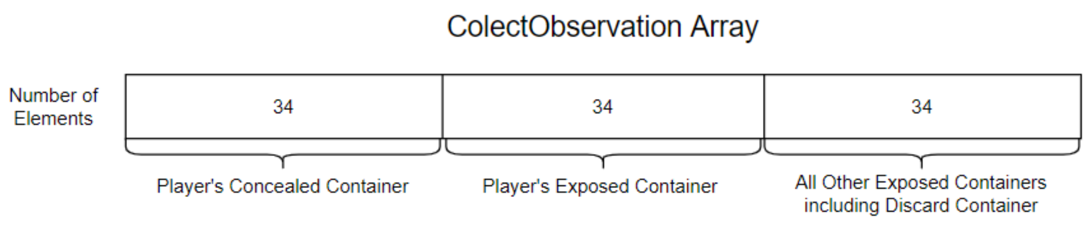
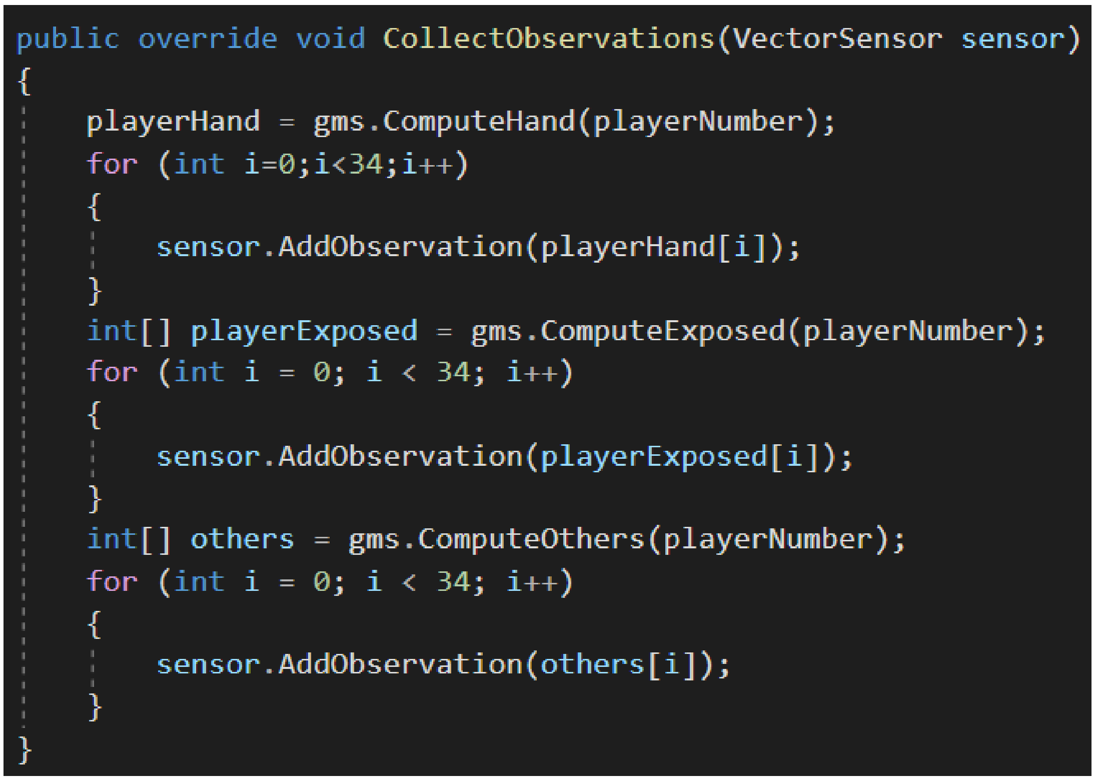

ColectObservation Array
Number of Elements
34
34
34
Player's Concealed Container
Player's Exposed Container
All Other Exposed Containers including Discard Container

Figure 3.8: Collect Observation Array

Each element of the computed array is sent individually using AddObservation(), and the implementation is shown in Figure 3.9.



```
public override void CollectObservations(VectorSensor sensor)
{
    playerHand = gms.ComputeHand(playerNumber);
    for (int i=0;i<34;i++)
    {
        sensor.AddObservation(playerHand[i]);
    }
    int[] playerExposed = gms.ComputeExposed(playerNumber);
    for (int i = 0; i < 34; i++)
    {
        sensor.AddObservation(playerExposed[i]);
    }
    int[] others = gms.ComputeOthers(playerNumber);
    for (int i = 0; i < 34; i++)
    {
        sensor.AddObservation(others[i]);
    }
}
```

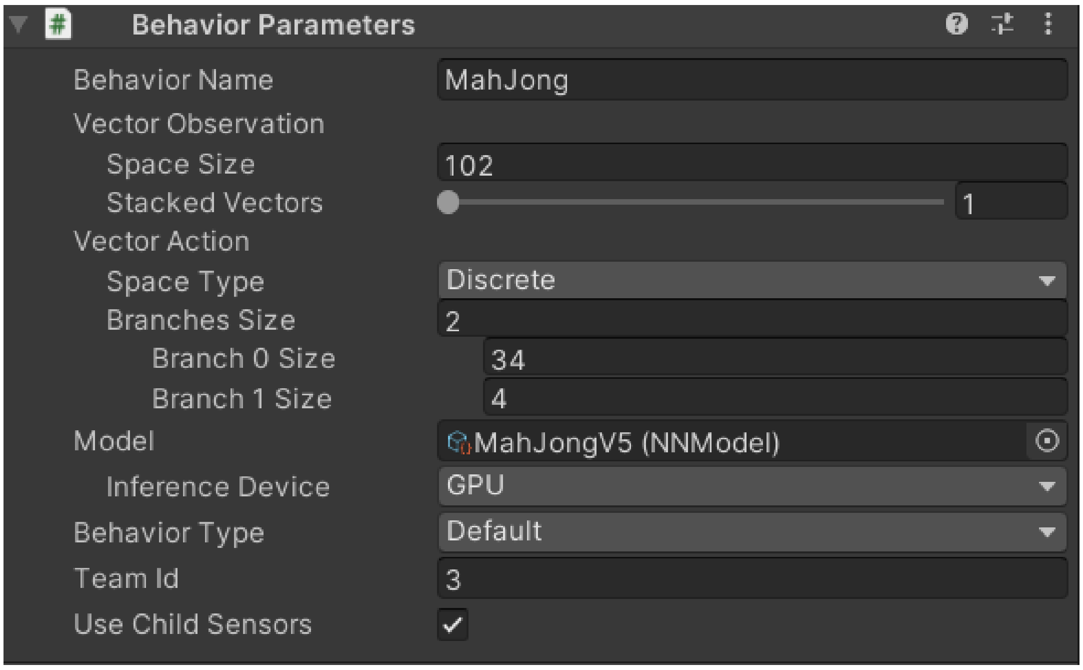
Figure 3.9: Snapshot of Code

### 3.3.3 OnActionReceived()

The actions that the agent can take have to be defined. The actions that the agent is allowed to take are:

- Discarding a tile, 34 possibilities
- Choice of Chow, 4 possibilities
- Choice of Pong, 2 possibilities
- Choice of Kong, 2 possibilities

With the exception of tile discard, the others can be put together due to their similar nature. Also, the method used to mask actions for tile discard is different from the others, thus requiring a separate vector branch. Defining the values can be done in the inspector window in Unity Editor, shown in Figure 3.10.

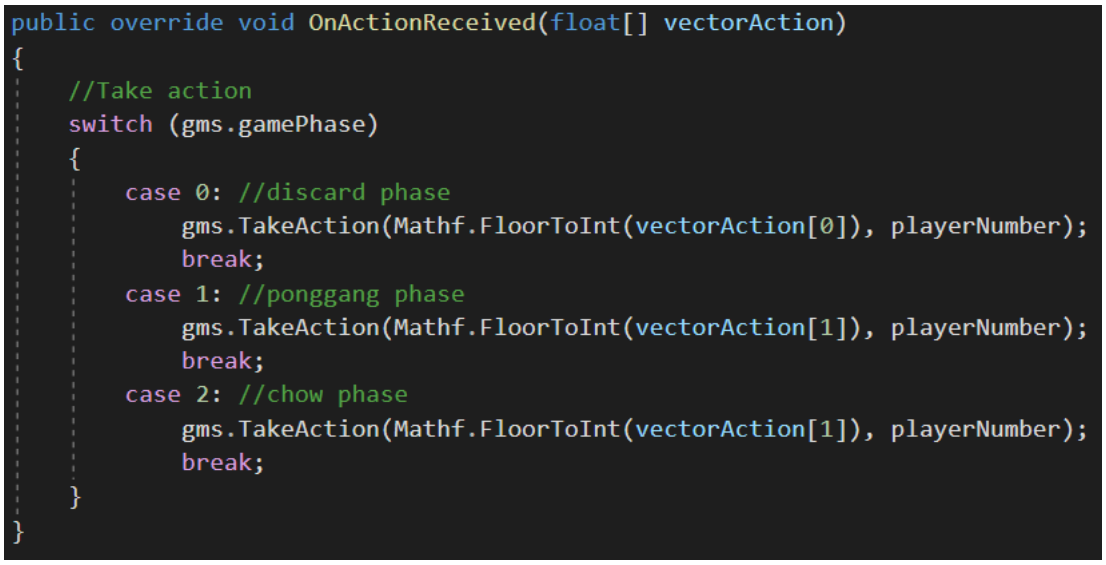


Behavior Parameters

| | |
|---|---|
| Behavior Name | MahJong |
| Vector Observation | |
| Space Size | 102 |
| Stacked Vectors | 1 |
| Vector Action | |
| Space Type | Discrete |
| Branches Size | 2 |
| Branch 0 Size | 34 |
| Branch 1 Size | 4 |
| Model | MahJongV5 (NNModel) |
| Inference Device | GPU |
| Behavior Type | Default |
| Team Id | 3 |
| Use Child Sensors | ✓ |

Figure 3.10: Behaviour Parameters in Unity Editor

Under Vector Action, branch 0 is used for discard while branch 1 is used for the others. Implementation of OnActionReceived() in Figure 3.11 uses vectorAction from its respective branch for usage.



```
public override void OnActionReceived(float[] vectorAction)
{
    //Take action
    switch (gms.gamePhase)
    {
        case 0: //discard phase
            gms.TakeAction(Mathf.FloorToInt(vectorAction[0]), playerNumber);
            break;
        case 1: //ponggang phase
            gms.TakeAction(Mathf.FloorToInt(vectorAction[1]), playerNumber);
            break;
        case 2: //chow phase
            gms.TakeAction(Mathf.FloorToInt(vectorAction[1]), playerNumber);
            break;
    }
}
```

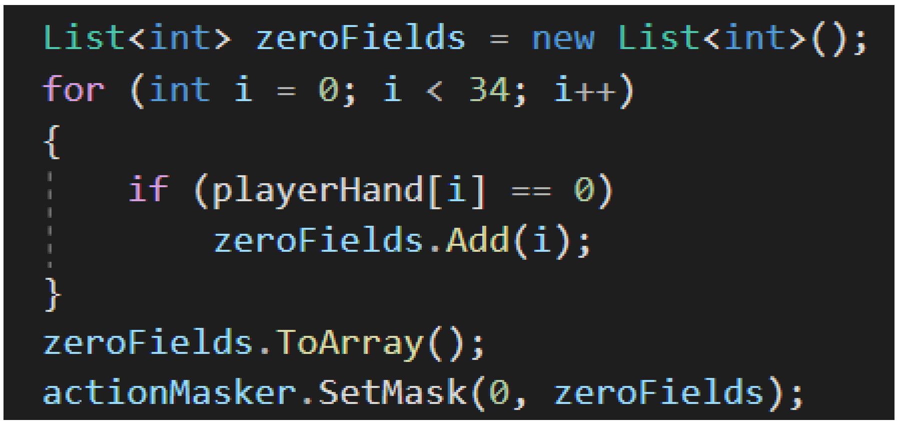
Figure 3.11: Snapshot of Code

To allow the agent to take action, it is implemented using the function TakeAction(). This function uses the value from vectorAction send it into the respective variables used to activate the next step. For the discard phase, the agent is also prevented from choosing tiles that they do not have by double-checking its action to its hand. This has to be implemented because the issue happened despite the action being masked.

### 3.3.4 CollectDiscreteActionMasks()

At any point of time, there can only be a maximum of 14 different tiles for discard, thus in order to prevent the agent from selecting impossible moves, such as selecting the tile when they do not have the tile, masking hides the tile option from the agent.

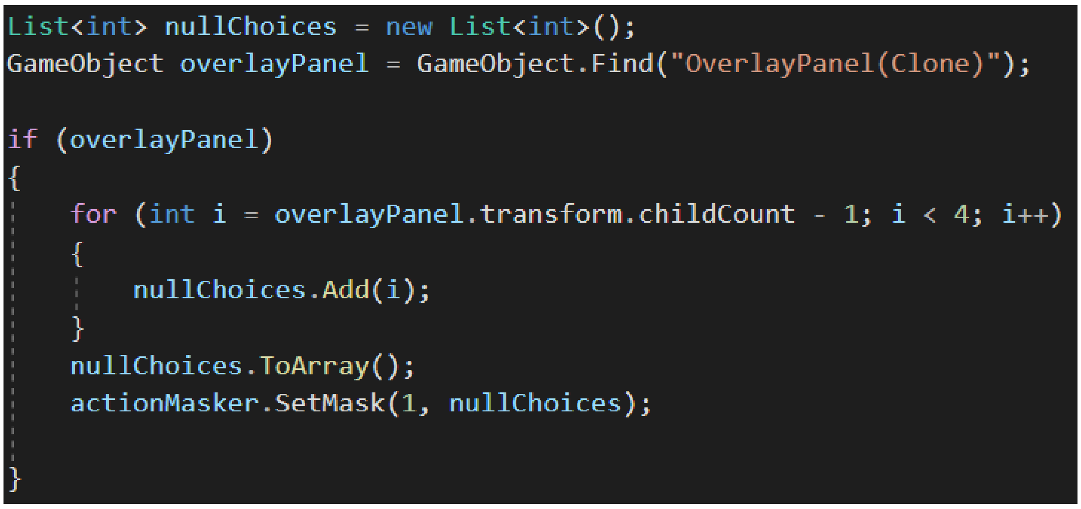
For the discard phase, the agent's concealed tiles are first checked, then all those indexes that contain 0 as their values will be masked. This is done using the code shown in Figure 3.12.



```
List<int> zeroFields = new List<int>();
for (int i = 0; i < 34; i++)
{
    if (playerHand[i] == 0)
        zeroFields.Add(i);
}
zeroFields.ToArray();
actionMasker.SetMask(0, zeroFields);
```

Figure 3.12: Snapshot of Code

However, the choice of pong/chow/kong makes use of buttons. Since the maximum number of buttons at any point in time is 4, a mask has to be set for the buttons that do not exist at that point in time. This implementation is done by the code shown in Figure 3.13.



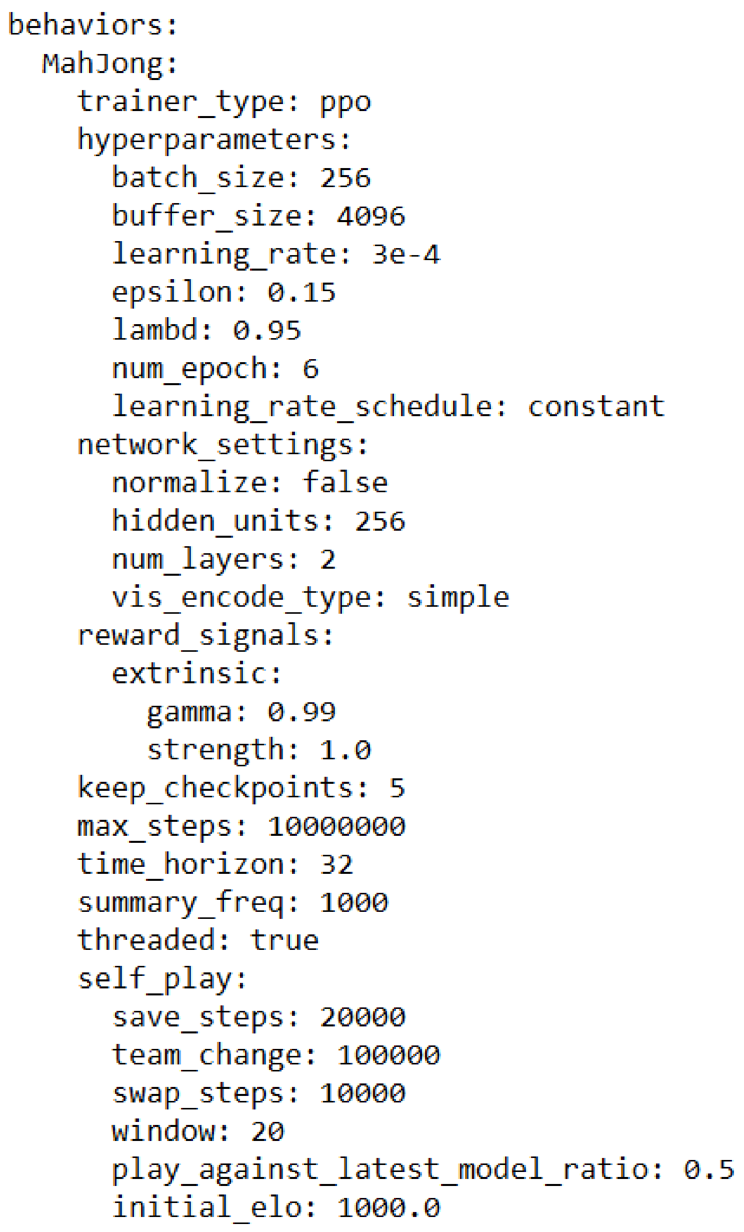
```
List<int> nullChoices = new List<int>();
GameObject overlayPanel = GameObject.Find("OverlayPanel(Clone)");

if (overlayPanel)
{
    for (int i = overlayPanel.transform.childCount - 1; i < 4; i++)
    {
        nullChoices.Add(i);
    }
    nullChoices.ToArray();
    actionMasker.SetMask(1, nullChoices);

}
```

Figure 3.13: Snapshot of Code

## 3.4 Trainer File Configurations



```
behaviors:
  MahJong:
    trainer_type: ppo
    hyperparameters:
      batch_size: 256
      buffer_size: 4096
      learning_rate: 3e-4
      epsilon: 0.15
      lambd: 0.95
      num_epoch: 6
      learning_rate_schedule: constant
    network_settings:
      normalize: false
      hidden_units: 256
      num_layers: 2
      vis_encode_type: simple
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
    keep_checkpoints: 5
    max_steps: 10000000
    time_horizon: 32
    summary_freq: 1000
    threaded: true
    self_play:
      save_steps: 20000
      team_change: 100000
      swap_steps: 10000
      window: 20
      play_against_latest_model_ratio: 0.5
      initial_elo: 1000.0
```

Figure 3.14: Trainer File Configurations for Mahjong

The trainer file used is shown in Figure 3.14. The values used for each parameter is decided based on its purpose, the default value, and its typical range the value should be.

## 3.5 Training Results and its Reward Function

There were different versions of the agents trained. 3 was done with different settings: 2 have the same reward functions but different observation parameters, the final one has a simpler reward function and the changed observation parameters.

### 3.5.1 1st Version

The first version was trained with the intention to reduce training time. The rewards are shown in Table 3.1.

| Event | Value | Reasoning |
|---|---|---|
| Win | +1 | Once the agent is not the winner of that round if there is one, they will be considered to have lost. Since this is a 4 player game, the odds of losing is extremely high, thus the rewards of losing were reduced. |
| Lose | -0.25 | |
| Draw | +0.001 | A draw can be considered as not losing, thus a very small reward is given. |
| Chow | +0.01 | A reward is given in order to steer the agent to choose to chow or pong if they are given a chance to, however this might lead to undesirable results if they are "building" their hand to win. |
| Pong | +0.01 | |
| Kong | +0.05 | In most rules, a kong usually entitles the player to instant monetary gains, thus a larger reward is given. |

Table 3.1: Reward Events, Values and Reasoning

The observation parameters are different from the one shown in Figure 3.8. This one does not separate the agent's own exposed tiles, thus aiming to tell the agent what had been exposed. This result in a smaller information array thus can potentially lead to shorter training times. The observation buffer is shown in Figure 3.15.

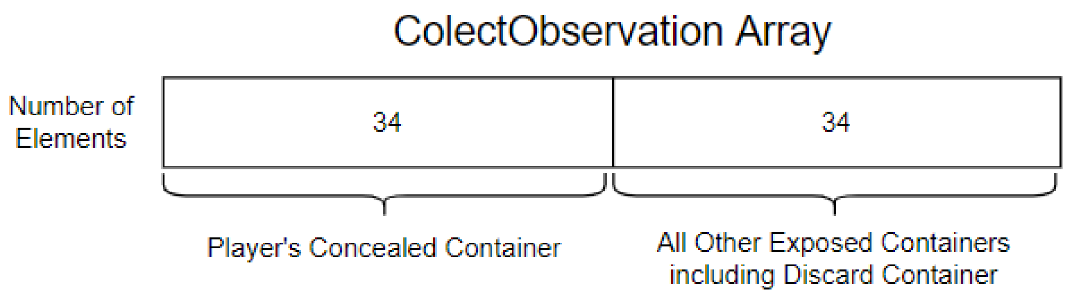

ColectObservation Array
Number of Elements
34
34
Player's Concealed Container
All Other Exposed Containers including Discard Container

Figure 3.15: Collect Observation Array

The training results for the first version is slow, partly because there was an error that prevent the start of the executable program most of the time, and when it starts, it could only train for a few hours before crashing again. Thus this was trained using the editor, which does not have any errors. However the implementation of the program make it not possible for multiple boards to be in the program at the same time, thus the training speed for this version was drastically slowed. The graphs from Tensorboard can be seen in Figure 3.16 and Figure 3.17.

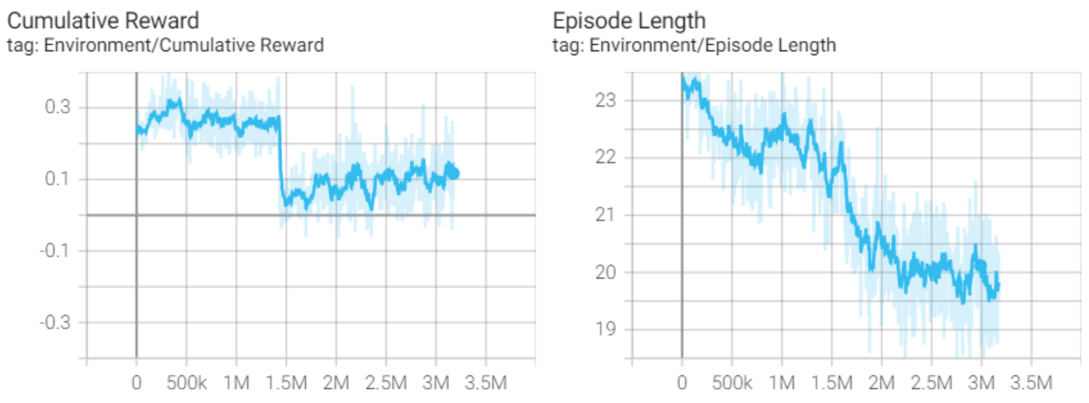


Figure 3.16: Cumulative Reward and Episode Length Graph

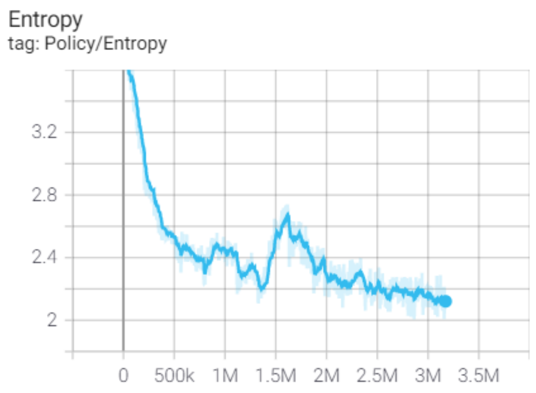


Figure 3.17: Entropy Graph

One major drawback of training from the editor is that any changes made to the reward function directly affect its training. Despite a steady drop of episode length, which correlates to winning before the episode ends, this version was dropped due to the sudden drop of the cumulative reward graph and spike of entropy, suggesting instability.

### 3.5.2 2nd Version

The second version was trained with the reward function used in the first version as seen from Table 3.1, and the observations buffer as seen from Figure 3.7. This version was able to be trained using the executable file, and the results of the training can be seen in Figure 3.18, Figure 3.19 and Figure 3.20.

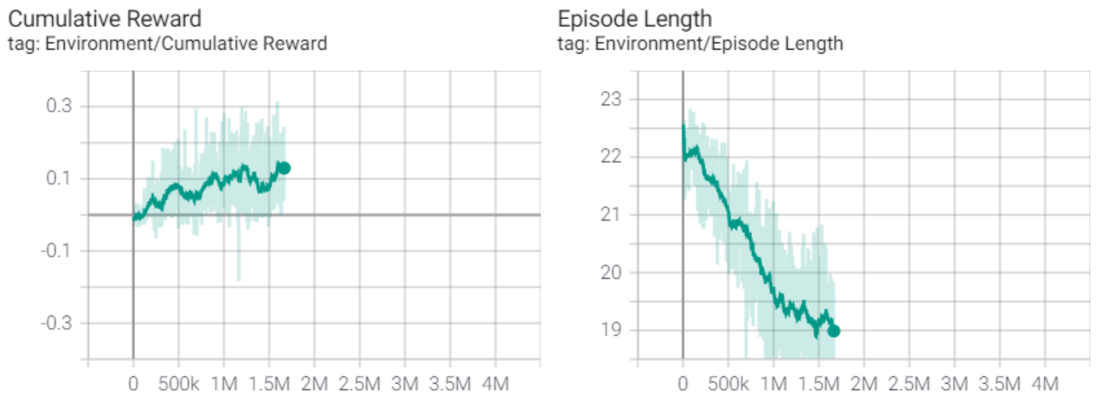
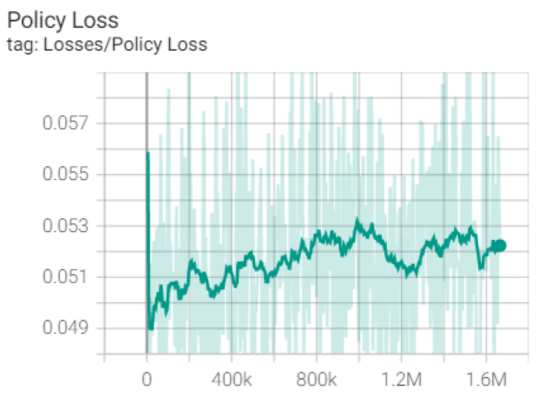


Figure 3.18: Cumulative Reward and Episode Length Graph



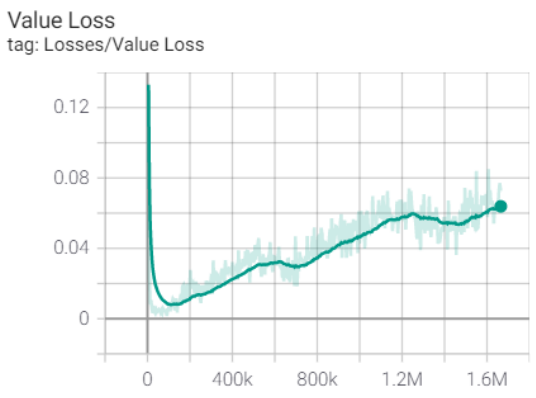


Figure 3.19: Policy Loss and Value Loss Graph

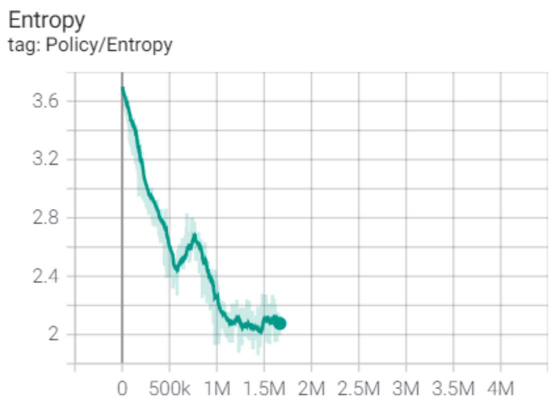


Figure 3.20: Entropy Graph

With the separated observation for the agent's own exposed tiles, the training sees a faster decrease in episode length and entropy as compared to the first version despite using the same reward functions. Small changes in the policy can be seen and the increased value loss may be due to increased agent skill and the ability to predict the value becomes harder. However, since this version is guided using the reward function to chow, pong and kong, another version is trained to see the difference between a guided version and a non guided version.

### 3.5.3 3rd Version

A third version is created to test out whether the agent is able to learn with minimal interference, thus removing the rewards given for chow, pong and kong. The reward for losing is also increased. This can be seen in Table 3.2.

| Event | Value | Reasoning |
|---|---|---|
| Win | +1 | Once the agent is not the winner of that round if there is one, they will be considered to have lost. Since the agent will maximise the rewards it will attain, no changes were made to these 2 values. |
| Lose | -1 | |
| Draw | +0.001 | A draw can be considered as not losing, thus a very small reward is given. |

Table 3.2: Reward Events, Values and Reasoning

Using the same observations used in the second version, the agent is trained and the graphs can be seen in Figure 3.21, Figure 3.22 and Figure 3.23.

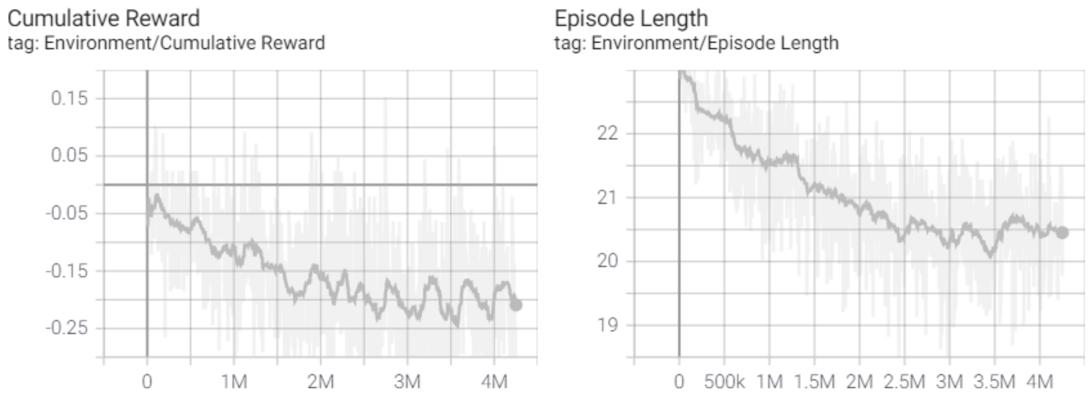
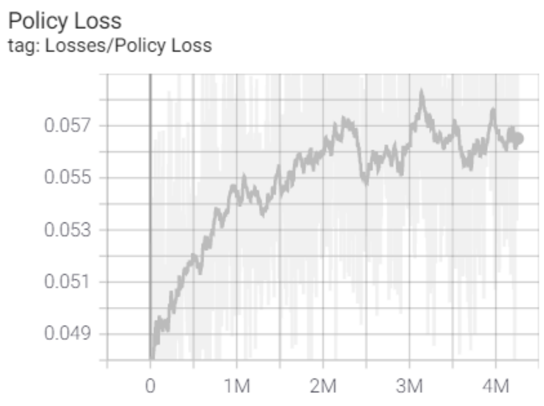


Figure 3.21 Cumulative Reward and Episode Length Graph



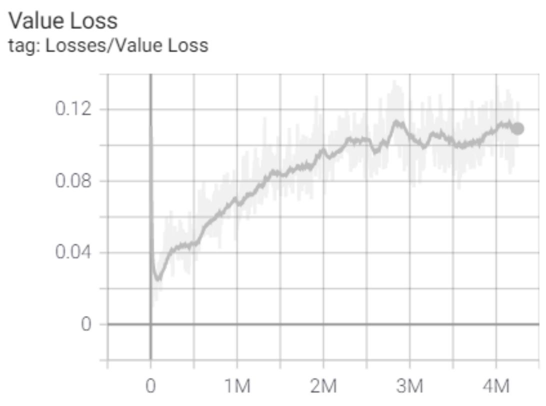


Figure 3.22: Policy Loss and Value Loss Graph

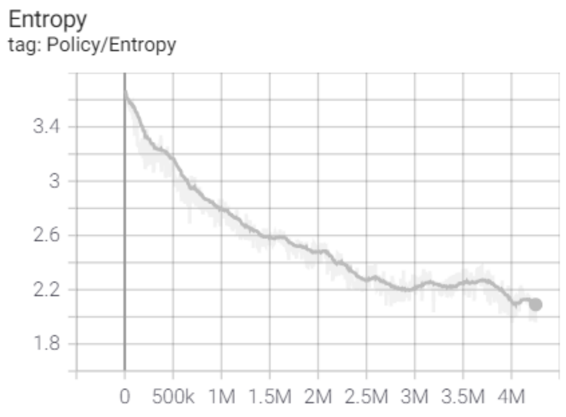


Figure 3.23: Entropy Graph

With these graphs, a comparison of different versions can be done. However, the results from cumulative rewards cannot be compared directly due to different reward functions, and therefore leading to very different reward graphs.

The fact that this version uses unguided rewards affects the rest of the graphs, making it learn slower than the other versions.

The episode length of this version is found to be longer than the previous versions. However, it also has the gentlest slope out of the 3 versions since the agent needs to explore the state space in order to get the best results.

The policy loss graph sees a larger change of policy as compared to the other versions due to a larger state space. This also affects the ability of the model to predict the value of the state as seen from Value Loss.

The Entropy graph shows that this version takes a longer time to reach the same randomness as the previous versions, however, this eliminates the problem of the agent choosing to chow, pong or gang overbuilding their tiles to win the game.

# Chapter 4

# Conclusions and Future Work

## 4.1 Conclusions

Overall, the 3rd version shows some promising results based on tests that were done, which achieved the objectives of this project. With the help of the ML agents package, it allows a person without much knowledge about AI to be able to use an algorithm to train an RL AI for their own use. In addition, if the AI can be trained to human-like proficiency with such a simplified interface, it opens up a whole avenue for the future of games.

## 4.2 Recommendation in Future Work

Improvements to the agent can be done using edited reward functions and a new NN can be trained with the help of the older NN, to speed up training and not start from scratch. In addition, curiosity, also known as intrinsic rewards, can also be enabled in order to push the agent towards exploring new states. This is because of sparse rewards: if the agent does not make the decision to get the reward, the agent does not learn. Thus by activating curiosity, this situation can be improved since there are a lot of state spaces that the agent can explore.

# References

[1] N. Yee, "Motivations for Play in Online Games", CyberPsychology & Behavior, vol. 9, no. 6, pp. 772-775, 2006. Available: 10.1089/cpb.2006.9.772.

[2] P. Mozur, "Google's AlphaGo Defeats Chinese Go Master in Win for A.I.", Nytimes.com, 2020. [Online]. Available: https://www.nytimes.com/2017/05/23/business/google-deepmind-alphago-go-champion-defeat.html. [Accessed: 17- Sep- 2020].

[3] "16.1 The Problem of Imperfect Information and Asymmetric Information", Opentextbc.ca, 2020. [Online]. Available: https://opentextbc.ca/principlesofeconomics/chapter/16-1-the-problem-of-imperfect-information-and-asymmetric-information/. [Accessed: 18- Sep- 2020].

[4] J. Li et al., "Suphx: Mastering Mahjong with Deep Reinforcement Learning", 2020. Available: https://arxiv.org/abs/2003.13590. [Accessed 17 September 2020].

[5] "OpenAI Five", OpenAI, 2018. [Online]. Available: https://openai.com/blog/openai-five/. [Accessed: 21- Sep- 2020].

[6] J. Schulman, F. Wolski, P. Dhariwal, A. Radford and O. Klimov, "Proximal Policy Optimization Algorithms", arXiv.org, 2021. [Online]. Available: https://arxiv.org/abs/1707.06347. [Accessed: 16- Mar- 2021].

[7] Juliani, A., Berges, V., Teng, E., Cohen, A., Harper, J., Elion, C., Goy, C., Gao, Y., Henry, H., Mattar, M., Lange, D. (2020). Unity: A General Platform for Intelligent Agents. arXiv preprint arXiv:1809.02627. https://github.com/Unity-Technologies/ml-agents.

[8] Teo, S., 2021. Multiuser Online Game with AI.

[9] S. Schuchmann, "Sebastian-Schuchmann/Self-Play-TicTacToe-AI-ML-Agents-", GitHub, 2021. [Online]. Available: https://github.com/Sebastian-Schuchmann/Self-Play-TicTacToe-AI-ML-Agents-. [Accessed: 16- Mar- 2021].