Multiple AI Competition in Self Developed Game

Term 1 Report

ESTR4998/4999

November 27, 2020

Partners: XIAO Tianyi

LUO Lu

Instructor: Prof. Andrej Bogdanov

Abstract

Abstract text

1 Background

1.1 Related Works

As the development of science technology, the game also evolves from the ancient chess game to the fantastic AAA video games nowadays. And with the development of artificial intelligence, bulding an agent to play a game is always a popular and interesting topic. Up to now, there are many research on playing game with nerual netwrok, like AlphaGO(Silver et al), playing Atari game(Mnih et al), and playing Dota2(Berner et al).

Inspired by previous works, we found that Reinforcement Learning is widely used in game agent training. Therefore, we decide to build reinforcement learning model in our project. Also, the game to be played by AI will be developed by ourselves, which is a soccer game.

Our goal is to build an agent which is able to play the game with

certain strategy. And an agent which can beat normal human being would be satisfactory. What's more, by adjusting configuration of the game, we wish to know if our agents could cooperate with their teammates in the multiple players mode(like 2v2 or 5v5).

1.2 TensorFlow

TensorFlow is a free and open-source software library for machine learning, which was first realeased in 2017. TensorFlow is commonly used in deep neural networks. And in this project, keras, which works as the interface for TensorFlow, is used to build our nerual network.

1.3 Pygame

Pygame is a set of Python modules for making computer games, which was first realeased in 2020. Though Pygame may not support fantastic 3D game, however, it is simple and easy to learn and use.

Additionally, considering the training process of AI, a game platform based on Python would be more suitable than other main stream game development platforms nowadays, like Unity or Unreal Engine 4. That

is the reason why we choose Pygame as our game development platform.

2 Introduction

2.1 Agents Design

2.1.1 Reinforcement learning

Reinforce learning is an area of machine learning and artificial intelligence, which concerns about how agents will take actions to achieve the best outcome in a environment. Unlike the supervised learning, reinforce learning does not need labelled data, and it focuses on exploration and exploitation. Agents can take randomly actions and receive correspond rewards to explore current environment. Agents can also make decisions by exploiting the current knowledge which comes from the exploration.

A basic reinforcement is modeled as Markov Decision Process, and a MDP consists of 4 parts:

1) A finite set of environment and agent states, named S

2) A finite set of action of agents, named A

3) Transition functions T

4) Reward function R

Reinforcement learning is widely used in many area, including animal psychology, game theory and create bots for video games. In our project, we will implement at least two kinds of reinforcement learning model: Q-learning and Deep Q-Network(DQN), discuss the advantage and disadvantage of each of them, and compare the different of them about their response when we change the rewards function or the configuration of our game.

Implementation detail of Reinforcement learning

Most of the reinforcement learning method are model-free, which means the algorithm does not require the prior known about the transition and reward functions and the agents need to estimate the models by interacting with the black box, environment.

The pseudo code of reinforcement learning is shown in Algorithm 1.

In this pseudo code, $\tilde{T}$, $\tilde{R}$, $\tilde{Q}$ and $\tilde{V}$ are the estimates of the agent, and they should be initialized before the training section. For each

Algorithm 1 Reinforcement Learning (Zoph et al, 28)

Initialize $\tilde{T}, \tilde{R}, \tilde{Q}$ *and/or* $\tilde{V}$
for each episode **do**
    $s \in S$ is initialized as the starting state
    $t := 0$
    repeat
        choose an action $a \in A(s)$
        perform action a
        observe the new state s' and received reward r
        update $\tilde{T}, \tilde{R}, \tilde{Q}$ *and/or* $\tilde{V}$
        using the experience $\langle s, a, r, s' \rangle$
        $s := s'$
        $t := t + 1$
    until s' is a goal state or t reaches the limitation

episode, we reset the environment, and initialize it to the starting state. Then the agents will repeatedly choose an action randomly or based on the knowledge, observe the return of the environment and update the estimates. For some environment, there are some goal states, and the episode will stop when one of the goal state is reached. Notice that goal states are terminal states,not the best or winning states.

2.1.2 Q-learning

Q-learning is one of the most basic and popular temporal difference learning method to estimate Q-value function(Zoph et al, 31). Tempo-

ral difference method use estimates of other value to learn their value estimates, and the update rule of TD method is:

$$V_{k+1}(s) = V_k(s) + \alpha(r + \gamma V_k(s') - V_k(s))$$

In this equation, k is the times of iteration, s is the state to be updated, α is the learning rate which should be gradually decreased as iterating, γ is the discount factor and r is the received reward.

In Q-learning, the basic idea is similar to the temporal difference learning. The difference is that value estimates V becomes Q-value function, and the estimates of other values become the previous agent's Q-value function. Therefore, the update rule of Q-learning is a variation of TD learning:

$$Q_{k+1}(s_t, a_t) = Q_k(s_t, a_t) + \alpha(r_t + \gamma max_a Q_k(s_{t+1}, a) - Q_k(s_t, a_t))$$

The difference between Q-value estimate and the value estimate in TD is that in the same state, the Q-value for different actions can be different. The transition function is determined, which means for a state s_t and the agent takes an action a_t, the agent will walk to a

certain state s_{t+1} and receive reward r_t. The discount factor γ will determine how much the expected future rewards affect the current estimates. The algorithm will focus more on short-term rewards if the γ is set lower.

The advantage of Q-learning is that it's exploration-insensitive, which means we can get the optimal policy as long as the α is set properly and every state-action pairs can be visited infinite times(Zoph et al, 31).

However, the drawback of Q-learning is also obvious. We need to store every state-action pairs in the Q table, but the numbers of states and actions are huge in many environments because the dimensions of the state are high. For each pair, we need to visit it adequate times, to update the Q-value, which is time-consuming, even impossible. In our project, we will show how this disadvantages affect the performance of Q-learning.

Implementation detail of Q-learning

The pseudo code of Q-learning is shown as Algorithm 2.

Q-learning is a kind of Reinforcement learning, so the main procedure is similar. However, there are some details to be noticed.

Algorithm 2 Reinforcement Learning (Zoph et al, 31)

Require: γ, α

Initialize Q (e.g. $Q(s,a) = 0 \ for \ \forall s \in S, \forall a \in A$)
for each episode **do**
    Modify α and γ
    $s \in S$ is initialized as the starting state
    repeat
        choose a random action or the best action $a \in A(s)$ based on the exploration strategy.
        perform action a
        observe the new state s' and received reward r
        $Q_{k+1}(s,a) = Q_k(s,a) + \alpha(r + \gamma \cdot max_{a \in A(s')} Q_k(s,a') - Q_k(s',a))$
        using the experience $\langle s, a, r, s' \rangle$
        $s := s'$
    until s' is a goal state or t reaches the limitation

Q-table	a_1	a_2	...	a_n
s_1	0	1	...	0
s_2	1	0	...	0
...	...	...	...	...
s_n	0	0	...	1

Table 1: Q-table

First, the estimates of Q-learning is Q-table. A Q-table will be initialize before the training section. Q-table is a table consists of the states of the environment and the action. For example, one possible Q-table is similar to Table 1.

Second, in the start of each episode, we may need to decrease the learning rate - α, to ensure the Q-table can converge finally. We also need to decrease the probability of random action, and the agent will make decisions more based on the knowledge got from exploration.

2.1.3 Deep Q-Network

As the previous part mentioned, the performance of Q-learning in high dimensional environment is not satisfied. However, the previous tries at using neutral network to represent the action-value function were failed. The reinforcement learning is unstable or even diverge because of the correlations present in the sequence of observation(Mnih et al).

This problem was finally solved by DeepMind, and the solution is called DQN. DeepMind introduced two techniques to remove the correlations. One is experience replay, inspired from a biological mechanism. By using this technique, the agent will save amount of experiences(state-action-reward-state) in the dataset, called "memory". Then, the al-

gorithm will randomly choice a small batch of experiences from the memory to apply Q-learning. There are some advantage to use this strategy compared the traditional Q-learning, and one of them is that this strategy can significantly remove the correlation and make the approximator stable.

The second technique is to use two separate networks. One is called target network Q, which is cloned from the original network Q every C updates. The target network is used for updating network Q, in the right side of update rule. This method can also decrease the correlation and make the algorithm more stable.

Implementation detail of DQN

The pseudo code of DQN provided by DeepMind is shown in Algorithm 3.

Algorithm 3 Deep Q-Network(Mnih et al)

Setup replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights θ'

for each episode **do**

    Initialize sequence $s_1 = x_1$ and preprocessed sequence $\phi_1 = \phi(s_1)$

    $t := 1$

    repeat

        choose a random action or the best action $a_t \in A(s)$ based on the exploration strategy.

        perform action a_t

        observe the new input x_{t+1} and received reward r_t

        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

        store the experience $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ in D

        Set $y_j = r_j + \gamma max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta')$

        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ

        Every C steps reset $\hat{Q} = Q$

    until s' is a goal state or t reaches the limitation

3 Design

The design of our FYP is based on two parts, which are the game part and AI part.

3.1 Game Design

3.1.1 Game Mode

In consideration of the cost of game development, basically the time cost, we decide to implement a game with straightforward structure. For the purpose of AI training, the game should have one clear goal and controllable user inputs, otherwise the workload and cost of the FYP could be hard to measure. Then as the result of teamwork discussion, soccer game is chosen as the game mode.

3.1.2 Game Rule

Team and players There are two teams (team-0 and team-1) in the game. For each team, ther are N players ($1 \leq N \leq 7$).

Goal If a ball pass through the goal of a team, then opposite team would get a score. And in each turn of the game, the team who

get most scores will win the game, otherwise it ends in a draw. Therefore, for each team, they should try their best to get more scores and prevent the opposite team to get any score.

time Every turn of game has a time limit. As soon as it reaches time limit, this turn of game will be forcely over.

3.1.3 Player Action

For each player, it can get the ball when it is free, steal the ball when it is catched by another player, and shoot the ball when it is catching the ball.

catch When a ball is not catched by any player, any player can try to get the ball. As soon as a player touch the ball, the player will get the ball.

Steal When a ball is catched by a player, other players can steal the ball from player. As long as another player touch the player with ball, the ball would be stolen. However, after a player just get the ball, there will be a short invincible period. Only after the invincible period, can other players steal the ball.

shoot When a player is catching the ball, the player can shoot the ball

away. It can shoot the ball along the eight directions, which are left, right, up, down, upper-left, upper-right, lower-left, lower-right.

And for the player just shoot the ball, it need to wait for a short period to be able to get the ball again.

3.1.4 Other Details About Game

boundary When the ball reach the boundary, it will bounce back. Players are not able to get out of boundary.

Initialization and Reset When each turn of game begins, every player would be assigned to an initial position, and ball will be placed in the center of the field. And when any of two teams get a score, the positions of players and ball will also be reset to initial positions.

3.2 AI Design

3.2.1 Objection

The objection of our project is to build agents which is able to play the game with cretain strategy, to find how and why the configuration of

environment effect the agents and to compare the difference between different model.

Therefore we design 2 kinds of agents in this project, the agent using Q-learning and the agent using DQN. The former is simpler, and don't need the use of machine learning. The latter is an artificial intelligence and expected to perform better.

3.2.2 Structure of Agents

Sensors In some researches, agents are created for existing video game, so the sensors of them are image recognizing CNN which are fed by screenshots of the game. For example, the researchers in DeepMind use CNN to read information from the screenshots of Atari games(Mnih et al).

However, in our project, the environement are designed for training agents from the very beginning. The agents can directly get the information, including the position of every agents and ball, by using some method. In this way, we simplify the sensor part, and the quality of CNN will not effect the training results which are most concerned.

Actuators The output of algorithm will be a list of actions, containing

the direction of moving and shotting choice. The actuator need to convert the list to another list with length of 5, to simulate the button pressed by human.

Q-table for Q-learning The main consider about the Q-table design is the size. Data receive from sensor has 6 dimonsions, and the size of each dimonsion is really huge. It's unrealistic to build a Q-table containing every details of the game, so we need to simplify the states to keep the size of the table reasonable and also containing adequate information.

There are two scheme to simplify. One is to divide the court into small pieces, so that we can ignore the trivial position detail. The size of every pieces needs to be carefully considered. If pieces are too small, the size of Q-table is still huge, and if pieces are too large, the positions of every instances are too inaccurate.

The second scheme is that we still divide the court into pieces, but just store the position of current player. Then we calculate the relative distance of other players and ball, and record the logarithm of the distance in every dimensions. The advantage is that we can save more space and the information stored are more similar to what human feel like. However, the accuracy is

a problem too.

In the project, we use the second scheme currently.

4 Implement

4.1 Game Implementation

In our pygame implementation of the soccer game, we build two important classes **Player** and **Ball**.

4.1.1 Player

The Player class inherits from Sprite, which is a pre-defined class of pygame module. Below are methods of Player.

___init___ In the ___init___ method, we define and initialize the related variavles of a player, assign an id and initial position to the player, and load the image of players.

```
class Player(Sprite):
    def __init__(self, team, initial_pos_x, initial_pos_y, pid, player_image):
        super(Player, self).__init__()
        self.id = pid
        self.team = team  # team-0: attack right door / team-1: attack left door
        self.v = Velocity(0.0, 0.0)
        self.player_image = pygame.image.load(player_image)
        self.rect = self.player_image.get_rect()
        self.rect.centerx = initial_pos_x
        self.rect.centery = initial_pos_y
        self.timer = pygame.time.Clock()
        self.cd_time = conf.shoot_cd_time
        self.shoot_dir = 99
```

Figure 1: implementation of ___init___ in Player

input_handler In the input_handler, an input array is passed into the method. The input array contains information about user input, including the four moving directions along x and y axes, and if the user want shoot the ball. If no user input on one axis, or two opposite directions (like up and down, or left and right) show up at the same time, the player will not have velocity on that axis. Otherwise, the velocity will be added on the corresponding directions. If the user want to shoot the ball, the related shoot_dir will be calculated through xy_to_dir, and then be dealt with if the ball belongs to this player.

```
def input_handler(self, input_array):
    if input_array[2] == 1 and input_array[3] == 0:
        self.v.x = -conf.player_v
    elif input_array[3] == 1 and input_array[2] == 0:
        self.v.x = conf.player_v
    else:
        self.v.x = 0
    if input_array[0] == 1 and input_array[1] == 0:
        self.v.y = -conf.player_v
    elif input_array[0] == 0 and input_array[1] == 1:
        self.v.y = conf.player_v
    else:
        self.v.y = 0
    if input_array[4] == 1:
        self.shoot_dir = xy_to_dir(self.team, input_array[3] - input_array[2],
                                   input_array[0] - input_array[1])
    else:
        self.shoot_dir = 99
```

Figure 2: implementation of init_handler in Player

```
def xy_to_dir(team, x, y):
    # 01-Right, 11-RightUp, 10-Up, 12-LeftUp, 02-Left, 22-LeftDown, 20-Down, 21-RightDown
    res = 0
    if x > 0:
        res = res + 1
    elif x < 0:
        res = res + 2
    if y > 0:
        res = res + 10
    elif y < 0:
        res = res + 20
    if res != 0:
        return res
    # default dir when no other keyboard input
    if team == 1:
        return 2
    else:
        return 1
```

Figure 3: implementation of xy_to_dir

update In the update, the position of a player would be updated according to its velocity. And boundary check will be executed, in case that the player cross over the boundary.

```
def update(self):
    pos_x = self.rect.centerx + self.v.x
    pos_y = self.rect.centery + self.v.y

    left_bound = conf.width * 0.125
    right_bound = conf.width * 0.875
    upper_bound = conf.height * 0.125
    lower_bound = conf.height * 0.875
    if pos_x < left_bound:
        pos_x = left_bound
    if pos_x > right_bound:
        pos_x = right_bound
    if pos_y < upper_bound:
        pos_y = upper_bound
    if pos_y > lower_bound:
        pos_y = lower_bound

    self.rect.centerx = int(pos_x)
    self.rect.centery = int(pos_y)
```

Figure 4: implementation of update in Player

shoot_update In shoot_update, after player shoot the ball, the timer will tick, for check_shoot_cd to check time.

```
def shoot_update(self):
    self.timer.tick()
    self.cd_time = conf.shoot_cd_time
```

Figure 5: implementation of shoot_update in Player

check_shoot_cd If a player want to get the ball, the system will check if it just shoot the ball by using the timer, which begins to tick in shoot_update.

```
def check_shoot_cd(self):
    if self.timer.tick() > self.cd_time:
        self.cd_time = conf.shoot_cd_time
        return True
    else:
        self.cd_time = self.cd_time - self.timer.get_time()
        return False
```

Figure 6: implementation of cehck_shoot_cd_time in Player

render In render, the update function will be called. Then the player will be rendered through the screen, which is passed to the render method.

```
def render(self, screen):
    self.update()
    screen.blit(self.player_image, self.rect)
```

Figure 7: implementation of render in Player

4.1.2 Ball

The Ball class inherits from Sprite, which is a pre-defined class of pygame module. Below are methods of Ball.

___init___ In the ___init___ method, we define and initialize the related variavles of the ball, assign initial position to the player, and load the image of players.

```
class Ball(Sprite):
    def __init__(self, initial_pos_x, initial_pos_y):
        super(Ball, self).__init__()
        self.ball = pygame.image.load(conf.ball_image)
        self.rect = self.ball.get_rect()
        self.rect.centerx = initial_pos_x
        self.rect.centery = initial_pos_y
        self.v = Velocity(0.0, 0.0)  # v = v - at
        self.catcher = -1
        self.timer = pygame.time.Clock()
        self.remain_time = 0
```

Figure 8: implementation of ___init___ in Ball

belong In the belong, an player id is passed into this method, and it will be checked that if the ball belongs to the player with this id.

```
def belong(self, pid):
    return pid == self.catcher
```

Figure 9: implementation of belong in Ball

caught In the caught, the ball is caught by the player with given player id. And the information about tha ball would be updated.

```
def caught(self, pid):
    self.catcher = pid
    self.timer.tick()
    self.remain_time = conf.ball_cd_time
```

Figure 10: implementation of caught in Ball

copy_pos Position of ball will be updated according to the x and y value passed.

```
def copy_pos(self, x, y):
    self.rect.centerx = x
    self.rect.centery = y
```

Figure 11: implementation of copy_pos in Ball

check_time_up In check_time_up, the remain time of invincible period will be checked. If the invincible time is up, the method

will return True, which means other players can steal the ball from the player catching the ball.

```
def check_time_up(self):
    self.remain_time = self.remain_time - self.timer.tick()
    if self.remain_time <= 0:
        self.remain_time = 0
        return True
    else:
        return False
```

Figure 12: implementation of check_time_up in Ball

shoot_ball Update relative information of the ball, after a player shoot the ball, and update velocity of the ball through dir_to_xy.

```
def shoot_ball(self, dir):
    self.catcher = -1
    self.v.x, self.v.y = dir_to_xy(dir)
```

Figure 13: implementation of shoot_ball in Ball

```python
def dir_to_xy(d):
    # 01-Right, 11-RightUp, 10-Up, 12-LeftUp, 02-Left, 22-LeftDown, 20-Down, 21-RightDown
    x = 0
    y = 0
    if d == 1:
        x = conf.player_power
    elif d == 2:
        x = -conf.player_power
    elif d == 10:
        y = -conf.player_power
    elif d == 20:
        y = conf.player_power
    elif d == 11:
        x = conf.player_power * 0.78
        y = - conf.player_power * 0.78
    elif d == 12:
        x = -conf.player_power * 0.78
        y = - conf.player_power * 0.78
    elif d == 21:
        x = conf.player_power * 0.78
        y = conf.player_power * 0.78
    elif d == 22:
        x = - conf.player_power * 0.78
        y = conf.player_power * 0.78
    return x, y
```

Figure 14: implementation of dir_to_xy

update_pos In update_pos, position of ball will be updated according to it velocity, and boundary rebounce will be done. Also, the velocity will also be updated, according to the friction of the ground through update_v.

```
def update_pos(self):
    # boundary and revert velocity here
    if 3.5 / 15 * conf.height > self.rect.centery \
            or self.rect.centery > 11.5 / 15 * conf.height:
        if self.rect.centerx < conf.width * 0.125:
            self.rect.centerx = conf.width * 0.125
            self.v.x = update_v(self.v.x * -1, conf.friction)
        if self.rect.centerx > conf.width * 0.875:
            self.rect.centerx = conf.width * 0.875
            self.v.x = update_v(self.v.x * -1, conf.friction)
    if self.rect.centery < conf.height * 0.125:
        self.rect.centery = conf.height * 0.125
        self.v.y = update_v(self.v.y * -1, conf.friction)
    if self.rect.centery > conf.height * 0.875:
        self.rect.centery = conf.height * 0.875
        self.v.y = update_v(self.v.y * -1, conf.friction)
    # update velocity according to friction
    if self.v.x != 0 and self.v.y != 0:
        self.rect.centerx = self.rect.centerx + int(self.v.x)
        self.rect.centery = self.rect.centery + int(self.v.y)
        self.v.x = update_v(self.v.x, conf.friction)
        self.v.y = update_v(self.v.y, conf.friction)
    elif self.v.x != 0:
        self.rect.centerx = self.rect.centerx + int(self.v.x)
        self.v.x = update_v(self.v.x, conf.friction)
    elif self.v.y != 0:
        self.rect.centery = self.rect.centery + int(self.v.y)
        self.v.y = update_v(self.v.y, conf.friction)
```

Figure 15: implementation of update_pos in Ball

```
def update_v(v, f):
    if int(v) == 0:
        return 0
    if v > 0:
        v = v - f
        if v <= 0:
            v = 0
    elif v < 0:
        v = v + f
        if v >= 0:
            v = 0
    return v
```

Figure 16: implementation of update_v

render In render, update_pos is called. Then, ball is rendered through the screen passed to this method.

```
def render(self, screen):
    self.update_pos()
    screen.blit(self.ball, self.rect)
```

Figure 17: implementation of render in Ball

in_door In this function, it is checked that if any team get a score. If team-0 get a score, return 0, if team-1 get a score, return 1. Otherwise, return -1.

```
def in_door(self):
    if 3.5 / 15 * conf.height < self.rect.centery < 11.5 / 15 * conf.height:
        if self.rect.centerx < conf.width * 0.125:
            return 1
        if self.rect.centerx > conf.width * 0.875:
            return 0
    return -1
```

Figure 18: implementation of in_door in Ball

4.1.3 Main Part

First, information about the game is initialized, and important variables storing information about the game are built. Here a function named initialuze_game will be called.

```
def initialize_game():
    for i in range(1, N + 1):
        team_now = 0
        image = conf.player_image_blue
        if i > N / 2:
            team_now = 1
            image = conf.player_image_red
        pos = conf.init_pos[i]
        p = Player(team_now, int(screen_rect.centerx * pos[0]),
                   int(screen_rect.centery * pos[1]), i, image)
        players.add(p)
        print("player: id={}, team={}".format(p.id, p.team))
        agent = AgentsQT(p.id)
        agents.append(agent)
```

Figure 19: implementation of initialize_game

In the main while loop of game execution, first each player will deal with its input passed from AI, through the relative method of Player and Ball.

```
for p in players.sprites():
    #input_array = get_input(p.id)
    prev_pos_x[p.id - 1] = p.rect.centerx
    prev_pos_y[p.id - 1] = p.rect.centery
    action[p.id - 1] = agents[p.id - 1].make_decision(state[p.id - 1])
    input_array = get_input_ai(p.id, action[p.id - 1])
    p.input_handler(input_array)
    if p.shoot_dir < 99:
        if ball.belong(p.id):
            p.shoot_update()
            ball.shoot_ball(p.shoot_dir)
            rewards[p.id - 1] -= 1000
        p.shoot_dir = 99
```

Figure 20: implementation of main part about dealing with input

Then, the system will detect if there is any collision between player and ball, and judge if any player get or steal the ball according to the state of ball and the player catching the ball (if any). Then position of the ball will be updated, and sysytem will check if any team get a score.

```
# deal with collision
stealer_list = []
holder = None
stealer = None
for p in players.sprites():  # check if anyone want to steal the ball
    if pygame.sprite.collide_rect(ball, p):
        if ball.belong(p.id):
            holder = p
        elif p.check_shoot_cd():
            stealer_list += [p]
if len(stealer_list) == 1:
    stealer = stealer_list[0]
elif len(stealer_list) > 1:
    stealer = stealer_list[random.randint(0, len(stealer_list) - 1)]

if stealer is None and holder is not None:  # still hold the ball
    ball.copy_pos(holder.rect.centerx, holder.rect.centery)
elif stealer is not None:  # steal the ball
    if ball.belong(-1):  # if ball is free
        ball.caught(stealer.id)
        rewards[stealer.id - 1] += 1500
    elif ball.check_time_up():  # if ball is stolen
        ball.caught(stealer.id)
```

Figure 21: implementation of main part about collision

Finally, iamge of every player and ball will be rendered un the screen.

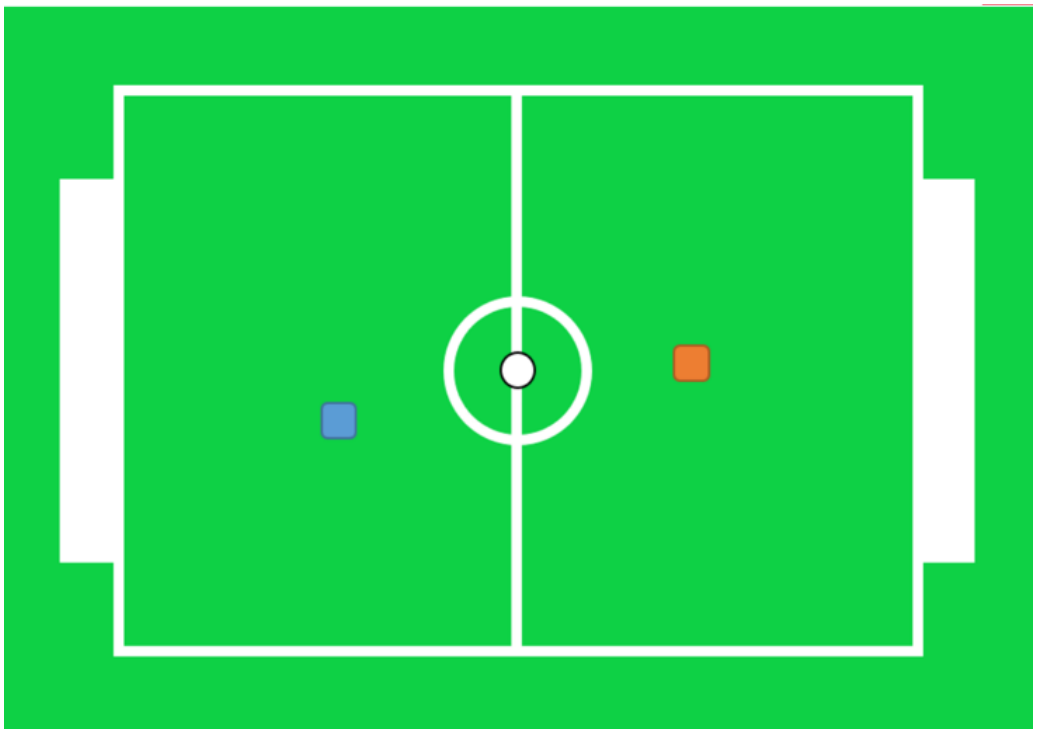

Figure 22: Game

4.2 AI Implementation

4.3 Q-Learning Agents

___init___ When we new a new Q-learning agent, we will initialize the Q-table, learning rate, discount factor and exploration strategy.

```python
class AgentsQT():
    def __init__(self, id):
        # create Q table
        # The sturcture of Q table:
        #   player's position (9, 9)
        #   opponent's relative position (7, 7)
        #   ball's relative position (7, 7)
        #   moving direction
        #   action 0->nothing 1->kick?
        self.id = id
        self.q_table = np.zeros((9, 9, 7, 7, 7, 7, 9, 2))
        # learning rate
        self.alpha = 1
        # discount factor
        self.gamma = 0.7
        # exploration strategy
        self.greedy = 0.9
```

Figure 23: ___init___()

Sensor The sensor of a agent should be able to get position information for all players and ball from game environment. However, the termporary implementation only consider the 1 vs 1 game mode.

```
def getGameState(pid, players, ball):
    ret_state = [0, 0, 0, 0, 0, 0]
    for p in players:
        if p.id == pid:
            ret_state[0] = p.rect.centerx
            ret_state[1] = p.rect.centery
        else:
            ret_state[2] = p.rect.centerx
            ret_state[3] = p.rect.centery
    ret_state[4] = ball.rect.centerx
    ret_state[5] = ball.rect.centery
    return ret_state
```

Figure 24: get_game_state()

States Processor We use get_state() method to simplify states. we fisrt divide the court into 9 * 9 pieces, to find current player in which pieces. Then, we calculate the distance of ball and opponent in each directions, using log10() to compress the information.

```python
def get_state(self, state):
    return_state = np.zeros((6,), dtype=int);
    player_x = state[0]
    player_y = state[1]
    opponent_x = state[2]
    opponent_y = state[3]
    ball_x = state[4]
    ball_y = state[5]

    interval_x = 1/12 * conf.width
    return_state[0] = (player_x - (0.125 * conf.width)) // interval_x
    interval_y = 1/12 * conf.height
    return_state[1] = (player_y - (0.125 * conf.height)) // interval_y

    diff_x = opponent_x - player_x
    diff_y = opponent_y - player_y
    if diff_x > 0:
        return_state[2] = math.ceil(math.log10(abs(diff_x) + 1)) + 3
    elif diff_x == 0:
        return_state[2] = 3
    else:
        return_state[2] = 3 - math.ceil(math.log10(abs(diff_x) + 1))
    if diff_y > 0:
        return_state[3] = math.ceil(math.log10(abs(diff_x) + 1)) + 3
    elif diff_y == 0:
        return_state[3] = 3
    else:
        return_state[3] = 3 - math.ceil(math.log10(abs(diff_x) + 1))

    diff_ball_x = ball_x - player_x
    diff_ball_y = ball_y - player_y
    if diff_ball_x > 0:
        return_state[4] = math.ceil(math.log10(abs(diff_x) + 1)) + 3
    elif diff_ball_x == 0:
        return_state[4] = 3
    else:
        return_state[4] = 3 - math.ceil(math.log10(abs(diff_x) + 1))
    if diff_ball_y > 0:
        return_state[5] = math.ceil(math.log10(abs(diff_x) + 1)) + 3
    elif diff_ball_y == 0:
        return_state[5] = 3
    else:
        return_state[5] = 3 - math.ceil(math.log10(abs(diff_x) + 1))
```

Figure 25: get_state()

Update Rule The update rule is the same as the equation mentioned in introduction. Learning rate and discount factor will be updated before the training section in every episodes.

```
def update_q_table(self, old_state, current_action, next_state, r):
    next_max_value = np.max(self.q_table[next_state[0], next_state[1], next_state[2],
        next_state[3], next_state[4], next_state[5]])
    self.q_table[old_state[0], old_state[1], old_state[2], next_state[3], next_state[4],
        next_state[5], current_action] = (1 - self.alpha) * self.q_table[
        old_state[0], old_state[1], old_state[2], next_state[3], next_state[4],
        next_state[5], current_action] + self.alpha * (r + next_max_value)
```

Figure 26: update_q_table()

Exploration Strategy The exploration strategy will be updated before the training section of every episodes. In the early phase, the agent will be more like to choose random works, but as the training goes, the agents will have higher and higher probability to choose the local best action.

```
if (random):
    if np.random.rand(1) < self.greedy:
        ret_act = np.random.choice(range(18))
    else:
        ret_act = act.index(max(act))
else:
    ret_act = act.index(max(act))

if ret_act ≥ 9:
    return [ret_act - 9, 1]
else:
    return [ret_act, 0]
```

Figure 27: exploration strategy

Actuator This method will receive a list with length of 2 and output a list whose size is 5. It will convert the moving direction to the button should be pressed.

```
def get_input_ai(pid, action):
    ret_array = [0, 0, 0, 0, 0]
    if action[0] == 1 or action[0] == 2 or action[0] == 8:
        ret_array[0] = 1
    if action[0] == 2 or action[0] == 3 or action[0] == 4:
        ret_array[3] = 1
    if action[0] == 4 or action[0] == 5 or action[0] == 6:
        ret_array[1] = 1
    if action[0] == 6 or action[0] == 7 or action[0] == 8:
        ret_array[2] = 1
    if action[1] == 1:
        ret_array[4] = 1
    return ret_array
```

Figure 28: get_input_ai()

4.4 Deep Q-Network Agents

The implementation of DQN agents are not finished yet, we need to normalize the environement to the standard can be used by tensorflow.

___init___ When we new a new Deep Q-network agent, we will initialize the memory space and other parameters for machine learning.

```python
class AgentsDQN():
    def __init__(self, action_set):
        self.gamma = 1
        self.model = self.init_netWork()
        self.batch_size = 128
        self.memory = deque(maxlen=2000000)
        self.greedy = 1
        self.action_set = action_set
```

Figure 29: ___init___()

Network Structure The network will have two or more layer(depend on the training result).

```python
def init_netWork(self):
    """
    构建模型
    :return:
    """
    model = models.Sequential([
        layers.Dense(64 * 4, activation="tanh", input_dim=self.observation_space.shape[0]),
        layers.Dense(18, activation="linear")
    ])

    model.compile(loss='mean_squared_error',
                  optimizer=optimizers.Adam(0.001))
    return model
```

Figure 30: network

Actuator Same as Q-learning agents.

This method will receive a list with length of 2 and output a list whose size is 5. It will convert the moving direction to the button should be pressed.

```
def get_input_ai(pid, action):
    ret_array = [0, 0, 0, 0, 0]
    if action[0] == 1 or action[0] == 2 or action[0] == 8:
        ret_array[0] = 1
    if action[0] == 2 or action[0] == 3 or action[0] == 4:
        ret_array[3] = 1
    if action[0] == 4 or action[0] == 5 or action[0] == 6:
        ret_array[1] = 1
    if action[0] == 6 or action[0] == 7 or action[0] == 8:
        ret_array[2] = 1
    if action[1] == 1:
        ret_array[4] = 1
    return ret_array
```

Figure 31: get_input_ai()

5 Current Conclusion

In this term?

Q-learning may not work well in our game, because the huge size of Q-table. if we want to store all the details in the table, the size of the table will be $(664 \times 465)^3 \times 8 \times 2 = 4.7095 \times 10^{17}$, which is incredible

huge. Even if we simplify the table, there still are 3,111,696 cells in the table.

The size of the table make the time and space complexity far beyond our expectation, but it is not the toughest problem. The main issue make this algorithm performing bad is the times of states reached by agents. Because of the scheme we use to simplify tables and the property of random walks, the agents seldom reach some of the states, which make the updating of the table slower.

6 Future Plan

6.1 Implementation of DQN

So far, we have implementaed Q-learning. However, the performance of Q-learning is not satisfactory. Therefore, we need to implement DQN, to improve the performance of agent.

6.2 Multiple Agents

Up to now, our agents are all trains on the 1v1 mode of the soccer game. In the future, we will also train different kinds of agents on the

2v2, 3v3, 5v5 and 7v7 modes, and compare what is different in strategy and their performance in different modes.

6.3 Standardization of Interface Between Game and Agent

The main part of game is implemented correctly. However, the interfaces between game and agent should be more standard, especially for the implementation of DQN.

References

[1] Silver, David, et al. "Mastering the game of Go with deep neural networks and tree search." nature 529.7587 (2016): 484-489.

[2] Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." arXiv preprint arXiv:1312.5602 (2013).

[3] Berner, Christopher, et al. "Dota 2 with large scale deep reinforcement learning." arXiv preprint arXiv:1912.06680 (2019).

[4] Zoph, Barret, and Quoc V. Le. "Neural architecture search with reinforcement learning." arXiv preprint arXiv:1611.01578 (2016).

[5] Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." nature 518.7540 (2015): 529-533.