Multiple AI Competition in Self Developed Game

Term 1 Report

ESTR4998/4999

November 26, 2020

Partners: XIAO Tianyi

LUO Lu

Instructor: Prof. Andrej Bogdanov

1 Abstract

To determine the atomic weight of magnesium via its reaction with oxygen and to study the stoichiometry of the reaction (as defined in 1.1):

1.1 Definitions

Stoichiometry The relationship between the relative quantities of substances taking part in a reaction or forming a compound, typically a ratio of whole integers.

Atomic mass The mass of an atom of a chemical element expressed in atomic mass units. It is approximately equivalent to the number of protons and neutrons in the atom (the mass number) or to the average number allowing for the relative abundances of different isotopes.

2 Background

2.1 Related Works

2.2 Tensorflow

2.3 Pygame

Mass of empty crucible	7.28 g
Mass of crucible and magnesium before heating	8.59 g
Mass of crucible and magnesium oxide after heating	9.46 g
Balance used	#4
Magnesium from sample bottle	#1

3 Introduction

3.1 Game Development Platform

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. Therefore, we choose Pygame as our game development platform.

4 Design

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

4.1 Game Design

4.1.1 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.

4.1.2 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.

4.1.3 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.

4.1.4 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.

4.2 AI Design

5 Implement

5.1 Game Implementation

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

5.1.1 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

```python
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

5.1.2 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

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.

```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

```
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

5.1.3 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.

```python
# 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.

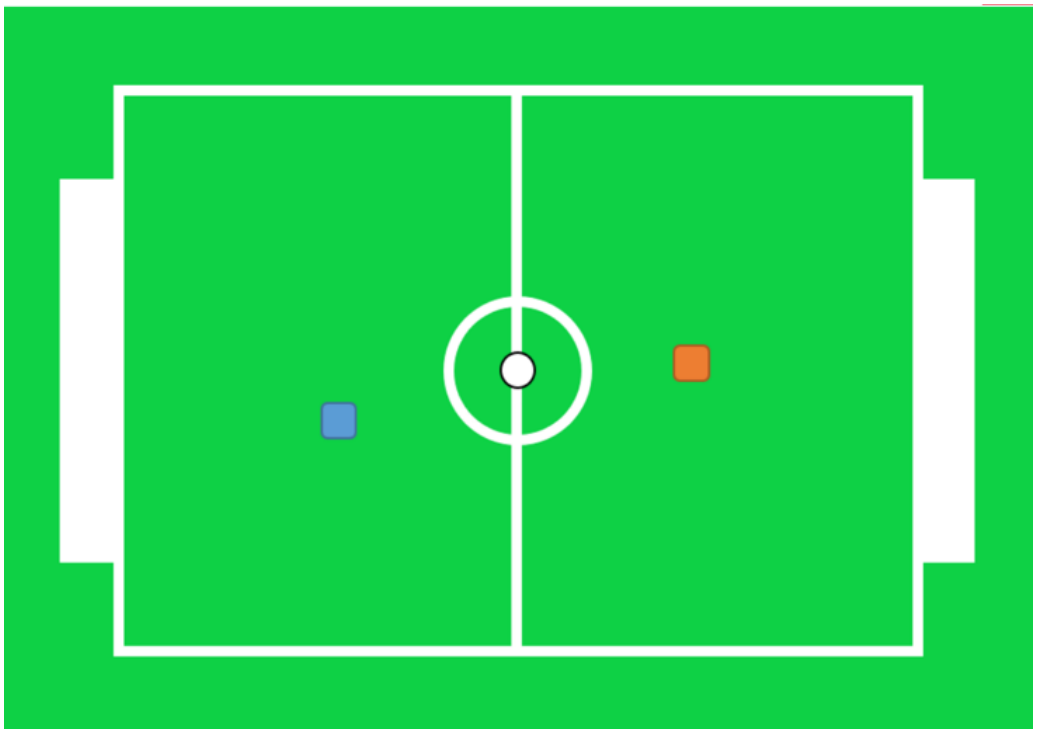

Figure 22: Game

5.2 AI Implementation

6 Current Conclusion

a. The *atomic weight of an element* is the relative weight of one of its atoms compared to C-12 with a weight of 12.0000000..., hydrogen with a weight of 1.008, to oxygen with a weight of

16.00. Atomic weight is also the average weight of all the atoms of that element as they occur in nature.

b. The *units of atomic weight* are two-fold, with an identical numerical value. They are g/mole of atoms (or just g/mol) or amu/atom.

c. *Percentage discrepancy* between an accepted (literature) value and an experimental value is

$$\frac{\text{experimental result} - \text{accepted result}}{\text{accepted result}}$$

7 Future Plan

placeholder

The most obvious source of experimental uncertainty is the limited precision of the balance. Other potential sources of experimental uncertainty are: the reaction might not be complete; if not enough time was allowed for total oxidation, less than complete oxidation of the magnesium might have, in part, reacted with nitrogen in the air (incorrect reaction); the magnesium oxide might have absorbed water from the air, and thus weigh "too much." Because the result obtained

is close to the accepted value it is possible that some of these experimental uncertainties have fortuitously cancelled one another.

References

Smith, J. M. and Jones, A. B. (2012). *Chemistry*. Publisher, 7th edition.