# 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

# 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.

## 3.2 Agents Design

### 3.2.1 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 ani-

mal 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 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

**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

goal states are terminal states,not the best or winning states.

### 3.2.2 $Q$-learning

$Q$-learning is one of the most basic and popular temporal difference learning method to estimate Q-value function(Zoph et al, 31). Temporal 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, $\alpha$ is the learning rate which should be gradually decreased as iterating, $\gamma$ 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 $\gamma$ will determine how much the expected future rewards affect the current estimates. The algorithm will focus more on short-term rewards if the $\gamma$ is set lower.

| $Q$-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

The advantage of $Q$-learning is that it's exploration-insensitive, which means we can get the optimal policy as long as the $\alpha$ 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.

**Algorithm 2** Reinforcement Learning (Zoph et al, 31)

**Require:** $\gamma$, $\alpha$

Initialize $Q$ (e.g. $Q(s,a) = 0 \ for \ \forall s \in S, \forall a \in A$)
**for** each episode **do**
    Modify $\alpha$ and $\gamma$
    $s \in S$ 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

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

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 - $\alpha$, 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.

### 3.2.3 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 algorithm 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 $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta'$
**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 $\theta$
        Every C steps reset $\hat{Q} = Q$
    **until** $s'$ is a goal state or $t$ reaches the limitation

# 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 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.

### 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

### 4.2.1 Objection

The objection of our project is 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 a artificial intelligence.

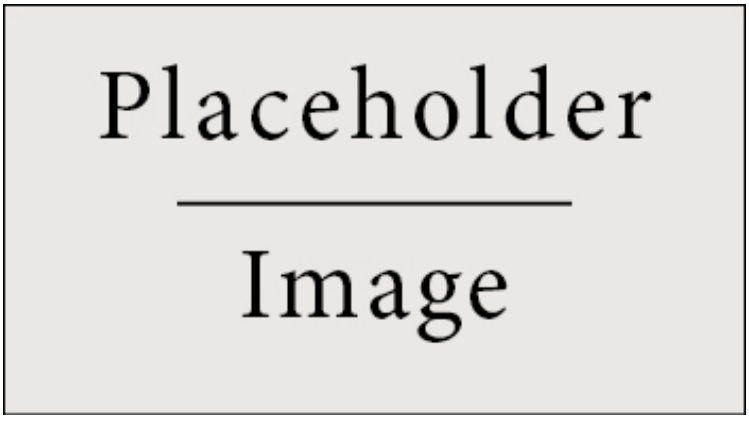


Figure 1: Figure caption.

# 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.

**___init___** In the ___init___ function, we define 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 2: implementation of ___init___

**inpur_handler**

**update**

**check_shoot_cd**

**shoot_update**

**render**

# 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

# References

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

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