2021/22

Student Name: Hang Yang

Student ID: 201522538

Project Title: COMP390
Competitive Game of Life

Supervisor: Louwe Kuijer

DEPARTMENT OF COMPUTER SCIENCE

The University of Liverpool, Liverpool L69 3BX

Dedicated to my dad and mom

Acknowledgements

I would like to thank my supervisor, Dr Louwe Kuijer, for all his help and advice. I would also like to thank my parents, without whose help this would not have been possible.

Finally, I would like to thank myself.

Competitive Game of Life

DEPARTMENT OF COMPUTER SCIENCE

The University of Liverpool, Liverpool L69 3BX

Abstract

In this project, an improved version of the game of life based on Conway's game of life is proposed, transforming a zero-player game into a two-player competition game. This game is implemented as a generic game environment for reinforcement learning.

The dissertation will also discuss the basic principles of reinforcement learning and methods such as experience replay, target network and other methods to increase the robustness of DQN. The final DQN agent has a win rate of approximately 85%. In addition, a method to automatically generate the background music of the game is proposed and implemented.

Finally, it should be mentioned that a model of life simulation was proposed in this dissertation, which failed in the end.

Contents

Chapter 1

Introduction

1.1 Overview

Conway's Game of Life is a celebrated two-dimensional cellular automaton proposed by Conway[1]. As it represents the birth, survival and death of individual cells on a 2D surface based on simple rules, it is the content related to Complex system and Emergence and it is also Turing-complete[2].

In this project, I will develop a two-player competitive version of enhanced Conway's Game of Life which I call Competitive Game of Life (CGoL), where the new rule set gives each cell the probability of birth and survival, and both birth and survival are competitive. Each player can place colored tokens (cells) to the grid at intervals in each generation. Over time, The winner is the player who kills all life of the competitor on the 2D space.

Meanwhile, a reinforcement learning algorithm(Deep Q Network agent) will be applied to the environment of CGoL for human player to compete with.

1.2 Aims & Objectives

- Propose a new model of life simulation, based on an in-depth study of cellular automata and artificial life(soft).
- Implement CGoL as a non-graphic game environment.
- Implement DQN algorithm to the game environment.
- Implement a UI environment of CGoL for the player to interact with the AI.

Chapter 2

Background & Motivation

I separated the project progress into 3 phases to better present my work on this project. in this section, I will introduce my research topic and motivation behind it, as well as the background of the study respectively.

2.1 Stage 1: Life simulation & Game of life

John Conway introduced Game of life at 1970[1].The game attracted a large number of enthusiasts to study the pattern of the game. In 2002, P. Rendell proved that Conway's game of life is Turing-complete[2], The same year, Stephen Wolfram published *A New Kind of Science*, which gives empirical and systematic study of cellular automaton and he also indicate that computation modelling method based upon algorithms that iterate on simple rules to simulate complex phenomenon is relevant to other fields of science[3].

I also looked emergence relevant agent-based model (ABM) like Boids[4], Schelling's model of segregation[5], and Sugarscape[6].These are evidences that simple programs are enough to capture the essence of almost any complex system[3].

The above background makes me very interested in life simulation, and I plan to focus on a certain aspect of this topic for in-depth research. At the beginning of the project, I spent a lot of time in the research of emergence and cellular automata. At this stage, my goal is to put forward a new model to imitate the birth, survival and competition of cell, and this model should have a more comprehensive description of life individual, rather than simply two states (death or survive) on one grid, and the new model needs to be capable of complex evolution. So I mainly put forward an idea for life stimulation which will be mentioned briefly in section3.2.1.

Notice that the subject of Artificial life and cellular automata share a closely tied history. Cellular automata were used in the early research of artificial life, and it is still used today for its advantages of scalability and parallelizatio[7]. Based on *Artificial life: organization, adaptation and complexity from the bottom up*[8], there are three basic types of artificial life: soft, which comes from software; hard, which comes from hardware; and wet, which comes from biochemistry. And the game ruleset I'm focusing in this project is in the category soft.

However, with the limitation of my scientific research ability and the difficulty of in-depth research, I realized that I could not come up with an achievable model in my current state. Therefore, I shifted my focus to the Conway Game of Life and refined the game so that its emergence became more complex.

There are 2^8 rules for Elementary Cellular Automaton (one-dimension), as mentioned before, Wolfram have researched them and gives detailed illustrations on his book A New Kind of Science[3]. For two-dimensional Cellular Automaton, there are 2^{18} possible Life-like rules. Lots of rules show exciting features which have been documented by Mirek Wojtowicz[9]. However, The stochastic model has never been studied or discussed. Although such model is difficult to be reproduced in experiments, it still has experimental significance. Therefore, at this stage, I put forward a 2 player's game of life stochastic model, where a threshold is used in the new ruleset to determine birth and survival, and increases the complexity and playability of the game. Also, the new model (ruleset) will be covered in section3.2.2.

2.2 Stage 2: Reinforcement Learning

The second stage, where I put in the most work. Because the setting of a two-player board game has been established, an AI player algorithm is necessary to interact with the human player.

In this environment of competitive game of life, the greedy algorithm can get good scores, which will be evaluated in the implementation section and compared with the reinforcement learning algorithm. However, in order to challenge myself and lay a solid foundation for Artificial Life, agent-based model (ABM) and other related fields, so as to prepare for further study, I decided to use reinforcement learning algorithm to develop AI for this project.

An article "Emergent Tool Use from Multi-Agent Interaction"[10] by OpenAI introduce a game that multi-agent playing a simple game of hide-and-seek. By training in the simulated environment for 43.4 million episodes, the agents learned some special strategies, some of which have not yet been discovered by humans. This kind of unsupervised emergent complexity shows that multi-agent co-adaptation could yield very complex and intelligent behaviour in the future.

Along with such ideas, competition and even cooperation between multiple intelligences is possible through self-supervised learning to find game strategies with an advantage. For example, competition and cooperation can occur in a game of life environment where two players pk two players. This example shows the broad scalability of this project, but this project will still focus on the competitive relationship between the two players.

Deep Reinforcement Learning in Strategic Board Game Environments [11] explains how to apply reinforcement learning (RL) to strategic board games and provides practical tips for implementation. For two-player zero-sum games, Littman[12] proposed a convergent Q-Learning method. further more, The paper *Human-level control using deep reinforcement learning*[13] introduces a new artificial agent called a deep Q network (DQN) that can learn substantial policies directly from high-dimensional inputs using deep reinforcement learning.

Through these articles of reinforcement learning and the understanding of ABM in the previous study, deep Q network (DQN) has become the practical direction of AI Player development. The project will be focused on this stage, which will include a complete mathematical model and concepts in section 3.3, The key code snippet and three key features holds in the algorithm are described in section 4.

2.3 Stage 3: UI & Sound of Life

This stage is mainly about the deployment of the project. I have designed the UI of Competitive Game of life(CGoL), and some of the design concepts and illustrations will be shown in section3.4 . Secondly, there is a highlight in the UI design, which is The sound of life, the design concept and the specific technology used in it will be shown in section3.4.2.

Chapter 3

Design

Basically, This section will be divided into four parts: 3.1 will give an overview of the system structure. 3.2 will introduce the ruleset and data structure of Competitive Game of life, and 3.3 will show the basic concept of RL as well as algorithms, finally, 3.4 will demonstrate the design of the user interface, the sounds of life, and the flow of navigation.

The development of CGoL, that is to translate the rule set into an actual computer program similar to playgameoflife[14].

3.1 Structure

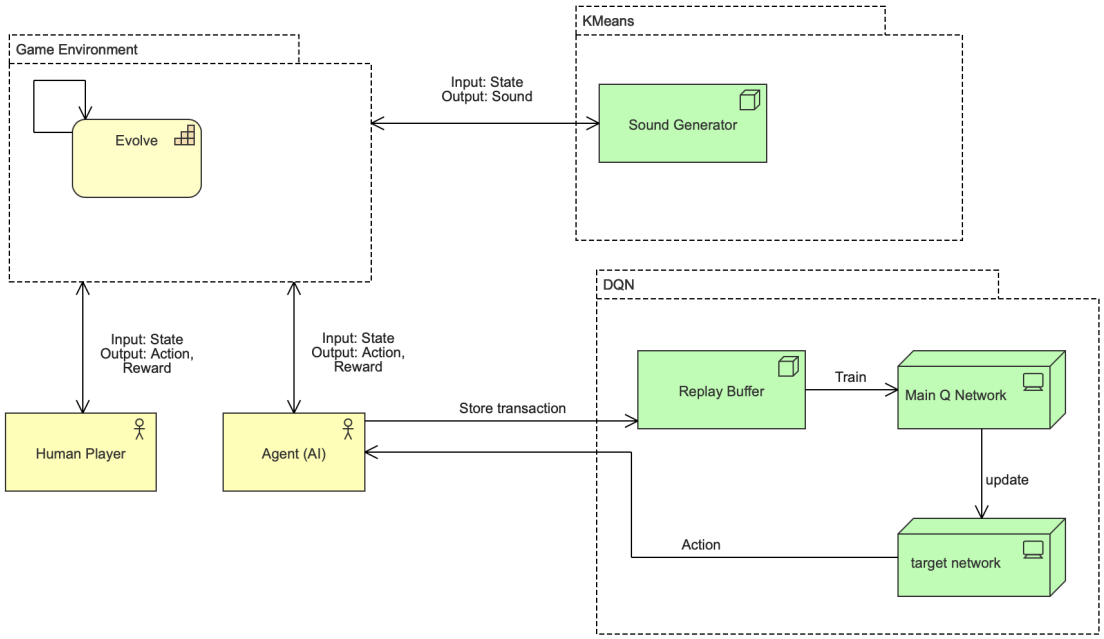

Figure 3.1: The structure of the system

The structure of this project is similar to the three stages mentioned above, with three main components:

- The Competive Game of Life(CGoL), which is a game environment using cellular automata.
- The deep reinforcement learning agent, which is the AI that fights against the human player
- The UI and the game background generated by game state.

3.2 Game design

Although Stage one of the project was full of frustrations and failures, there are certain aspects of the outcome that I think are worth mentioning in this section.

3.2.1 Life simulation

I mainly proposed an idea for life simulation: Life of Pakua, which is failed in the end. However, the idea is still remarkable for the course of the whole project, as it shows my critical ability and creativity. More of self-reflection will be in Section7.

As the name suggests, this idea was inspired by Pakua. Pakua is a set of eight symbols used in Taoist cosmology to symbolise the essential principles of and reality existence, considered as a collection of eight interconnected notions. Each consists of three lines, each of which is either "broken" or "unbroken," with 0 or 1 generating binary numerals 000 to 111, which are comparable to octal digits 0 to 7[15].

The eight trigrams is showed as following:

八卦 Bāguà—The eight trigrams (Top bar is most significant binary digit)

	乾 Qián	巽 Xùn	離 Lí	艮 Gèn	兌 Duì	坎 Kǎn	震 Zhèn	坤 Kūn
Trigram lines	☰ 111	☴ 110	☲ 101	☶ 100	☱ 011	☵ 010	☳ 001	☷ 000
Nature	Heaven/Firmament 天 Tiān	Wind 風 Fēng	Fire 火 Huǒ	Mountain 山 Shān	Lake/Marsh 澤 Zé	Water 水 Shuǐ	Thunder 雷 Léi	Ground 地 Dì

Figure 3.2: Eight trigrams of Pakua

It is a tool used in Chinese culture to deduce the relationship between various things in the world space time. Each trigram represents a certain thing. The eight trigrams are combined with each other to form the 64 trigrams, which are used to symbolise various natural and human phenomena.

So the idea is using each trigram as a unit, a set of a fixed number of trigrams is used to represent the game environment, in which a genetic algorithm-like process is carried out according to ruleset(eg. Crossover, Mutation), in which trigrams can combine by a fixed method during each generation of evolution and can also compete by a fixed method (e.g. dismantling a competitor's trigram or absorbing a competitor's trigram) and, eventually, the set may emerge as a final organism. Alternatively, a large set may contain multiple sets of trigram.

This idea can be implemented, and if trigram is large enough in the set, it can produce the final creature and dominate most of the creatures. However, The problems is, unlike genetic algorithm, which the problems is pre-defined, and we are able to map out a solution to a real problem through the Chromosome, which gives the abstract chromosome very intuitive properties, the final life of Pakua might be too abstract and difficult to visualize itself.

The second problem is, like the conway game of life, this final life can have some of the properties of life, which is defined by some previously given rules, and these emergent rules can trigger quantitative changes in the environment , but not enough to have properties that can trigger qualitative changes in a way that DNA and proteins trigger biological life.

3.2.2 Rule set of CGoL

First of all, I will briefly show the idea of GoL, then I will introduce the rule set of competitive version of GoL(CGoL). Noted that if there is only one player on CGoL, then the game reduces to the original Game of Life(GoL).

Based on Golly/RLE format, CGoL can be represented as

$$B2(\frac{1}{2})3/S1(\frac{1}{2})234(\frac{1}{2})$$

where x($\frac{1}{2}$) means there is $\frac{1}{2}$ probability to occur for x. I will give an interesting example of CGoL:

Instead of Conway's game of life, after three generations of evolution, an individual can evolve in dozens of ways, which shows the randomness of life evolution. As showed in Fig2, a life has potential to be prosperous, but it also has potential to die.

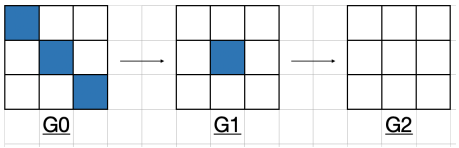

Figure 3.3: Conway's game of life

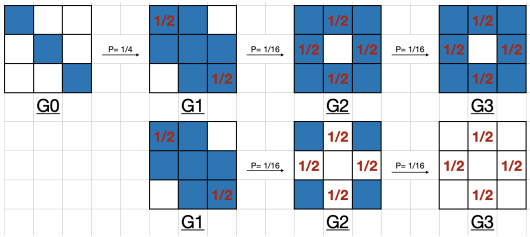

Figure 3.4: CGoL

Considering the competitive version of CGoL, the rules are symmetric for two players(Red and Blue), the rules here are from red's point of view:

- Brith (a cell is empty)
  1. if the cell has 3 red neighbours:
     (a) if the number of blue neighbours is not 3 or 2, then a red cell is born.
     (b) if the number of blue neighbours is 3, then there is $\frac{1}{2}$ probability that a red cell is born.
     (c) If the number of blue neighbours is 2, then there are $\frac{3}{4}$ probability that a red cell is born and $\frac{1}{4}$ probability that a blue cell is born.
  2. if the cell has 2 red neighbours:
     (a) if the number of blue neighbours is not 2, then a red cell is born.
     (b) if the number of blue neighbours is 2, then there are $\frac{3}{8}$ probability that a red cell is born and $\frac{3}{8}$ probability that a blue cell is born.
- Survival. (a cell is occupied by a red token)
  1. if the number of red neighbours is greater than blue neighbours and the difference is 1 or 4, then there is $\frac{1}{2}$ probability that the red survives.
  2. if the number of red neighbours is greater than blue neighbours and the difference is 2 or 3, then the red survives.
  3. if the number of red neighbours is equal to blue neighbours, then there is $\frac{1}{2}$ probability that the red survives.

Intuitively, CGoL has two improvements over Conway's Game of Life:

- Probability is used in the new rule set to determine birth and survival, which increases the complexity and playability of the game.
- Both birth and survival are competitive, and unlike the existing competition model that players can only decide the initial state of the game, however, CGoL has dynamic interaction with player, that is, in the interval of each generation, player can observe the survival state of their cells and change their strategies accordingly.

3.3 Reinforcement Learning

Reinforcement learning is a type of machine learning approach in which agents use trial and error to discover the optimal policy. RL may be effectively used for sequential decision-making problems by interacting with the environment[16].

Reinforcement learning algorithms can be categorised as off-policy and on-policy approaches. In off-policy RL algorithms, the behaviour policy used for choosing actions is different from the learning policy. In on-policy RL algorithms, however, behaviour policy is the same as learning policy. In addition, reinforcement learning can also be categorised as value-based and policy-based approaches. Agents in value-based RL algorithms update the value function to learn an appropriate policy, whereas agents in policy-based RL algorithms learn the policy directly[17]. In this project, Deep Q-learning(DQN) algorithm which is a typical off-policy value-based method will be considered.

In this section, The terms used in Reinforcement Learning will be introduced first, then MDP, which provides a mathematical framework for solving the RL problem, Q-function and Bellman equation will be also discussed, and finally, a general DQN algorithm will be presented and explained.

3.3.1 Terminology

- **Agent** : an AI that make decisions and also it is the learner in RL. Agents interact with the environment by taking actions and receive rewards based on their actions in each time-step.
- **Environment** : It is the demonstration of Game of Life. It is simulated environment with which the agent will interact .
- **State** : This is a condition at a specific time-step in the environment. Whenever an agent takes an action, the environment gives the agent rewards and a new state where the agent reached by taking the action.
- **Rewards** : A numerical values that the agent receives on performing some action at some state in the environment. The rewards can be positive or negative based on the actions of the agent.
- **Discount Factor** : The discount factor γ, is a real value $\in [0, 1]$, relates the rewards to time. It determines how much importance is to be given to the immediate reward and future rewards. This basically helps us to avoid infinity as a reward in continuous tasks.
- **Returns** : agent is looking for maximizing the cumulative rewards instead of the reward agent receives from the current state (also called immediate reward). This cumulative rewards is called returns. So returns can be defined by using discount factor as follows :

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1},$$

3.3.2 MDP

Markov-Decision Process (MDP) formulate Competition Game of Life in Reinforcement Learning mathematically. Actually, MDP allows modeling virtually any complex environment by assuming the environment strictly holds the Markov property. The Markov property means that evolution of the Markov process in the future only depends on the present state and does not depend on the past state. The Markov process does not remember the past if the present state is given, mathematically expression is given as following, S means sets of States:

$$P[S_{t+1} \mid S_t] = P[S_{t+1} \mid S_1, \ldots, S_t]$$

MDP is a 4 elements tuple (S, A, P_a, S_a), where

- S is a set of states of the environment, it is called the state space.
- A is a set of actions of the environment, it is called the action space.
- $P_a(s, s') = Pr(s_{t+1} = s' | s_t = s, a_t = a)$ is the probability that taking an action a at time t leads to a transition from s to s'. It is also called transition function.
- $R_a(s, s')$ is the immediate reward received from taking an action a that leads to a transition from s to s'. It is also called reward function.

Then a policy function π is a (potentially probabilistic) mapping from state space S to action space A.

There is a trick I used for the MDP needs to be mentioned here, I am not putting the evolutionary steps of the CGoL in the MDP. i.e. the state after taking an action is deterministic (non-probabilistic) and then the environment (CGoL) will evolve itself and produces the next state and rewards. Assume

the next state of s after evolution of CGoL is $E(s)$, then the transition function in this case will be $P_a(s, E(s'))$, and the reward function in this case will be $R_a(s, E(s'))$.

The implementation of MDP will be showed in 4.2.2.

3.3.3 Bellman equation

The Q-function, known as the action-value function, defines the value of taking action a in state s under a policy π. Based on [18], an agent using Q-learning in a finite MDP repeatedly observes a state s, chooses a legal action a, and then observes an immediate reward r and a transition to a new state s'. To be more specific, The Q Value of a state can be decomposed into the immediate reward agent receive for a transition from a state to another state, plus discounted Q value of the future state(s) based on the policy. Q-function is denoted by: $Q^\pi = (s, a)$:

$$Q^\pi(s,a) = E_\pi\{R_t|s_t = s, a_t = a\} = E_\pi\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s, a_t = a\}$$

Q-learning is guaranteed to converge to the optimum value function Q^*, if an adequate approach for exploring state-action pairings is applied, and its associated greedy policy is therefore an optimum policy. π^*[18].

Based on the idea above, we will have Bellman equation for the Action-value function. This equation simplifies the computation of the value function such that, rather of summing across numerous time steps, we may discover the best solution to a complicated issue by breaking it down into simpler, recursive subproblems and finding their optimal solutions[19]. Mathematical expression:

$$q_\pi(s,a) = \mathbb{E}_\pi\left[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a\right]$$

Following that, an optimum policy is determined as: $\pi(s) = argmax Q^*(s, a)$, where:

$$Q^*(s,a) = R(s,a) + \gamma \max_{a'} Q^*(s', a')$$

That is to say, once the optimal Q-function is determined, the optimal policy may be simply determined by selecting the action a that maximises $Q(s, a)$ given state s.

However, since CGoL's state space is 3^81 and CGoL's action space is $9 * 9$, a Q table to hold all Q values becomes no longer possible. In this case, Deep neural networks(DNN) is used for function approximation[13], the method is also called Deep Q Learning(DQN) which will be introduced in next section.

3.3.4 Deep Q Learning (DQN)

Deep Q-Learning (DQN) [13] is a variation of the classic Q-Learning algorithm that holds three key features: (1) a deep (convolutional) neural net architecture for Q-function approximation; (2) using mini-batches of random previous training data instead of real-time experience; and (3) using older network parameters to estimate the Q-values of the next state.[20]

These three key features will be implemented in the project. It will be further discussed in the DQN implementation section: 4.3.1 DNN Structure, 4.3.3 Experience Replay and 4.3.4 Target Network. Here, Pseudocode for DQN will be given based on Mnih[13] and Melrose,etc.[20]

Algorithm 1 Deep Q-learning with experience replay

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for episode 1, M **do** Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in the emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store experience $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of experiences $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the weights θ
    Every C steps reset $\hat{Q} = Q$
  end for
end for

3.4 UI and The sound of life

In this part, section3.4.1 will show some screen mockups and briefly introduce the flow of navigation, then section3.4.2 will introduce the design of the background music, the sound of life.

3.4.1 UI design

Since the focus of this project is on how to implement an AI Player, the purpose of the UI is only to visualize the competition between the two players. However, some of the philosophy of design has also been considered.

Monument Valley is an indie puzzle game developed and published by Ustwo Games, known for its minimalism and depth of meaning. Inspired by Monument Valley[21], I wanted a game that would be very simplified, but with It is very rich in meaning and will provoke the player to think deeply about the origin and evolution of life.

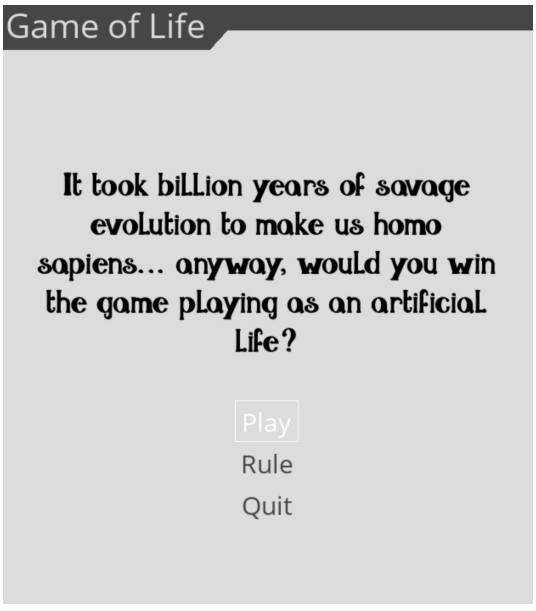

Figure 3.5: Index Page

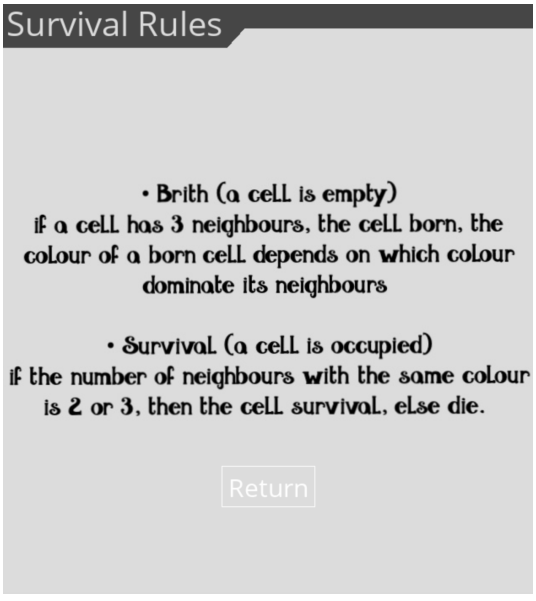

Figure 3.6: Rule Page

Basically, after player enters the game, there is a menu and the player can choose to start the game or view the game rules.

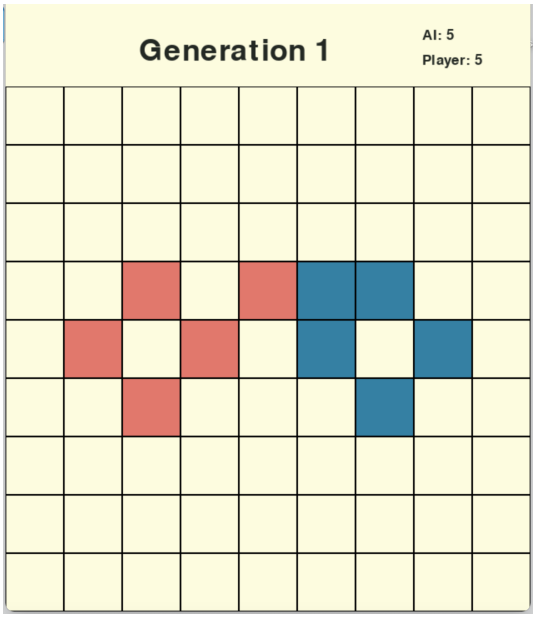

Figure 3.7: Playing0

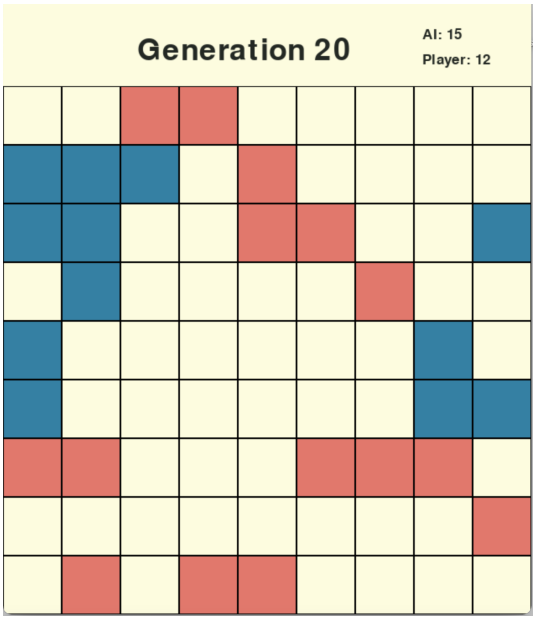

Figure 3.8: Playing1

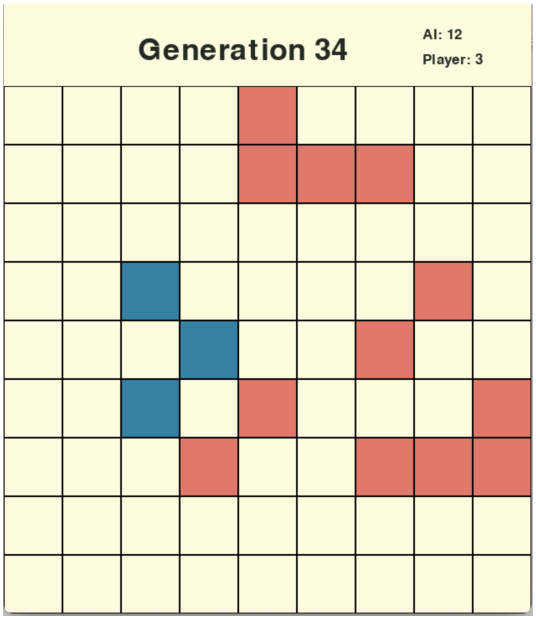

Figure 3.9: Playing2

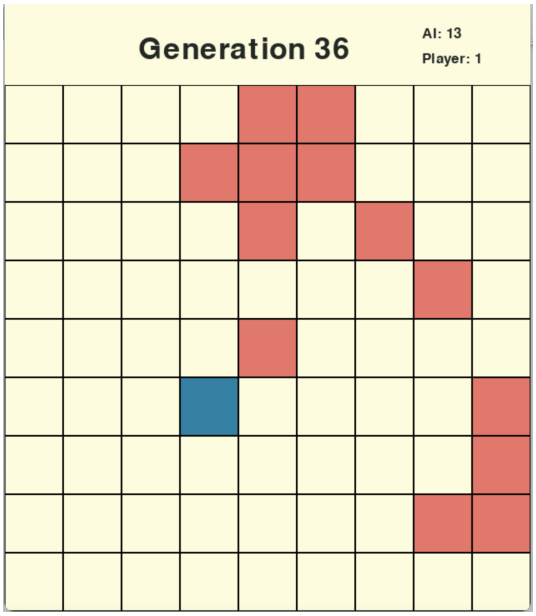

Figure 3.10: Playing3

The four figures above show the interface where the player competes with the AI. The interface shows the generation of the game, and the number of surviving cells of the player and AI in the current generation.

Figure 3.11: AI win Page

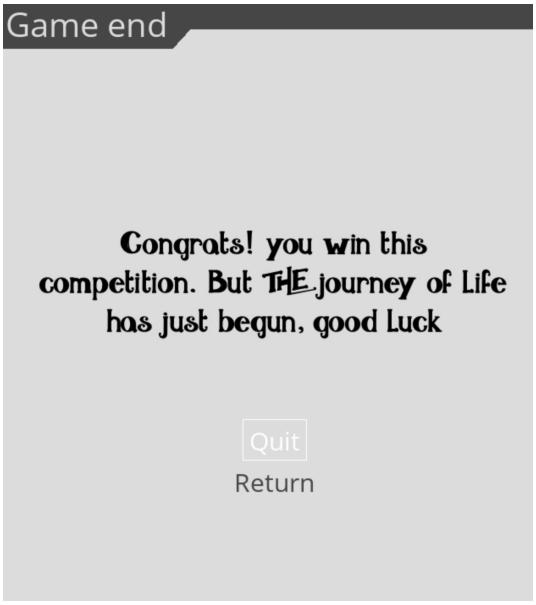

Figure 3.12: Player win Page

At the end of the game comes the summary of the game, which reflects on the evolution of life.

3.4.2 The sound of life

By studying each state of the game environment, we can intuitively know that once a player in a state has certain patterns, they have a higher win rate. Following this idea, we can associate a certain number of piano keys to each state, correlate states with sounds according to musical theory and psychological knowledge, and obtain a musical model of each competitive state, that is, mapping each state to some automatically generated musical melody.

When designing sound of life, the simplest idea was to define the intensity of the game by calculating the difference in the number of surviving cells between the two players, and to create a level of musical tension by using different spans of piano keys.

Another idea is inspired by supervised learning. By using experience replay to tag each state with a win rate, and using a DNN model to learn the win rate of each state so that the model can converge to the essential win rate of the state, and the DNN model can be used to predict the win rate of each state, then we can use the win rate to correlate the automatically generated music.

Chapter 4

Implementation

In this part, section4.1 will give an overview of the relationship between each file. Base on Fig4.6, section4.2 will discuss the implementation of game environment, section4.3 will cover all the DQN part, and the implementation of UI will be mentioned in section4.4. Note that many dead ends that didn't work will be mentioned in this section.

4.1 Structure overview

The game environment runs throughout the project. The keras_dqn.py file defines DQN agent. After instantiating DQN Agent, train DQN through model_train.py and save the model. Note that the training of this project can be divided into stages. This means that the replay.py file can reload the model and continue training. Finally, evaluate the model by using model_eval.py. Notice that a greedy player or random policy player is used to interact with the DQN Agent.

The other part is the UI. ui.py uses PyGame to visualize the game environment. In simple terms, ui.py converts an array of each state into a graphical interface. Pygame. py uses package pygame [22] to build the framework of the game menu and the logic of the flow navigations.

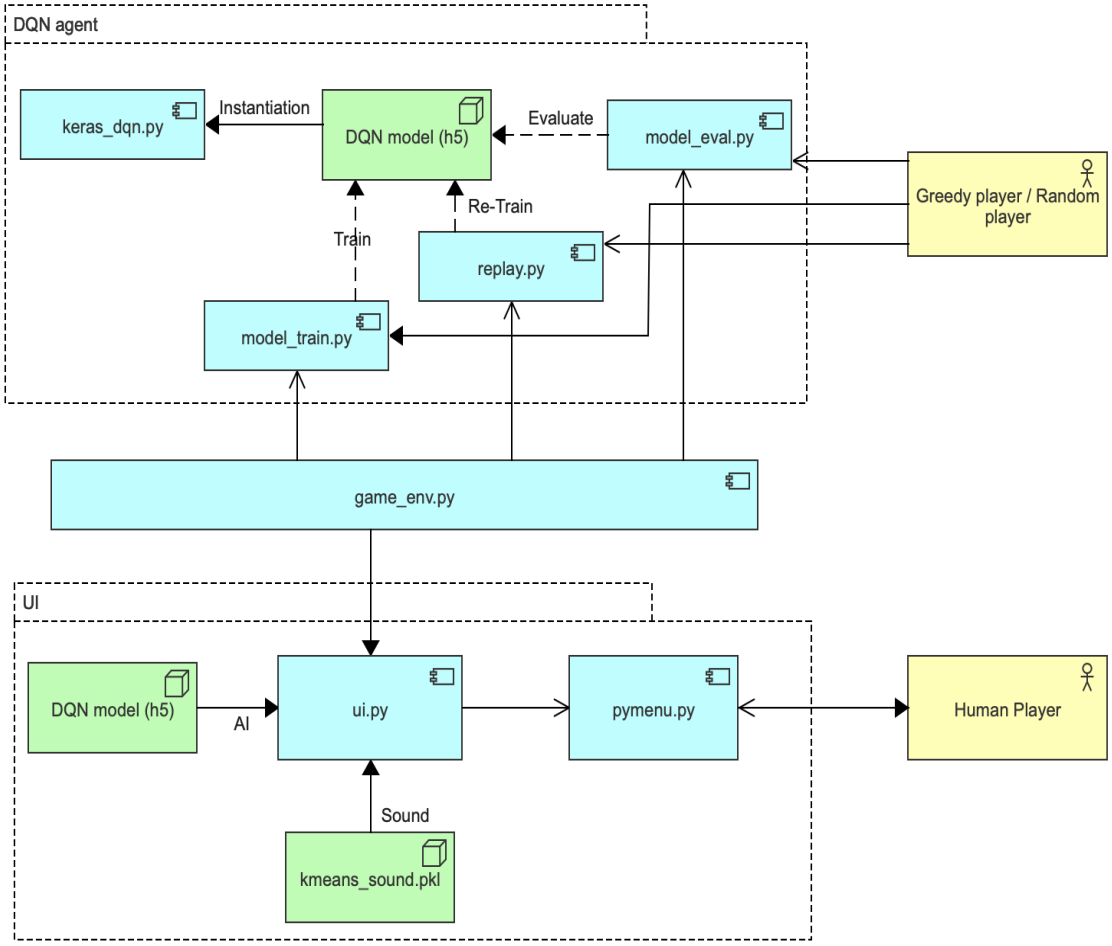

Figure 4.1: File Structure

4.2 Game environment

In this part, I will introduce the implementation of the game environment. Specifically, section4.2.2 will provide the code snippet of MDP implementation and the evolving ruleset of Competitive game of life(CGoL), and section4.2.4 will present some other useful methods provided to the DQN Agent.

4.2.1 Data Structure

Each state is stored in a 2-d array:

```
conway = State(initial_board())
print(conway.board)
```

```
[[0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 1 0 0 0 2 0 0]
 [0 1 0 1 0 2 0 2 0]
 [0 0 1 0 0 0 2 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]]
```

Figure 4.2: data structure of a state

Then I will use step() function to take an action:

```
print(conway.step(24))
print(conway.board)
```

```
(array([[0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 0, 0, 2, 2, 2, 0],
       [0, 1, 0, 1, 0, 2, 0, 2, 0],
       [0, 0, 1, 0, 0, 0, 2, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0]]), 4, False)
[[0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 1 0 0 2 2 2 0]
 [0 1 0 1 0 2 0 2 0]
 [0 0 1 0 0 0 2 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0]]
(base) niceanyh@pc-12-139 dqn_keras_v2 %
```

Figure 4.3: an state after taking an action

The above steps are also used for unit testing.

4.2.2 Evolve function

The rules of the original conway game of life can be represented as:

$$B3/S23$$

And CGoL, the design of section 3.2.2 can be represented as:

$$B2(\frac{1}{2})3/S1(\frac{1}{2})234(\frac{1}{2})$$

.

The early implementation of CGoL was based on the rules of CGoL:

```
    if pause == False:
        for x in range(self.rows):
            for y in range(self.columns):
                state = self.grid_array[x][y]
                neighbours = self.get_neighbours(x, y)

                if state == 0: #empty cell -> born
                    if neighbours == 3:
                        next[x][y] = 1
                    #the threshold increase, the less cell born --0.5
```

```
                    elif neighbours == 2 and np.random.rand() > 0.5:
                        next[x][y] = 1
                    else: #keep the same state
                        next[x][y] = state
                elif state == 1: #living cell -> die
                    if neighbours > 4 or neighbours == 0:
                        next[x][y] = 0
                    #when threshold increase,the less cell die --0.5
                    elif (neighbours == 1 or neighbours == 4 ) and np.random.rand() >
                        0.5:
                        next[x][y] = 0
                    else: #keep the same state
                        next[x][y] = state
        self.grid_array = next
```

However, after the visualisation was carried out(Fig4.4), some problems emerged: the possibilities of both birth and survival have been increased, and this rule leads to a crowded and confusing picture of life, making the experience of playing the game less enjoyable.

Figure 4.4: The visualisation of rule with possibility 1

When the threshold is changed to 0.01, that is:

$$B2(0.01)3/S1(0.01)234(0.01)$$

The game becomes closer to the original Conway Life game:

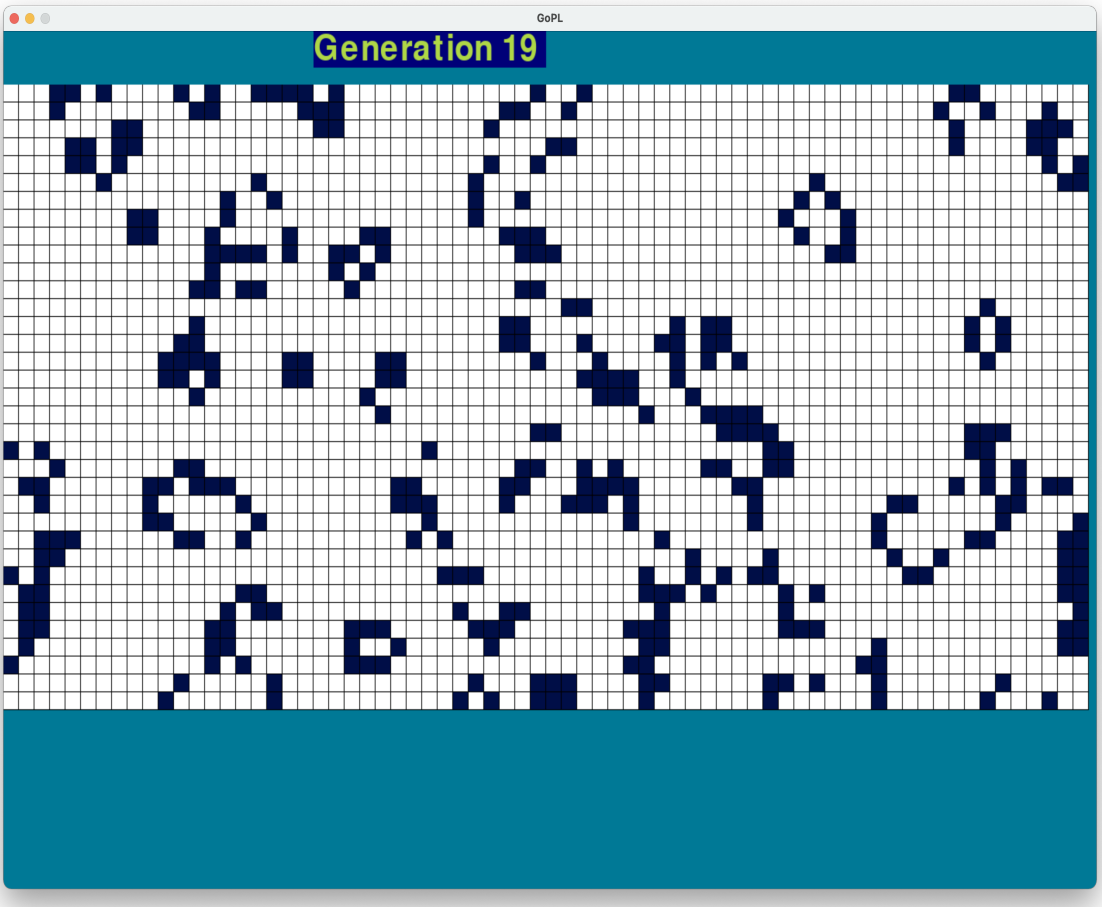

Figure 4.5: The visualisation of rule with possibility 2

Considering that these rules were used to design a game, some changes have been made as follows:

- Remove the idea of probability from the rules of the game.
- Use the difference in the number of cells to determine the status of a cell, with the cell belonging to which player depending on which player dominates that neighborhood.

The code snippet shown below is used in the current version of CGoL:

```
    def evolve(self):
        next = np.zeros(shape=(9,9),dtype="int")
        for x in range(len(self.board[0])):
                for y in range(len(self.board)):
                    cell_state = self.board[x][y]
                    p1_neighbours = self.get_neighbours( x, y)[0]
                    p2_neighbours = self.get_neighbours( x, y)[1]
                    neighbours = p1_neighbours + p2_neighbours

                    if cell_state == 0: #empty cell -> born
                        if neighbours == 3:
                            if p1_neighbours > p2_neighbours:
                                next[x][y] = 1
                            else: next[x][y] = 2

                    elif cell_state == 1: #living cell -> die
                        if p1_neighbours >= 4 or p1_neighbours <= 1:
                            next[x][y] = 0
                        else: #keep the same state
```

```
                            next[x][y] = 1
                    elif cell_state == 2:
                        if p2_neighbours >= 4 or p2_neighbours <= 1:
                            next[x][y] = 0
                        else: #keep the same state
                            next[x][y] = 2
        return next
```

4.2.3 Step function

The Game of Life Environment have a step(play1_action, player2_action) − > next_state method that applies actions of both two players to the environment, and returns the following information about the next state:

- **observation**: This is another environment state that the players can observe to choose its actions at the next step.
- **reward**: The agent is learning to maximize the sum of these rewards across multiple steps.
- **done**: This is a boolean value that represent if the game is done.

Step function is the implementation of MDP and it is the core API provided to DQN Agent, code snippet is provided below:

```
    def step(self,action_id_p1,action_id_p2= None,player=1,singlePlayer=True):
            action1 = get_action(action_id_p1)
            #todo check if an action is valid
            # if(self.board[action[0]][action[1]]!=0):
            self.board[action1[0]][action1[1]]=player
            if(singlePlayer==False):
                action2 = get_action(action_id_p2)
                self.board[action2[0]][action2[1]]=2

            observation_ = self.evolve() # conway game evolve to next gen
            self.board = observation_
            done = self.is_end()
            # if done: reward = living_reward(observation_,player)
            # else: reward=0
            reward = living_reward(observation_,player)
            if done:
                if reward == 0: reward = -100
                else: reward*=10
            return observation_,reward,done
```

4.2.4 Other functions

- get_action_id()

  The first thing to mention is that an action in the environment is intuitively understood as a tuple A(x, y), representing the placement of a tocken in row x, column y. For convenience, each action is given an id:

```
    def get_action_id(action):
        id  = action[0]*9+action[1]
        return id

    def get_action(id):
        action = np.array([id//9,id%9])
        return action
```

- legal_actions()

  This function is used to get legal actions that can be placed in a state.

```
    def legal_actions(self): #return a list of legal actions id
        legal_actions=[]
        for x in range(len(self.board[0])):
                for y in range(len(self.board)):
                    if self.board[x][y] == 0 :
                        action = 10*x+y
                        legal_actions.append(action)
        return legal_actions
```

- living_reward()

  This function is used to get the number of survival cells belong to a player in the current state.

```
    def living_reward(board,player):
        counter=0
        for x in range(len(board[0])):
                for y in range(len(board)):
                    cell_state = board[x][y]
                    if cell_state == player:
                        counter+=1
        return counter
```

- is_end() This function is used to check if the game come to end.

```
    def is_end(self):
        num_player1 = living_reward(self.board,1)
        num_player2 = living_reward(self.board,2)
        if (num_player1 != 0 and num_player2 != 0):
            return False
        else: return True
```

4.3 Deep Reinforcement Learning

In this section I will show some details of the implementation of the DQL Agent, including the neural network structure in 4.3.1, rules for exploration and exploitation in 4.3.2, experience replay in 4.3.3, target network in 4.3.4 and finally, in 4.3.5, I will show how I have save the DQN model so that it can be trained in different stages with different parameters and applied to the game environment (pygame).

4.3.1 DNN Structure

```
    def _build_model(self):
        # Neural Net for DQN Model
        model = Sequential()
        model.add(Dense(300, input_dim=9, activation='relu'))
        model.add(Dense(300, activation='relu'))
        model.add(Dense(self.action_size, activation='linear'))
        model.compile(loss='mse',
                      optimizer=Adam(lr=self.learning_rate))
        return model
```

4.3.2 Exploration and Exploitation

Exploration and Exploitation is a important dilemma in RL. In this section firstly I will introduce the basic idea of exploration and exploitation and then I will focus on how my algorithm solve the dilemma.

- Exploration is more of a long-term benefit idea that it allows the agent to explore its knowledge about each action, which might lead to long-term reward.

- Exploitation exploits the agent's current estimated reward and follows the greedy strategy to maximise reward. However, because the agent is greedy with the predicted reward rather than the actual reward, it is possible that it will not receive the highest reward.

The dilemma raises an inevitable question about how much to exploit and how much to explore in a RL problem. In this project, the idea is, instead of a fixed rate of exploration, I use a decaying exploration rate to allow the agent to explore in the early stages of training. And over episode of learning, decay the exploration rate as it learns so that the agent can make more and more reliable decisions over time based on what it learns.

```
self.epsilon = 1.0  # exploration rate at the beginning
self.epsilon_min = 0.005 # fixed exploration rate
self.epsilon_decay = 0.999
```

In each episode of replay buffer, do:

```
if self.epsilon > self.epsilon_min:
    self.epsilon *= self.epsilon_decay #decrease exploration rate
```

By using this method, agent has a fixed exploration rate of 0.005 after around 150 episodes of training.

One thing to mention is that epsilon decay at 0.999 was not the best parameter, and without computational force, epsilon decay could become closer to 1 (e.g. 0.99999). Due to the huge state space of the environment, The DQN agent theoretically requires at least $n * 3^{81}$ steps to fully learn the entire Q table, where n depends on the learning rate and DNN structure.

This has become a weakness of this project, which does not take into account the GPU acceleration and optimization of DQN, resulting in many parameters that are not optimal. This will be mentioned further in section 5.

4.3.3 Experience Replay

As mentioned in section3.3 , DNNs are used to approximate the Q function, and Bellman equation provides a solid method to compute target Q. The problem is therefore modelled as a DNN. However, a fundamental requirement for optimisers(such as SDG) in DNNs is that the training data be independent and identically distributed. When the naïve Q-learning agent interacts with the environment, it can produce a continuous list of transitions that are highly correlated[23]. As a result, experience replay is proposed.

Based on *Revisiting Fundamentals of Experience Replay*[24], the experience replay is a fixed-size buffer that stores the most recent transitions of the environment collected by following the policy. It increases the sample efficiency of the algorithm by allowing transitions to be reused for training multiple times instead of being considered only once and immediately dumped. In addition, experience replay also has the advantage of improving the network's stability during training.

In this project, the experience replay is implemented as a replay buffer that stores a fixed number of transitions. The basic approach is to train the DNN by randomly selecting transition from buffer, which is the approach used in this project, but there are other strategies like prioritized experience replay[25], that uses a certain priority strategy to sample transition.

```
class DQNAgent:
    def __init__(self, state_size, action_size):
        self.memory = deque(maxlen=2000)
```

The replay buffer is implemented by **deque** in Python collections library.

```
def memorize(self, state, action, reward, next_state, done):
    self.memory.append((state, action, reward, next_state, done))
```

Notice that the transition mentioned above is a tuple $T(S, A, R, S', D)$, where D is a boolean value that defines whether the game ends at the current state.

```
    def replay(self, batch_size):
        minibatch = random.sample(self.memory, batch_size)
        for state, action, reward, next_state, done in minibatch:
            target = reward
            if not done:
                # predict the future discounted reward
                target = (reward + self.gamma *
                          np.amax(self.target_network.predict(next_state)[0]))
            # make the agent to approximately map
            # the current state to future discounted reward
            target_f = self.model.predict(state)
            target_f[0][action] = target
            self.model.fit(state, target_f, epochs=1, verbose=0)
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay
```

The code snippet above shows sampling a small batch of transitions from the replay buffer, which break the correlation between subsequent steps in the environment. And then calculate target Q value using Bellman equation. we also need to update the actual Q value to the predicted Q network as the target network.

4.3.4 Target Network

As mentioned in 3.3.4, due to the huge state space in the game, neural network is used in this project. By using a function approximator instead of an exact Q function or Q table. In Q learning, each timestep will only update one state-action value, In DQN, however, each empirical replay will update many state-action values.

That is, when we update the weights of the neural network to bring $Q(s, a)$ closer to the expected outcome, we are already indirectly changing the weights learned by $Q(s', a')$ and other nearby states, which can make our training very unstable[23]. In such a case, target network is applied to the project.

The idea is that the prediction of Q Value at each timestep is performed by introducing another network called target network with the same structure of main Q-network, and the predicted Q value will be used as input to train the main Q-network. It is important to note that the weights of the target network do not need to be trained, but are periodically synchronized with the main Q-network.By doing so, the stability of the training is improved.

The replay function will be shown again, but this time the comment focuses on target network:

```
    def replay(self, batch_size):
        minibatch = random.sample(self.memory, batch_size)
        for state, action, reward, next_state, done in minibatch:
            target = reward
            if not done:
                target = (reward + self.gamma *
                          np.amax(self.target_network.predict(next_state)[0]))
            target_f = self.target_network.predict(state)
            #use target_network to produce target Q value
            target_f[0][action] = target
            self.model.fit(state, target_f, epochs=1, verbose=0)
            #use main Q network to train the weights
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay
```

Target network is synchronized with the main Q-network in every 50 episodes:

```
    for e in range(EPISODES):
        if len(agent.memory) > batch_size and e % 50 == 0:
                agent.model.save_weights("temp.h5")
                agent.target_network.load_weights("temp.h5")
```

4.3.5 H5 Model

In the current version of the code, the DQN model is saved and loaded by saving the weights of the DNN model, which is easily implemented by using the load_weights and save_weights that come with

Keras:

```
    def load(self, name):
        self.model.load_weights(name)

    def save(self, name):
        self.model.save_weights(name)
```

However, saving DQN model was an important issue in the early days of the project.

In the early stage of the project, I used the TF Agent library to build DQN model, which was very simple and did not even need to understand the principle of DQN. The tf Agent model can be successfully trained thousands of episodes, but when I tried to use Checkpointer and PolicySaver to save the model, an undebug error occurred in the system. I reckon the problem is either about *tf_agents.specs.BoundedArraySpec* or is about *tf_agents.specs.ArraySpec* , but the problem was still not solved. Finally, I gave up using Tf Agent.

I also tried using TensorFlow to build DNN model and tried to use *pickle* to save the model, but I failed, and I know there's something wrong with tensors of the environment that similar to the problem I encountered in tf Agent. In two days of study, I still could not find a solution to the problem, finally I turned to Kears.

The following is a code snippet of building DNN model with TensorFlow V1, which I didn't realize at the time was very slow and cumbersome:

```
self.s = tf.compat.v1.placeholder(tf.float32, [None, self.n_features], name='s')  # input
        self.q_target = tf.compat.v1.placeholder(tf.float32, [None, self.n_actions], name='Q_target')  # for calculating loss
        with tf.compat.v1.variable_scope('eval_net'):
            # c_names(collections_names) are the collections to store variables
            c_names, n_l1, w_initializer, b_initializer = \
                ['eval_net_params', tf.compat.v1.GraphKeys.GLOBAL_VARIABLES], 300, \
                tf.random_normal_initializer(0., 0.3), tf.constant_initializer(0.1)  # config of layers

            # first layer. collections is used later when assign to target net
            with tf.compat.v1.variable_scope('l1'):
                w1 = tf.compat.v1.get_variable('w1', [self.n_features, n_l1], initializer=w_initializer, collections=c_names)
                b1 = tf.compat.v1.get_variable('b1', [1, n_l1], initializer=b_initializer, collections=c_names)
                l1 = tf.nn.relu(tf.matmul(self.s, w1) + b1)

            with tf.compat.v1.variable_scope('l2'):
                w2 = tf.compat.v1.get_variable('w2', [n_l1, self.n_features], initializer=w_initializer, collections=c_names)
                b2 = tf.compat.v1.get_variable('b2', [1, self.n_features], initializer=b_initializer, collections=c_names)
                self.q_eval = tf.nn.relu(tf.matmul(l1, w2) + b2)


        with tf.compat.v1.variable_scope('loss'):
            self.loss = tf.reduce_mean(tf.compat.v1.squared_difference(self.q_target, self.q_eval))
        with tf.compat.v1.variable_scope('train'):
            self._train_op = tf.compat.v1.train.RMSPropOptimizer(self.lr).minimize(self.loss)
```

Through this saving model example, I realized the importance of systematic thinking and having a mature technology stack, more of which will be discussed in section7.

4.3.6 Other functions

- act()

  This function is used to get an action that could lead to predicted maximum reward.

```
    def act(self, state, invalid_action,apporach=1):
        #given a state, find the best action for agent to take
        if apporach ==1 and np.random.rand() <= self.epsilon:
            return random.randrange(self.action_size)
        act_values = self.model.predict(state)
        for ia in invalid_action:
            act_values[0][ia] = - math.inf
        return np.argmax(act_values[0])  # returns an action id
```

- random_act()

  This function is used to get an random action.

```
    def random_act(self,invalid_action):
        temp = list(np.array(range(0,self.action_size,1)))
        for ia in invalid_action:
            temp.remove(ia)
        action = random.choice(temp)
        return action
```

4.4 UI and The sound of life

4.4.1 UI

Since some screenshots of the game have been shown in section3.4.1, this section will focus on how it was implemented and the problems encountered in the implementation. Overall, the game uses pygame as the UI framework, and a pymenu package is used to build the game's menu and redirect page, their relationship is shown below:

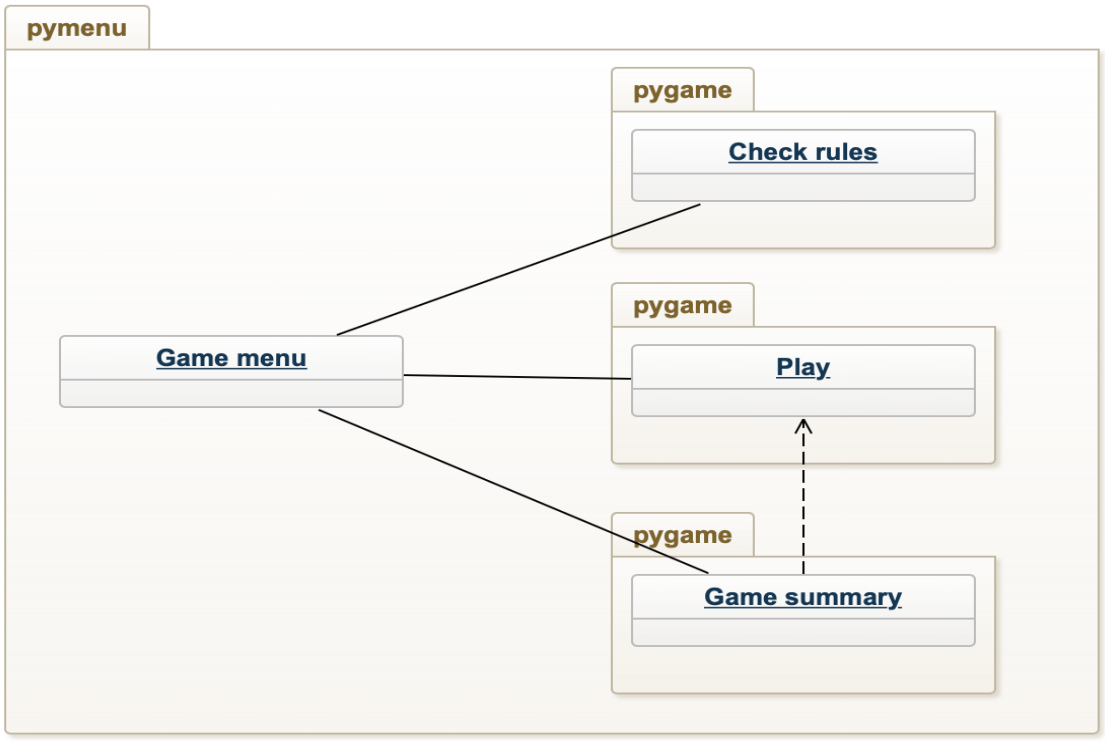

Figure 4.6: UI structure

Fig4.4 shows the early version of UI development, where there is a larger grid compared to the demo version, the reason for the reduced grid is that the limitation of computational power. Learning a large environment by my DQN agent could take days.

4.4.2 The sound of life

Due to the amount of time spent on reinforcement learning in this project, the implementation of the sounds of life was reduced to a kmeans classifier classifying all the states stored in the experience replay buffer into 27 groups, corresponding to 27 piano notes.

In the course of training, add state to an array at each timestep, and at the end of training, the array is used to train kmeans model:

```
    #during training, at each timestep, do:
    state_kmeans.append(next_state)
    #at the end of training, do:
    kmean_sound.train(state_kmeans)

    # load the model
    # model = pickle.load(open("kmeans_sound.pkl", "rb"))

    def train(x):
        kmeans = KMeans(n_clusters=21, random_state=0).fit(x)
        pickle.dump(kmeans, open("kmeans_sound.pkl", "wb"))
```

The following code shows how the kmeans model can be used:

```
    if event.type == evolve_event and not pause:
        # given a state, predict a sound index by kmeans
        sound_index = int(kmeans.predict(Board.board.flatten().reshape(1,81)))
        music = sound[sound_index]
        pygame.mixer.Sound.play(music)
        pygame.mixer.music.stop()
```

Finally, the evaluation of the sound of life will be given in the next section.

Chapter 5

Testing & Evaluation

5.1 Game Testing

Firstly, two main testing methods are used to test the game: unit testing and acceptance testing.

- Unit testing

  Unit testing is used throughout the development phase, i.e. the code is tested at the function level during the development phase. In this project, unit test mainly uses 'print()' to test each methods, an example see at section4.2.1. As this project was done by me alone, there was no communication as well as collaboration, so the unit test report did not make it into the plan.

- Acceptance testing

  In this project, acceptance testing is used after development to get feedback on the game by gathering volunteers to play the game and to get performance reports from the DQN agent. Specifically, surveys were obtained for a total of seven people, who won 4 out of 21 games. This means that the DQN agent has a roughly 81% win rate against human being. In addition, four participants in the survey suggested that the game was not interesting enough, which provided the motivation for the later version of The sound of life.

5.2 DQN Evaluation

Unlike gym, which has a standard game library to evaluate reinforcement learning algorithms, and because the DQN agent in this project is only used against human players, the main object of analysis in this section will be the cumulative win rate of the agent.

First I will provide training plots with several different parameters:

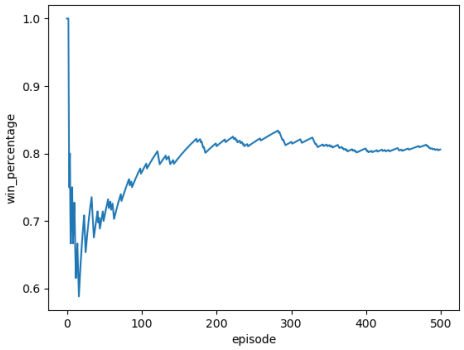

Figure 5.1: lr=0.01,emin=0.05,edecay=0.997

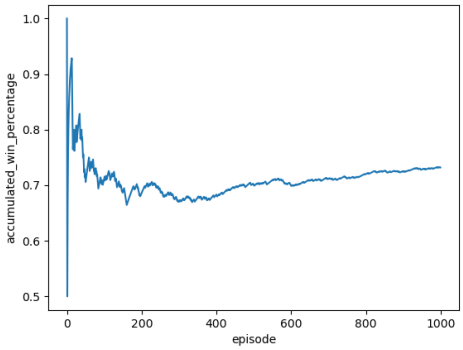

Figure 5.2: lr=0.0001

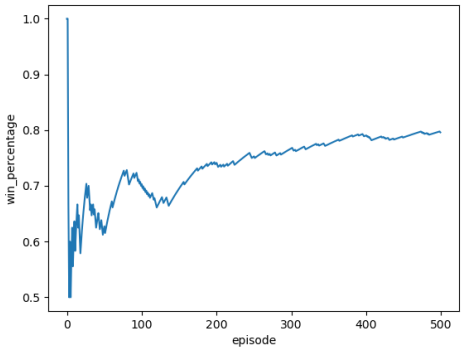

Figure 5.3: lr=0.001,emin=0.05,edecay=0.997

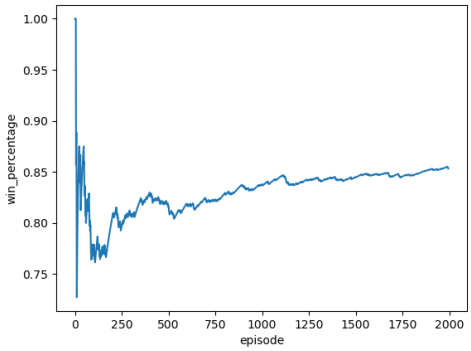

Figure 5.4: 5000+re-train 2000 episodes

It should be noted here that the training time for 5000 episodes is around 12 hours, and due to the limitation of computing power, the DQN model used in this project only learns 7000 episodes. This actually reflects a disadvantage of the DQN implementation: this project does not use GPU acceleration or optimization techniques(such as tensorflow.function) to improve the training speed.

In addition, the mainstream reinforcement learning algorithm is deployed to the cloud to use more computing power resources, while this project is completely run locally, which is also a disadvantage of the DQN implementation.

Chapter 6

Conclusion

To summarise, a life simulation model was proposed early in the project to serve as a competitive game of life environment, but failed in its practical application. Then a new ruleset was proposed to improve the conway's game of life, enhancing the gameplay and dynamics of the game.

The project succeeded in implementing a competitive version of game of life, where a human player could place life in a 2D space to compete with a reinforcement learning agent (DQN agent) trained after 7000 episodes.

It is important to highlight some of the methods used to improve the performance of the DQN agent, which was eventually able to achieve a win rate of over 85%.

Chapter 7

BCS Criteria & Self-Reflection

7.1 BCS Criteria

1. An ability to apply practical and analytical skills gained during the degree programme.
    - As shown in section4.1, the implementation of DQN, the implementation of game environment, and the implementation of sound generator shows my practical skills, that is, successfully implement these ideas from scratch.
    - The evaluation of the DQN agent and the testing of the game are presented in section5 shows my analytical skills.
2. Innovation and/or creativity.
    - In section3.2.1, I put up with an brand new artificial life model: Life of Pakua.
    - Instead of implementing a game simply following the ruleset of Conway Game of Life, in section3.2.2, I came up with my own ruleset to improve the Conway Life Game.
    - In section3.4.2, I proposed mapping each state to some automatically generated musical melody to make the game more playable and fun, rather than having a set soundtrack like most games.
3. Synthesis of information, ideas and practices to provide a quality solution together with an evaluation of that solution.
    - In section2, I put each aspect of the whole project into an academic context, providing a literature review and motivations for the whole project, section3 shows all the ideas that design the project.
    - In section5, the DQN algorithm is evaluated, and the game itself is tested and analyzed. In addition, section3.2.1 also analyzes the proposed model.
4. Your project meets a real need in a wider context.
    - The ruleset of CGoL (section3.2.2) is a study in the topic of cellular automaton and emergence, which introduced probability, expanded the scope of cellular automata.
    - Life of Pakua (section3.2.1) could be a possible preliminary study of 'soft' life under the theme of Artificial Life.
    - The Game itself satisfy demand for competitive version of cellular automaton, it expands the Conway Life of Game, making it a competing version of the 2-players game.
5. An ability to self-manage a significant piece of work
    - more detail will be given in section7.2
6. Critical self-evaluation of the process.
    - more detail will be given in section7.2

7.2 Self-Reflection

- Management

  In this part I would like to discuss project management and self-management, and I want to point out more about the inspiration I got from this project.

  I'll start by showing the project management method used in the project.

  1. I used Agile to manage the project. I think the feature of evolutionary development will make project development more efficient. For example, implementing a simple GoL game environment in the early stages of development would facilitate the development of the DQN module, while at a later stage the CGoL could be refined by improving the rules of the game, in such a way that research and development could be carried out simultaneously.
  2. Using github to manage this project, including managing the commit history, remote clone repositories are used to train models on computers in the school library, and branches provide an isolated environment for every change to my code base, which is very convenient.

  Now I will mentioning some self-management lessons from this project:

  1. Events that seem to take a lot of energy actually keep me inspired. Also, for example, talking to classmates, friends and teachers is a good way to get ideas. The reason why I think of it as self-management is that this inspiration can be applied in the future to motivate me to participate in more academic activities.
  2. I think that emotion management is an important factor in completing a project. This includes how to keep yourself motivated, how to deal with the frustration of not being able to solve problems in the long term, and how to live with other stresses.

- Self-evaluation

  In this part I will first talk about what I have learnt and how this has affected me. I will then talk about the strengths that I have shown in this project and finally my shortcomings in developing this project be pointed out in detail.

  Firstly, in addition to the academic knowledge of reinforcement learning, cellular automata, and artificial life, I have learnt through this project that:

  1. I used conda more frequently for environment configuration and package management, and became proficient in using Mac terminal and some Linux commands. These skills will provide me with a foundation for future data science related studies. I also found that using jupter notebook or colab would be more convenient and efficient for some machine learning and deep learning related tasks, which will be my focus in the future.
  2. I realised the importance of keeping up to date with the latest method stacks. As mentioned in section b, I used TensorFlow v1 to build DNNs, and the process was very cumbersome, while using the latest version of tensorflow to build DNNs was much easier. There are often many ways to solve a problem, so I think I should learn new libraries and tools more extensively in the future to maximise my development efficiency in the future.

  Second, while I evaluate myself after finishing the project, I realise that I have several strengths:

  1. I think I have a very creative and curious mind. I can connect the details of life with some ideas, in section b I connected yin and yang to the life simulation, and I also came up with an improved version of game of life, which I don't think I would have had to do, but I was very curious about what would have happened if I had tried that.

  Finally, there are also many weaknesses that need to be pointed out and improved in the future:

  1. The lack of clarity of the objective in the preliminary planning of this project led to much of the work being divorced from the core objectives at a later stage. I think another reason is that I have more ideas, which may be a bad thing, and frequent changes of focus can lead to serious consequences during the development process
  2. Lack of access to more python packages. Some packages offer efficient utilities that I didn't find, leading to a lot of tedious work.

Bibliography

[1] M. GARDNER, "The fantastic combinations of jhon conway's new solitaire game 'life," *Sc. Am.*, vol. 223 : 4, pp. 20–123, 1970. [Online]. Available: https://ci.nii.ac.jp/naid/10011784148/en/

[2] P. Rendell, *Turing Universality of the Game of Life*. London: Springer London, 2002, pp. 513–539. [Online]. Available: https://doi.org/10.1007/978-1-4471-0129-1_18

[3] S. Wolfram, *A new kind of science*. Wolfram media Champaign, IL, 2002, vol. 5.

[4] C. Reynolds. (2021, Oct.) Boids. [Online]. Available: https://en.wikipedia.org/wiki/Boids

[5] T. Schelling. (2021, Jun.) Schelling's model of segregation. [Online]. Available: https://en.wikipedia.org/wiki/Schelling%27s_model_of_segregation

[6] R. Axtell. (2021, Oct.) Sugarscape. [Online]. Available: https://en.wikipedia.org/wiki/Sugarscape

[7] Wikipedia. (2022, Mar.) Artificial life. [Online]. Available: https://en.wikipedia.org/wiki/Artificial_life

[8] M. A. Bedau, "Artificial life: organization, adaptation and complexity from the bottom up," *Trends in cognitive sciences*, vol. 7, no. 11, pp. 505–512, 2003.

[9] M. Wojtowicz. (2001, Sep.) Cellular automata rules lexicon. [Online]. Available: http://www.mirekw.com/ca/rullex_life.html

[10] B. Baker, I. Kanitscheider, T. Markov, Y. Wu, G. Powell, B. McGrew, and I. Mordatch, "Emergent tool use from multi-agent interaction," *Machine Learning, Cornell University*, 2019.

[11] K. Xenou, G. Chalkiadakis, and S. Afantenos, "Deep reinforcement learning in strategic board game environments," in *European Conference on Multi-Agent Systems*. Springer, 2018, pp. 233–248.

[12] M. L. Littman, "Markov games as a framework for multi-agent reinforcement learning," in *Machine learning proceedings 1994*. Elsevier, 1994, pp. 157–163.

[13] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[14] Unknown. Conway's game of life. [Online]. Available: https://playgameoflife.com

[15] wikipedia. (2022, Apr.) Bagua. [Online]. Available: https://en.wikipedia.org/wiki/Bagua#Western_Bagua

[16] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[17] K. Shao, Z. Tang, Y. Zhu, N. Li, and D. Zhao, "A survey of deep reinforcement learning in video games," *arXiv preprint arXiv:1912.10944*, 2019.

[18] G. Tesauro, "Extending q-learning to general adaptive multi-agent systems," *Advances in neural information processing systems*, vol. 16, 2003.

[19] Jordi. (2020, Jun.) The bellman equation. [Online]. Available: https://towardsdatascience.com/the-bellman-equation-59258a0d3fa7

[20] M. Roderick, J. MacGlashan, and S. Tellex, "Implementing the deep q-network," *arXiv preprint arXiv:1711.07478*, 2017.

[21] M. Čujdíková, "Game monument valley-intersection of mathematics and art," in *ECGBL 2020 14th European Conference on Game-Based Learning*. Academic Conferences limited, 2020, p. 116.

[22] P. Shinners, "Pygame," http://pygame.org/, 2011.

[23] Jordi. (2020, Aug.) Deep q-network(dqn) -experience replay and target networks. [Online]. Available: https://towardsdatascience.com/deep-q-network-dqn-ii-b6bf911b6b2c

[24] S. Zhang and R. S. Sutton, "A deeper look at experience replay," *arXiv preprint arXiv:1712.01275*, 2017.

[25] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.