

---

# Artificail Intelligence Application Report of Self-learning Snake using Double Deep Q Network

---

**Jingkang Yang**

M. Eng Student

Software Systems Engineering

University of Regina

Regina, SK, Canada

yang242j@uregina.ca

## Abstract

This paper introduces an attempt to construct a simple snake game by applying the Pygame library, and introducing a Double Deep Q Network algorithm to construct an artificial intelligence agent to control the snake for unsupervised learning. This paper compares the advantages and disadvantages of different algorithms while introducing the implementation details.

## 1 Introduction

With the continuous development of artificial intelligence, there are more and more application fields for artificial intelligence, and the performance can be significantly improved. The application fields are not limited to robotic applications, but also includes gaming entertainment applications. This artificial intelligence application project aims to apply the concepts of reinforcement learning to one of the earliest and most basic human interactive computer games, Snake. Just as Alpha-Go, created by the DeepMind team, beat the world champion at Go, the world's most complex board game, this project will replicate Alpha-Go's success on Snake. The goal of the project is for snakes to learn and play on their own without human intervention.

The algorithm applied in this project is called Dual Deep Q-Network, which is a combination of Reinforcement Q-Learning and Deep Neural Network. It has both the advantages of reinforcement learning being unsupervised self-learning and the advantages of deep neural networks, which have unlimited potential to learn everything no matter how complex the problem is.

## 2 Knowledge and data representation

To solve this problem, to make a snake agent to play and learn the game, we need to specify a lot of design details beforehand.

### 2.1 Action reward

By giving the agent feedback and letting it know how good or bad the action is in the current game state, the reward for every action the agent makes is critical. The most commonly used reward mechanisms are a positive reward if the snake eats food, a negative reward if the snake dies, and a zero reward if it is just alive. However, in my opinion, this simple mechanism is not desirable. During the training process, there can be more than three important stages. For some reason, snakes tend to do circular movements in the early training stages. Additionally, we need to give some feedback when the snake is alive but not eating. So, I got this idea from Square Robots [1].

<i>Condition</i>	<i>Reward</i>
<i>Hit wall OR Eat snake body OR Health point = 0</i>	<b>-100</b>
<i>Eat food</i>	<b>+30</b>
<i>Move toward food</i>	<b>+1</b>
<i>Move away from food</i>	<b>-5</b>

Table 1: Reward Table

Snakes are not only rewarded with thirty points for eating one food. There are several other reward criteria as well. Each body cell of the snake has a hundred hit points, and it loses one hit point each time it moves. The longer the snake, the longer it can live. If any of the death conditions are met, or its health drops to zero, the snake dies and is rewarded with a penalty of minus one hundred. To motivate the snake to search for food, the snake will receive a positive reward of plus one for moving toward the food, and a penalty reward of minus five for moving away from the food.

### 2.2 State representation

How the current state environment is represented is critical, as this is the only way for the agent to know the environment. There are many ways to represent states. The most robust and commonly used is a list of eleven booleans that represent some basic information about the current game state. The first four booleans are used to indicate the direction of the snake. The next three booleans are used to indicate the danger state of the front, left, and right map cells. The reason there is no back danger detection is that the back is always the body of the snake, and the body of the snake is something that the snake cannot touch. And the last four booleans are used to indicate the relative orientation of the snake and the food. Through this state representation, the agent can gain some basic knowledge of the environment. However, since snakes can only see danger within a cell in three directions, and

can only know the relative orientation of the food, snakes often put themselves in danger, trapping themselves.

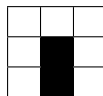


Table 2: Danger Censoring Region

```
def get_state(self) -> list:
    """
    Collect values to discribe the current game state
    19 bool values to discribe the game state
    [
        snake goes up,
        snake goes down,
        snake goes left,
        snake goes right,
        danger ahead,
        danger left,
        danger left-up,
        danger left-down,
        danger right,
        danger right-up,
        danger right-down,
        food left-up,
        food up,
        food right-up,
        food left,
        food right,
        food left-down,
        food down,
        food right-down
    ]
    """
```

Figure 1: Nigteen Bool State Values

By thinking this way, I used nineteen booleans to represent the state. I added four additional booleans to describe the danger status of the four diagonal cells, and four additional booleans to enhance relative food orientation awareness. With such a state representation, the snake can avoid some trapped situations. However, this is still not the optimal state representation strategy. The best way to do this is to convert the entire game environment, each cell, to a matrix of floats or booleans, giving the snake agent a global view of the game environment.

### 2.3 Action representation

As with the previous two setups, there are several different approaches to action representation. The easiest way is to use the same four arrow keys as the user interacts to control the snake's movement. The benefit of this approach is minimal modification of the game code. However, I don't like this approach because the agent always tends to make the snake move in the opposite direction rather than making a circle.

This leads to the second control method, which uses only three movements to control the relative turning of the snake. This method no longer controls the absolute direction of the snake like using the

arrow keys, but controls the snake's turning based on the snake's current moving direction. Relative directions include moving forward, turning left, and turning right. But it is not a good idea to use a 2D vector as input for later model training, because the model cannot accurately identify the meaning of the vector.

The last one is to use three one-hot encoded values based on the second action control method to represent actions in three relative directions. This solves the problem of the model not being able to read vector inputs.

### **3 Approach, Techniques and Algorithms**

After I decided on the details, I needed to decide how to build the game and the agent. Or in other words, which external modules and algorithms are needed to help me structure the project.

#### **3.1 Snake game construction**

First of all, I need a Snake game for the agent to play and learn. There are two options for me to decide. I could decide to import the external Snake game mod from an environment library like gym, or I could create it myself from scratch. Since the built-in game mods might not give me the environment values I need, or I might need to modify the code a little, I decided to make my own Snake.

##### **3.1.1 Pygame and pygame\_menu module**

The module that I include to help me build Snake and UI menus is called pygame. This module is very useful for building a simple game using python. I can't say how advanced it is, but the ease of learning is unquestionable, that's why so many beginners like to use it for their first python game project.

There is another module called pygame\_menu. Although it has pygame in its name, it doesn't actually belong to pygame. It is an independent mod made by a third-party development team based on pygame. This mod saves me a lot of time in developing user interface.

##### **3.1.2 Snake and food reborn**

Snakes and food are the two most important elements in this game. According to the game definition, food can respawn anywhere on the map, but the snake's respawn location is undefined. Most snake games have a fixed snake spawn location, as these games are built for human players and the randomness is not user friendly. But in this agent training project, since every time the snake dies the

agent will restart the game, I don't want the agent to get used to the snake being spawned in a fixed location, so I let the snake spawn randomly anywhere on the map, like food.

### **3.1.3 User and agent play steps**

Although the agent tries to simulate how a professional player would play the game, the agent and human user have different ways of interacting. Therefore, I have defined two different game steps for the user and the agent. User steps need to read keyboard directional control key interactions, there are four possible directions, the game will ignore the opposite direction and perform any acceptable action. The agent step requires action input from the agent, and simply executes the action and returns the computed reward and round score.

### **3.1.4 Danger detection**

There are three death conditions in the game. Snakes are not supposed to hit walls. Snakes cannot eat themselves. To make the game more challenging, I added another condition. Snakes have health points, and each time the snake takes a step, the health point decreases. If the health points is zero, the snake dies. Each time the snake eats a food, its health resets, and the health is equal to the length of the snake multiplied by the health of each snake cell. By default each cell has one hundred health points.

## **3.2 Agent algorithm**

There are many algorithms that fit this topic, play and learn Snake. In this section, I'll introduce some alternatives and explain why or why not to use them. In order for Snake to learn by itself, it is best to use unsupervised reinforcement learning. Also, Q-learning techniques are best suited and most commonly used for such problems.

### **3.2.1 Basic Q-learning**

Basic Q-learning is one of the most fundamental reinforcement learning techniques. It creates a Q-learning table to store each state and its corresponding action, the action reward, the computed Q-value, and the next state by performing the specified action. Every time an agent sends a state and requires an action, the table returns the action with the current maximum Q value. By iterating over all possible states and actions, the agent performs well and quickly. However, Q-learning also has some disadvantages.

First, Q-learning is a bootstrapping algorithm that learns one estimate based on another estimate. Unlike supervised learning, which uses labels or ground truth to guide agent learning, Q-learning is a

type of unsupervised reinforcement learning that learns from feedback from previous attempts, which can be implicit and untimely.

Second, since the agent decides actions based on the maximum value of the previous Q-values, the agent is likely to overestimate the new Q-value and lead to maximizing bias through learning. The bootstrap mechanism exponentially amplifies the damage caused by overestimation.

Lastly, Q-learning is an off-policy algorithm with different strategies in target estimation and action selection. The advantage of off-policy is that it can diverge for a short period of time, but the results can be unreliable.

Therefore, the basic Q-learning does solve the problem, and it's easier to implement compared to other alternatives. However, for complex problems with more possible state-action pairs, it is not the optimal solution.

### **3.2.2 Double Q-learning**

Double Q-learning is an algorithm that takes basic Q-learning one step further by attempting to address one of the shortcomings listed above. Double Q-learning addresses overestimation by using two independent Q-value estimators and using one to update the other. Because the estimators are independent and identical, the agent can estimate and select actions unbiased.

### **3.2.3 Deep Q Network**

Deep Q Network is another attempt at the basic Q-learning algorithm, trying to increase the potential application limit by introducing neural network in it. Neural networks have the potential to learn all incredible things. By combining the Q-learning algorithm with a neural network, the agent can train the network to predict the Q-value for each possible states, allowing the state description to be flexible to handle different environments. On top of this, deep neural networks can process data matrices or even image data if the number of layers in the neural network is increased to make it deeper, or if convolutional neural networks are involved in it.

Deep Q-networks make Q-tables flexible and enable Q-learning to handle complex problems. However, there is still an overestimation problem due to the estimation based on the same model estimates.

### **3.2.4 Double Deep Q Network**

Through the above attempts, Double Deep Q Network emerged. Double Deep Q Network, not only solves the problems of overestimation and fixed Q-table listed above, but also introduces a new feature, experience replay, which makes the estimation more accurate and closer to the true value.

**Double neural network** In addressing the overestimation problem and the involvement of neural networks, Double Deep Q Network uses two neural network models to handle action evaluation and action selection, respectively. This decouples the two estimation parts from "guess from my old guess".

The neural network integrated in the algorithm can be of various shapes. There are a number of ways to configure a neural network for each specific application. For the Snake game, I did not introduce a deep neural network as the algorithm name suggests. The deeper the neural network, the more training time it will take. Also, if the complexity of the neural network does not match the application, the model is likely to overfit or underfit.

The neural network I introduce in this project is a sequential model with two fully connected hidden dense layers, each of which consists of 256 hidden nodes, and uses a rectified linear activation function. I think this layer structure setup is suitable for Snake games without causing it to overfit or underfit.

The neural network library is from Tensorflow, just a personal preference.

**Bellman equation** The double deep Q-network modifies the future evaluation part of the Bellman equation from evaluating the future state Q-value with the current action to estimating the selected best action using the target model. By doing this, the agent can update the current state estimate with the updated Q-value.

**Replay memory** Double Deep Q Network stores each transition sample (i.e. old state, action, reward, and new state) into a pool of experience replays and trains in random batches after each round. Using this experience replay technique, not only can each transition sample be reused, but the temporal correlation between each transition sample can be broken.

**Model synchronization** Since there are two models used in the Double Deep Q network, the algorithm synchronizes the weights of the action target model with the weights of the action evaluation model being trained. The synchronization period of this project is set to 100 rounds, that is, for every 100 rounds, the two models are synchronized once.

### 3.3 Action selection - Epsilon greedy

Since all reinforcement learning requires traversing the map to collect state-action-value pairs as explorations, letting the algorithm decide which node to iterate or not to iterate is not ideal, as it tends to follow the same path rather than unknown states. Therefore, I introduced the epsilon greedy algorithm into the action selection mechanism, giving it some randomness. The initial value of the epsilon parameter is 1.0 with a decay rate of 0.99. This means that every time the agent finishes

a batch of training, epsilon will be multiplied by its decay rate to reduce randomness. And the minimum value of the epsilon parameter is set to 0.01, which is to keep the 1% randomness of the action at all times. This is because during the agent training, there is no guarantee that each action selected is the optimal solution, and it still needs to be continuously explored.

$$\epsilon_n = \epsilon_{init} * \epsilon_{decay}^n$$

Figure 2: Epsilon decay equation

As the number of training sessions increases, the randomness of actions decreases accordingly. Multiply the Epsilon by 100 to get how many random actions are in every 100 moves.

#### 4 A Structural diagram and explanation

The project has two parts, the Snake Game and the Agent Algorithm, which are connected through the main menu class of the user interface, as shown in the UML diagram. Includes three main functions, representing the three goals of the project. The first goal is that the user should be able to play Snake freely by interacting with the arrow keys. This requires a fully functional Snake game built in. The second goal is to train the agent without time constraints for as long as the user wants. Apart from that, the user should be able to terminate training and resume training from the most recent training results. This training function is the core purpose of this project. Last but not least, the user should be able to observe the snake growing during the training demo.

The Snake game module has three subclasses to implement the corresponding functions of each entity respectively. Snake class is able to control movement, drawing, and resetting the snake's body and position. Food class can generate and draw a food at random coordinates in the map, but this location cannot be covered by the snake body. Map class currently only has one job, and that is to set the map color for the entire playground. With the synergy of all three subclasses, the Snake game class can focus on making the game work. In order to enable the game to be interacted by the user using the keyboard arrow keys and by the agent by passing action commands, the game control integrates two independent game running logics for different control methods.

In order to solve the problem identified at the beginning of this paper, a double deep Q-network algorithm is introduced and implemented. The algorithm has two main steps, the training step and the learning step.

The training step is the step of collecting and memorizing data for the subsequent learning step to fit two neural network models. The agent first collects the current state values with a predefined matrix structure, introduced in the data representation section. Then, by using the epsilon greedy algorithm described in Section 3.5, the agent decides to perform either a random action or a model-estimated



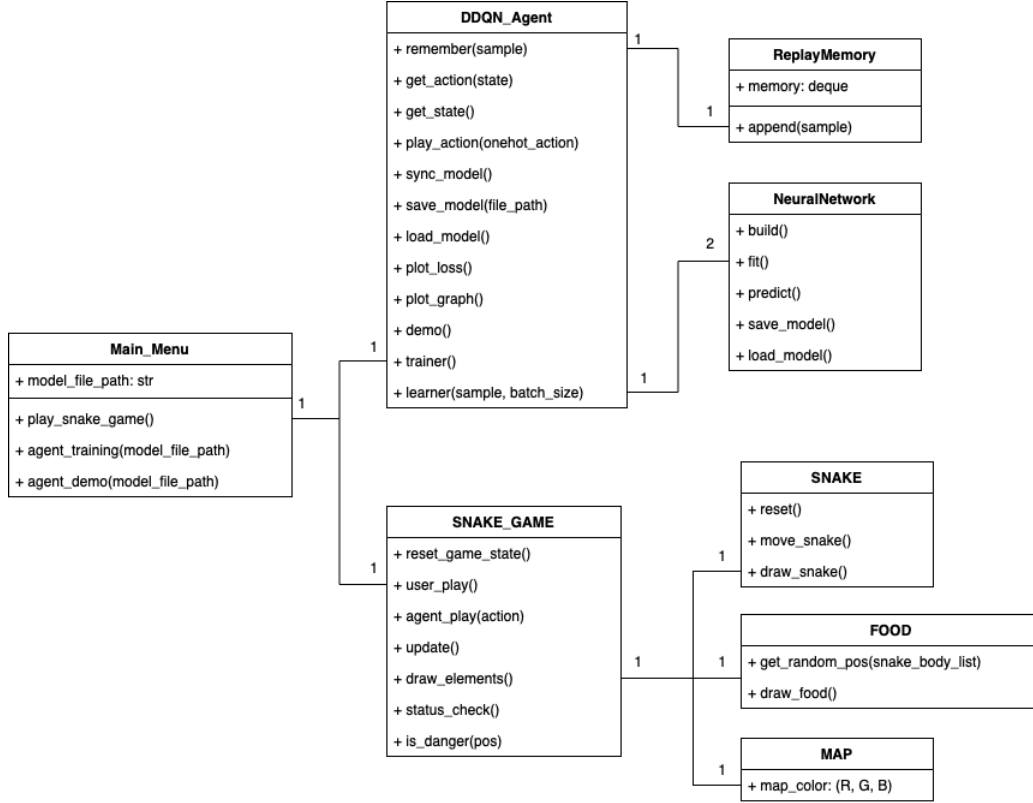


Figure 3: UML structural explanation

action. By performing the selected action, the game environment will return a list of values, including the action reward, round over status, and the current round score. The agent then collects another set of state values as the post-action state values. Armed with this data, the agent can construct a record of transition samples and remember it in the experience replay memory pool for later training. Then, the agent will decide whether to continue the next round of sample data collection according to the round status. If the action results in the end of the round, reset the game environment and proceed to the learning step.

The learning step is the core of the Double Deep Q Network algorithm. First, the agent checks whether there are enough transition samples in the memory pool. If there are enough samples, the agent will randomly sample a certain number of samples as mini-batches for model training. The algorithm requires two neural network models, the target model and the action evaluation model, which were introduced in Section 3.2.4.1. By involving two independent models to decouple the action target and action evaluation, the algorithm is able to minimize the effect of overestimation of Q values that all single-model Q-learning algorithms have. The Q-value update step is to use the target model to estimate the future state value of the maximum Q-value action, use the action evaluation model to estimate the future state, then multiply by the gamma discount rate and add the action reward. The main goal is to minimize the error loss between the predicted Q-values and the

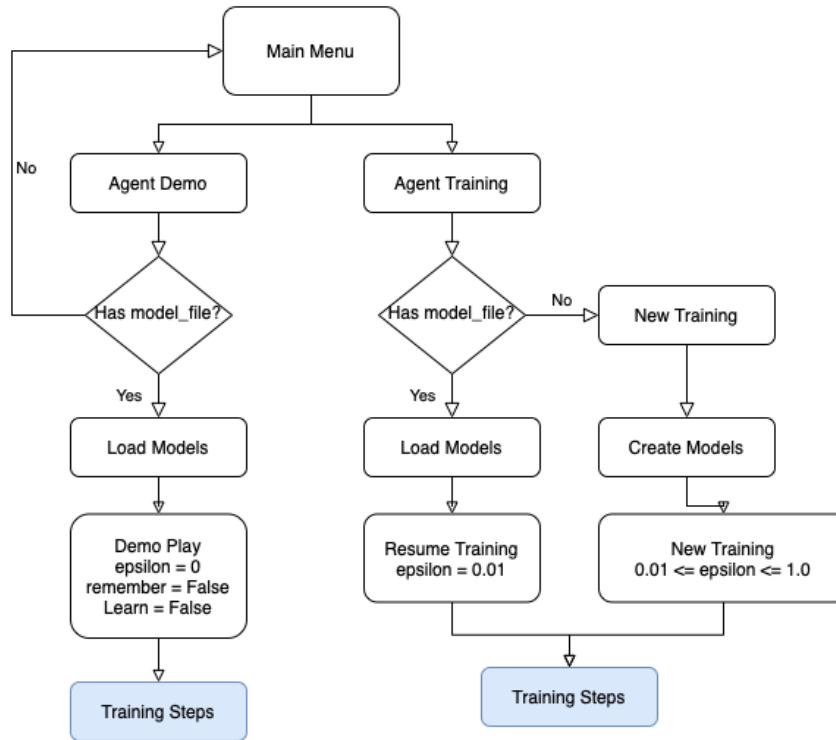


Figure 4: Main menu flowchart

updated Q-values by using gradient descent on the mean squared error loss. Every hundred rounds of the game, the weights of the target model will be synchronized with the weights of the trained action evaluation model. In this way, the action target model acts as a fixed training target, while without it the training target would be dynamic.

## 5 How to use your “modules”

To run the program, user should install some libraries.

- **PYGAME** v2.1.2
- **PYGAME-MENU** v4.2.5
- **TENSORFLOW** v2.6.0

There are five buttons in the main menu, which correspond to the five main functions of the program.

- **PLAY** The user should be able to manually play the Snake game.
- **NEW AI TRAINING** The user should be able to train the AI agent and the agent should learn while play the game.

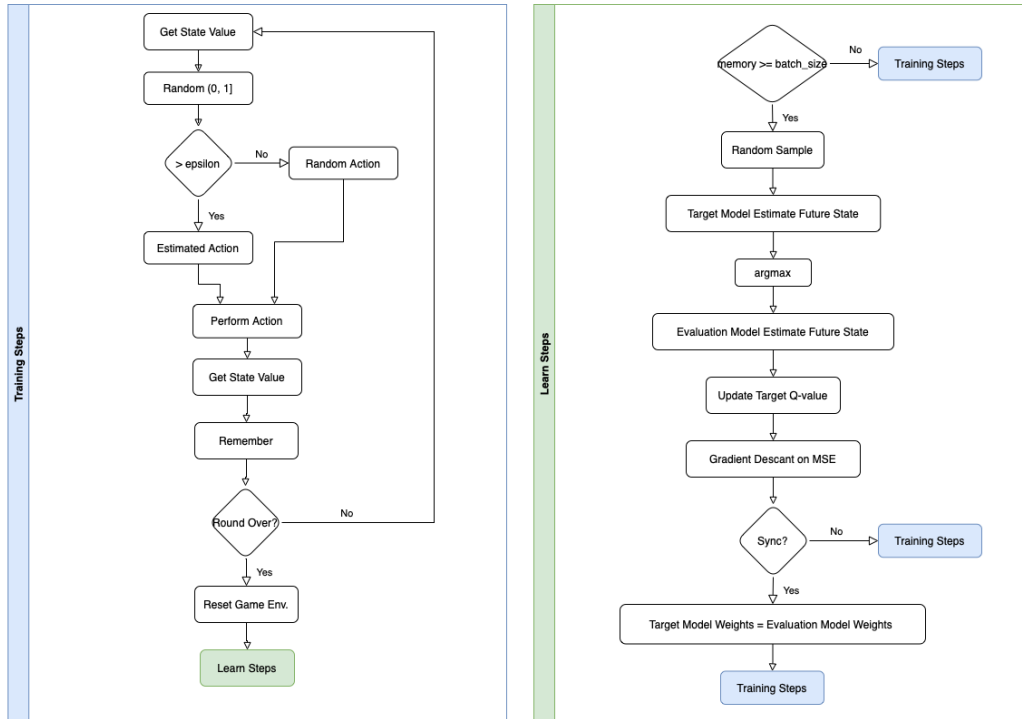
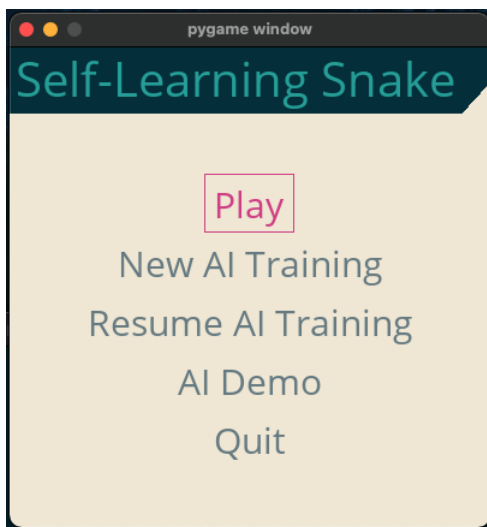
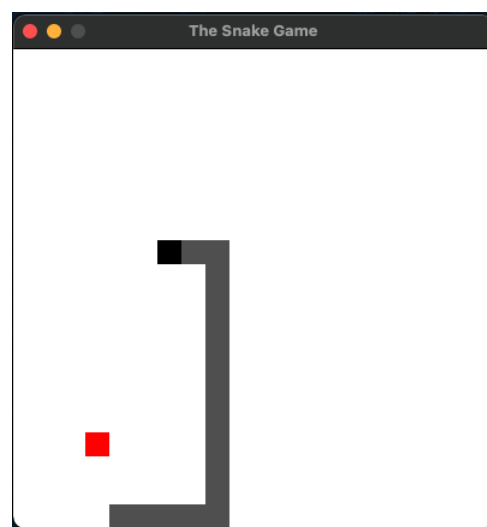


Figure 5: Train and learn flowchart

- **RESUME AI TRAINING** The user should be able to resume the training and the training should continue from where last time stops.
- **DEMO** The program should be able to demonstrate the well trained agent.
- **QUIT** The user should be able to safely close the program and not cause any error.



(a) Main menu image



(b) Snake playing screenshot

Figure 6: Game playing screenshot

When the agent is training, the user can view the training progress on the terminal, including the number of training rounds, round scores, highest records, and epsilon values. epsilon is a random parameter that causes the agent to randomly perform an action at intervals. Users can also view training loss trends and score trends in the plots directory. The trend graph will refresh as the agent trains. The current best trained model file will be saved in a directory named model, and the model file defaults to "ddqn\_snake.h5". The demo model file is named "ddqn\_snake\_demo.h5" by default. Therefore, users can use the newly trained model to display the training results by renaming.

The "config.py" file contains the default values for all hyper-parameters. Each hyper-parameter has an explanation behind it, making it easy for users to understand what it means. Users can customize the appearance and operation of the program by modifying the numbers.

## 6 Sample Sessions

During training, the agent loops continuously unless the user closes the program or manually returns to the main menu by pressing the escape key.

The program prints some useful messages in the terminal. For example, the number of rounds, the current round score, training records, and epsilon randomness. If epsilon reaches a preset minimum value, the countdown starts to stop early. By default, there are five hundred rounds of patience before terminating the training. The user can change the default early stop value to a very large number so that the program does not stop prematurely.

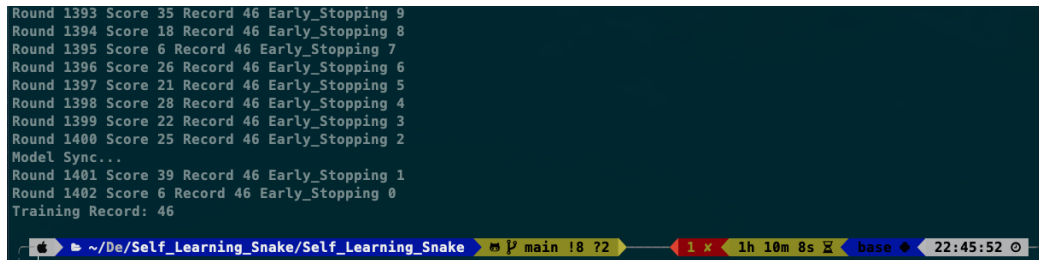
A terminal window showing the output of a snake training program. The output consists of multiple lines of text, each representing a training round. Each line contains four values: Round, Score, Record, and Early\_Stopping. The Round number increases from 1393 to 1402. The Score fluctuates between 6 and 39. The Record remains constant at 46. The Early\_Stopping value decreases from 9 to 0. After Round 1402, the text 'Training Record: 46' is displayed. The terminal window has a dark background with light-colored text. The bottom of the window shows a status bar with the file path '~/.Self\_Learning\_Snake/Self\_Learning\_Snake', the current directory 'main', and the file '8 72'. There are also some icons and a timestamp '22:45:52'.

Figure 7: Snake training terminal screenshot

## 7 Sample Listing

The code for this project is well documented with comments and class introductions. In this section, I will briefly describe the two main classes in this project and the methods of them. Screenshots are not indicate the complete code.

```

104 class SNAKE_GAME:
105     """ SNAKE_GAME class
106     Define how the game operates.
107     - SNAKE_GAME.reset_game_state()
108     - SNAKE_GAME.user_play()
109     - SNAKE_GAME.agent_play()
110     - SNAKE_GAME.update()
111     - SNAKE_GAME.draw_elements()
112     - SNAKE_GAME.status_check()
113     - SNAKE_GAME.is_danger()
114     """
115 > def __init__(self, surface, agent=False) -> None:--
130
131 > def reset_game_state(self):--
149
150 def user_play(self):
151     """SNAKE_GAME.user_play()
152
153     INPUT: None
154
155     OUTPUT: game_over, game_score
156
157     Define the operating steps for user interaction
158     - Collect user input:
159         - four arrow keys for direction control
160         - esc for stop game and back to main_menu
161         - update the game when receives user event
162     - fill the surface color
163     - draw each elements, snake and food
164     - update the pygame display
165     - return game_over status and game_score
166     """
167     # Collect user inputs
168     for event in pygame.event.get():
169         if event.type == pygame.QUIT: # Quit the game

```

Figure 8: Snake game class code

Figure 8 is a screenshot of the code of Snake's main control class and its comments. This class includes the reset and update functions of the game, while also taking into account two different ways of interacting for both human user and agent user.

```

39 FC_DIM = config.FC_DIM # @param {type:"integer"}
40 EARLY_STOPPING = config.EARLY_STOPPING # @param {type:"integer"}
41
42 > class ReplayMemory(object):--
74
75 > class NeuralNetwork(object):--
160
161 class DDQN_Agent(object):
162     """DDQN_Agent
163     Defines the algorithm to train the agent as well as make the Double-Deep-Q-Network learn
164
165     Methods:
166     - self.remember(sample_list) -> Append the sample list into the memory pool
167     - self.get_action(state) -> Return the onehot action with the max predicted Q-values
168     - self.get_state() -> Return state-value list for the current game environment
169     - self.play_action(onehot_action) -> Return the stop_training status,
170         -> reward after perform the action,
171         -> game_over status,
172         -> game_score after perform the action
173     - self.sync_model() -> Synchronize the weights of evaluation_model to the target_model
174     - self.save_model(file_path) -> Save the model to the given file_path
175     - self.load_model() -> Load the model from the pre_defined file_path in the __init__()
176     - self.plot_loss() -> Plot and save the loss graph
177     - self.plot_graph(score_list, mean_score_list) -> Plot and save the score graph
178     - self.demo() -> Play the game with 0 epsilon and no learning
179     - self.trainer() -> Agent training algorithm/steps
180     - self.learn(training_sample, batch_size) -> Double-Deep-Q-Network learning algorithm
181
182     """
183     def __init__(self, game_env, model_file_path=None) -> None:
184         # Init Instances
185         self.env = game_env
186         self.memory = ReplayMemory()
187         self.dqn_eval_model = NeuralNetwork(input_shape=STATE_LEN, output_shape=ACTION_RANGE, fc_dim=FC_DIM, learning_rate=LR)
188         self.dqn_targ_model = NeuralNetwork(input_shape=STATE_LEN, output_shape=ACTION_RANGE, fc_dim=FC_DIM, learning_rate=LR)

```

Figure 9: Agent class code

Figure 9 is a code screenshot of an artificial intelligence agent with a double deep Q-network algorithm implementation. This class consists of two main steps. Agent training steps focus on data collection and storage. And the model learning step is to fit the computed updated truth values to the action evaluation model by minimizing the mean squared error loss between the predicted values and the updated truth values.

## **8 Discussion**

Through the development of this project, I really learned a lot. Before this project, I knew some reinforcement learning algorithms. But what I don't know is, how to implement this algorithm in detail and how to adjust some hyper-parameters by observing the results.

### **8.1 Game developing in Python**

Game development has always been an interest of mine, but never had the opportunity to learn how to implement it. With this opportunity, I finally wrote my first little game, even though it was done with the help of the pygame library.

### **8.2 Pros and Cons**

The advantage of my approach, Double Deep Q Network, is that it addresses two major problems that other Q-learning algorithms cannot avoid. This algorithm certainly has more potential to explore, but its performance is limited due to the simplicity of the application environment.

The disadvantage of my method is that it takes a long time to get acceptable results. Another downside is that my state representation and neural network are too simple for the algorithm to take full advantage of.

## **9 Conclusion**

This project aims to build a snake game and an artificial intelligence agent to train snakes to play by themselves. To achieve this goal, through research on reinforcement learning, the algorithm was identified as a dual deep Q-network. This algorithm successfully solves some common problems found in many other Q-learning algorithms.

## **10 Future Work**

Although the project goals have been achieved, there are still many details that need to be corrected and refined.

### **10.1 State representation**

The state representation used in this project can be increased to a nine-by-nine matrix around the snake's head, and beyond. By doing this, the snake can avoid most of the cornering situations. Apart from that, the state can be the entire game map, which will provide the agent with all useful information to decide the best route.

### **10.2 Deep Neural Network**

As the complexity of the state representation increases, the neural network layers should be increased. If the neural network could add more layers or even convolutional layers, then the agent could definitely learn more from the transition samples.

### **10.3 Model weights synchronization**

The model weight synchronization method used in this project is the simplest method, where the target model accepts all trained weights from the evaluation model. This is not ideal, as the trained weights may have incorrect estimates. Therefore, if the synchronous method can accept the weights in part but not all, the training results can be closer to the ground truth.

### **10.4 Resume model training graph**

There is an obvious problem with the model training resume function. When model training resumes, it doesn't pick up exactly where it left off. Since many of the relevant hyper-parameters and recorded values were not stored in the model file, random exploration and image plotting could not be recovered.

## References

- [1] Square Robots. "AI learns to play SNAKE using Reinforcement Learning" YouTube, 20 June. 2019, <https://www.youtube.com/watch?v=8cdUree20j4>.
- [2] Python Engineer. "Teach AI To Play Snake - Reinforcement Learning Tutorial With PyTorch And Pygame (Part 1 to 4)" YouTube, 20, December. 2020, <https://www.youtube.com/watch?v=PJl4iabBEz0&t=899s>.
- [3] Choi, M., Jung, S., & Lim, K. (2020). A seamless adaptive video streaming technique based on the double DQN. *The Journal of Korean Institute of Communications and Information Sciences*, 45(10), 1700–1707. <https://doi.org/10.7840/kics.2020.45.10.1700>
- [4] Harder, H. de. (2022, March 25). Snake played by a deep reinforcement learning agent. Medium. Retrieved April 9, 2022, from <https://towardsdatascience.com/snake-played-by-a-deep-reinforcement-learning-agent-53f2c4331d36>
- [5] Kelly, S. (2019). Game project: Snake. Python, PyGame, and Raspberry Pi Game Development, 181–212. [https://doi.org/10.1007/978-1-4842-4533-0\\_16](https://doi.org/10.1007/978-1-4842-4533-0_16)
- [6] Reinforcement learning and deep reinforcement learning. (2021). *Deep Learning in Science*, 282–307. <https://doi.org/10.1017/9781108955652.016>
- [7] Sewak, M. (2019). Deep Q Network (DQN), double DQN, and dueling DQN. *Deep Reinforcement Learning*, 95–108. [https://doi.org/10.1007/978-981-13-8285-7\\_8](https://doi.org/10.1007/978-981-13-8285-7_8)
- [8] Zuo, G., Du, T., & Lu, J. (2017). Double DQN method for object detection. 2017 Chinese Automation Congress (CAC). <https://doi.org/10.1109/cac.2017.8243989>