INM 707 Reinforcement Learning Coursework

By Chit Lun, CHOW

Basic Tasks

1. Definition of environment and problem to be solved

The problem to be solved is to train an agent to effectively play the snake game through reinforcement learning. The goal of the game is to move through the grid (game board) and collect food while avoiding hitting obstacles such as walls (grid limits) or its own "body" that will grow with the food it collects. The challenge lies in the dynamic of the game as the result changes with each action the agent takes.

Definition of the Environment

In this project, the environment is based on pygame platform. The environment is a 20 pixels square grids, with a resolution of 1280 by 720. The grid will be a 64 by 36.

The agent must learn to make decisions based on the current state, including the snake's position and direction, location of the food, and the positions of the walls.

As Reinforcement learning algorithm is used to train the agent by giving feedback through rewards or penalties based on the outcome of each action. The agent will learn to adapt its behavior to maximize its cumulative reward over time (return) that in this game would translate as a maximum score.

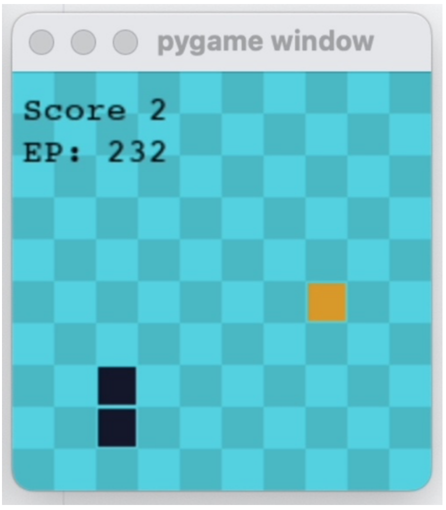

Figure 1: Screenshot from the game using pygame platform.

Therefore, the agent will learn how to play through the state transitions functions that will define how the snake moves through the grid designed as part of the environment, and the reward function that quantifies progress by giving a numerical reward that would help the agent learn about the movements (actions) that execute.

The environment, as shown on Figure 1, designed for the game consist of three elements:

- **Grid**: 400 squares (20 x 20)
- **Snake:** Center of the screen
- **Food**: It will be display as a single square on the screen that has a different color, and its position is determined by a randomize function that will change its location after each step that the snake take.

2. Definition of State Transition Function and Reward Function

The reinforcement learning algorithm that is implemented is Q-learning which optimize the learning process based on a Markov decision process (MDP). Therefore, the agent being deploy on the defined environment with a possible set of states (**s**) that can be attain by performing specific actions (**a**), considering the MDP the change of state only depends on the

action executed and the current state. Hence, the change on the current state will not depend on any previous action of state. In each change of state there is a reward (or punishment) associated to the action performed.
Regarding this game, the Q- Learning algorithm will define what is the best action to follow based on the previous outcome. This is achieved by defining a Q-matrix that will contain every possible action and the reward that could be obtain, and the Q-Learning algorithm will update that matrix on each iteration looking to optimize the results.

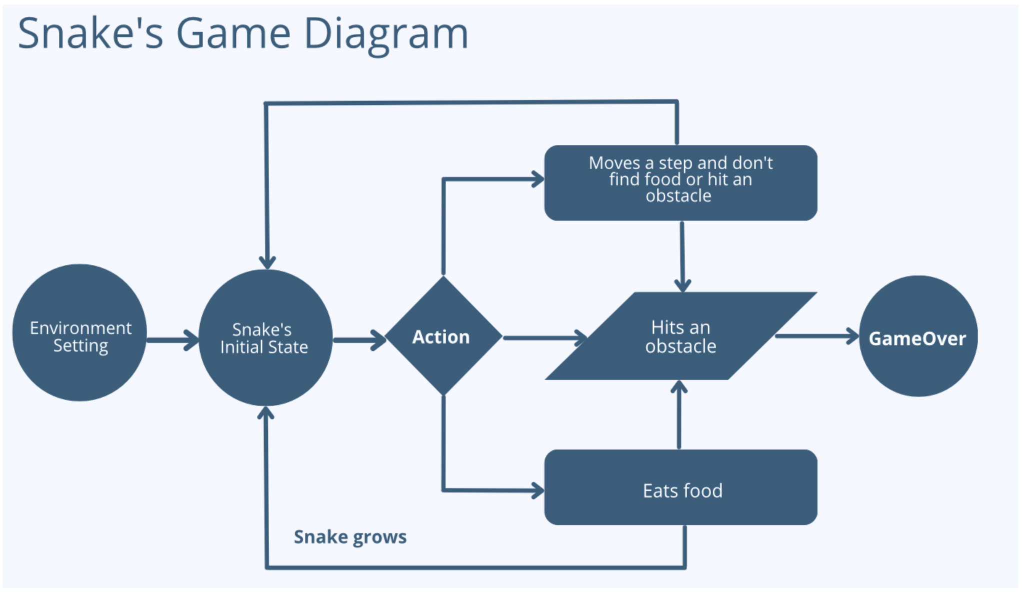

Figure 2: Diagram of the game process

In this game, the possible actions for the agent to take at playing snake are defined by the 4 directions that it could move on the grid (up, down, left, or right), and the state would be defined by the reward (**r**) that find, unless is the terminal state (game over) as it can be visualized on the diagram above.

<u>State Definition</u>
There are a numbers of state definitions that has been developed. To make sure that state for the Q-learning is computationally feasible, the states-space is defined as the following tables suggested:

State Variable	Notation	Explanation	Possible Values
Current_up	c_{up}	Snake is moving upward	0, 1
Current_down	c_{down}	Snake is moving downward	0, 1
Current_right	c_{right}	Snake is moving right	0, 1
Current_left	c_{left}	Snake is moving left	0, 1
Food_up	f_{up}	Food is above the snake	0, 1
Food_down	f_{down}	Food is below the snake	0, 1
Food_right	f_{right}	Food is on the right of the snake	0, 1
Food_left	f_{left}	Food is on the left of the snake	0, 1
Danger_ahead	d_{ahead}	Danger ahead	0, 1
Danger_right	d_{right}	Danger on the right	0, 1
Danger_left	d_{left}	Danger on the left	0, 1

Mathematically, the state space is defined as the following:

$$S = \{s \mid s = (c_{up}, c_{down}, c_{right}, c_{left}, f_{up}, f_{down}, f_{right}, f_{left}, d_{ahead}, d_{right}, d_{left})\} \subset \mathbb{Z}^{11}$$

To measure the size of the state space, there will be around $2^{11} = 2048$ different states if just calculating the combination of every possible values. It is possible to simplify. If a snake is going in a single direction, say going up, all other directional states are all zeros. For the states indicating the relative position of the food, there can be 10 possible states. Therefore, there are only 4 possibilities for the directions, and the number of possible states is just $C_1^4 \times (3^2 + 1) \times 2^3 = 320$.

Rewards Definition

The rewards that can be obtained by the action can be frame on the following table:

Action	Reward	Score
Move a step without eating food or hitting an obstacle	0	0
Move a step eating food	10	1
Move a step and hit an obstacle	-10	0

Table 1: Rewards gained by each action.

A score column is attached to table 1 to explain the relation that held with the reward obtained by the agent through each action as it is consider later as a performance measure.

3. Q- Learning parameters and Policies

The learning process consist mainly of 2 components: exploration and exploitation. To facilitate the learning, it would be easier for the agent to have a random exploration at the beginning of the training stage, which will be explained while detailing the policies that are use. However, through the experience of each move, the agent can observe the change between states and rewards, which is provided after each time-step is completed. The role of Q-learning is to build a look-up table for agent's reference when deciding best possible action, and update it when new experience comes in.

The Q- Learning function is formulated through the Bellman's equation as it follows:

$$Q(s_t, a_t) = \mathrm{Q(s_t, a_t)} + \alpha \left[\left(r + \gamma * \max_a Q(s_{t+1}, a) \right) - Q(s_t, a_t) \right]$$

Having **s** (state), **a**(actions) and **r** (reward) already defined, the equation above calculates $Q(s_t, a_t)$ that represent the present Q-value giving a state and action that its updated through the parameters **α** (learning rate) and **γ** (discount factor that determines the agent's priority towards immediate reward or the long term return), and it's optimize by comparing to the maximum reward that can be achieved in the next state, which is the maximum Q-value by performing an action a.

Parameters

For an initial experiment, the values for the parameters were set as follows:

α = 0.01 and **γ** = 0.95

These parameters where set to provide a gradual learning over 30000 episodes that led the agent to achieve the highest score before reaching the terminal state (game over).
It also considers a goal to achieve 100 as a score to proof a successful learning.
While both parameters can be set between [0,1], the ones initially chosen to provide a low learning rate and a high discount rate prioritizing the return over the immediate reward.

<u>Policies</u>
This reinforcement learning process also considers a behavior and a learning policy. And specifically on the Q-learning algorithm both policies are set through the same parameter, ε (epsilon). In this implementation the policy considered is the ε–greedy policy as it allows to balance and parametrize the exploration (data collection) and exploitation (reward seeking) performed by the agent. This explains on the agent deciding on a random action with a probability of ε, and deciding on a greedy action with a probability of 1-ε.

For the initial experiment, ε is set to 1 whilst applying the epsilon decay technique that helps in balancing the exploration and exploitation by decreasing the value of ε in each episode. Therefore, the behavior policy is altered by encouraging the agent to explore more in the beginning of training and exploiting after, while the ε decreases. To control this process, a minimum of ε was set at 0.001.

4. Q- Learning Performance

As a result of the first implementation of the Q-learning algorithm, the results were lower than those expected.
However, the goal score of 100 is projected to be achieved following the tuning of hyper parameters (α and γ) and the implementation of deep learning (DQN).

However, as shown in Figure 3, the algorithm's performance has already exhibit positive results with a mainly sustained score over 20 after 6000 episodes.
This is likely due to the Epsilon decay, as illustrated in Figure 4, which reached its threshold of 0.01 in episode 6179.

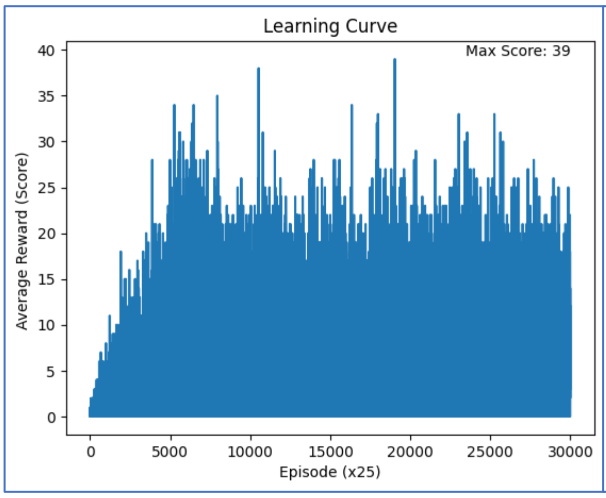

Figure 3: Learning curve of first implementation. Average Score through 30000 episodes.

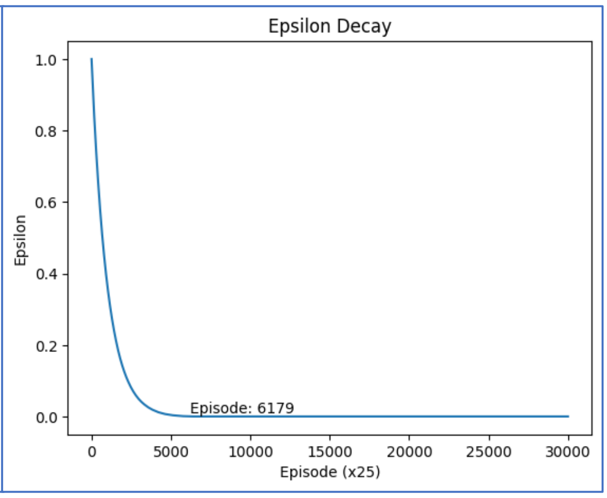

Figure 4: Epsilon decay curve of first implementation over 30000 episodes.

This suggest that the agent collected useful data that allowed it to consistently exploit the environment in later episodes, resulting in an optimal policy with the given parameters (α = 0.01, γ = 0.95 and ε discounted from 0.999).
However, it's critical to note that maximum score obtained was only 39, which is yet far from the goal. Accordingly, it's ratified the need for further improvements and optimizations.

5. Parameters and policy tuning

After the base implementation of the algorithm another set of parameters were considered to improve the results.

The new parameters consider are as follows:

Alpha (α)	0.001	0.005	0.01
Gamma (γ)	0.3	0.65	0.95
Epsilon discount	0.999	0.9995	0.99975

Results show a significant improvement in the performance compared to the base experiment while finding the best performing parameters at α = 0.005, γ = 0.3 and ε discounted at 0.999. These parameters were chosen considering a change on the learning strategy. Whilst adding 2 lower values for α, it is expected to direct the agent to a slower and progressive learning. Additionally, for γ the 2 new values considered were also lower, intending to guide the agent towards prioritizing the immediate reward instead of the return.
Finally, the epsilon discount value considered 2 higher values that could lead to promote the exploration process done before exploitation trying to enhance the policy optimization.

The results obtained with the new parametrization are detailed on the following section.

6. Quantitative and qualitative analysis

In the first trial the results were considerably lower that was is expected to be achieved. However, the epsilon decay showed a positive relation towards the overall performance. Hence, the reason to maintain the same policy whilst modifying the values to be discounted from ε considered a small variation on the initial value.
Despite this, for α and γ the variation on the initial parametrization is considerably higher expecting to achieve a bigger difference on the overall performance.

On figure 5, the results obtained for the 27 experiments performed are displayed pondering every combination of the parameters and showing the best performance when α is set to 0.005, γ = 0.3 and ε discounted from 0.999.
This parametrization achieved the goal set on 100 points of score by obtaining 101 as maximum score.
The best result can be explained as the learning rate was lower (-50%) and the discount factor was also considerably lower than first trial (-68,42%). This translates onto a slower learning

through each episode and a higher priority towards looking for an immediate reward instead of the accumulation of it.
The discount value applied to ε on the best result is the same that was use on the first implementation, which confirms the previous analysis that showed a good correlation between performance and the epsilon decay.

Therefore, the main effect of the results variation relies on the parameters of the Q-function rather than the policy.
What stands out on the result grid (figure 5), is that discount value for $\varepsilon = 0.999$ visibly shows a better performance in most of the scenarios.
Similarly, the value of γ visibly shows an effect on the overall performance on various experiment while at a lower value improves the score that is achieved.

Finally, α values only show a perceivable effect while is set on its lowest rate (0.001) as the results for this one is lower in each variation of γ and ε discounted value, while at the higher values (0.005 and 0.01) the effect of the parameters it can't be notice visually on the grid.

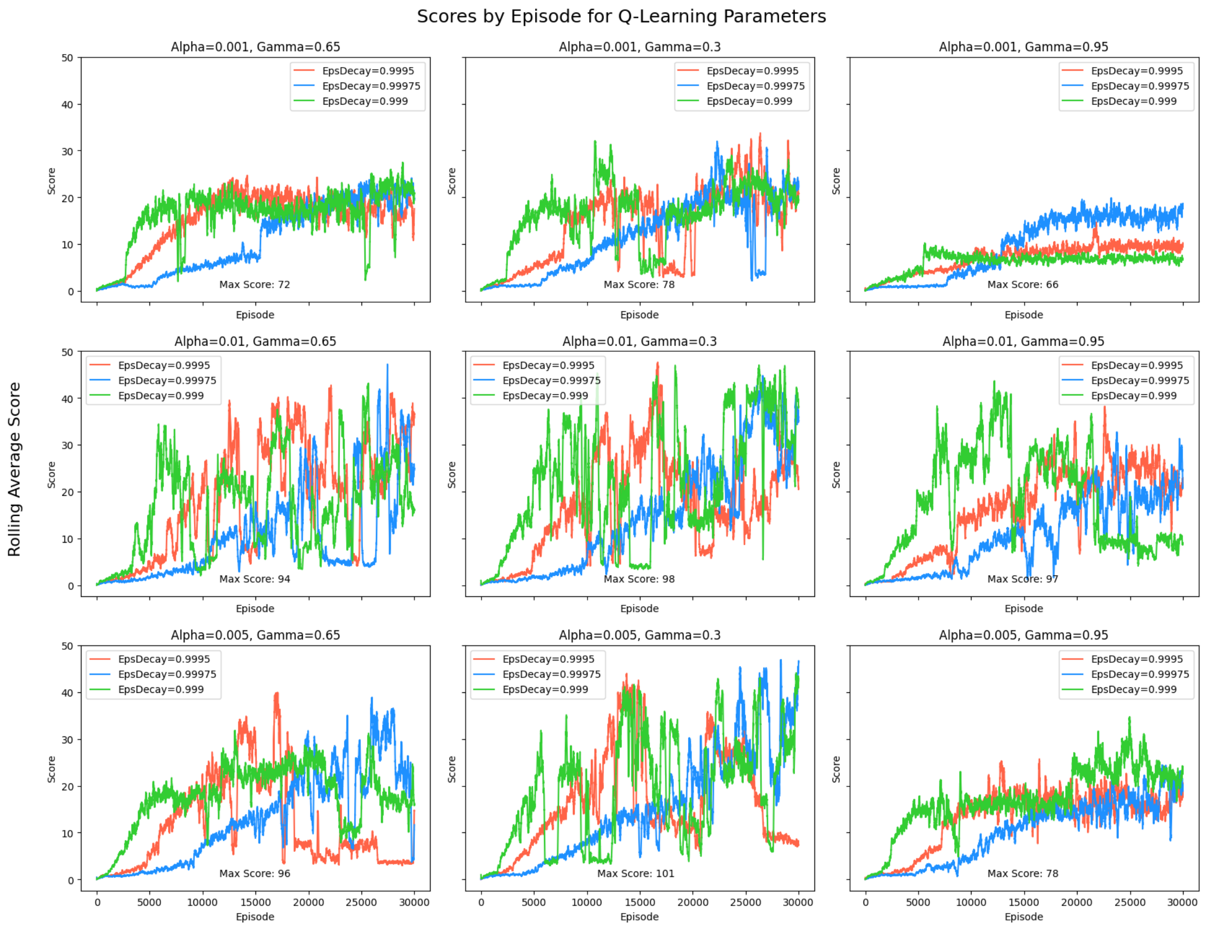

Figure 5: Grid of plots representing the results obtained with the new parameters. Is important to note that the scores are displayed as rolling average score. Then, the maximum scores described are not visible on the plots as they are contained within the rolling averages.

To conclude, the parameter optimization performed over the 27 new experiments provided useful insights about the process of the agent to learn to play the game successfully.

The optimal parameters found, translated to the agent's way of learning, indicate that the best performance (score) requires a slower pace to learn, and it improves while the short-term gain or the immediate reward provides a better incentive than the long run return.

Advanced Tasks: DQN

7. Implement Deep Q- Learning (DQN) with two improvements

In the basic tasks, the agent was aimed to evaluate a large size of Q tables. However, the number of entries that the Q-learning is going to update is far more than the total number of possible states. Therefore, the DQN is expected to have an improvement over the classical Q-learning.

Before implementing the DQN, there are a few things that needed to be changed in favor the agent's training. First, the state variables have been reduced from 12 to 11 that maps to only 3 possible actions. The Deep Q-Agent will only consider the dangers ahead because the snake advance in a single direction and the back of the snakes is always itself or an empty space. When the snakes grow, the agent can be confused at predicting the Q-value when there's potential danger behind the snake. Therefore, removing a redundant state can be more beneficial when using a neural network. For the actions space, the snake can only perform 3 actions when it is moving because it cannot go back and ate itself. Hence, removing 1 action can help reduce the ambiguities.

In this session, the DQN will be improved with two strategies: Double DQN and integration of Noisy nets.

First, the agent will be integrated with double DQN as the first improvement. The agent will be equipped with two networks of the same structure, one to generate predictions and the other one is used to evaluate the other. Using the DQN helps in reducing the risk of over estimations of the Q-values, which the ordinary DQN would have a higher chance to do. The two networks will have the same structures and same initial states, with 11 inputs, 256 hidden neurons and 3 output neurons. Each of them will have their own optimizers for their own model parameters.

After the use of the double DQN, the agent will be further improved with the use of the Noisy net. The aim of using this strategy here is to avoid or approach less or nearly no exploration moves at the time where the training is almost finish. From the previous stage, the epsilon decay is a fixed value, which doesn't have flexibility when the training phase is nearly completed. Mathematically, $\epsilon_{t+1} = \lambda\epsilon_t$, where λ is the decay constant. Then, considers every time step t and the initial value of exploration ϵ_0, $\epsilon_{t+1} = \lambda^{t+1}\epsilon_0$. However, the method of using this exponential decay can be naïve because that doesn't imply the agent have no rooms of improvement after a substantial amount of training. Although the state space is finite here, considering the actual distance of the food and moving when there are no rewards without eating the food, the estimation can be insufficient to represent the state, because of suboptimal actions. The noisy net layer helps to maintain the possibility of explorations while maintaining the expected optimal action. Mathematically,

$$y = (\mu_w + \sigma_w \odot \epsilon_w)\, x + (\mu_b + \sigma_b \odot \epsilon_b)$$

Where ⊙ represent the element-wise multiplication. The noisy net layer will be integrated along with the double DQN.

8. Quantitative and qualitative analysis

The quantitative analysis will assess the performance in terms of learning time, rolling average scores and the highest achievable scores.

Computational Cost

The result of the rolling average of mean episode time has a significant difference among the 3 candidate algorithms. First, due to the exploration now being integrated with the network itself, the agent with noisy net has the slowest episode time in the competition. Something worth notice here is that agent with noisy net has its lowest possible value for exploring without the use of network. The reason of this approach is to see the willingness of how the network in deciding to explore, with sacrifice of longer episode time. Interestingly, the double DQN agent have made relatively faster episode time comparing to the agent that only has a single DQN (figure 6). The difference became more significant when that came close the end of the training, where the mean episode time of the two agents can differed for almost 0.05 seconds. Adding randomness can increase the episode speed as that is skipped in the prediction of the neural network, but that also bring noises to the memories of training, where the movement decisions are independent of the states.

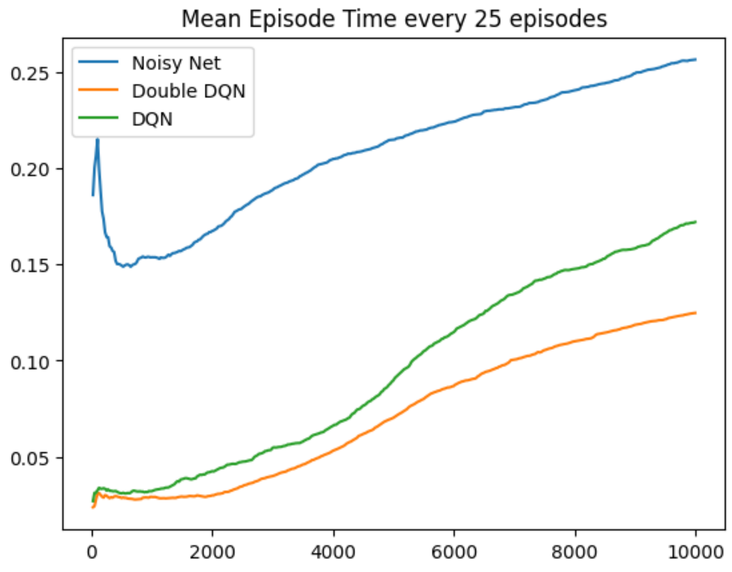

Figure 6: Plot displaying the mean time calculated for 25 episodes in the 3 different algorithms.

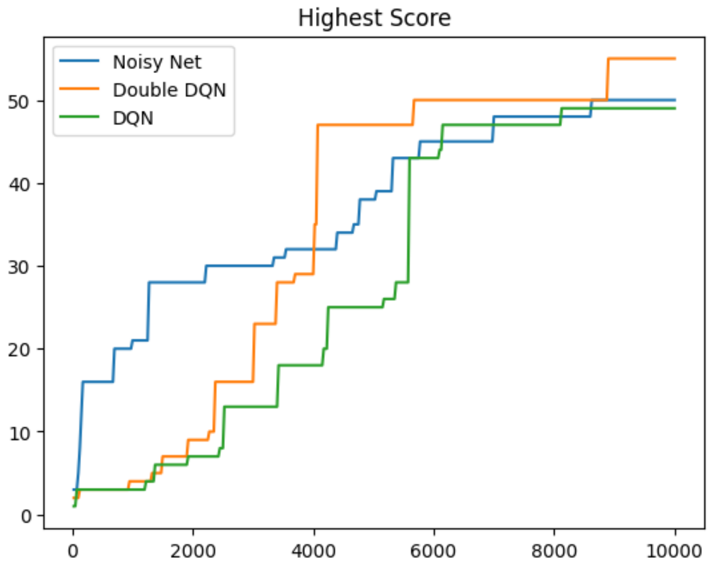

Figure 7: Plot displaying the highest score achieved by the 3 different algorithms.

Learning Strategies

The highest scores achieved by the candidates is an indicator of the time where the agents experience a breakthrough of their learning in short run. This also shows the characteristics of their learning approach. The agent with noisy net shows a steady

growth, in terms of its highest achievements (figure 7). Although it shows a rapid growth in the first 2000 episodes it slowed down afterwards and then keep finding a way through to another breakthroughs.
On the side of the DQNs, the double DQN has shown a large break through at around 4000 episodes. In contrast, the single DQN found its breakthrough nearly after 2000 episodes later. In general, DDQN and DQN agents possess a similar strategy – keep searching until a breakthrough is found. After overcoming the bottleneck, they gradually search for further improvements. The double DQN was outperforming among the other candidates which attained nearly a score of 60, while the noisy net agent only got 50 and DQN has the lowest record among the three.

Performance Characteristics
The rolling averages shows the trend of the agents learning performances. Through the learning episodes, 3 agents have shown significant volatility of their performance. The DQN agent has demonstrated their uncertainties enlarged when more episodes are trained, while the double DQN network shown better performance in comparison. Unlike the DQN, the noisy network tried to learn solely depending on the output of itself, which has gained a significant advantage in the first 4000 episodes.

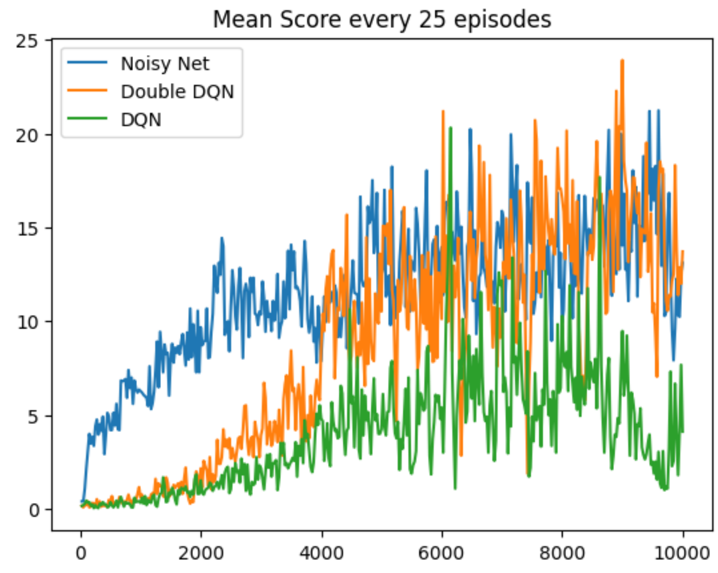

Figure 8: Plot of mean score achieved every 25 episodes for the 3 algorithms.

After the double DQN has found the breakthrough, the agents have a similar performance until the end of the training phase. The behavior of the noisy net shows that it can learn from the environment and be able to react. It also can learn when the epsilon randomness is absent for skipping the network decision. In contrast, the DQNs could heavily rely on the initial value of the epsilon. The agent could possibly find itself loss when the amount of exploration is not suitable. For example, making an agent greedy by setting the epsilon to 1 could generate a lot of noise for the training, which leads to inaccurate estimation of the Q-value. On the other extreme, giving a small epsilon would cause that the agent takes a long time to find a breakthrough, which would require longer time to find the optimal Q-values.

Conclusion
To summarize, the two improvements show their effects positively when combined to form the noisy net with double DQN. The double DQN has enhance the performances in comparison to the single DQN agent along with a faster computational time. On the other hand, the noisy net introduces a built-in ε–greedy policy within the decision network itself. With its nature of relying on the decision network, it shows a significant advantage when training with less episodes, especially when training resources is limited. In addition, the

performance volatility is generally less than DQN, meaning less noise. Hence, the improvements implemented could have a positively influenced the DQN agent performance.

9. Implementation of 2 RL algorithms with RLLib and Atari Learning Environment

Environment

In this session, the breakout-v4 is used for evaluating the 2 candidates of the RL algorithms. The agent of this environment has an objective of breaking the walls by paddling the ball to make it bounce back up. With each time it breaks a single piece of wall, it gains rewards of 1. However, if the ball lands to the bottom of the screen, the agent will lose 1 live, which will have 5 in total for an episode.

The agent will have 4 different actions in each timesteps: doing nothing, fire, move right, and move left. For the firing, the agent would have 5 chances for shooting. It cannot shoot after a ball has been emitted. The agent can move left or right, or doing nothing to paddle the ball when it is bouncing downward.

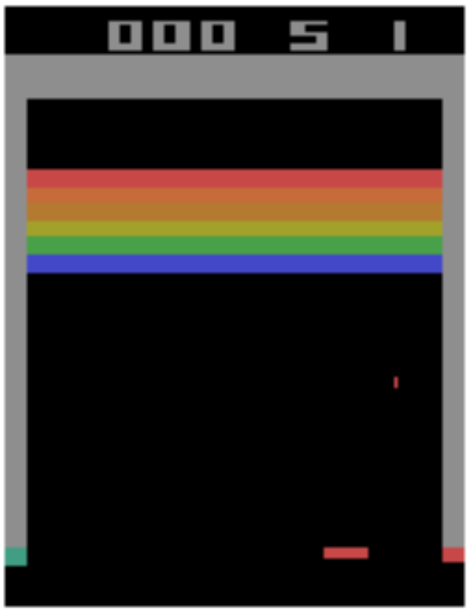

Fig.9 Interface of the Game Environement

There are several state variables here: position of the player, the ball direction and position and breakable wall positions. However, there are still random variables such as the bounce back direction of the ball by the paddle.

Choice of Algorithms and Description

Deep Q-Network

A DQN can be useful here because of its ability to capture states with image. With the benefits of memories, it would help to determine the direction of the ball by using memory buffers. Although this has been used in previous tasks, It can also be part of benchmarking with the other candidate

Advantage Actor-Critic (A2C)

The A2C framework is a relatively light-weight methods that adopt the use of asynchronous gradient descent for the optimizing the neural network. From [1], the A2C has shown a significant advantage on the computational performance in terms of training time. Comparing to DQN, its variant A3C has a significant advantage of its training time for at least 5 times.

10. Result Analysis

A2C Performance

The A2C network has a generally shorter time for training when comparing to the DQN of the RLlib. In terms of the episode length, it first start with more number of episodes, which can probably means it has more rewards. After the first iterations, the episode length starts to drop down significantly but gradually improving after few more training iterations. However, less episode length implies the less rewards the agent can get. However, from the graph it shows the strategy that the agent takes survival as a priority to improve its learning. After the big drop of the episode duration, there's a slightly positive trend for increasing the episode length. This can mean that the agent sometimes can find a breakthrough for its learning but it cannot convert the experience for future performances.

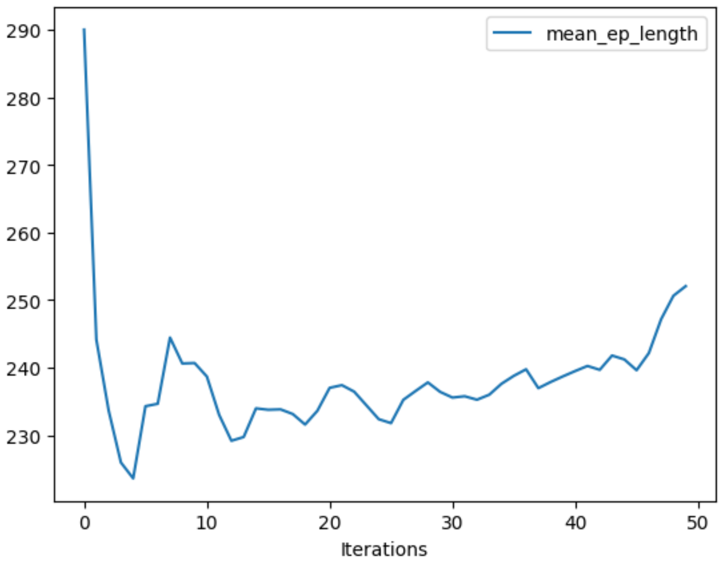

Fig.10 Mean Episode Length of each training iterations

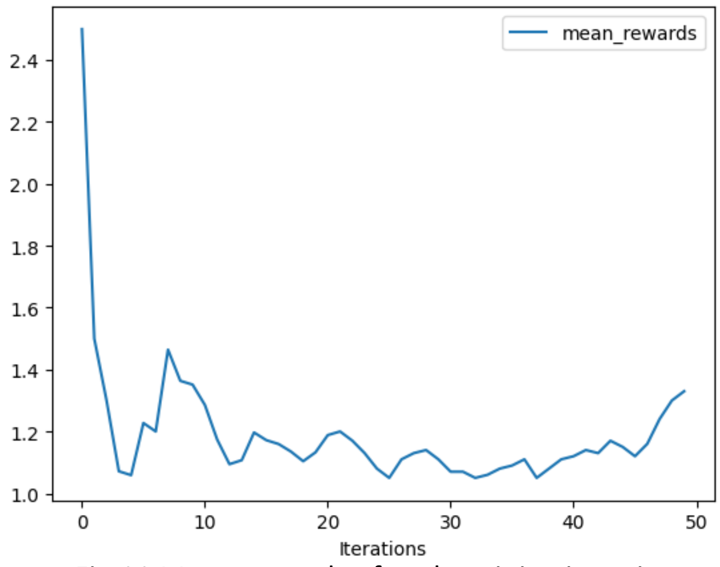

Fig.11 Mean rewards of each training iterations

References

[1] V. Mnih *et al.*, “Asynchronous Methods for Deep Reinforcement Learning,” 2016.
[2] V. Mnih *et al.*, “Playing Atari with Deep Reinforcement Learning,” 2013.

[3] GitHub Repo Link: https://github.com/chitlchow/inm707-deep-reinforcement-learning-project.git

plt.show()

Code printout

Main.py for Q-learning

```
import numpy as np
import pygame
from src.snake import Snake
from src.food import Food
from src.QLAgent import QLearner
import pandas as pd
import pickle
import time


# Helper Function
# Helper Function
def drawGrid(surface):
    for y in range(0, int(grid_height)):
        for x in range(0, int(grid_width)):
            if (x+y)%2 == 0:
                r = pygame.Rect((x*gridsize, y*gridsize), (gridsize,gridsize))
                pygame.draw.rect(surface,(93,216,228), r)
            else:
                rr = pygame.Rect((x*gridsize, y*gridsize), (gridsize,gridsize))
                pygame.draw.rect(surface, (84,194,205), rr)

def get_state(snake, food):
    # Initialize the array of the states
    states = []
    # 1: States of current direction
    actions = [up, down, right, left]
    for action in actions:
        states.append(int(snake.direction == action))

    # 2: Check food position relative to the snake
    food_direction_states = snake_food_direction(snake, food)

    for dir_state in food_direction_states:
        states.append(int(dir_state))

    # 3: Dangers ahead
    dangers = check_dangers(snake)
    for danger in dangers:
        states.append(int(danger))

    return tuple(states)

def check_dangers(snake):
    # head position
    x, y = snake.positions[0]
    danger_ahead = False
    danger_right = False
    danger_left = False

    actions = [up, right, down, left]
    # Check danger ahead
    front_pos = (x + snake.direction[0]*gridsize, y + snake.direction[1]*gridsize)
    # If it crash it selfs
    if  front_pos in snake.positions \
        or front_pos[0] >= screen_width \
        or front_pos[1] >= screen_height \
        or front_pos[0] < 0 \
        or front_pos[1] < 0:
        danger_ahead = True

    #  Check right
    current_dir_index  = actions.index(snake.direction)
    right_step = actions[(current_dir_index + 1) % 4]
    right_pos = (x + right_step[0]*gridsize, y + right_step[1]*gridsize)
    if right_pos in snake.positions \
        or right_pos[0] >= screen_width \
        or right_pos[1] >= screen_height \
        or right_pos[0] < 0 \
        or right_pos[1] < 0:
        danger_right = True
```

```python
    # Check left
    left_step = actions[(current_dir_index - 1) % 4]
    left_pos = (x + left_step[0]*gridsize, y + left_step[1]*gridsize)

    if left_pos in snake.positions \
        or left_pos[0] >= screen_width \
        or left_pos[1] >= screen_height \
        or left_pos[0] < 0 \
        or left_pos[1] < 0:
        danger_left = True

    dangers = [danger_ahead, danger_right, danger_left]
    return dangers

def snake_food_direction(snake, food):
    head_pos = snake.positions[0]
    food_pos = food.position

    # Compute the difference in coordinates
    delta_x = head_pos[0] - food_pos[0]
    delta_y = head_pos[0] - food_pos[0]

    # Prepare variables
    food_up = False
    food_down = False
    food_right = False
    food_left = False

    if delta_x < 0:
        food_right = True
        food_left = False
    elif delta_x > 0:
        food_right = False
        food_left = True
    if delta_y > 0:
        food_up = False
        food_down = True
    elif delta_y < 0:
        food_up = True
        food_down = False

    return [food_up, food_down, food_right, food_left]

def reset_game(snake, food):
    snake.length = 1
    snake.positions = [(screen_width/2, screen_height/2)]
    food.randomize_position()

screen_width = 400
screen_height = 400

gridsize = 20
grid_width = screen_width/gridsize
grid_height = screen_height/gridsize

up = (0,-1)
down = (0,1)
left = (-1,0)
right = (1,0)

score = 0
num_episodes = 10000
game_speed = 10000
# Main program for the game

alpha = 0.01
gamma = 0.95
epsilon_discount = 0.9992
time_steps = []
def game_loop(alpha, gamma, epsilon_discount):

    clock = pygame.time.Clock()
    # screen = pygame.display.set_mode((screen_width, screen_height), 0, 32)

    learner = QLearner(screen_width, screen_height, gridsize, alpha, gamma, epsilon_discount)
    # surface = pygame.Surface(screen.get_size())
    # surface = surface.convert()
    # drawGrid(surface)
    high = 0
    score = 0
    snake = Snake(screen_width, screen_height)
    food = Food(screen_width, screen_height)
    training_history = []
    # Score label on screen
    # score_display = pygame.font.SysFont("monospace",16)
    steps_without_food = 0
    episode = 1

    for episode in range(1, num_episodes + 1):
        # print(episode)
        score = 0
        steps_without_food = 0
        start = time.time()
        crash = False
        learner.episode_history = []
        learner.reward_history = []
        learner.clear_history()
        food.randomize_position()
        while not crash or steps_without_food == 1000:
            clock.tick(game_speed)
            reward = 0
            current_state = get_state(snake, food)
            action = learner.get_action(current_state)
            snake.turn(learner.actions[action])
            crash = snake.move()
            new_state = get_state(snake, food)

            if crash or steps_without_food == 1000:
                reward = -10

            if snake.get_head_position() == food.position:
                reward = 10
                score += 1
                if score > high:
                    high = score
                snake.length += 1
                steps_without_food = 0
                food.randomize_position()
            else:
                steps_without_food += 1
            # Update Q-tables
            learner.update_Q_valeus(current_state, new_state, action, reward)
        end = time.time()
        ep_time = end - start()
        learner.history.append(score)
        learner.update_epsilon()
        reset_game(snake, food)
        if episode % 25 ==0:
            print("EP: {}, Mean Score: {}, epsilon: {}, Highest: {}".format(
                episode,
                np.mean(np.array(learner.history)),
                learner.epsilon,
                high
            ))
            with open("training_history/episode-{}.pickle".format(episode), 'wb') as file:
                pickle.dump(learner.Q_tables, file)
            learner.history = []


            # Reset score
        training_history.append((episode, score, learner.epsilon))

    training_history = pd.DataFrame(training_history, columns=['Episodes', 'Score',
'Epsilon'])
    training_history.to_csv('result-dataset/training_history-({}, {}, {}).csv'.format(alpha,
gamma, epsilon_discount))

game_loop(alpha, gamma, epsilon_discount)
```

```
== Result Analysis

import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import glob
----
#fetching data from the first implementation
training_data = pd.read_csv('result-dataset/training_history-(0.01, 0.95, 0.999).csv')
----
#plot of the learning curve showing the changing in scores considering the mean over 25
episodes
max_reward = training_data['Score'].max()

fig, ax = plt.subplots()
ax.plot(training_data.index, training_data['Score'])
ax.text(training_data.index[-1], max_reward, f'Max: {max_reward:.2f}', ha='right',
va='bottom')
ax.set_xlabel('Episode (x25)')
ax.set_ylabel('Average Score')
ax.set_title('Learning Curve')
plt.show()
----
#plot of the epsilon decay through the 30000 episodes adding the episode where it reaches
the threshold
threshold_episode = training_data['Episodes'][np.argmax(training_data['Epsilon'] <= 0.001)]
threshold_episode = int(threshold_episode)

fig, ax = plt.subplots()
ax.plot(training_data['Episodes'], training_data['Epsilon'])
ax.text(threshold_episode, 0.001, f'Episodes: {threshold_episode}', ha='right', va='bottom')
ax.set_xlabel('Episode (x25)')
ax.set_ylabel('Epsilon')
ax.set_title('Epsilon Decay')
plt.show()
----
#consolidating the training history for every parameter tuning using glob

csv_pattern = 'result-dataset/*.csv'
csv_files = glob.glob(csv_pattern)

dfs = []
for csv_file in csv_files:
    df = pd.read_csv(csv_file)
    df['filename'] = csv_file
    dfs.append(df)
```

```
training = pd.concat(dfs, ignore_index=True)

#fetching the parameter values use in each training from the filename using regex method
values = training['filename'].str.extract(r'.*\((.*), (.*), (.*)\)\.csv')
training['alpha'] = values[0]
training['gamma'] = values[1]
training['epsilon_discount'] = values[2]
----
#fixing format of parameter values
training['alpha'] = training['alpha'].astype(float)
training['gamma'] = training['gamma'].astype(float)
training['epsilon_discount'] = training['epsilon_discount'].astype(float)
----

#validating max score obtained in parameter tuning and the parameters used
max_index = training['Score'].idxmax()
max_row = training.loc[max_index]
alpha = max_row['alpha']
gamma = max_row['gamma']
epsilon_discount = max_row['epsilon_discount']
print(training['Score'].max())
print(alpha, gamma, epsilon_discount)
----

----
#plot results for different parameter tuning using rolling averages.
alphas= training['alpha'].unique()
gammas = training['gamma'].unique()
epsilon_decays = training['epsilon_discount'].unique()
fig, axs = plt.subplots(3, 3, figsize=(20, 15), sharex=True, sharey=True)
colors = ['tomato', 'dodgerblue', 'limegreen']
for i, alpha in enumerate(alphas):
    for j, gamma in enumerate(gammas):
        max_score = -np.inf  # Initialize max score to negative infinity
        for k, epsilon_decay in enumerate(epsilon_decays):
            data = training[(training['alpha'] == alpha) & (training['gamma'] == gamma) &
                            (training['epsilon_discount'] == epsilon_decay)]
            data['Rolling_Avg'] = data['Score'].rolling(window=100, min_periods=1).mean()
            axs[i, j].plot(data['Episodes'], data['Rolling_Avg'], label=f'EpsDecay={epsilon_decay}',
                           color=colors[k])
            axs[i, j].set_title(f'Alpha={alpha}, Gamma={gamma}')

            # Update max score if current score is higher
            current_score = data['Score'].max()
            if current_score > max_score:
                max_score = current_score
            axs[i, j].legend()
            axs[i, j].set_title(f"Alpha={alpha}, Gamma={gamma}")
            axs[i, j].set_xlabel("Episode")
            axs[i, j].set_ylabel("Score")
            axs[i, j].legend()

            for ax in axs.flat:
                ax.set_ylim([0, max_score+10])
        axs[i, j].text(0.6, 0.05, f'Max Score: {max_score}',
                transform=axs[i, j].transAxes, ha='right', va='bottom', fontsize=10)

fig.suptitle('Scores by Episode for Q-Learning Parameters', y=0.925,x=0.5,fontsize=18)
fig.text(0.1, 0.5, 'Rolling Average Score', va='center', rotation='vertical', fontsize=16)

plt.subplots_adjust(wspace=0.1, hspace=0.2)
plt.show()
== Result Analysis
```

Main-DQN.py

```
import numpy as np
import pygame
from src.snake import Snake
from src.food import Food
from src.DQN import DQN_Agent
import pandas as pd
import pickle
import time

# Helper Function
def drawGrid(surface):
    for y in range(0, int(grid_height)):
        for x in range(0, int(grid_width)):
            if (x+y)%2 == 0:
                r = pygame.Rect((x*gridsize, y*gridsize), (gridsize,gridsize))
                pygame.draw.rect(surface,(93,216,228), r)
            else:
                rr = pygame.Rect((x*gridsize, y*gridsize), (gridsize,gridsize))
                pygame.draw.rect(surface, (84,194,205), rr)

def get_state(snake, food):
    # Initialize the array of the states
    states = []
    # 1: States of current direction
    actions = [up, down, right, left]
    for action in actions:
        states.append(int(snake.direction == action))

    # 2: Check food position relative to the snake
    food_direction_states = snake_food_direction(snake, food)

    for dir_state in food_direction_states:
        states.append(int(dir_state))

    # 3: Dangers ahead
    dangers = check_dangers(snake)
    for danger in dangers:
        states.append(int(danger))

    return tuple(states)
```

```python
def check_dangers(snake):
    # head position
    x, y = snake.positions[0]
    danger_ahead = False
    danger_right = False
    danger_left = False

    actions = [up, right, down, left]
    # Check danger ahead
    front_pos = (x + snake.direction[0]*gridsize, y + snake.direction[1]*gridsize)
    # If it crash it selfs
    if  front_pos in snake.positions \
        or front_pos[0] >= screen_width \
        or front_pos[1] >= screen_height \
        or front_pos[0] < 0 \
        or front_pos[1] < 0:
        danger_ahead = True

    #  Check right
    current_dir_index  = actions.index(snake.direction)
    right_step = actions[(current_dir_index + 1) % 4]
    right_pos = (x + right_step[0]*gridsize, y + right_step[1]*gridsize)
    if right_pos in snake.positions \
        or right_pos[0] >= screen_width \
        or right_pos[1] >= screen_height \
        or right_pos[0] < 0 \
        or right_pos[1] < 0:
        danger_right = True

    # Check left
    left_step = actions[(current_dir_index - 1) % 4]
    left_pos = (x + left_step[0]*gridsize, y + left_step[1]*gridsize)

    if left_pos in snake.positions \
        or left_pos[0] >= screen_width \
        or left_pos[1] >= screen_height \
        or left_pos[0] < 0 \
        or left_pos[1] < 0:
        danger_left = True

    dangers = [danger_ahead, danger_right, danger_left]
    return dangers

def snake_food_direction(snake, food):
    head_pos = snake.positions[0]
    food_pos = food.position

    # Compute the difference in coordinates
    delta_x = head_pos[0] - food_pos[0]
    delta_y = head_pos[0] - food_pos[0]

    # Prepare variables
    food_up = False
    food_down = False
    food_right = False
    food_left = False

    if delta_x < 0:
        food_right = True
        food_left = False
    elif delta_x > 0:
        food_right = False
        food_left = True
    if delta_y > 0:
        food_up = False
        food_down = True
    elif delta_y < 0:
        food_up = True
```

```
            food_down = False

        return [food_up, food_down, food_right, food_left]

    def reset_game(snake, food):
        snake.length = 1
        snake.positions = [(screen_width/2, screen_height/2)]
        food.randomize_position()

    screen_width = 400
    screen_height = 400

    gridsize = 20
    grid_width = screen_width/gridsize
    grid_height = screen_height/gridsize

    up = (0,-1)
    down = (0,1)
    left = (-1,0)
    right = (1,0)

    num_episodes = 10000
    game_speed = 1000

    alpha = 0.001
    gamma = 0.9
    epsilon_discount = 0.9992

    # Show graphics if set to True
    graphics = False

    # Main program for the game

    def game_loop(alpha, gamma, epsilon_discount):
        learner = DQN_Agent(learning_rate=alpha, gamma=gamma,
    epsilon_decay=epsilon_discount)
        snake = Snake(screen_width, screen_height)
        food = Food(screen_width, screen_height)
        if graphics:
            clock = pygame.time.Clock()
            screen = pygame.display.set_mode((screen_width, screen_height))
            surface = pygame.Surface(screen.get_size())
            surface.convert()
            drawGrid(surface)
        training_histories = []
        start = time.time()
        high = 0
        for episode in range(1, num_episodes + 1):

            crash = False
            score = 0
            steps_without_food = 0
            start = time.time()

            learner.episode_history = []
            learner.reward_history = []
            learner.clear_memory()
            game_over = False

            while not crash or steps_without_food == 1000:
                reward = 0

                # Agent making moves
                current_state = get_state(snake, food)
                action = learner.get_action(current_state)
                snake.turn(learner.actions[action])
                crash = snake.move()
                new_state = get_state(snake, food)
```

```python
            # Check if the snake eat the food
            if snake.get_head_position() == food.position:
                snake.length += 1
                score += 1
                if score >= high:
                    high = score
                    # Save the best model achieved new highest scores
                    learner.model.save_model(f_name='snapshot-score-
{}.pth'.format(high))
                reward = 10

                # Reset the counter
                steps_without_food = 0
                food.randomize_position()
            else:
                steps_without_food += 1

            # Case where episodes is going to be terminated
            if crash or steps_without_food == 1000:
                # Break the loop if crashing or timeout
                reward = -10
                game_over = True


            # Update the Q-values

            learner.reward_history.append(reward)

            # Memorize step
            learner.memorize(current_state,
                             reward=reward,
                             action=action,
                             new_state=new_state,
                             game_over=int(game_over))
            # learner.train_step(current_state, action, reward, new_state)
            learner.train_short_memories(current_state, action, reward, new_state,
int(game_over))
            if graphics:
                clock.tick(game_speed)
                drawGrid(surface)
                snake.draw(surface)
                food.draw(surface)
                screen.blit(surface, (0,0))
                pygame.display.update()

        learner.train_long_memories()
        learner.clear_episode_memories()
        end = time.time()
        ep_time = end - start

        # Reset the environment
        learner.score_history.append(score)
        learner.ep_time_history.append(ep_time)
        learner.update_epsilon()
        reset_game(snake, food)
        # print("EP: {}, Score: {}".format(episode, score))
        if episode % 25 == 0:
            print("EP: {}, Mean Score: {:.2f}, epsilon: {:.4f}, episode time:
{:.6f}, Highest: {}"\
                  .format(episode,
                          np.mean(np.array(learner.score_history)),
                          learner.epsilon,
                          np.mean(np.array(learner.ep_time_history)),
                          high))
            training_histories.append((
                episode, np.mean(np.array(learner.score_history)), learner.epsilon,
                np.mean(np.array(learner.ep_time_history)), high)
            )
            # Clear the history of last 25 episodes
            learner.score_history = []

    training_history = pd.DataFrame(training_histories, columns=['ep',
'mean_score_last25_ep', 'epsilon', 'ep_time', 'all_time_highest'])
    training_history.to_csv('result-dataset/training_history_dqn.csv')

game_loop(alpha, gamma, epsilon_discount)
```

Main-DQN.py

```python
import numpy as np
import pygame
from src.snake import Snake
from src.food import Food
from src.DDQN import double_DQN_agent
import pandas as pd
import pickle
import time

# Helper Function
def drawGrid(surface):
    for y in range(0, int(grid_height)):
        for x in range(0, int(grid_width)):
            if (x+y)%2 == 0:
                r = pygame.Rect((x*gridsize, y*gridsize), (gridsize,gridsize))
                pygame.draw.rect(surface,(93,216,228), r)
            else:
                rr = pygame.Rect((x*gridsize, y*gridsize), (gridsize,gridsize))
                pygame.draw.rect(surface, (84,194,205), rr)

def get_state(snake, food):
    # Initialize the array of the states
    states = []
    # 1: States of current direction
    actions = [up, down, right, left]
    for action in actions:
        states.append(int(snake.direction == action))

    # 2: Check food position relative to the snake
    food_direction_states = snake_food_direction(snake, food)

    for dir_state in food_direction_states:
        states.append(int(dir_state))

    # 3: Dangers ahead
    dangers = check_dangers(snake)
    for danger in dangers:
        states.append(int(danger))

    return tuple(states)

def check_dangers(snake):
    # head position
    x, y = snake.positions[0]
    danger_ahead = False
    danger_right = False
    danger_left = False

    actions = [up, right, down, left]
    # Check danger ahead
    front_pos = (x + snake.direction[0]*gridsize, y + snake.direction[1]*gridsize)
    # If it crash it selfs
    if  front_pos in snake.positions \
        or front_pos[0] >= screen_width \
        or front_pos[1] >= screen_height \
        or front_pos[0] < 0 \
        or front_pos[1] < 0:
        danger_ahead = True

    #  Check right
    current_dir_index  = actions.index(snake.direction)
    right_step = actions[(current_dir_index + 1) % 4]
    right_pos = (x + right_step[0]*gridsize, y + right_step[1]*gridsize)
    if right_pos in snake.positions \
        or right_pos[0] >= screen_width \
        or right_pos[1] >= screen_height \
        or right_pos[0] < 0 \
        or right_pos[1] < 0:
        danger_right = True

    # Check left
    left_step = actions[(current_dir_index - 1) % 4]
    left_pos = (x + left_step[0]*gridsize, y + left_step[1]*gridsize)

    if left_pos in snake.positions \
        or left_pos[0] >= screen_width \
        or left_pos[1] >= screen_height \
        or left_pos[0] < 0 \
        or left_pos[1] < 0:
        danger_left = True

    dangers = [danger_ahead, danger_right, danger_left]
    return dangers

def snake_food_direction(snake, food):
    head_pos = snake.positions[0]
    food_pos = food.position

    # Compute the difference in coordinates
    delta_x = head_pos[0] - food_pos[0]
    delta_y = head_pos[0] - food_pos[0]

    # Prepare variables
    food_up = False
    food_down = False
    food_right = False
    food_left = False

    if delta_x < 0:
        food_right = True
        food_left = False
    elif delta_x > 0:
        food_right = False
        food_left = True
    if delta_y > 0:
        food_up = False
        food_down = True
    elif delta_y < 0:
        food_up = True
        food_down = False

    return [food_up, food_down, food_right, food_left]

def reset_game(snake, food):
    snake.length = 1
    snake.positions = [(screen_width/2, screen_height/2)]
    food.randomize_position()

screen_width = 400
screen_height = 400

gridsize = 20
grid_width = screen_width/gridsize
grid_height = screen_height/gridsize

up = (0,-1)
down = (0,1)
left = (-1,0)
right = (1,0)

num_episodes = 10000
game_speed = 1000

alpha = 0.001
gamma = 0.9
```

```
epsilon_discount = 0.9992

# Show graphics if set to True
graphics = False

# Main program for the game

def game_loop(alpha, gamma, epsilon_discount):
    learner = double_DQN_agent(learning_rate=alpha, gamma=gamma,
epsilon_decay=epsilon_discount)
    snake = Snake(screen_width, screen_height)
    food = Food(screen_width, screen_height)
    if graphics:
        clock = pygame.time.Clock()
        screen = pygame.display.set_mode((screen_width, screen_height))
        surface = pygame.Surface(screen.get_size())
        surface.convert()
        drawGrid(surface)
    training_histories = []
    start = time.time()
    high = 0
    for episode in range(1, num_episodes + 1):

        crash = False
        score = 0
        steps_without_food = 0
        start = time.time()

        learner.episode_history = []
        learner.reward_history = []
        learner.clear_memory()
        game_over = False
        swap = False
        while not crash or steps_without_food == 100:
            reward = 0
            swap = not swap

            # Agent making moves
            current_state = get_state(snake, food)
            action = learner.get_action(current_state)
            snake.turn(learner.actions[action])
            crash = snake.move()
            new_state = get_state(snake, food)

            # Check if the snake eat the food
            if snake.get_head_position() == food.position:
                snake.length += 1
                score += 1
                if score >= high:
                    high = score
                    # Save the best model achieved new highest scores
                    learner.model1.save_model(f_name='model1-snapshot-score-
{}.pth'.format(high))
                    learner.model2.save_model(f_name='model2-snapshot-score-
{}.pth'.format(high))
                reward = 10

                # Reset the counter
                steps_without_food = 0
                food.randomize_position()
            else:
                steps_without_food += 1

            # Case where episodes is going to be terminated
            if crash or steps_without_food == 100:
                # Break the loop if crashing or timeout
                reward = -10
                game_over = True
            # Update the Q-values

            learner.reward_history.append(reward)

            # Memorize step
            learner.memorize(current_state,
                             reward=reward,
                             action=action,
                             new_state=new_state,
                             game_over=int(game_over))
            # learner.train_step(current_state, action, reward, new_state)
            learner.train_short_memories(current_state, action, reward, new_state,
int(game_over), swap)
            if graphics:
                clock.tick(game_speed)
                drawGrid(surface)
                snake.draw(surface)
                food.draw(surface)
                screen.blit(surface, (0,0))
                pygame.display.update()

        learner.train_long_memories(swap)
        learner.clear_episode_memories()
        end = time.time()
        ep_time = end - start

        # Reset the environment
        learner.score_history.append(score)
        learner.ep_time_history.append(ep_time)
        learner.update_epsilon()
        reset_game(snake, food)
        # print("EP: {}, Score: {}".format(episode, score))
        if episode % 25 == 0:
            print("EP: {}, Mean Score: {:.2f}, epsilon: {:.4f}, episode time:
{:.6f}, Highest: {}"\
                  .format(episode,
                          np.mean(np.array(learner.score_history)),
                          learner.epsilon,
                          np.mean(np.array(learner.ep_time_history)),
                          high))
            training_histories.append((
                episode, np.mean(np.array(learner.score_history)), learner.epsilon,
                np.mean(np.array(learner.ep_time_history)), high)
            )
            # Clear the history of last 25 episodes
            learner.score_history = []

    training_history = pd.DataFrame(training_histories, columns=['ep',
'mean_score_last25_ep', 'epsilon', 'ep_time', 'all_time_highest'])
    training_history.to_csv('result-dataset/training_history_ddqn.csv')

game_loop(alpha, gamma, epsilon_discount)
```

main-DDQN.py

```python
import numpy as np
import pygame
from src.snake import Snake
from src.food import Food
from src.DDQN import double_DQN_agent
import pandas as pd
import pickle
import time

# Helper Function
def drawGrid(surface):
    for y in range(0, int(grid_height)):
        for x in range(0, int(grid_width)):
            if (x+y)%2 == 0:
                r = pygame.Rect((x*gridsize, y*gridsize), (gridsize,gridsize))
                pygame.draw.rect(surface,(93,216,228), r)
            else:
                rr = pygame.Rect((x*gridsize, y*gridsize), (gridsize,gridsize))
                pygame.draw.rect(surface, (84,194,205), rr)

def get_state(snake, food):
    # Initialize the array of the states
    states = []
    # 1: States of current direction
    actions = [up, down, right, left]
    for action in actions:
        states.append(int(snake.direction == action))

    # 2: Check food position relative to the snake
    food_direction_states = snake_food_direction(snake, food)

    for dir_state in food_direction_states:
        states.append(int(dir_state))

    # 3: Dangers ahead
    dangers = check_dangers(snake)
    for danger in dangers:
        states.append(int(danger))

    return tuple(states)

def check_dangers(snake):
    # head position
    x, y = snake.positions[0]
    danger_ahead = False
    danger_right = False
    danger_left = False

    actions = [up, right, down, left]
    # Check danger ahead
    front_pos = (x + snake.direction[0]*gridsize, y + snake.direction[1]*gridsize)
    # If it crash it selfs
    if  front_pos in snake.positions \
        or front_pos[0] >= screen_width \
        or front_pos[1] >= screen_height \
        or front_pos[0] < 0 \
        or front_pos[1] < 0:
        danger_ahead = True

    #  Check right
    current_dir_index  = actions.index(snake.direction)
    right_step = actions[(current_dir_index + 1) % 4]
    right_pos = (x + right_step[0]*gridsize, y + right_step[1]*gridsize)
    if right_pos in snake.positions \
        or right_pos[0] >= screen_width \
        or right_pos[1] >= screen_height \
        or right_pos[0] < 0 \
        or right_pos[1] < 0:
        danger_right = True

    # Check left
    left_step = actions[(current_dir_index - 1) % 4]
    left_pos = (x + left_step[0]*gridsize, y + left_step[1]*gridsize)

    if left_pos in snake.positions \
        or left_pos[0] >= screen_width \
        or left_pos[1] >= screen_height \
        or left_pos[0] < 0 \
        or left_pos[1] < 0:
        danger_left = True

    dangers = [danger_ahead, danger_right, danger_left]
    return dangers

def snake_food_direction(snake, food):
    head_pos = snake.positions[0]
    food_pos = food.position

    # Compute the difference in coordinates
    delta_x = head_pos[0] - food_pos[0]
    delta_y = head_pos[0] - food_pos[0]

    # Prepare variables
    food_up = False
    food_down = False
    food_right = False
    food_left = False

    if delta_x < 0:
        food_right = True
        food_left = False
    elif delta_x > 0:
        food_right = False
        food_left = True
    if delta_y > 0:
        food_up = False
        food_down = True
    elif delta_y < 0:
        food_up = True
        food_down = False

    return [food_up, food_down, food_right, food_left]

def reset_game(snake, food):
    snake.length = 1
    snake.positions = [(screen_width/2, screen_height/2)]
    food.randomize_position()

screen_width = 400
screen_height = 400

gridsize = 20
grid_width = screen_width/gridsize
grid_height = screen_height/gridsize

up = (0,-1)
down = (0,1)
left = (-1,0)
right = (1,0)

num_episodes = 10000
game_speed = 1000

alpha = 0.001
gamma = 0.9
```

```python
epsilon_discount = 0.9992

# Show graphics if set to True
graphics = False

# Main program for the game

def game_loop(alpha, gamma, epsilon_discount):
    learner = double_DQN_agent(learning_rate=alpha, gamma=gamma,
epsilon_decay=epsilon_discount)
    snake = Snake(screen_width, screen_height)
    food = Food(screen_width, screen_height)
    if graphics:
        clock = pygame.time.Clock()
        screen = pygame.display.set_mode((screen_width, screen_height))
        surface = pygame.Surface(screen.get_size())
        surface.convert()
        drawGrid(surface)
    training_histories = []
    start = time.time()
    high = 0
    for episode in range(1, num_episodes + 1):

        crash = False
        score = 0
        steps_without_food = 0
        start = time.time()

        learner.episode_history = []
        learner.reward_history = []
        learner.clear_memory()
        game_over = False
        swap = False
        while not crash or steps_without_food == 100:
            reward = 0
            swap = not swap

            # Agent making moves
            current_state = get_state(snake, food)
            action = learner.get_action(current_state)
            snake.turn(learner.actions[action])
            crash = snake.move()
            new_state = get_state(snake, food)

            # Check if the snake eat the food
            if snake.get_head_position() == food.position:
                snake.length += 1
                score += 1
                if score >= high:
                    high = score
                    # Save the best model achieved new highest scores
                    learner.model1.save_model(f_name='model1-snapshot-score-
{}.pth'.format(high))
                    learner.model2.save_model(f_name='model2-snapshot-score-
{}.pth'.format(high))
                reward = 10

                # Reset the counter
                steps_without_food = 0
                food.randomize_position()
            else:
                steps_without_food += 1

            # Case where episodes is going to be terminated
            if crash or steps_without_food == 100:
                # Break the loop if crashing or timeout
                reward = -10
                game_over = True
            # Update the Q-values

            learner.reward_history.append(reward)

            # Memorize step
            learner.memorize(current_state,
                             reward=reward,
                             action=action,
                             new_state=new_state,
                             game_over=int(game_over))
            # learner.train_step(current_state, action, reward, new_state)
            learner.train_short_memories(current_state, action, reward, new_state,
int(game_over), swap)
            if graphics:
                clock.tick(game_speed)
                drawGrid(surface)
                snake.draw(surface)
                food.draw(surface)
                screen.blit(surface, (0,0))
                pygame.display.update()

        learner.train_long_memories(swap)
        learner.clear_episode_memories()
        end = time.time()
        ep_time = end - start

        # Reset the environment
        learner.score_history.append(score)
        learner.ep_time_history.append(ep_time)
        learner.update_epsilon()
        reset_game(snake, food)
        # print("EP: {}, Score: {}".format(episode, score))
        if episode % 25 == 0:
            print("EP: {}, Mean Score: {:.2f}, epsilon: {:.4f}, episode time:
{:.6f}, Highest: {}"\
                  .format(episode,
                          np.mean(np.array(learner.score_history)),
                          learner.epsilon,
                          np.mean(np.array(learner.ep_time_history)),
                          high))
            training_histories.append((
                episode, np.mean(np.array(learner.score_history)), learner.epsilon,
                np.mean(np.array(learner.ep_time_history)), high)
            )
            # Clear the history of last 25 episodes
            learner.score_history = []

    training_history = pd.DataFrame(training_histories, columns=['ep',
'mean_score_last25_ep', 'epsilon', 'ep_time', 'all_time_highest'])
    training_history.to_csv('result-dataset/training_history_ddqn.csv')

game_loop(alpha, gamma, epsilon_discount)
```

main-DDQN-NoisyNet.py

```python
import numpy as np
import pygame
from src.snake import Snake
from src.food import Food
from src.DDQN_NoisyNet import NoisyNet_agent
import pandas as pd
import pickle
import time

# Helper Function
def drawGrid(surface):
    for y in range(0, int(grid_height)):
        for x in range(0, int(grid_width)):
            if (x+y)%2 == 0:
                r = pygame.Rect((x*gridsize, y*gridsize), (gridsize,gridsize))
                pygame.draw.rect(surface,(93,216,228), r)
            else:
                rr = pygame.Rect((x*gridsize, y*gridsize), (gridsize,gridsize))
                pygame.draw.rect(surface, (84,194,205), rr)

def get_state(snake, food):
    # Initialize the array of the states
    states = []
    # 1: States of current direction
    actions = [up, down, right, left]
    for action in actions:
        states.append(int(snake.direction == action))

    # 2: Check food position relative to the snake
    food_direction_states = snake_food_direction(snake, food)

    for dir_state in food_direction_states:
        states.append(int(dir_state))

    # 3: Dangers ahead
    dangers = check_dangers(snake)
    for danger in dangers:
        states.append(int(danger))

    return tuple(states)

def check_dangers(snake):
    # head position
    x, y = snake.positions[0]
    danger_ahead = False
    danger_right = False
    danger_left = False

    actions = [up, right, down, left]
    # Check danger ahead
    front_pos = (x + snake.direction[0]*gridsize, y + snake.direction[1]*gridsize)
    # If it crash it selfs
    if  front_pos in snake.positions \
        or front_pos[0] >= screen_width \
        or front_pos[1] >= screen_height \
        or front_pos[0] < 0 \
        or front_pos[1] < 0:
        danger_ahead = True

    #  Check right
    current_dir_index  = actions.index(snake.direction)
    right_step = actions[(current_dir_index + 1) % 4]
    right_pos = (x + right_step[0]*gridsize, y + right_step[1]*gridsize)
    if right_pos in snake.positions \
        or right_pos[0] >= screen_width \
        or right_pos[1] >= screen_height \
        or right_pos[0] < 0 \
        or right_pos[1] < 0:
        danger_right = True

    # Check left
    left_step = actions[(current_dir_index - 1) % 4]
    left_pos = (x + left_step[0]*gridsize, y + left_step[1]*gridsize)

    if left_pos in snake.positions \
        or left_pos[0] >= screen_width \
        or left_pos[1] >= screen_height \
        or left_pos[0] < 0 \
        or left_pos[1] < 0:
        danger_left = True

    dangers = [danger_ahead, danger_right, danger_left]
    return dangers

def snake_food_direction(snake, food):
    head_pos = snake.positions[0]
    food_pos = food.position

    # Compute the difference in coordinates
    delta_x = head_pos[0] - food_pos[0]
    delta_y = head_pos[0] - food_pos[0]

    # Prepare variables
    food_up = False
    food_down = False
    food_right = False
    food_left = False

    if delta_x < 0:
        food_right = True
        food_left = False
    elif delta_x > 0:
        food_right = False
        food_left = True
    if delta_y > 0:
        food_up = False
        food_down = True
    elif delta_y < 0:
        food_up = True
        food_down = False

    return [food_up, food_down, food_right, food_left]

def reset_game(snake, food):
    snake.length = 1
    snake.positions = [(screen_width/2, screen_height/2)]
    food.randomize_position()

screen_width = 400
screen_height = 400

gridsize = 20
grid_width = screen_width/gridsize
grid_height = screen_height/gridsize

up = (0,-1)
down = (0,1)
left = (-1,0)
right = (1,0)

num_episodes = 10000
game_speed = 1000
```

```python
alpha = 0.001
gamma = 0.9
epsilon_discount = 0.9992

# Show graphics if set to True
graphics = False

# Main program for the game

def game_loop(alpha, gamma, epsilon_discount):
    learner = NoisyNet_agent(learning_rate=alpha, gamma=gamma,
epsilon_decay=epsilon_discount)
    snake = Snake(screen_width, screen_height)
    food = Food(screen_width, screen_height)

    if graphics:
        clock = pygame.time.Clock()
        screen = pygame.display.set_mode((screen_width, screen_height))
        surface = pygame.Surface(screen.get_size())
        surface.convert()
        drawGrid(surface)

    training_histories = []
    high = 0
    for episode in range(1, num_episodes + 1):
        score = 0
        steps_without_food = 0
        start = time.time()

        learner.episode_history = []
        learner.reward_history = []
        learner.clear_episode_memories()
        game_over = False
        swap = False
        while not crash or steps_without_food == 1000:
            reward = 0
            swap = not swap

            # Agent making moves
            current_state = get_state(snake, food)
            action = learner.get_action(current_state)
            snake.turn(learner.actions[action])
            crash = snake.move()
            new_state = get_state(snake, food)

            # Check if the snake eat the food
            if snake.get_head_position() == food.position:
                snake.length += 1
                score += 1
                if score >= high:
                    high = score
                    # Save the best model achieved new highest scores
                    learner.model1.save_model(f_name='noisy-model1-snapshot-score-
{}.pth'.format(high))
                    learner.model2.save_model(f_name='noisy-model2-snapshot-score-
{}.pth'.format(high))
                reward = 10

                # Reset the counter
                steps_without_food = 0
                food.randomize_position()
            else:
                steps_without_food += 1

            # Case where episodes is going to be terminated
            if crash or steps_without_food == 1000:
                # Break the loop if crashing or timeout
                reward = -10
```

```
                    game_over = True


                # Update the Q-values

                learner.reward_history.append(reward)

                # Memorize step
                learner.memorize(current_state,
                                 reward=reward,
                                 action=action,
                                 new_state=new_state,
                                 game_over=int(game_over))


                # learner.update_priorities(learner.priorities, td_error)

                # learner.train_step(current_state, action, reward, new_state)
                learner.train_short_memories(current_state, action, reward, new_state,
int(game_over), swap)

                if graphics:
                    clock.tick(game_speed)
                    drawGrid(surface)
                    snake.draw(surface)
                    food.draw(surface)
                    screen.blit(surface, (0,0))
                    pygame.display.update()

            learner.train_long_memories(swap)
            learner.clear_episode_memories()
            end = time.time()
            ep_time = end - start

            # Reset the environment
            learner.score_history.append(score)
            learner.ep_time_history.append(ep_time)
            learner.update_epsilon()
            reset_game(snake, food)
            # print("EP: {}, Score: {}".format(episode, score))
            if episode % 25 == 0:
                print("EP: {}, Mean Score: {:.2f}, epsilon: {:.4f}, episode time:
{:.6f}, Highest: {}"\
                      .format(episode,
                              np.mean(np.array(learner.score_history)),
                              learner.epsilon,
                              np.mean(np.array(learner.ep_time_history)),
                              high))
                training_histories.append((
                    episode, np.mean(np.array(learner.score_history)), learner.epsilon,
                    np.mean(np.array(learner.ep_time_history)), high)
                )
                # Clear the history of last 25 episodes
                learner.score_history = []

    training_history = pd.DataFrame(training_histories, columns=['ep',
'mean_score_last25_ep', 'epsilon', 'ep_time', 'all_time_highest'])
    training_history.to_csv('result-dataset/training_history_ddqn_noisy.csv')

game_loop(alpha, gamma, epsilon_discount)
```

QLAgent.py

```
import numpy as np
from src.snake import Snake
import random
import json

up = (0,-1)
down = (0,1)
left = (-1,0)
right = (1,0)

class QLearner:
    def __init__(self, display_width, display_height, grid_size, alpha,
gamma, epsilon_discount):
        self.display_width = display_width
        self.disply_height = display_height
        self.grid_size = grid_size

        # Definition of the states:
        #       Current Directions: up, down, right, left
        #       Food positions relative to the snake: up, down, right,
left
        #       Dangers around : up, down, right, left
        # Actions: Up, Down, right, and left
        # Consider taking the states as a binary variables, each will
have 2,
        # then you will have 2^12 *4 = 16384 states
        # I'm going to define the dimension of the table as the
following:
        # current_up: 0, 1
        # current_down: 0,1
        # current_right: 0, 1
        # current_left: 0, 1
        # food_up: 0, 1
        # food_down: 0, 1
        # food_right: 0, 1
        # food_left: 0, 1
        # danger_ahead
        # danger_right: 0, 1
        # danger_left: 0, 1
        # actions: up, down, left, right

        self.Q_tables = np.zeros((2,2,2,2,2,2,2,2,2,2,2,3))    # Depends
on the State and Actions

        # Q learning Parameters
        self.epsilon = 1.0
        self.alpha = alpha
        self.gamma = gamma
        self.epsilon_discount = epsilon_discount
        self.min_epsilon = 0.001
        # State and Action for Q values
        self.history = []

        # The action space, from state to actions
        self.actions = {
            0: 'straight',
            1: 'turn_right',
            2: 'turn_left'
        }

    def get_action(self, state):
        # Get random action by exploiting
        if random.random() < self.epsilon:
            return random.choice([0,1,2])
        # Return matrix of available actions
        return np.argmax(self.Q_tables[state])

    def update_epsilon(self):
        self.epsilon = max(self.epsilon * self.epsilon_discount,
self.min_epsilon)

    # Reset the learner
    def clear_history(self):
        self.history = []

    def update_Q_valeus(self, old_state, new_state, action, reward):
        self.Q_tables[old_state][action] = (1 - self.alpha) *
self.Q_tables[old_state][action] + \
                                          self.alpha  * (reward +
self.gamma * max(self.Q_tables[new_state]))
```

DQN.py

```python
import torch
from torch import nn
import os
import numpy as np
import random
from collections import deque

# Set up for the model to use any tensor processing device

device = torch.device('cpu')

class DQN_Agent():
    def __init__(self, learning_rate, gamma, epsilon_decay):
        # ANN model and parameters for training
        self.model = DQ_Network(11, 256, 3)
        self.eval_model = DQ_Network(11, 256, 3)

        self.learning_rate = learning_rate
        self.optimizer = torch.optim.Adam(self.model.parameters(),
lr=self.learning_rate)

        # Q learning parameters
        self.gamma = gamma
        self.epsilon = 0.7
        self.epsilon_decay = epsilon_decay
        self.min_epsilon = 0.01

        self.loss_func = nn.MSELoss()


        # Short memories
        self.short_memories_size = 0
        self.short_memories = deque(maxlen=10)
        self.episode_memories = deque()

        # Histories
        self.ep_time_history = []
        self.score_history = []
        self.reward_history = []
        self.actions = {
            0: 'straight',
            1: 'turn_right',
            2: 'turn_left'
        }

    def get_action(self, state):
        # Get random action by exploiting
        if random.random() < self.epsilon:
            return random.choice([0,1,2])
        # Return matrix of available actions
        else:
            state_vector = np.array(state)
            state_vector =
torch.from_numpy(state_vector).type(torch.Tensor).to(device)
            # print(torch.argmax(self.model(state_vector)).item())
            return torch.argmax(self.model(state_vector)).item()
    def update_epsilon(self):
        self.epsilon = max(self.epsilon* self.epsilon_decay,
self.min_epsilon)


    def train_short_memories(self, states, actions, rewards,
next_states, game_overs):
        self.train(states, actions, rewards, next_states, game_overs)

    def train_long_memories(self):
        if len(self.episode_memories) > 1000:
            sample = random.sample(self.episode_memories, 1000)
        else:
            sample = self.episode_memories
        states, actions, rewards, next_states, game_overs =
zip(*sample)
        self.train(states, actions, rewards, next_states, game_overs)

    def train(self, states, actions, rewards, next_states, game_overs):
        states = torch.tensor(states, dtype=torch.float).to(device)
        next_states = torch.tensor(next_states,
dtype=torch.float).to(device)
        rewards = torch.tensor(rewards, dtype=torch.float).to(device)
        actions = torch.tensor(actions, dtype=torch.long).to(device)

        if len(states.shape) == 1:
            states = torch.unsqueeze(states, 0)
            next_states = torch.unsqueeze(next_states, 0)
            rewards = torch.unsqueeze(rewards, 0)
            actions = torch.unsqueeze(actions, 0)
            game_overs = (game_overs, )

            # print(actions.size(0))
        q_pred = self.model(states)
        q_expected = q_pred.clone()
        # print(rewards)
        for i in range(len(game_overs)):
                # This is a (10, 1) vector
            if game_overs != 1:
                q_new = rewards[i] + self.gamma *
torch.max(self.model(next_states[i]))* (1 - game_overs[i])
            q_expected[i][actions[i]] = q_new
            # print(q_new)
        self.optimizer.zero_grad()
        loss = self.loss_func(q_pred, q_expected)
        # print(loss)
        loss.backward()
        self.optimizer.step()


    def memorize(self, current_state, action, reward, new_state,
game_over):
        self.short_memories.append((current_state, action, reward,
new_state, game_over))
        self.episode_memories.append((current_state, action, reward,
new_state, game_over))

    def clear_memory(self):
        self.short_memories = deque(maxlen=10)
        self.short_memories_size = 0
    def clear_episode_memories(self):
        self.episode_memories = deque()

class DQ_Network(nn.Module):
    # The network is to predict the Q-value - not the actions
    def __init__(self, input_dim, hidden_dim ,output_dim):
        super(DQ_Network, self).__init__()
        self.linear1 = nn.Linear(input_dim, hidden_dim)
        self.linear2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        out = torch.relu(self.linear1(x))
        out = self.linear2(out)
        # Returns the Q-value of each actions
        return out

    def save_model(self, f_name="model.pth"):
        model_folder_path = 'DRL_models'
        if not os.path.exists(model_folder_path):
            os.mkdir(model_folder_path)
        f_name = os.path.join(model_folder_path, f_name)
        torch.save(self.state_dict(), f_name)
```

DDQN.py

```
import torch
from torch import nn
import os
import numpy as np
import random
from collections import deque

# Set up for the model to use any tensor processing device

device = torch.device('cpu')

class double_DQN_agent():
    def __init__(self, learning_rate, gamma, epsilon_decay):
        # ANN model and parameters for training
        self.model1 = DQNetwork(11, 256, 3)
        self.model2 = DQNetwork(11, 256, 3)
        self.model2.load_state_dict(self.model1.state_dict())
        self.learning_rate = learning_rate
        self.model1_optimizer =
torch.optim.Adam(self.model1.parameters(), lr=self.learning_rate)
        self.model2_optimizer =
torch.optim.Adam(self.model2.parameters(), lr=self.learning_rate)

        # Q learning parameters
        self.gamma = gamma
        self.epsilon = 0.7
        self.epsilon_decay = epsilon_decay
        self.min_epsilon = 0.01

        self.loss_func = nn.MSELoss()


        # Short memories
        self.short_memories_size = 0
        self.short_memories = deque(maxlen=10)
        self.episode_memories = deque()

        # Histories
        self.ep_time_history = []
        self.score_history = []
        self.reward_history = []
        self.actions = {
            0: 'straight',
            1: 'turn_right',
            2: 'turn_left'
        }

    def get_action(self, state, swap=False):
        # Get random action by exploiting
        if random.random() < self.epsilon:
            return random.choice([0,1,2])
        # Return matrix of available actions
        else:
            state_vector = np.array(state)
            state_vector =
torch.from_numpy(state_vector).type(torch.Tensor).to(device)
            if swap:
```

```
            # print(torch.argmax(self.model(state_vector)).item())
                return torch.argmax(self.model2(state_vector)).item()
            return torch.argmax(self.model1(state_vector)).item()

    def update_epsilon(self):
        self.epsilon = max(self.epsilon* self.epsilon_decay,
self.min_epsilon)


    def train_short_memories(self, states, actions, rewards,
next_states, game_overs, swap):
        self.train(states, actions, rewards, next_states, game_overs,
swap)

    def train_long_memories(self, swap):
        if len(self.episode_memories) > 1000:
            sample = random.sample(self.episode_memories, 1000)
        else:
            sample = self.episode_memories
        states, actions, rewards, next_states, game_overs =
zip(*sample)
        self.train(states, actions, rewards, next_states, game_overs,
swap)

    def train(self, states, actions, rewards, next_states, game_overs,
swap):
        states = torch.tensor(states, dtype=torch.float).to(device)
        next_states = torch.tensor(next_states,
dtype=torch.float).to(device)
        rewards = torch.tensor(rewards, dtype=torch.float).to(device)
        actions = torch.tensor(actions, dtype=torch.long).to(device)

        if len(states.shape) == 1:
            states = torch.unsqueeze(states, 0)
            next_states = torch.unsqueeze(next_states, 0)
            rewards = torch.unsqueeze(rewards, 0)
            actions = torch.unsqueeze(actions, 0)
            game_overs = (game_overs, )

        if swap:
            q_pred = self.model2(states)
        else:
            q_pred = self.model1(states)

        q_expected = q_pred.clone()
        for i in range(len(game_overs)):
                # This is a (10, 1) vector
            if game_overs != 1:
                if swap:
                    # Evaluate by model 1
                    q_new = rewards[i] + self.gamma *
torch.max(self.model1(next_states[i]))* (1 - game_overs[i])
                else:
                    # Evaluate by model 2
                    q_new = rewards[i] + self.gamma *
torch.max(self.model2(next_states[i])) * (1 - game_overs[i])
            q_expected[i][actions[i]] = q_new

        if swap:
            self.model2_optimizer.zero_grad()
        else:
            self.model1_optimizer.zero_grad()
        loss = self.loss_func(q_pred, q_expected)
        # print(loss)
        loss.backward()
        if swap:
            self.model2_optimizer.step()
        else:
            self.model1_optimizer.step()


    def memorize(self, current_state, action, reward, new_state,
game_over):
        self.short_memories.append((current_state, action, reward,
new_state, game_over))
        self.episode_memories.append((current_state, action, reward,
new_state, game_over))

    def clear_memory(self):
        self.short_memories = deque(maxlen=10)
        self.short_memories_size = 0
    def clear_episode_memories(self):
        self.episode_memories = deque()

class DQNetwork(nn.Module):
    # The network is to predict the Q-value - not the actions
    def __init__(self, input_dim, hidden_dim ,output_dim):
        super(DQNetwork, self).__init__()
        self.linear1 = nn.Linear(input_dim, hidden_dim)
        self.linear2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        out = torch.relu(self.linear1(x))
        out = self.linear2(out)
        # Returns the Q-value of each actions
        return out

    def save_model(self, f_name="model.pth"):
        model_folder_path = 'DDQN_models'
        if not os.path.exists(model_folder_path):
            os.mkdir(model_folder_path)
        f_name = os.path.join(model_folder_path, f_name)
        torch.save(self.state_dict(), f_name)
```

DDQN_NoisyNet.py

```
import torch
from torch import nn
import torch.nn.functional as F
import os
import numpy as np
import random
from collections import deque

# Set up for the model to use any tensor processing device

device = torch.device('cpu')

class NoisyNet_agent():
    def __init__(self, learning_rate, gamma, epsilon_decay):
        # ANN model and parameters for training
        self.model1 = NoisyNet(11, 256, 3)
        self.model2 = NoisyNet(11, 256, 3)
        self.model2.load_state_dict(self.model1.state_dict())
        self.learning_rate = learning_rate
        self.model1_optimizer =
torch.optim.Adam(self.model1.parameters(), lr=self.learning_rate)
        self.model2_optimizer =
torch.optim.Adam(self.model2.parameters(), lr=self.learning_rate)

        # Q learning parameters
        self.gamma = gamma
        self.epsilon = 0.1
        self.epsilon_decay = epsilon_decay
        self.min_epsilon = 0.01

        self.loss_func = nn.MSELoss()


        # Short memories
        self.short_memories_size = 0
        self.short_memories = deque(maxlen=10)
        self.episode_memories = deque()

        # Histories
        self.ep_time_history = []
        self.score_history = []
        self.reward_history = []
        self.actions = {
            0: 'straight',
            1: 'turn_right',
            2: 'turn_left'
        }

    def get_action(self, state, swap=False):
        # Get random action by exploiting
        if random.random() < self.epsilon:
            return random.choice([0,1,2])
        # Return matrix of available actions
        else:
            state_vector = np.array(state)
            state_vector =
torch.from_numpy(state_vector).type(torch.Tensor).to(device)
```

```
            if swap:
            # print(torch.argmax(self.model(state_vector)).item())
                return torch.argmax(self.model2(state_vector)).item()
            return torch.argmax(self.model1(state_vector)).item()

    def update_epsilon(self):
        self.epsilon = max(self.epsilon* self.epsilon_decay,
self.min_epsilon)


    def train_short_memories(self, states, actions, rewards,
next_states, game_overs, swap):
        self.train(states, actions, rewards, next_states, game_overs,
swap)

    def train_long_memories(self, swap):
        if len(self.episode_memories) > 1000:
            sample = random.sample(self.episode_memories, 1000)
        else:
            sample = self.episode_memories
        states, actions, rewards, next_states, game_overs =
zip(*sample)
        self.train(states, actions, rewards, next_states, game_overs,
swap)

    def train(self, states, actions, rewards, next_states, game_overs,
swap):
        states = torch.tensor(states, dtype=torch.float).to(device)
        next_states = torch.tensor(next_states,
dtype=torch.float).to(device)
        rewards = torch.tensor(rewards, dtype=torch.float).to(device)
        actions = torch.tensor(actions, dtype=torch.long).to(device)

        if len(states.shape) == 1:
            states = torch.unsqueeze(states, 0)
            next_states = torch.unsqueeze(next_states, 0)
            rewards = torch.unsqueeze(rewards, 0)
            actions = torch.unsqueeze(actions, 0)
            game_overs = (game_overs, )

        if swap:
            q_pred = self.model2(states)
        else:
            q_pred = self.model1(states)

        q_expected = q_pred.clone()
        for i in range(len(game_overs)):
                # This is a (10, 1) vector
            if game_overs != 1:
                if swap:
                    # Evaluate by model 1
                    q_new = rewards[i] + self.gamma *
torch.max(self.model1(next_states[i]))* (1 - game_overs[i])
                else:
                    # Evaluate by model 2
                    q_new = rewards[i] + self.gamma *
torch.max(self.model2(next_states[i])) * (1 - game_overs[i])
            q_expected[i][actions[i]] = q_new
        if swap:
            self.model2_optimizer.zero_grad()
        else:
            self.model1_optimizer.zero_grad()
        loss = self.loss_func(q_pred, q_expected)
        # print(loss)
        loss.backward()
        if swap:
            self.model2_optimizer.step()
        else:
            self.model1_optimizer.step()


    def memorize(self, current_state, action, reward, new_state,
game_over):
        self.short_memories.append((current_state, action, reward,
new_state, game_over))
        self.episode_memories.append((current_state, action, reward,
new_state, game_over))

    def clear_memory(self):
        self.short_memories = deque(maxlen=10)
        self.short_memories_size = 0
    def clear_episode_memories(self):
        self.episode_memories = deque()

class NoisyNet(nn.Module):
    # The network is to predict the Q-value - not the actions
    def __init__(self, input_dim, hidden_dim ,output_dim):
        super(NoisyNet, self).__init__()
        self.layer1 = NoisyLinear(input_dim, hidden_dim)
        self.layer2 = NoisyLinear(hidden_dim, output_dim)

    def forward(self, x):
        out = torch.relu(self.layer1(x))
        out = self.layer2(out)
        # Returns the Q-value of each actions
        return out

    def save_model(self, f_name="model.pth"):
        model_folder_path = 'DDQN_models'
        if not os.path.exists(model_folder_path):
            os.mkdir(model_folder_path)
        f_name = os.path.join(model_folder_path, f_name)
        torch.save(self.state_dict(), f_name)


class NoisyLinear(nn.Module):
    def __init__(self, in_features, out_features, sigma_init=0.017):
        super(NoisyLinear, self).__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.sigma_init = sigma_init
        self.weight_mu = nn.Parameter(torch.FloatTensor(out_features,
in_features))
        self.weight_sigma =
nn.Parameter(torch.FloatTensor(out_features, in_features))
        self.bias_mu = nn.Parameter(torch.FloatTensor(out_features))
        self.bias_sigma = nn.Parameter(torch.FloatTensor(out_features))
```

```python
        self.reset_parameters()

    def reset_parameters(self):
        mu_range = 1 / self.in_features ** 0.5
        sigma_init = self.sigma_init / self.in_features ** 0.5
        self.weight_mu.data.uniform_(-mu_range, mu_range)
        self.weight_sigma.data.fill_(sigma_init)
        self.bias_mu.data.uniform_(-mu_range, mu_range)
        self.bias_sigma.data.fill_(sigma_init)

    def forward(self, x):
        weight_eps = torch.empty_like(self.weight_sigma).normal_()
        bias_eps = torch.empty_like(self.bias_sigma).normal_()
        weight = self.weight_mu + self.weight_sigma * weight_eps
        bias = self.bias_mu + self.bias_sigma * bias_eps
        return F.linear(x, weight, bias)
```

Sanke.py

```python
import random
import pygame
import sys

up = (0,-1)
down = (0,1)
left = (-1,0)
right = (1,0)
gridsize = 20

class Snake():
    def __init__(self, screen_width, screen_height):
        self.length = 1
        self.screen_width = screen_width
        self.screen_height = screen_height
        self.positions = [((screen_width/2), (screen_height/2))]
        self.direction = random.choice([up, down, left, right])
        self.color = (17, 24, 47)
        # Special thanks to YouTubers Mini - Cafetos and Knivens Beast
for raising this issue!
        # Code adjustment courtesy of YouTuber Elija de Hoog

    def get_head_position(self):
        return self.positions[0]

    def turn(self, point):
        if point == 'turn_right':
            if self.direction == up:
                self.direction = right
            elif self.direction == down:
                self.direction = left
            elif self.direction == right:
                self.direction = down
            elif self.direction == left:
                self.direction = up
        elif point == 'turn_left':
            if self.direction ==  up:
                self.direction = left
            elif self.direction == left:
                self.direction = down
            elif self.direction == down:
                self.direction = right
            elif self.direction == right:
                self.direction = up
        else:
            self.direction
        # if self.length > 1 and (point[0]*-1, point[1]*-1) ==
self.direction:
        #     return
        # else:
        #     self.direction = point

    # Return True if crash
    def move(self):
        cur = self.get_head_position()
        x,y = self.direction
        new = (
                (cur[0] + (x * gridsize)),
                (cur[1] + (y * gridsize))  # %
pygame.display.Info().current_h
        )
        # Reset if the snake eat itself
        if len(self.positions) > 2 and new in self.positions[2:]:
            return True

        elif new[0] > self.screen_width \
                or new[1] > self.screen_height \
                or new[0] < 0\
                or new[1] < 0:
            return True
        else:
            self.positions.insert(0,new)
            if len(self.positions) > self.length:
                self.positions.pop()

        return False


    def draw(self,surface):
        for p in self.positions:
            r = pygame.Rect((p[0], p[1]), (gridsize,gridsize))
            pygame.draw.rect(surface, self.color, r)
            pygame.draw.rect(surface, (93,216, 228), r, 1)
```

food.py

```
import random
import pygame

gridsize = 20
class Food():
    def __init__(self, screen_width, screen_height):
        self.position = (0,0)
        self.screen_width = screen_width
        self.screen_height = screen_height
        self.color = (223, 163, 49)
        self.randomize_position()

    def randomize_position(self):
        grid_width = self.screen_width / gridsize
        grid_height = self.screen_height / gridsize
        self.position = (random.randint(0, grid_width-1)*gridsize,
random.randint(0, grid_height-1)*gridsize)

    def draw(self, surface):
        r = pygame.Rect((self.position[0], self.position[1]),
(gridsize, gridsize))
        pygame.draw.rect(surface, self.color, r)
        pygame.draw.rect(surface, (93, 216, 228), r, 1)
```