*Team: George-Dorian Damian*

*Author: George-Dorian Damian*

*SNAKE*

DEEP Q-NETWORK

# Introduction

This project represents a Deep Q-Network (DQN) system that trains itself to play the Snake game. It is build using JavaScript (the game itself) and Python (the Back End where the DQN is located).

The Front End is responsible for showing the user how the system works in real-time, while the Back End makes all the calculations. These two communicate with each other using a WebSocket. Firstly, the Snake game provides to the Back End the current state of the game. The Python server uses the DQN to provide the best Q-Value action and sends a response with this action. Finally, the game sends with the help of the WebSocket a message containing some information relevant for the training of the DQN.

The DQN is trained using rewards and punishments. In our case the reward is given when the agent makes the snake eat a apple and the punishment is given when the agent makes the snake collide with a wall or itself.

# DESIGN

A diagram of a diagram

Description automatically generated with medium confidence

Figure 1 - Usecase Diagram

A diagram of a computer

Description automatically generated

Figure 2 - Front End Class Diagram

A diagram of a server

Description automatically generated

Figure 3 - Back End Class Diagram

A diagram of a company

Description automatically generated

Figure 4 - Sequence Diagram

# Front End

This application's Front End manages the part the user sees. Its only responsibility is drawing the Snake game, providing the current state of the game and the feedback for the DQN training.

The state is represented by 11 Boolean values. The first 3 values are true if the positions next to the snake's head are obstacles. There is a value for each direction (left, forward, right). The next 4 values cannot be true simultaneously, as they represent the current direction that the snake is moving. The last 4 values indicate what direction is the apple compared to the head of the snake.

The Front End firstly encodes the state and then it sends it through the WebSocket to the Agent. Then it waits for the response given, which represents a direction for the snake ('left', 'forward', 'right'). Based on this direction, the snake must turn. After the turn, the Front End sends again a message to the Agent with some data relevant for the training of the model: the reward/punishment for the DQN, a Boolean that is true when the game is over, the current score, the new state.

# MODEL

The code implements the core components of a Deep Q-Learning algorithm using PyTorch.

The Linear\_QNet class defines a simple neural network with one hidden layer used to approximate the Q-values. These Q-values represent the expected future reward for each action in each state.

The QTrainer class manages the training process, updating the Q-values through the Bellman equation. This is achieved by comparing the predicted Q-values with the target Q-values, which are computed using the reward received and the maximum Q-value of the next state. The training process uses the Adam optimizer and Mean Squared Error (MSE) loss to minimize the difference between predicted and target Q-values, enabling the model to learn the optimal policy for the agent.

# AGENT

This project implements a reinforcement learning agent using **Deep Q-Learning (DQN)** to interact with a simulated environment. The agent selects actions based on its current state, following an **epsilon-greedy** strategy that balances exploration (random actions) and exploitation (action selection based on the learned model). Over time, the agent improves its decision-making through experience replay, storing state-action-reward sequences and sampling from them to train the model.

The agent communicates with the client via **WebSocket**, receiving the state and sending the selected action. After receiving feedback (reward, done status, and score), the agent updates its model using the **Q-learning** algorithm. The model is trained to predict future rewards, and the agent improves its performance by minimizing the error between predicted and actual rewards.

The agent’s model is periodically saved every 50 games, and it tracks the highest score achieved. This setup enables the agent to learn optimal strategies for the environment over time, with real-time interaction and continual improvement based on feedback.

# Performance

For the Snake game with the 10x10 grid:

After training for 50 games the model started sensing the importance of eating apples.

After training for 150 games the highest score was 22.

After training for 500 games the highest score was 41.

# References

Snake game made in HTML, JavaScript: github.com/mohdriyaan/snake-game