Reinforcement Learning for Snake Game AI

**Project Report**

**Submitted By:**

M. Sami Faisal (22K-6009) Syed Muhammad Sufyan (22K-6012)

Muhammad Yasir (22K-5090)

**Course:** Artificial Intelligence

**Instructor:** Ms. Almas Ayesha Ansari

**Submission Date:** May 9, 2025

**Department of Computer Science**

**National University of Computing and Emerging Sciences**

**Contents**

[Abstract](#_bookmark0) 2

1. [Introduction](#_bookmark1) 2
2. [Methodology](#_bookmark2) 2
   1. [Reinforcement Learning Framework](#_bookmark3) . . . . . . . . . . . . . . . . . . . . . 2
   2. [Neural Network](#_bookmark4) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
   3. [Training Process](#_bookmark5) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
3. [Implementation](#_bookmark6) 3
4. [Results and Evaluation](#_bookmark7) 4
5. [Conclusion](#_bookmark8) 4
6. [References](#_bookmark9) 4

# Abstract

This project explores the application of reinforcement learning to train an intelligent agent to play the classic Snake game. Using a Q-learning approach with a neural network, the agent learns to navigate the game environment, avoid collisions, and maximize its score by consuming food. The implementation leverages Python with PyTorch for the neural network and Pygame for the game interface. The agent’s performance is evaluated based on its score and learning progress over multiple games, with results visualized through score trends. This report discusses the methodology, implementation details, results, and limitations of the approach.

# Introduction

The Snake game is a classic arcade game where a player controls a snake to eat food, grow in length, and avoid collisions with itself or the game boundaries. This project aims to develop an AI agent that learns to play Snake autonomously using reinforcement learning, specifically Q-learning with a deep neural network (Deep Q-Learning). The agent is trained to make decisions based on the game state, balancing exploration and exploitation to optimize its score. The report details the reinforcement learning framework, the neural network architecture, the training process, and the evaluation of the agent’s performance.

# Methodology

## Reinforcement Learning Framework

The agent is trained using Q-learning, a model-free reinforcement learning algorithm. The key components are:

* + - **State**: An 2D vector capturing the game state, including danger in- dicators (straight, right, left), current direction (left, right, up, down), and food location relative to the snake’s head.
    - **Action**: One of three actions: move straight, turn right, or turn left, represented as a one-hot vector [1,0,0], [0,1,0], or [0,0,1].
    - **Reward**: A reward of +10 for eating food, -10 for game over (collision or timeout), and 0 otherwise.
    - **Q-Value**: The expected cumulative reward for taking an action in a given state, approximated by a neural network.

The agent uses an epsilon-greedy policy, starting with high randomness (epsilon = 80) and decreasing it as the number of games increases, shifting from exploration to exploitation.

## Training Process

The training involves:

1. **Short-Term Memory**: After each game step, the agent trains on the tuple (state, action, reward, next state, done) to update Q-values.
2. **Long-Term Memory**: A replay buffer (deque with max size 100,000) stores ex- periences. Periodically, a mini-batch of 1,000 experiences is sampled for training.
3. **Bellman Update**: The target Q-value is computed as *Q*new = *r*+*γ·*max(*Q*(next state)) if not done, where *γ* = 0*.*9 is the discount factor.

The model is saved when a new high score is achieved.

# Implementation

The project is implemented in Python 3.8+ with the following libraries:

* Pygame: For rendering the game interface.
* PyTorch: For building and training the neural network.
* NumPy: For state vector computations.
* Matplotlib: For plotting training progress. The codebase consists of four modules:

1. snake\_game.py: Implements the Snake game logic, including movement, collision detection, and rendering.
2. model.py: Defines the Linear\_QNet and QTrainer for training.
3. agent.py: Implements the Agent class, handling state extraction, action selection, and memory management.
4. plotter.py: Visualizes the agent’s scores and mean scores over games.

The game runs at 25 frames per second, with a grid size of 640x480 pixels and a block size of 20 pixels.

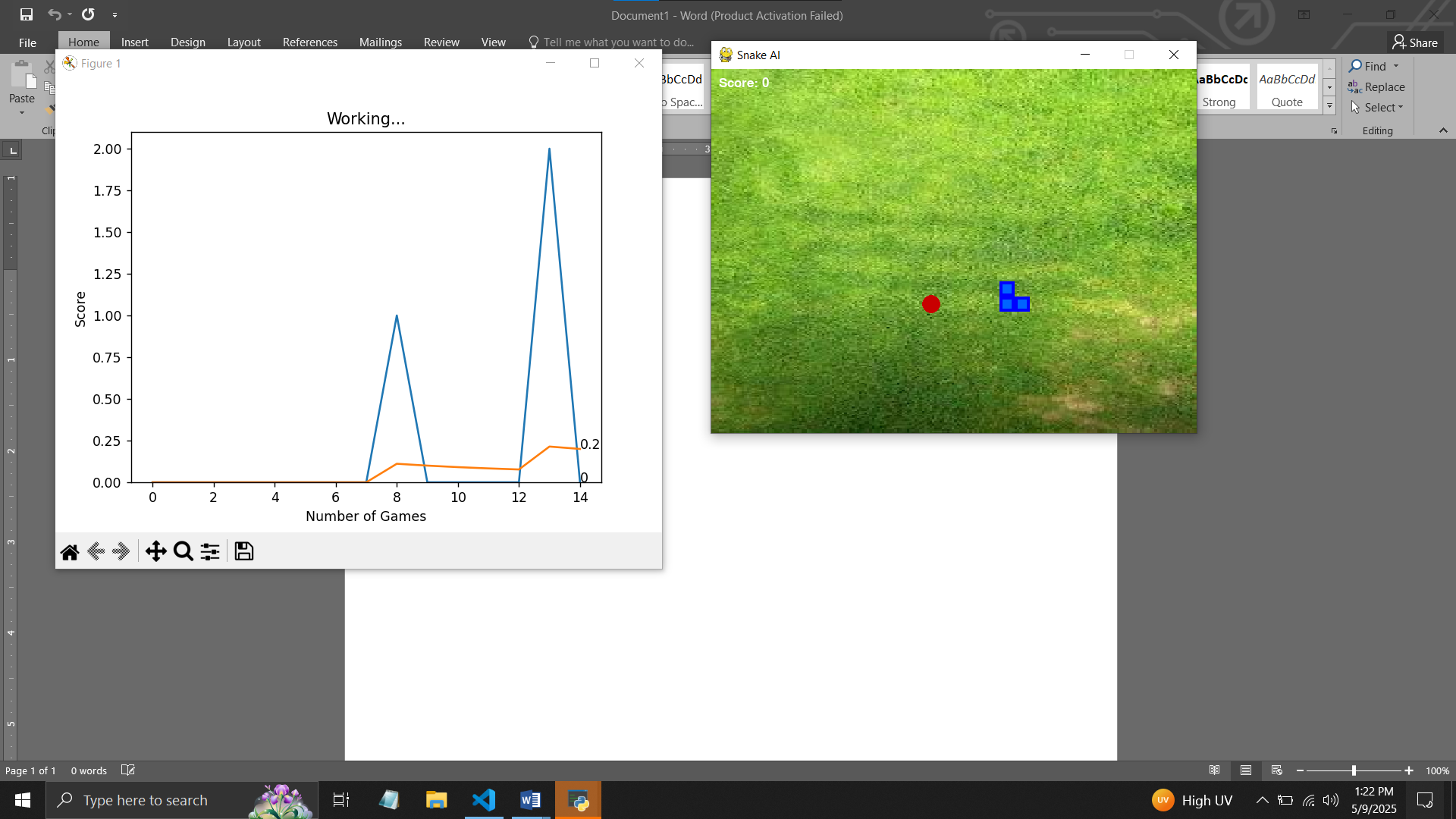
# Results, Analysis and Evaluation

The agent was trained over multiple games, with performance evaluated by the score (number of food items consumed) and the record high score. Key observations:

* **Initial Performance**: Early games showed low scores (0–2) due to random actions (high epsilon).
* **Learning Progress**: After 100 games, scores increased, with occasional peaks (e.g., 5–10), indicating improved decision-making.
* **Long-Term Trends**: Mean scores stabilized, reflecting consistent performance, though high variance persisted due to the game’s stochastic nature.

A plot of scores versus game number would show individual game scores as a volatile line and mean scores as a smoother, upward-trending curve. The highest recorded score improved over time, demonstrating the agent’s ability to learn effective strategies. Lim- itations include sensitivity to hyperparameters (e.g., epsilon decay rate, learning rate) and occasional instability in training due to the exploration-exploitation trade-off.

The below screenshot show the snake score when the games starts. It can be observed that in the start the snake is exploring a lot of paths before reaching to its goal (avoiding collisions and eating apples).



After 80-100 games the snake will be trained enough to play and score higher to at least 30-50 points in each game. This will be obtained from the data stored in the memory of the program. It estimates and then move to the most suitable location based on estimation from the Q-Learning and model training.



# Conclusion

This project successfully implemented a reinforcement learning agent to play the Snake game using Deep Q-Learning. The agent learned to navigate the environment, avoid collisions, and maximize its score, with performance improving over time. The use of a neural network to approximate Q-values and a replay buffer for stable training was effec- tive. Future improvements could include hyperparameter tuning, incorporating double Q-learning to reduce overestimation, or exploring alternative algorithms like Deep De- terministic Policy Gradients (DDPG). The project highlights the power of reinforcement learning in solving complex, dynamic tasks.

# References

1. Sutton, R. S., Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. MIT Press.
2. Mnih, V., et al. (2015). Human-level control through deep reinforcement learning.

*Nature*, 518(7540), 529–533.

1. PyTorch Documentation: <https://pytorch.org/docs/stable/index.html>
2. Pygame Documentation: <https://www.pygame.org/docs/>