Python snake game by RL

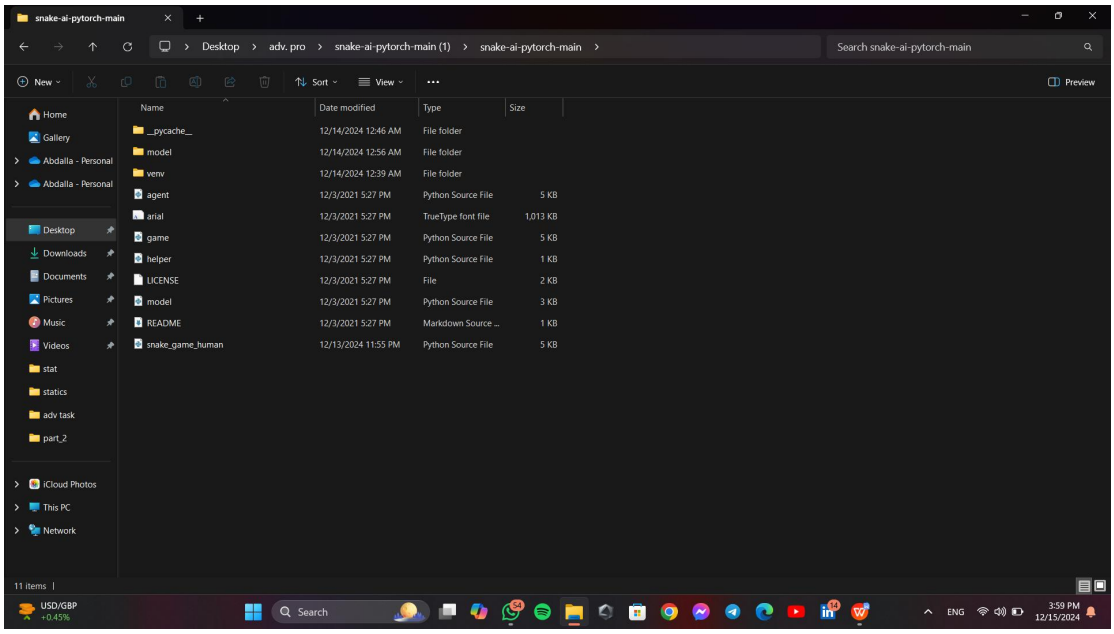
Code 1

Video:
[Python + PyTorch + Pygame Reinforcement Learning – Train an AI to Play Snake]

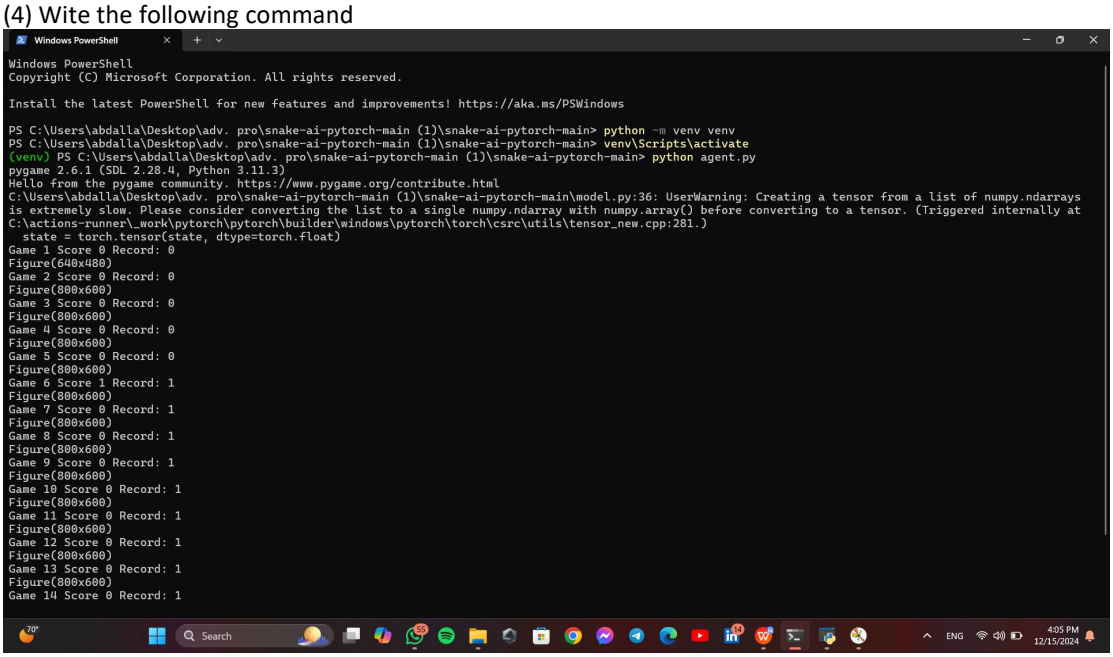
"gethub link of the code description of video"

How to run code :

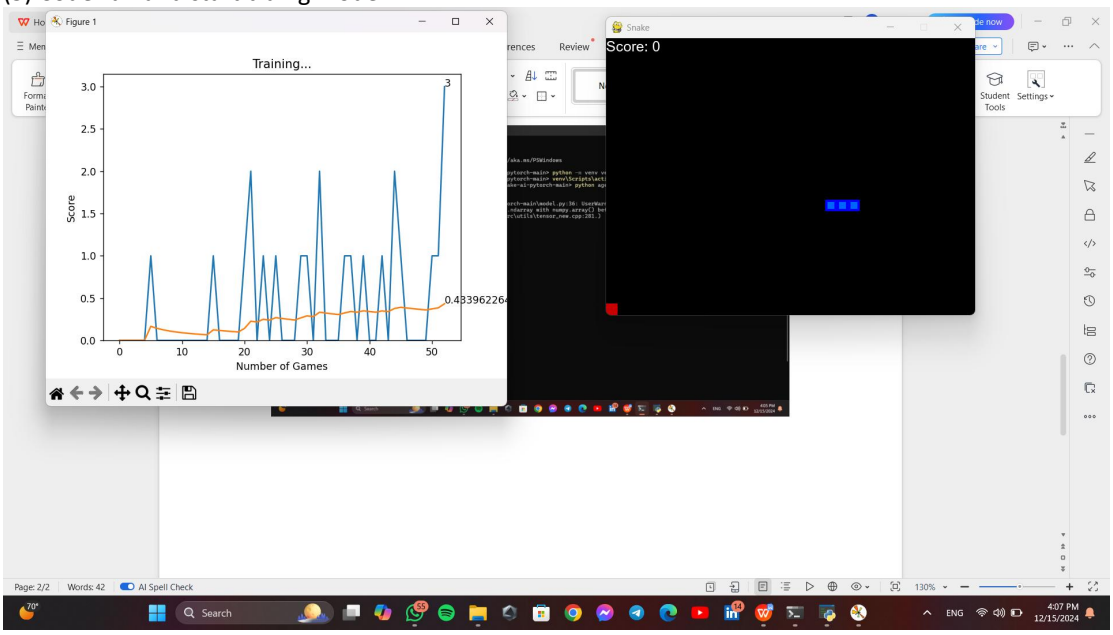
(1) Unzip code folder
(2) Open folder
(3) Right click and choose "open terminal"



(4) Wite the following command



(5) Code run and start traing model



Code workflow and structure

- **agent.py**: The file where the reinforcement learning agent is implemented.
- **arial.ttf**: A font file, possibly used for displaying text in the game.
- **game.py**: Contains the logic for the Snake game environment.
- **helper.py**: Likely includes utility functions to assist in various tasks.
- **model.py**: Defines the neural network architecture used by the agent.
- **snake_game_human.py**: A script for playing the Snake game manually, without AI

1. agent.py

This file implements the reinforcement learning agent using a Q-learning approach:

Agent class manages the training and decision-making of the agent.
Attributes:
n_games: Tracks the number of games played.
epsilon: Controls exploration (randomness).
gamma: Discount factor for future rewards.
memory: A deque to store experiences for replay.
model: An instance of Linear_QNet for decision-making.
trainer: An instance of QTrainer for optimizing the model.
Methods:
get_state: Extracts the current game state as input for the model.
remember: Stores experiences for replay.
train_long_memory: Uses a batch of experiences for training.
train_short_memory: Trains on the most recent experience.
get_action: Chooses an action based on exploration-exploitation trade-off.

2. game.py

Defines the Snake game environment, compatible with both AI and human control:

SnakeGameAI class:
Attributes:
Screen size (w, h), block size, and speed settings.
Snake and food positions.
Methods:
play_step: Progresses the game by one step based on the given action.
reset: Resets the game for a new episode.
_place_food: Randomly positions the food.
_move: Updates the snake's position based on the action.
_is_collision: Checks if the snake collides with walls or itself.
_update_ui: Handles rendering the game on the screen.

3. model.py

Defines the neural network used for the Q-learning agent:
Linear_QNet class:
A simple feedforward neural network with:
Input layer size: Represents the game state
Hidden layer size: Processes the input with non-linear transformations.
Output layer size: Corresponds to possible actions (up, down, left, right).
Methods:
forward: Performs the forward pass to predict Q-values.
save: Saves the trained model to a file.
load: Loads a previously saved model.

Workflow

First strep

the project structure is organized:

- agent.py - Reinforcement Learning (RL) agent.
- game.py - Game environment.
- model.py - Neural network for Q-learning.
- helper.py - Utility functions for data visualization and analysis.

Secound step

Game Environment Design

Create the game environment using Pygame in game.py:

- Define the snake's movement, food placement, and boundary conditions.
- Implement the logic for collision detection (with itself or walls).

Create a play_step() method to:

- Take an action (left, right, straight).
- Update the snake's position.
- Check for rewards (eating food).
- Determine if the game is over.

Third step

3. Agent Development

Design the RL agent in agent.py:

State Representation:

Define the game's state using features like the snake's position, food position, and direction.

Action Space:

Map actions (e.g., turn left, turn right, go straight) to integers.

Memory:

Use a deque to store past experiences (state, action, reward, next state).

Policy:

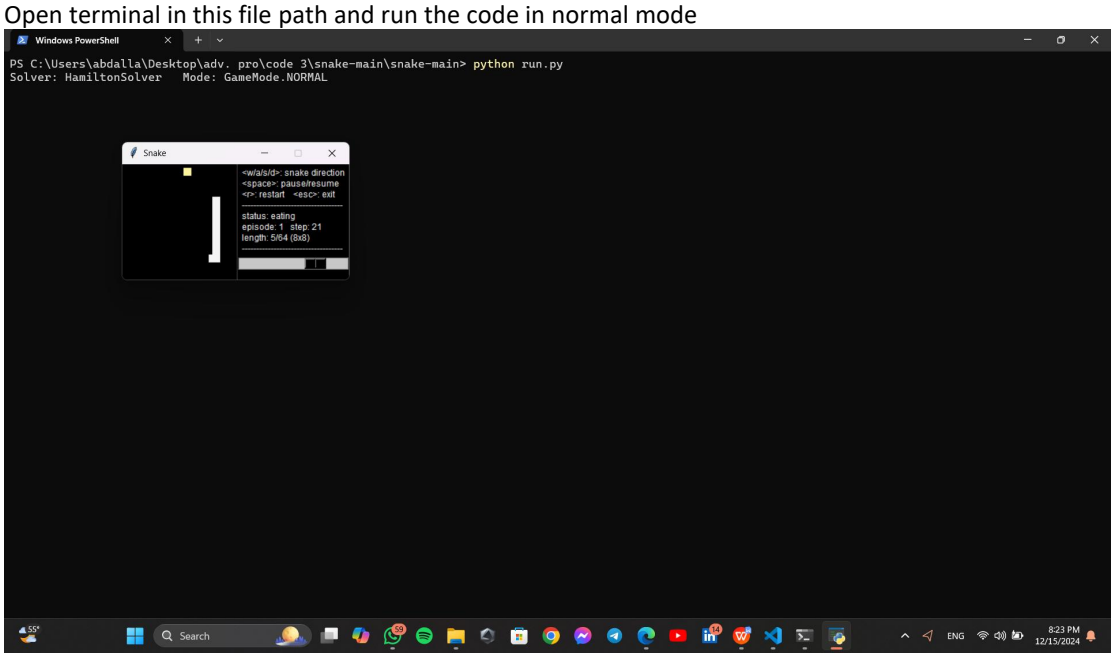
Implement an epsilon-greedy strategy to balance exploration (random actions) and exploitation (predicted actions).

Code 2

Video : null

How to run code

Open terminal in this file path and run the code in normal mode



Code workflow and structure

Code Structure
Root Level
run.py: Likely the main entry point to execute the project.
requirements.txt: Lists Python dependencies for the project.
README.md: Provides project documentation and usage instructions.
LICENSE: Specifies licensing terms.
docs/: Contains documentation, presentation files, and images to explain algorithms, solvers, and results.
tests/: Houses unit tests for various modules.
Core Modules

snake/: Main package for game logic and utilities.

game.py: Contains the main game mechanics.
gui.py: Implements the graphical interface of the game.

snake/base/: Provides foundational components.

direc.py: Handles directions for the snake.
map.py: Manages the game map.
point.py: Represents points (coordinates) in the grid.
pos.py: Position-related utilities.
snake.py: Represents the snake's data and behavior.

snake/solver/: Implements AI solvers for the game.

base.py: Base class for solvers.
greedy.py: Greedy algorithm-based solver.
hamilton.py: Hamiltonian cycle solver.
path.py: Pathfinding algorithms.
dqn/: Deep Q-Network (DQN)-based solver.
history.py: Tracks DQN training history.
logger.py: Handles DQN logging.
memory.py: Implements replay memory for reinforcement learning.
snakeaction.py: Encapsulates snake actions for DQN.

snake/util/: Provides utility modules.

sumtree.py: Implements a sum tree for efficient prioritization.
Tools
plot_dqn_compare.py: Compares performance of different DQN configurations.
plot_dqn_history.py: Visualizes DQN training history.
print_ckpt.py: Processes and prints checkpoints, likely related to DQN models.

Workflow

Game Initialization:

The game logic is initialized using game.py.
The GUI for the game is set up with gui.py.

Gameplay:

Base modules in snake/base/ handle map creation, snake movement, and collision detection.
Player or AI controls the snake to navigate the grid.

AI Solvers:

The solver package provides different strategies for automated gameplay.
Greedy: Aims for immediate goals (e.g., shortest path to food).
Hamiltonian: Ensures the snake follows a safe cyclic path.
DQN: Uses reinforcement learning to optimize gameplay.

Reinforcement Learning:

dqn/ modules manage training, memory replay, and action decisions.
Training performance can be analyzed using the tools in the tools/ directory.

Testing and Validation:

Automated tests for each module ensure reliability (found in tests/).

Visualization:

Results and comparisons of algorithms or training runs can be visualized using scripts in tools/ and images in docs/images/.

Features

Multiple AI Solvers:

Greedy, Hamiltonian, and DQN solvers demonstrate different approaches to automating the Snake game.

Reinforcement Learning:

Includes a DQN-based solver to learn optimal strategies through experience.

GUI Integration:

A user-friendly graphical interface for playing or observing AI strategies.

Customizability:

Modular structure allows easy modification or addition of new solvers and features.

Data Visualization:

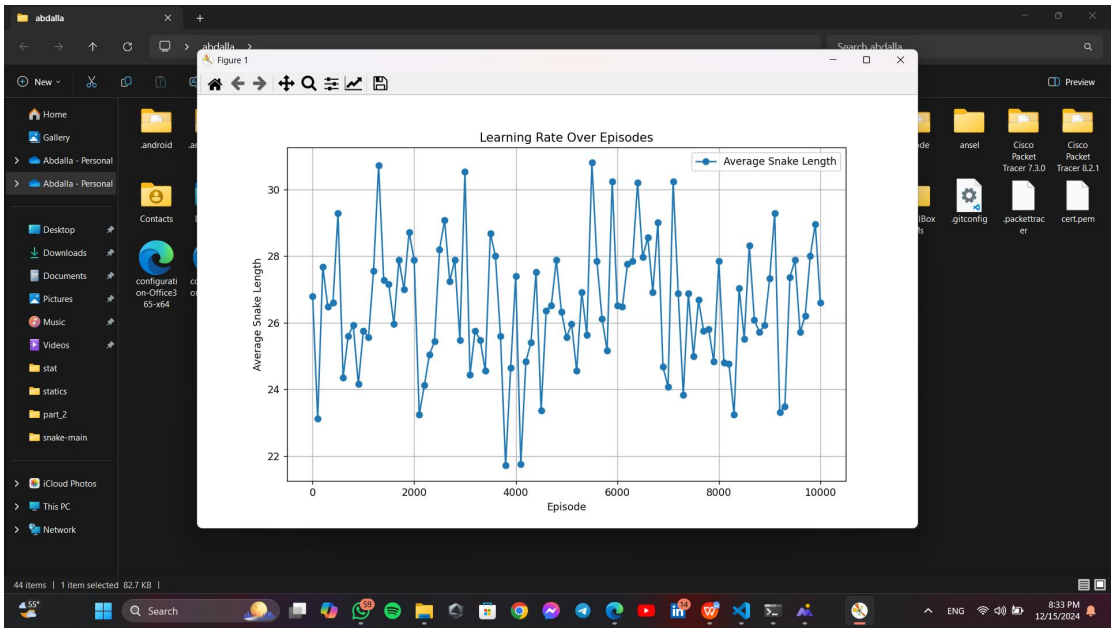
Provides scripts to analyze and visualize training performance and solver comparisons.

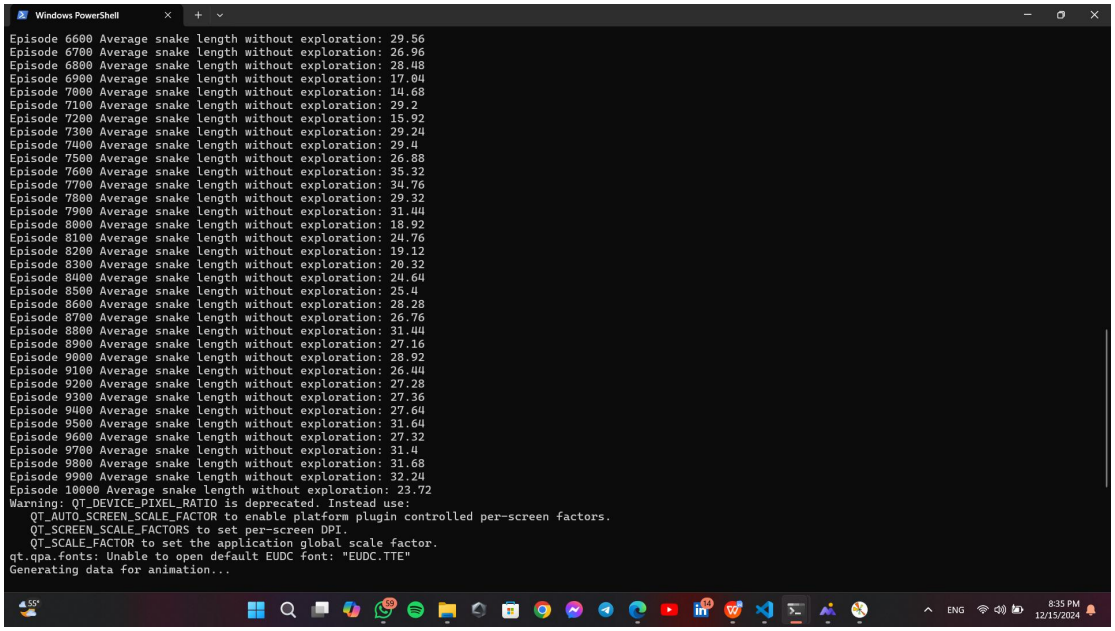
Unit Testing:

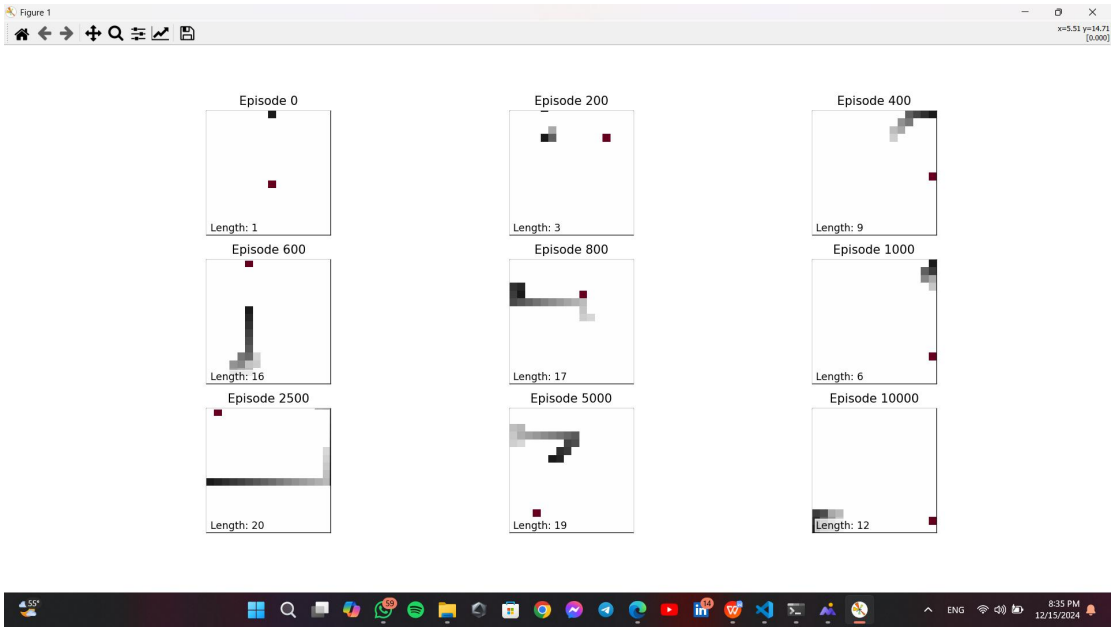
Comprehensive tests ensure the accuracy and stability of core components.

Code3

Video: null







Videos on snake game :

How to train AI to play snake using deep reinforcement learning

Python Game Programming Tutorial: Snake Game Part 1

Refances

https://github.com/patrickloeber/snake-ai-pytorch
Snake-Reinforcement-Learning
eidenyoshida/Snake-Reinforcement-Learning: Applying basic reinforcement learning principles using the Snake game in Python
eidenyoshida/Snake-Reinforcement-Learning: Applying basic reinforcement learning principles using the Snake game in Python
Rishit-katiyar/snake-reinforcement-learning: This repository contains a Python implementation of the classic Snake game using reinforcement learning techniques. The game is built with Pygame and includes features such as obstacle generation, Q-learning, and visualization of training progress.