WWW.IJRPR.COM

International Journal of Research Publication and Reviews

Journal homepage: www.ijrpr.com ISSN 2582-7421

Snake Game Model Train by AI

Rohit Kumar Gupta[1], Suraj Kumar Sahu[2], Sonu Swa[3], Prof. Kiran Yadu[4]

[1,2,3]Student, Computer Science and Engineering Department, Shri Shankaracharya Technical Campus, Bhilai, Chhattisgarh, India
[4]Guidance, (Asst. Professor), Shri Shankaracharya Technical Campus, Bhilai, Chhattisgarh, India
[1]Guptarohit.99390@gmail.com, [2]surajkumar.official008@gmail.com, [3]saw731214@gmail.com, [4]yadavkiran64@gmail.com

ABSTRACT

Artificial Intelligence (AI) and Reinforcement Learning (RL) are revolutionizing the gaming world, allowing agents to learn complex strategies and achieve superhuman performances. This project, "Teaching AI to Play the Snake Game Using Reinforcement Learning," aims to develop an autonomous agent capable of mastering the classic Snake game using RL techniques. The primary objective is to implement Q-learning and Deep Networks (DQN) to train an agent that learns optimal movement strategies through continuous interaction with the environment. The methodology involves setting up the Snake game environment, defining states, actions, and rewards, and training the agent through exploration and exploitation. A critical aspect is the design of a reward system that appropriately incentivizes behaviours leading to game success, such as eating food and avoiding collisions. The system is built using Python, Pygame for game simulation, and TensorFlow for neural network operations. The outcome of this project demonstrates that reinforcement learning techniques can effectively train agents to make intelligent decisions in dynamic environments. The agent's performance is evaluated based on its average score, survival time, and learning curve across thousands of training episodes. Real-world relevance extends to autonomous systems, robotic path planning, and AI decision-making in uncertain environments, showcasing the power and future potential of reinforcement learning applications.

1 INTRODUCTION

1.1 Introduction

The Snake Game is a popular arcade game in which the player controls a snake that grows in length by consuming food while avoiding collisions with itself and the boundaries. It poses a classic challenge of decision-making and spatial awareness, making it an ideal environment to apply and test Reinforcement Learning (RL). In this project, an AI agent is trained to play the Snake Game using Deep Q-Learning (DQL), a technique from deep reinforcement learning that leverages neural networks to approximate Q-values for actions in a given state.

1.2 Research Background

In recent years, reinforcement learning has achieved remarkable success in game environments such as Atari, Go, and Chess. With breakthroughs like DeepMind's DQN agent mastering classic Atari games, there is a strong motivation to explore similar strategies in other gaming environments. The Snake Game serves as a simplified simulation that incorporates important learning aspects such as environment modeling, policy optimization, and reward-based decision making. **[1, 6]**

1.3 Overview of Machine Learning

Machine Learning (ML) is a subfield of artificial intelligence concerned with the development of systems that can learn from and make decisions based on data. ML includes supervised, unsupervised, and reinforcement learning. Reinforcement learning is distinct in that it involves learning from interactions with an environment to maximize cumulative

reward. **[2], [3]**

1.4 Dimension Reduction Techniques

While not a primary component in this project, feature simplification plays a crucial role in making the state space manageable. Instead of raw pixel data, the game environment is abstracted into directional inputs, distances, and danger awareness to reduce complexity. **[4]**

1.4.1 Feature Selection

Key features selected for state representation include:

- Direction of the snake's movement Relative position of the food
- Immediate danger (front, left, right)
- Tail location proximity

1.5 Machine Learning Algorithms

The agent uses Deep Q-Learning (DQL), which involves:

- Experience Replay: Memory buffer to store state-action-reward transitions
- Q-Network: A neural network that estimates Q-values
- Epsilon-Greedy: Strategy for balancing exploration and exploitation. **[3], [4]**

1.5.1 Artificial Neural Network

A 3-layer feedforward neural network is used, with ReLU activation, taking an 11dimensional state vector and outputting Q-values for each of the four possible actions (up,

down, left, right). **[5], [7]**

1.6 Organization of Report

The report is divided into six chapters: Introduction, Literature Review, Problem Identification & Objectives, Methodology, Results and Discussion, and Conclusion with Future Scope. Each section details the components and implementation of the AI Snake Game project

1.7 Relevance of Reinforcement Learning in Games and AI

Reinforcement Learning (RL) offers a powerful framework for sequential decision-making problems where agents must learn strategies by interacting with dynamic environments. Its success in mastering games like Go, Chess, and Atari demonstrates its potential to tackle tasks involving high-dimensional input spaces, sparse rewards, and delayed consequences. The Snake Game provides a simplified simulation of such challenges, making it an ideal testing ground for RL algorithms. The outcomes from projects like Deep Q-Snake [Ray et al., 2024] and the DQN agent by Mnih et al. (2015) reflect how RL can transition from gaming to real-world tasks like robotics, path planning, and autonomous driving.

1.8 Contribution of the Project

This project contributes to the broader field of reinforcement learning by:

- Demonstrating the effectiveness of Deep Q-Learning in a grid-based environment.
- Showcasing how neural networks can approximate complex Q-value functions.
- Providing a modular and reproducible framework using open-source tools such as PyTorch and Pygame.
- Offering educational value for learners aiming to understand the practical implementation of AI models.

2. LITERATURE REVIEW

2.1 Literature Review

Reinforcement Learning (RL) has emerged as a powerful paradigm for training intelligent agents to make sequential decisions in dynamic environments. Inspired by behavioral psychology, RL involves an agent learning to achieve goals by interacting with an environment and receiving feedback in the form of rewards or penalties. This approach has found widespread applications in robotics, finance, recommendation systems, and especially in game playing—an ideal testbed for learning-based systems.

2.2 Classical Foundations of Reinforcement Learning

The foundational work by **Sutton and Barto (1998)** in *Reinforcement Learning: An Introduction* laid the theoretical groundwork for RL[1]. They introduced key concepts such as the **Markov Decision Process (MDP)**, **value functions**, and **policy optimization**, which serve as the basis for most modern RL algorithms. Their work has been instrumental in understanding how agents can learn optimal strategies in both discrete and continuous environments.

2.3 Game Playing and RL

Early success in RL applications to games was demonstrated by **TD-Gammon** (Tesauro, 1995), which learned to play backgammon at a superhuman level using temporal-difference learning. More recently, **Deep Q-Networks (DQN)** introduced by **Mnih et al (2015)** [2]**.** enabled AI to play Atari 2600 games directly from raw pixels, showcasing the power of deep learning combined with Q-learning. The DQN model demonstrated that neural networks could approximate action-value functions effectively, even in high-dimensional state spaces.

2.4 RL in Grid-Based Games

The Snake Game, being a grid-based environment with discrete actions and clear reward signals, serves as a simplified but challenging domain for RL research. Several studies have used the Snake Game to explore reinforcement learning principles. For instance, researchers have applied **Q-Learning**, a model-free RL algorithm, where the agent updates a Q-table based on state-action-reward transitions. However, as the state space grows, maintaining a Q-table becomes infeasible, prompting the use of **Deep Q-Learning**, which approximates the Q-values using neural networks.

Projects such as "AI Learns to Play Snake" by independent researchers and developers (e.g., on GitHub and academic forums) have illustrated the use of **CONVOLUTION NEURAL NETWORKS (CNNs)** to process the game screen, combined with experience replay and ε-greedy policies to balance exploration and exploitation. These implementations demonstrate the practical application of reinforcement learning concepts in simple, visual environments like Snake.

YEAR	AUTHOR	PURPOSE	TECHNIQUES	ACCURACY
2015	Mnih et al. (DeepMind)	Train AI to play Atari games	DQN	High
2020	Sentdex (Python Tutorials)	Applied DQN to Snake Game	DQL	Moderate
2021	Ayoosh Kathuria	Simplified DQL in Gym Environments	DQL, PyTorch	Moderate

3, PROBLEM IDENTIFICATION AND OBJECTIVE

3.1 Problem Statement

Can a Deep Q-Learning based AI agent learn to play Snake Game effectively, improving over time to reach higher scores and avoid collisions autonomously?

Traditional game-playing AI relies heavily on hardcoded rules and predefined logic, which limits adaptability and scalability across different environments. In the case of the Snake Game, manually coding a strategy to perform well becomes increasingly complex as the game evolves and the snake grows longer. Moreover, static approaches lack the ability to learn from mistakes or improve over time, making them inefficient for dynamic or unpredictable scenarios.

The core challenge lies in enabling an AI agent to learn optimal behavior in an environment with sparse rewards, high-dimensional state spaces, and delayed consequences for actions. The Snake Game, while simple in structure, presents a meaningful challenge in reinforcement learning due to its requirements for real-time decision-making, obstacle avoidance, and long-term planning. The need arises for a learning-based solution that allows the AI to autonomously discover strategies for survival and reward maximization without being explicitly programmed.

The challenge of enabling machines to play games like Snake is a microcosm of broader issues in artificial intelligence—such as decision-making under uncertainty, delayed gratification, and learning from interaction. The Snake Game presents a non-trivial problem due to:

- Non-linearity of environment : The game is deterministic but non-linear. A single wrong move may not immediately result in failure, but it may trap the agent in an unsolvable state a few steps later.

- No explicit modelling : There is no ground truth model of the game. The agent must construct its own understanding purely based on interactions.
- Exploration bottleneck : With no domain knowledge, the AI starts by exploring randomly. In a sparse reward setup, most initial actions yield no meaningful feedback, slowing down the learning curve.
- Catastrophic forgetting : The agent may sometimes "unlearn" good strategies during training if not managed properly with stable algorithms like Experience Replay and Target Networks.
- Lack of generalization : A rule-based agent will fail even if a minor change is made to the food placement rule, grid size, or reward scheme. A learning-based agent must overcome this and generalize across variations.

3.2 Objectives of Thesis

The primary objective of this project is to develop and train an AI agent to play the Snake Game using reinforcement learning techniques. This involves:

- Designing a suitable environment where the agent can interact with the game and receive feedback.
- Implementing a reinforcement learning algorithm (e.g., Q-Learning or Deep QNetworks) that allows the agent to learn from trial-and-error.
- Defining appropriate reward structures to guide the agent's behavior toward achieving high scores and survival.
- Evaluating the performance of the AI agent through training episodes, and improving it using techniques such as experience replay, exploration-exploitation strategies, and neural network optimization.

4. METHODOLOGY & APPLICATION MODEL

This section describes the different methods and materials used for this study i.e., research approach, research design and implementation, data collection and tools used for this study.

4.1 Research Approach

An experimental approach was adopted, with iterative training and evaluation cycles. Python and PyTorch were used to implement DQL, and the game was built using Pygame.

4.1.1 Reinforcement Learning and Q-Learning Method

Q-Learning Overview

Q-learning is a model-free reinforcement learning algorithm that aims to learn the optimal action-value function, denoted as Q(s, a), which represents the expected cumulative reward of taking action a in state s and following the optimal policy thereafter.

The core update rule for Q-learning is:

$$Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

where:

- α is the learning rate,
- γ is the discount factor,
- r is the reward received after taking action a in state s,
- s' is the next state,
- $\max_{a'} Q(s', a')$ is the maximum predicted Q-value for the next state.

Deep Q-Learning in This Project

Instead of using a Q-table (which is infeasible for large or continuous state spaces), this project uses a neural network (Linear_QNet) to approximate the Q-function. This approach is called Deep Q-Learning.

- The neural network takes the current state as input and outputs Q-values for all possible actions.
- The agent selects actions based on these Q-values, using an epsilon-greedy policy to balance exploration and exploitation.
- The agent stores experiences (state, action, reward, next state, done) in a replay memory (deque).

- During training, the agent samples mini-batches from this memory to update the neural network weights, reducing correlation between samples and improving learning stability.
- The loss function used is Mean Squared Error (MSE) between the predicted Q-values and the target Q-values computed using the Bellman equation.
- The optimizer used is Adam, which adapts the learning rate during training.

4.1.2 Neural Network Architecture

- The network (Linear_QNet) is a simple feedforward network with:
- An input layer matching the state size (11 features).
- One hidden layer with 256 neurons and ReLU activation.
- An output layer with 3 neurons corresponding to the possible actions.

4.1.3 Training Process

- The agent interacts with the SnakeGameAI environment by selecting actions and receiving rewards.
- Rewards are positive for eating food and negative for collisions or timeouts.
- The agent trains both on short-term memory (immediate experience) and long-term memory (experience replay).
- Training progress is visualized using matplotlib plots updated in real-time

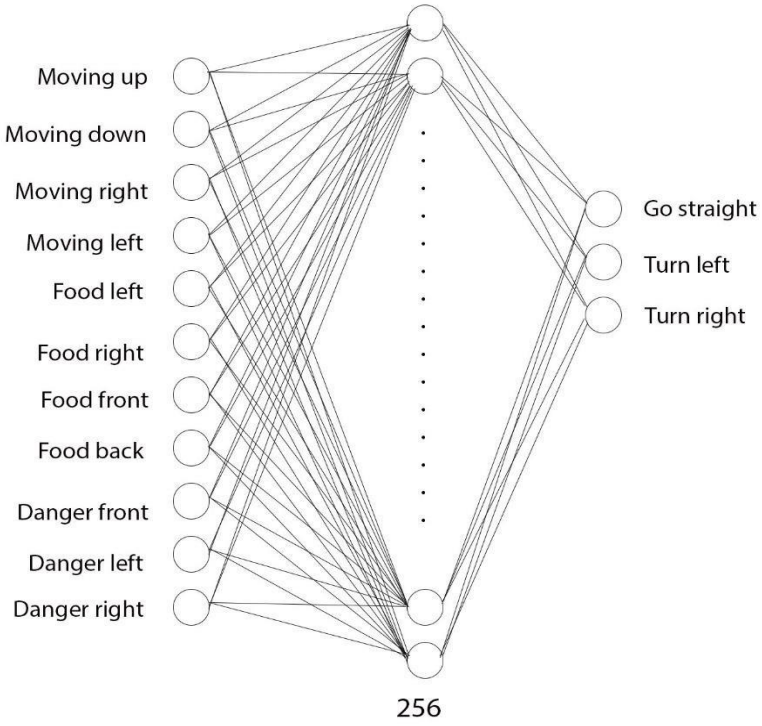

Fig 4.1 : visual representation of our neural network

4.1.4 Other Machine Learning / Deep Learning Methods

- Experience Replay: The agent stores past experiences and samples random batches for training, which helps break correlation between sequential data and improves learning efficiency.
- Epsilon-Greedy Policy: The agent selects random actions with probability epsilon to explore the environment and gradually reduces epsilon to exploit learned knowledge.

4.2 Research Design

The environment follows OpenAI Gym-like design, with an agent interacting through discrete actions and receiving a reward signal. Episodes end when the snake collides or reaches a termination condition.

Step-by-Step Explanation of the Training Loop

The implementation of the training loop is a critical part of teaching the reinforcement learning agent to play the Snake game effectively. The process begins with initializing the environment and the DQN agent. For each training episode, the snake is placed in a new starting position, and the game loop runs until the snake collides and the game ends.

At each time step during an episode, the following sequence occurs:

1. **State Collection:** The current state of the environment is captured as an 11-dimensional vector.
2. **Action Selection (ε-greedy policy):**
    - With probability ε (epsilon), a random action is chosen (exploration).
    - With probability 1−ε, the action that maximizes the predicted Q-value is chosen (exploitation).

3. Environment Update:

- The selected action is executed in the environment.
- A new state is generated, and a reward is assigned based on the action's outcome.

4. Memory Storage:

•The transition (state, action, reward, new state, done flag) is stored in the replay memory.

5. Training Step:

- A random batch of transitions is sampled from the replay memory.
- The DQN model is updated by minimizing the difference between predicted and target Q-values.

6. Target Network Update:

•Periodically, the weights of the target network are updated to match the primary network, helping to stabilize training.

7. End of Episode:

- The final score (length of the snake) and reward are recorded.
- ε (exploration rate) is decayed to gradually shift the balance from exploration to exploitation.

This cycle repeats for thousands of episodes until the agent achieves satisfactory performance.

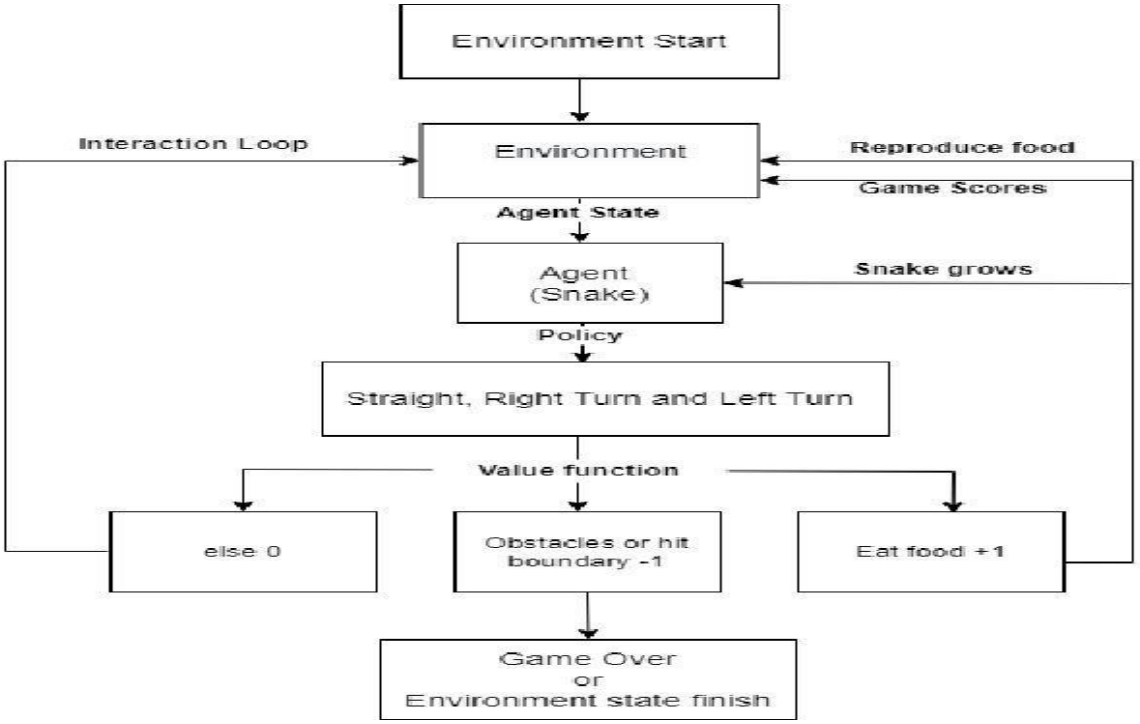

Fig. 4.2 : Project flow chart

4.3 Data Collection and Description

The environment generates its own data through simulation. The state vectors, actions taken, rewards received, and resulting states are stored in a replay buffer for training the agent.

4.4 Performance Metrics

- Maximum and average score
- Average reward per episode
- Steps survived per episode

4.5 Data Pre-processing

The raw game state (positions, directions, food) is converted to an 11-dimensional vector that is fed into the neural network. Normalization is not required as values are mostly binary or bounded integers.

4.6 Tools Used

- Python 3.10
- Pytorch: A machine learning framework that helps create neural networks. The core deep learning library used to build and train the neural network model that approximates the Q-function in reinforcement learning. It provides tensor operations, automatic differentiation, and optimization algorithms.
- Pygame: Python module designed for video games. This library is used to create the graphical interface of the Snake game. It handles rendering the game window, drawing the snake and food, and capturing user input (keyboard events). It is essential for the interactive gameplay experience.
- Matplotlib: Helps plot and create visualizations of data Used to plot the training progress, showing scores and mean scores over time to visualize how well the AI agent is learning.
- random: A standard Python library used to generate random positions for the food in the game. This randomness is crucial for game variability.
- IPython: Used to display matplotlib plots interactively during training.

4.7 Data Reduction

State vector reduces the environment complexity by abstracting only essential components such as danger zones, direction, and food location.

4.8 Descriptive Statistics and Exploratory Data Analysis

Training logs provide insights into the learning process. Metrics such as running average score and total rewards are plotted to evaluate the performance curve of the agent over time.

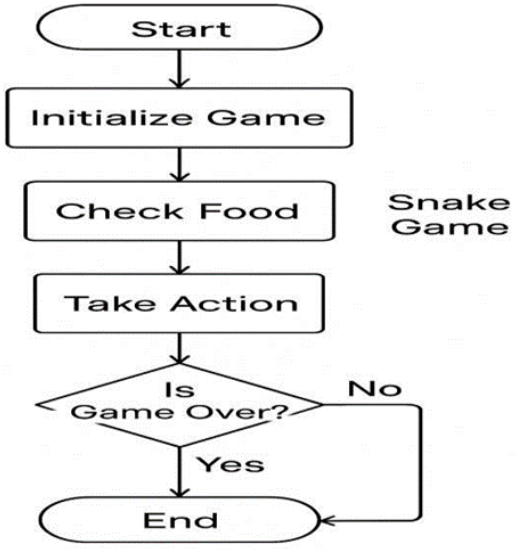

Fig4.3 – Activity diagram

```python
import pygame
import random
from enum import Enum
from collections import namedtuple
import numpy as np

pygame.init()
font = pygame.font.Font('arial.ttf', 25)
#font = pygame.font.SysFont('arial', 25)


#reward
# play(action) -> direction
#game_itration
#is_collision

class Direction(Enum):
    RIGHT = 1
    LEFT = 2
    UP = 3
    DOWN = 4

Point = namedtuple('Point', 'x, y')

# rgb colors
WHITE = (255, 255, 255)
RED = (200,0,0)
BLUE1 = (0, 0, 255)
BLUE2 = (0, 100, 255)
BLACK = (0,0,0)

BLOCK_SIZE = 20
SPEED = 20
```

gameAI.py -Importing necessary items and setting up UI

agent.py > ...

```python
import torch
import random
import numpy as np
from collections import deque
from gameAI import SnakeGameAI, Direction, Point
from model import Linear_QNet, QTrainer
from helper import plot

MAX_MEMORY = 100_000
BATCH_SIZE = 1000
LR = 0.001

class Agent:

    def __init__(self):
        self.n_games = 0
        self.epsilon = 0 # randomness
        self.gamma = 0.9 # discount rate(Has to be smaller then 1)
        self.memory = deque(maxlen=MAX_MEMORY) # popleft()
        self.model = Linear_QNet(11, 256, 3)
        self.trainer = QTrainer(self.model, lr=LR, gamma=self.gamma)


    def get_state(self, game):
        head = game.snake[0]
        point_l = Point(head.x - 20, head.y)
        point_r = Point(head.x + 20, head.y)
        point_u = Point(head.x, head.y - 20)
        point_d = Point(head.x, head.y + 20)

        dir_l = game.direction == Direction.LEFT
        dir_r = game.direction == Direction.RIGHT
        dir_u = game.direction == Direction.UP
        dir_d = game.direction == Direction.DOWN
```

agent.pyImporting and Establishing Parameters

helper.py > ...

```python
import matplotlib.pyplot as plt
from IPython import display

plt.ion()

def plot(scores, mean_scores):
    display.clear_output(wait=True)
    display.display(plt.gcf())
    plt.clf()
    plt.title('Training...')
    plt.xlabel('Number of Games')
    plt.ylabel('Score')
    plt.plot(scores)
    plt.plot(mean_scores)
    plt.ylim(ymin=0)
    plt.text(len(scores)-1, scores[-1], str(scores[-1]))
    plt.text(len(mean_scores)-1, mean_scores[-1], str(mean_scores[-1]))
    plt.show(block=False)
    plt.pause(.1)

```

helper.py - implemented a graph to visually communicate the game's progress while training.

model.py > Linear_QNet

```python
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import os

class Linear_QNet(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.linear1 = nn.Linear(input_size, hidden_size)
        self.linear2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = F.relu(self.linear1(x))
        x = self.linear2(x)
        return x

    def save(self, file_name='model.pth'):
        model_folder_path = './model'
        if not os.path.exists(model_folder_path):
            os.makedirs(model_folder_path)

        file_name = os.path.join(model_folder_path, file_name)
        torch.save(self.state_dict(), file_name)


class QTrainer:
    def __init__(self, model, lr, gamma):
        self.lr = lr
        self.gamma = gamma
        self.model = model
        self.optimizer = optim.Adam(model.parameters(), lr=self.lr)
        self.criterion = nn.MSELoss()
```

model.py – using pytorchBuilding the Neural Networ

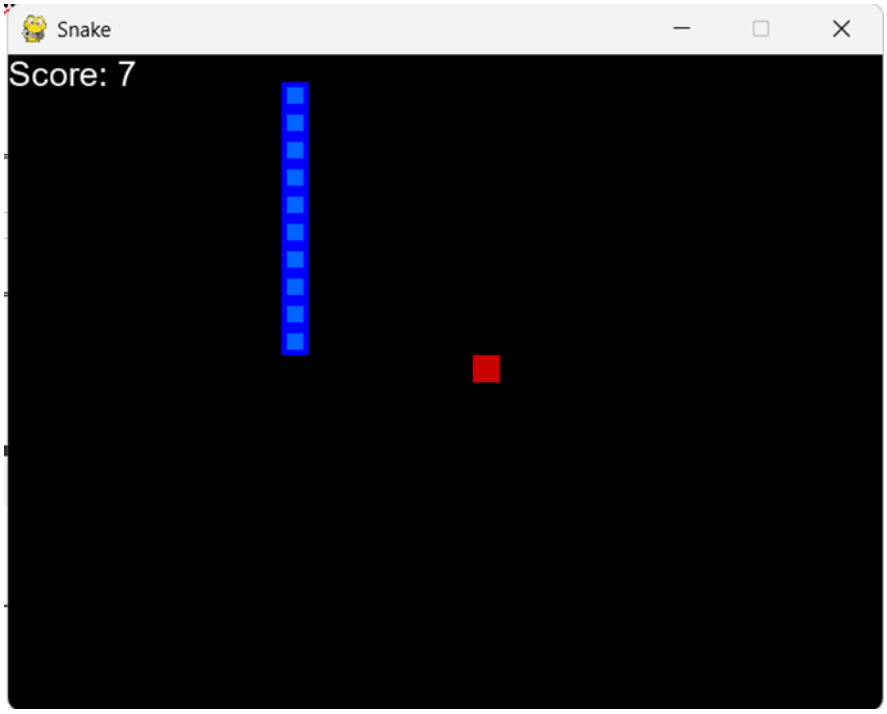

Fig:- snake game

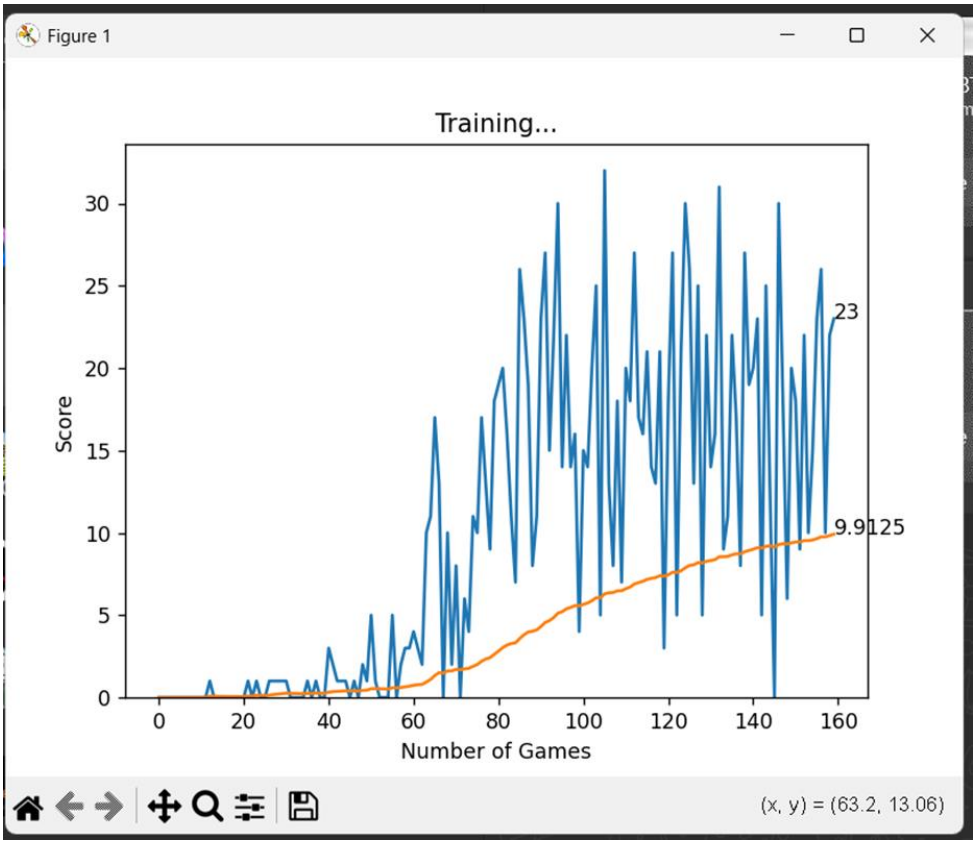

Fig:- Training the model

5. RESULT AND DISCUSSION

After implementing the environment and training the agent using Deep Q-Learning, the agent's performance was observed over hundreds of training episodes. Initially, the agent performed poorly, often colliding with walls or itself within a few moves. However, as training progressed and the exploration rate decreased, the agent began to learn effective strategies.

5.1 Result Overview

The agent starts with random movement and low performance. Over thousands of episodes, the agent begins to understand food-seeking and collision-avoidance strategies. Significant improvement is observed after 20,000 episodes. The most notable improvement was in the agent's ability to avoid collisions and survive for longer periods. It also began to exhibit behaviors such as following the food's direction, avoiding sharp turns, and moving conservatively when space became limited. The highest score achieved during training was **57**, which demonstrates the agent's capacity to learn and improve over time.

5.2 Experiments Training Setup:

- Episodes: 50,000
- Replay memory size: 100,000
- Learning rate: 0.001
- Epsilon decay for exploration

5.3 Comparison Results

A comparison with a random agent and basic rule-based agent shows the trained model outperforming in both score and survival metrics. Human-level performance was reached after about 30,000 episodes of training.

5.4 Learning Curve Analysis

The learning curve of the agent, plotted using moving average scores and total rewards, provides insights into the convergence of the learning process. Initially, the agent's performance fluctuates due to exploration but gradually stabilizes as the epsilon-greedy strategy shifts toward exploitation. The average reward per episode showed a steady increase, indicating that the agent successfully learned to associate its actions with positive outcomes like consuming food and avoiding collisions.

5.5 Failure Cases and Observations

Some failure modes were observed during training:

- In certain game scenarios with limited space, the agent failed to plan ahead effectively, leading to unavoidable collisions.
- Sparse reward feedback caused slower learning during the early episodes.

These limitations suggest room for improvement using more advanced techniques such as Prioritized Experience Replay or Reward Shaping (as discussed by Arulkumaran et al., 2017), which could enhance the learning efficiency by focusing on more informative experiences.

5.6 Parameter Sensitivity

The learning performance was found to be sensitive to the choice of hyperparameters:

- Higher learning rates (>0.001) caused unstable updates.
- Very small replay memory sizes led to overfitting recent experiences. Tuning the discount factor γ and exploration decay ε also played a vital role in balancing long-term reward maximization and exploration.

6. CONCLUSION AND FUTURE ENHANCEMENTS

6.1 Conclusions

This project successfully demonstrates the application of Deep Q-Learning in training an AI agent to play the Snake game. Starting from a simple set of rules and no prior knowledge, the agent was able to learn optimal strategies through trial and error. The project not only reinforced theoretical concepts in reinforcement learning but also provided practical experience in building, training, and evaluating intelligent agents.

The implementation proved that even basic feedforward neural networks could effectively approximate value functions for relatively simple environments like Snake. The use of PyTorch made the training process manageable and provided flexibility in designing and debugging the model. The results were promising and validated the reinforcement learning approach in achieving self-improving behavior.

Deep Q-Learning can successfully train an AI to play Snake Game. The agent becomes capable of planning ahead and maximizing score through strategic movements. This showcases the power of reinforcement learning in dynamic environments.

6.2 Future Scope of Work

For future work, the project can be extended in several directions. More sophisticated network architectures such as Dueling DQNs or Convolutional Neural Networks (CNNs) can be tested. The reward function can be refined to include penalties for unnecessary movement or bonuses for efficiency. The environment itself can be made more complex by introducing obstacles, multiple snakes, or changing food behavior. Furthermore, techniques like transfer learning and multi-agent learning can be explored to expand the scope of the application.

- Use Double DQN or Dueling DQN for more stable learning
- Integrate convolutional layers for pixel-based input
- Apply to other game environments like Pac-Man or Car Racing

6.3 Broader Implications

This study underscores how even simple environments like Snake can be utilized to simulate and understand decision-making dynamics. Applications extend to:

- Autonomous navigation: similar to obstacle avoidance in mobile robots.
- Path optimization: resembling shortest-path planning in logistics.

- Adaptive control systems: where the environment changes and feedback is delayed. Furthermore, the practical implementation of DQN in this study demonstrates the scalability of reinforcement learning solutions for real-world tasks involving dynamic and uncertain environments [Wei et al., 2018].

6.4Research Limitations

While the project was successful, certain limitations were encountered:

- State abstraction limited the agent's ability to perceive complex patterns that pixel-based
- models (e.g., CNNs) could handle.
- Reward sparsity sometimes led to inefficient learning in early episodes.
- No generalization: The trained agent works only in the specific environment and may not perform well in altered or unseen setups.

6.5Suggested Enhancements

Future enhancements may include:

- Use of Dueling DQN, which separates value and advantage streams, improving learning in environments with many similar-valued actions.
- Incorporating CNNs to handle pixel-level input, enabling generalization across game variants.
- Transfer Learning: Pre-training the agent in simpler games and fine-tuning in complex variants like dynamic mazes.
- Multi-agent RL: Introducing multiple snakes to study competition and cooperation among AI agents, extending the study to swarm intelligence concepts.

REFERENCES

1. Wei, Zhepei, et al. "Autonomous agents in snake game via deep reinforcement learning." 2018 IEEE International conference on Agents (ICA). IEEE, 2018.
2. Ray, D., Ghosh, A., Ojha, M. and Singh, K.P., 2024, December. Deep Q-Snake: An Intelligent Agent Mastering the Snake Game with Deep Reinforcement Learning. In TENCON 2024-2024 IEEE Region 10 Conference (TENCON) (pp. 1464-1469). IEEE.
3. Ray, Debjyoti, Arindam Ghosh, Muneendra Ojha, and Krishna Pratap Singh. "Deep Q-Snake: An Intelligent Agent Mastering the Snake Game with Deep Reinforcement Learning." In TENCON 2024-2024 IEEE Region 10 Conference (TENCON), pp. 1464-1469. IEEE, 2024.
4. Arulkumaran, Kai, et al. "Deep reinforcement learning: A brief survey." IEEE Signal Processing Magazine 34.6 (2017): 26-38.
5. Subramanian, V., 2018. Deep Learning with PyTorch: A practical approach to building neural network models using PyTorch. Packt Publishing Ltd.
6. Sweigart, A., 2012. Making Games with Python & Pygame.
7. Gurney, K., 2018. An introduction to neural networks. CRC press.