KALINGA INSTITUTE OF INDUSTRIAL TECHNOLOGY (KIIT)

Deemed to be University U/S 3 of UGC Act, 1956

A Project Report

On

"A Reinforcement Learning & Deep Q Learning based approach to the Snake Game & Building a Snake AI"

In Partial Fulfilment of the Requirement for the Award of

BACHELOR'S DEGREE IN
COMPUTER SCIENCE AND ENGINEERING

BY

Ashutosh Das	**21051884**
Debarka Mandal	**21051888**
Pritisha Giri	**21052439**
Bhagyashree Samantsinghar	**21052451**

UNDER THE GUIDANCE OF
Mr. Manas Ranjan Biswal

SCHOOL OF COMPUTER SCIENCE & ENGINEERING
KALINGA INSTITUTE OF INDUSTRIAL TECHNOLOGY
Bhubaneswar, ODISHA – 751024

November 2024

KALINGA INSTITUTE OF INDUSTRIAL TECHNOLOGY (KIIT)

Deemed to be University U/S 3 of UGC Act, 1956

A Project Report

On

"A Reinforcement Learning & Deep Q Learning based approach to the Snake Game & Building a Snake AI"

In Partial Fulfilment of the Requirement for the Award of

BACHELOR'S DEGREE IN
COMPUTER SCIENCE AND ENGINEERING

BY

Ashutosh Das	21051884
Debarka Mandal	21051888
Pritisha Giri	21052439
Bhagyashree Samantsinghar	21052451

UNDER THE GUIDANCE OF
Mr. Manas Ranjan Biswal

SCHOOL OF COMPUTER SCIENCE & ENGINEERING
KALINGA INSTITUTE OF INDUSTRIAL TECHNOLOGY
Bhubaneswar, ODISHA – 751024

November 2024

KIIT Deemed to be University
School of Computer Science & Engineering
Bhubaneswar, ODISHA 751024

CERTIFICATE

This is to certify that the project entitled

"A Reinforcement Learning & Deep Q Learning based approach to the Snake Game & Building a Snake AI"

Submitted by

Ashutosh Das	**21051884**
Debarka Mandal	**21051888**
Pritisha Giri	**21052439**
Bhagyashree Samantsinghar	**21052451**

is a record of Bonafide work carried out by them, in the partial fulfilment of the requirement for the award of Degree of Bachelor of Engineering (Computer Science & Engineering) at KIIT Deemed to be University, Bhubaneswar. This work is done during 2024-2024, under our guidance.

Date: 16/11/2024

(Mr. Manas Ranjan Biswal)
Project Guide

ACKNOWLEDGEMENT

We are profoundly grateful to Mr. Manas Ranjan Biswal of Affiliation for his expert guidance and continuous encouragement throughout to see that this project rights its target since its commencement to its completion. We are grateful to Dr.Arup Abhinna Acharya Dean of the Department, Computer Science and Communication, for providing us with the required facilities for the completion of the project work. We are very much thankful to the Director General,KIIT,Dr Biswajit Sahoo, for their encouragement and cooperation to carry out this work.

We express our thanks to Project Coordinator Dr.Jagannath Singh, for his Continuous support and encouragement. We thank all teaching faculty of IT Department, whose suggestions during reviews helped us in accomplishment of our project.

We would like to thank our parents, friends, and classmates for their encouragement throughout our project period. At last but not the least, we thank everyone for supporting us directly or indirectly in completing this project successfully

1. Ashutosh Das (21051884)
2. Debarka Mandal (21051884)
3. Pritisha Giri (21052439)
4. Bhagyashree Samantsinghar (21052451)

CONTENTS

ABSTRACT

In this project, an AI-powered snake game is developed using Deep Q-Learning (DQL), a technique from Reinforcement Learning (RL). The Snake Game is a classic arcade game that is a perfect example of Reinforcement Learning's potential because of its complicated state space and sequential decision-making. This project's goal is to use Python to create an intelligent agent that can learn the best ways to navigate a virtual world, avoid collisions, and collect rewards (food) in order to maximise its score. The AI agent is trained utilising Deep Q-Learning, a sophisticated model-free reinforcement learning technique that blends deep neural networks and Q-Learning. In order to enable the agent to effectively learn a policy that maximises the cumulative reward, the neural network approximates the Q-values for every action that could be taken. During training, the agent explores the game environment, learns from interactions by making mistakes, and stabilises learning with an experience replay buffer. Methods like as target networks, reward shaping, and epsilon-greedy strategy are used to improve training. The study gives insights into the use of RL techniques in game AI creation and shows how successful DQL is in a dynamic, multi-state context. With promising performance in playing the Snake Game on its own without human assistance, the results demonstrate the agent's capacity to learn the best movements and tactics. Deep reinforcement learning's promise in game AI is demonstrated in this effort, which also provides a basis for using similar methods to tackle more challenging real-world issues.

Introduction

Game development is one of the many industries that artificial intelligence (AI) has revolutionised. In this research, we investigate how to teach an AI agent to play the traditional Snake Game on its own using Deep Q-Learning (DQL), a reinforcement learning (RL) technique. Because of its dynamic state space and requirement for real-time decision-making, the game is a perfect setting for testing RL algorithms.

By interacting with the game environment, avoiding collisions, and eating to improve its score, the AI agent learns the best tactics. To handle the game's complex state space, the agent uses Deep Q-Learning, which approximates Q-values by combining Q-Learning with a deep neural network. To improve training stability and efficiency, methods including target networks, epsilon-greedy exploration, and experience replay are employed.

This project demonstrates the potential of deep reinforcement learning in game AI, highlighting the agent's ability to master the Snake Game and offering insights into the application of RL techniques for complex tasks.

Literature Review

Reinforcement Learning (RL) is a most powerful Machine Learning paradigm that allows agents to learn optimal behaviours through trial-and-error interactions with the environment. Through feedback delivered as rewards or penalties, RL agents can progressively tune their actions to optimise their cumulative rewards. The goal of this review is to explore its key concepts, algorithms, and recent advancements in RL.

- Agent and Environment: The agent is the entity that makes the decisions, and the environment is the outside system that the agent makes decisions about.
- The State and Action: The state is the situation in which the environment is right now. The action signifies an agent's selection that affects the state of the environment.
- Reward: A real number, providing the immediate feedback from taking an action. A positive reward encourages behaviours that we want to encourage, as negative rewards discourage them.
- Policy: A policy defines the agent action selection strategy, which is a mapping from states to actions. The policy can be deterministic (always selecting the same action for a given state) or it can be stochastic (selecting actions with some probability).
- Value Function: A value function predicts the expected future reward of a given state or state-action pair, quantifying the quality of a state or action in a given context.

Markov Decision Process (MDP)

MDPs provide a formalism for modelling sequential decision making problems. MDP is defined by tuple (S,A,P,R,γ):

- S:Afinite set of states.
- A:Afinite set of actions.
- Transition P(s'|s, a): The probability of going from state s, to state s' given that we took action a
- R(s,a) : The expected immediate reward received for taking action in state s.
- γ:Thediscount factor, which values the future

RL Algorithms

1. **Dynamic Programming**

    - **Value Iteration**: Iteratively computes the optimal value function V*(s) by applying the Bellman optimality equation:

      $$V^*(s) = \max_a \sum(s', r)\, P(s', r|s, a)[r + \gamma V^*(s')]$$

    - **Policy Iteration**: Iteratively improves the policy π:

      Policy Evaluation: Calculate the value function Vπ(s) for the current policy π.

      Policy Improvement: Improve the policy by acting greedily with respect to the current value function:

      $$\pi'(s) = \text{argmax}_a \sum(s', r)\, P(s', r|s, a)[r + \gamma V\pi(s')]$$

2. **Monte Carlo Methods**

    - **Monte Carlo Prediction**: Estimates the value function by averaging returns from multiple episodes.
    - **Monte Carlo Control**: Learns the optimal policy by using Monte Carlo estimation to evaluate policies and improve them.

3. **Temporal Difference (TD) Learning**

    - **SARSA**: Estimates the action-value function Q(s,a)Q(s, a)Q(s,a) by adjusting it using the current state-action pair, the next state, the subsequent action, and the received reward:

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

    - **Q-learning**: Finds the optimal action-value function Q∗(s,a)Q^*(s, a)Q∗(s,a) by updating it based on the current state-action pair, the next state, and the highest possible reward from the next state's actions:

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

4. **Policy Gradient Methods**

    - **Policy Gradient**: Directly optimises the policy parameters θ by maximising the expected reward:

      $$\theta \leftarrow \theta + \alpha \nabla \theta J(\theta)$$

    - **Actor-Critic Methods**: These methods blend policy gradient techniques with value function estimation to improve learning.
    - **Deep Reinforcement Learning**: Deep RL uses deep neural networks alongside reinforcement learning algorithms to tackle complex challenges. [1]

Overview of Reinforcement Learning in Gaming

The use of Reinforcement Learning (RL) in gaming is examined in this research, with a particular emphasis on how RL techniques are applied in the vintage Snake game. It emphasizes how important reinforcement learning is to creating intelligent agents that can learn from their mistakes and adjust to their surroundings. According to the literature study, a number of reinforcement learning techniques, including as SARSA and Q-learning, have been successfully applied in gaming environments to improve user engagement and agent performance.

Key Contributions and Findings

- **Reinforcement Learning Techniques**:
  Instead of using a greedy technique, the study examines the usage of SARSA as an on-policy algorithm that enables agents to learn from their behaviours based on the current policy. By making it possible for the snake to move around and react to user inputs more effectively, this technique has been demonstrated to increase gaming

effectiveness. Furthermore, it is noted that a noteworthy development that improves the potential of RL in gaming applications is the combination of Deep Q-learning and neural networks, specifically with Keras.

- **Challenges in Implementation:**
  The literature lists a number of difficulties in putting RL algorithms into practice, including the environment's complexity and the amount of time needed for action evaluation. These elements may cause the agent to function more slowly and have a harder time learning . The study also points out that although model-free techniques like Q-learning offer flexibility, they might have trouble figuring out iterations in more complicated situations, suggesting that more research into hybrid systems that blend several approaches is necessary .

- **Emerging Trends**:
  According to the review, there is a tendency to combine deep learning methods with reinforcement learning, which has the potential to enhance agent performance and training.

Although RL applications in gaming have advanced, the literature still identifies holes that need to be filled. More thorough research is required to determine the best settings for RL agent training as well as the long-term efficacy of different approaches. Closing these gaps may result in the creation of gaming agents that are more resilient and flexible and can manage a variety of difficulties. [2]

This article on **A Deep Q-Learning based approach applied to the Snake game** by the author- Alessandro Sebastianelli, Massimo Tipaldi, Silvia Liberata Ullo, Luigi Glielmo describes a Deep Q-Learning based approach for an agent to learn how to play the Snake game. Here are the key points :

1. **Deep Q-Learning (DQN**):
   A technique that combines Q-learning (reinforcement learning) with deep neural networks. It is used to address problems with large state and action spaces.

2. **State and Action Space**:
   - State: Represented by a vector of 11 binary values indicating danger proximity, movement direction, and relative food position based on sensors.
   - Action: Choose from "Straight," "Left," or "Right" to move the snake.

3. **Deep Neural Network Architecture:**
   - Input layer: 11 nodes for the state vector.
   - 3hiddenlayers: 100 nodes each with ReLU activation and Dropout for feature extraction.
   - Output layer: Predicts the best action using the softmax function.

4. **Training Process**:
   - Agent interacts with the environment (Snake game) and receives rewards for actions.
   - Stores state, action, reward, and next state for experience replay.
   - Trains the DQN using experience replay data to improve future action selection.

5. **Results:**
   - The agent achieved a good score (average of 56) playing the Snake game.
   - Tuning hyperparameters (learning rate, network architecture) was crucial for performance.
   - The role of Dropout layers (regularization) is still under investigation.
   - Further research will explore different loss functions and optimization algorithms. [3]

Reinforcement Learning in Decision-Making

The study "Modelling Decisions in Games Using Reinforcement Learning" investigates how human decision-making in a virtual setting can be modelled using reinforcement learning (RL) algorithms, specifically in the context of a platform jumper game. An overview of the pertinent literature covered in the paper is provided below:

- **Historical Context of AI in Games**:
  The paper starts off by discussing how AI was developed for classic board games like checkers and chess, where large game trees could be created thanks to established rules. This method, which is less suited to contemporary video games because of their complexity and unclear rules, was demonstrated by well-known programs like Deep Blue and Chinook.
- **Learning Approaches**:
  Imitation learning and reinforcement learning are the two main learning approaches that are distinguished in this work. While reinforcement learning concentrates on learning from the results of actions through rewards and punishments, imitation learning entails copying the movements of a skilled player. The capacity of Q-learning, a model-free reinforcement learning technique, to discover the best course of action through repeated interactions with the environment is highlighted. The evolution of deep reinforcement learning (DRL) algorithms is also covered in the literature. In particular, the advent of DeepQ algorithms, which directly process visual inputs using convolutional neural networks, is discussed. This development marks a substantial improvement over conventional techniques that rely on pre-extracted features by enabling agents to learn from raw gaming data.
- **Algorithm Comparison**:
  The study highlights the necessity of comparing deep RL with conventional algorithms, especially with regard to how well they can take human decision-making into account. It brings up issues that are not as well examined in the body of current work, such as the learning rates of various algorithms and their flexibility in response to shifting conditions . [4]

Reinforcement Learning

Reinforcement learning (RL) is a type of machine learning focused on training agents to make a sequence of decisions to achieve a goal, often maximizing some notion of cumulative reward. RL is inspired by behavioural psychology and operates through interactions between an agent and its environment. Unlike supervised learning, where models learn from labelled data, RL relies on feedback from actions rather than direct instruction.

Core Components of Reinforcement Learning:

- **Agent**:
  The entity that learns and makes decisions. The agent's goal is to learn an optimal strategy or policy to maximize cumulative rewards over time.
- **Environment**:
  The world or context in which the agent operates. The environment provides feedback to the agent about the consequences of its actions, allowing the agent to adjust its strategy.
- **State** (s):
  A representation of the current situation of the agent within the environment. The state can contain detailed information (e.g., positions, goals, etc.) and determines the available actions and expected rewards.
- **Action** (a):
  Any decision or move the agent can make within the environment. The choice of action directly impacts the next state and the reward received. The agent's goal is to choose actions that maximize long-term rewards.
- **Reward** (r):
  A scalar value that the environment provides to the agent as feedback on an action taken in a specific state. Positive rewards encourage certain actions, while negative rewards (penalties) discourage them. The agent seeks to maximize cumulative reward over time.

Key Concepts:

- **Discount Factor (γ\gamma γ):** Balances immediate and future rewards in cumulative calculations.
- **Exploration vs. Exploitation**: The agent must explore new actions to gather information while exploiting known actions for immediate reward.

Types of RL Algorithms:

- **Value-Based Methods**: Estimate value of state-action pairs (e.g., Q-Learning).
- **Policy-Based Methods**: Directly optimize the action-selection policy (e.g., REINFORCE, Actor Critic).
- **Model-Based Methods**: Use a model of the environment for planning.

Deep Reinforcement Learning (DRL):

- Combines deep learning with RL for handling complex, high-dimensional environments (e.g., DQN, PPO, DDPG).

Applications:

- Gaming: Achieving super human performance in games (e.g., AlphaGo).
- Robotics: Training robots for navigation and manipulation.
- Autonomous Vehicles: Enhancing decision-making in dynamic settings.
- Healthcare: Optimizing treatments and personalizing medicine.
- Finance: Portfolio management and trading optimization.

RL allows agents to learn optimal strategies over time by maximizing long-term rewards, making it suitable for complex, interactive tasks.

Example of Reinforcement Learning

Reinforcement Learning in Tic-Tac-Toe:

To apply RL to Tic-Tac-Toe, we can frame the game as follows:

- **Agent:** The AI player.
- **Environment**: The Tic-Tac-Toe board.
- **State**: The current configuration of the board.
- **Action**: Placing an 'X' or 'O' in an empty square.
- **Reward**:
  - Positive reward for winning the game.
  - Negative reward for losing the game.
  - Zero reward for a draw.

The goal of the RL agent is to learn a policy that maximizes its expected cumulative reward.

The Reinforcement Learning Process

The RL process involves the following steps:

1. **Initialization**: The agent starts with a random policy.
2. **Action Selection**: The agent selects an action based on its current policy.
3. **Environment Transition**: The environment transitions to a new state based on the action.
4. **Reward**: The agent receives a reward or penalty.
5. **Policy Update**: The agent updates its policy to improve future decisions.

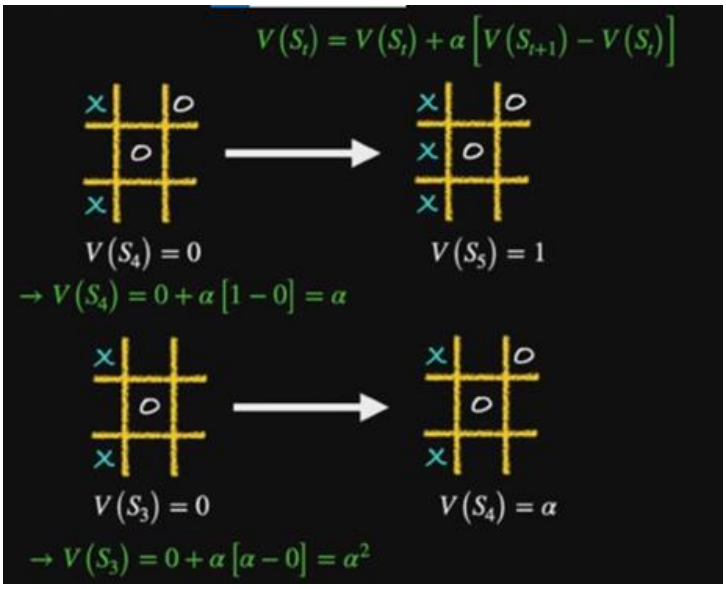
In the context of Tic-Tac-Toe, reinforcement learning (RL) enables an agent to improve its gameplay through repeated practice and a structured feedback loop. The board configurations serve as states, while possible moves represent actions. Rewards are assigned based on outcomes: winning yields a positive reward, losing a negative one, and drawing a neutral reward. Using Q-learning, the agent updates a Q-table to estimate action values, gradually refining its strategy. Through exploration exploitation and iterative learning, the agent optimizes decision-making to ultimately play Tic-Tac-Toe with high proficiency.

Assignment of values to the intermediate states

To assign the values to intermediate states we make the program play against itself and keep track of the states it went through while playing. This is done by using the given function:

$$V(S_{t}) = V(S_{t}) + \alpha[V(S_{t+1}) - V(S_{t})]$$

Example of how the value for an intermediate state is calculated using the final value:



To calculate intermediate Q-values in the reinforcement learning setup for Tic-Tac-Toe, each state action pair is updated based on the Bellman equation, taking into account the immediate reward and the maximum Q-value for future states. Specifically, after each game move, the agent updates the Q value for the current state-action pair by combining the observed reward and the highest Q-value from potential next moves. This iterative process allows the agent to learn optimal moves over time by reinforcing actions leading to winning or drawing outcomes while penalizing losing moves.

Challenges in Reinforcement Learning

There are several challenges in applying RL:

- **Exploration vs. Exploitation**: The agent needs to balance between exploring new actions to discover better strategies and exploiting known good actions to maximize immediate rewards.
- **Credit Assignment:** It can be difficult to determine which actions contributed to a particular outcome, especially in games with long sequences of moves.
- **Generalization**: The agent needs to learn a policy that works well in various game scenarios, not just the ones it has encountered during training.

Reinforcement learning is a powerful technique for solving decision-making problems in complex environments. By learning from its interactions with the environment, an RL agent can develop effective strategies to achieve its goals. The Tic-Tac-Toe example demonstrates how RL can be applied to a simple game, providing a foundation for understanding more complex applications.

Deep-Q Learning

Deep Q-Learning (DQN) is an advanced reinforcement learning algorithm that combines Q-learning with deep neural networks to handle complex, high dimensional environments. Originally developed by researchers at DeepMind, DQN has been successfully applied in various settings, including mastering video games and robotic tasks.

Here's a concise overview:

- **Goal:** Like traditional Q-learning, DQN aims to learn a Q-function Q(s,a),Q(s, a),Q(s,a) that estimates the expected cumulative reward (value) of taking action aaa in a state sss. The goal is to find the action policy that maximizes these values.
- **Why Deep Learning?** Q-learning struggles in environments with large or continuous state spaces. DQN addresses this by using a deep neural network as a function approximator to estimate the Q-values, enabling it to process complex input data (e.g., pixels in images) and environments with large action spaces.

Components of DQN:

- **Q-Network**: A neural network that takes the current state as input and outputs Q-values for each possible action.
- **Target Network**: A separate, periodically updated network that stabilizes training by reducing oscillations in Q-value estimates.
- **Experience Replay**: A memory buffer that stores past experiences (state, action, reward, next state). During training, mini-batches are sampled randomly from this buffer to break correlations between consecutive experiences and stabilize learning.

Training Process:

- The agent takes actions according to an ε-greedy policy, balancing exploration (random actions) and exploitation (choosing actions based on maximum Q-value).
- After each action, the agent stores the experience in the replay buffer.
- Periodically, the agent samples from the buffer to train the Q-network, using the Bellman equation to update Q-values:

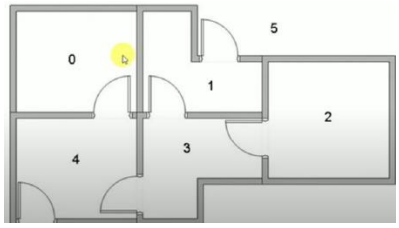
$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max\_\{a'\} Q(s', a') - Q(s, a)]$$

- The target network is periodically synchronized with the Q-network to improve stability.

In essence, DQN extends Q-learning's principles to complex environments using neural networks, enabling it to process richer, high-dimensional data, making it a foundational method in deep reinforcement learning.

Example of Deep-Q Learning

The objective is to train an agent to navigate a 5-room building. The agent's goal is to learn the optimal path from any starting room to a designated goal room. This is achieved by implementing the Q-learning algorithm.



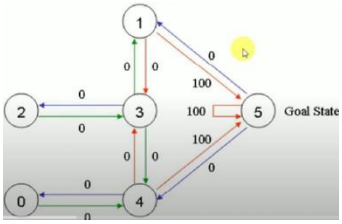
Environment Setup:

- **State Space**: Each room in the building represents a distinct state.
- **Action Space**: The actions available in each state are the possible doors to adjacent rooms.
- **Reward System**:
    - Positive reward: Reached the goal room.
    - Negative reward: Took a suboptimal action or failed to reach the goal within a certain number of steps.
    - Zero reward: For all other actions.

Solution:

So now we have to convert this building into state and actions where each room is represented as state and each door represents an action.

After converting the above diagram of the building into state and actions



So, these circles represent the states from 0 to 4 and 5 is the goal state so if the agent moves from state 4 to state 3 will be considered an action The path or door that immediately leads to the goal state have an instant reward of 100 and others doors that don't connect directly have a reward of zero. Each arrow in the diagram contains an instant reward value as shown. so now for this problem we need to find the optimal path from any state to the goal state.

And now we will apply the Q-Learning algorithm to the above state diagram:

Reward Matrix (R) :

This matrix represents the immediate reward received for taking a specific action in a particular state. The rows represent the current state (0 to 5), and the columns represent the possible actions (0 to 5). The values in the matrix indicate the reward:

- **-1**: Indicates that the action is not possible or leads to a negative outcome.
- **0**: Indicates a neutral outcome or no immediate reward.
- **100**: Indicates a positive reward, likely associated with reaching the goal state.

For the first step we need to write the reward matrix like in the diagram given below:

$$R = \begin{array}{c|cccccc} \text{State} & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & -1 & -1 & -1 & -1 & 0 & -1 \\ 1 & -1 & -1 & -1 & 0 & -1 & 100 \\ 2 & -1 & -1 & -1 & 0 & -1 & -1 \\ 3 & -1 & 0 & 0 & -1 & 0 & -1 \\ 4 & 0 & -1 & -1 & 0 & -1 & 100 \\ 5 & -1 & 0 & -1 & -1 & 0 & 100 \end{array} \quad \text{Action}$$

In order to fill the matrix we need to put the values when one state to another in this case from state 0 we can go to state 4 with reward of 0 so in row 0 column 4 we fill zero and for others it would be-1 as we can't reach those states directly.

Q learning algorithm

For each s, a initialize the table entry $\hat{Q}(s, a)$ to zero.

Observe the current state s

Do forever:

- Select an action a and execute it
- Receive immediate reward r
- Observe the new state s'
- Update the table entry for $\hat{Q}(s, a)$ as follows:

$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$$

- $s \leftarrow s'$

After this we need to assume a learning rate and a initial state for this problem we will assume:

- Learning rate = 0.8
- Initial state = Room 1

Now initialize matrix Q as zero matrix

$$R = \begin{array}{c|cccccc} \text{State} & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & -1 & -1 & -1 & -1 & 0 & -1 \\ 1 & -1 & -1 & -1 & 0 & -1 & 100 \\ 2 & -1 & -1 & -1 & 0 & -1 & -1 \\ 3 & -1 & 0 & 0 & -1 & 0 & -1 \\ 4 & 0 & -1 & -1 & 0 & -1 & 100 \\ 5 & -1 & 0 & -1 & -1 & 0 & 100 \end{array} \quad \text{Action} \qquad Q = \begin{array}{c|cccccc} & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Now in the reward matrix (R) start from row 1 as we had considered state 1 as initial state,

And any one action from two possibilities either the agent can go to 3 with a reward of zero or it can go to 5 for a reward of 100 By random selection, agent goes to state 5 as an action, after moving to 5 rewards of 100 is given and the current state becomes 5.

Then we need to check for possible states that the agent can visit in this scenario possible states to visit are states 1, 4 and 5.

Q-Learning Update Rule:

$$Q(state, action) = R(state, action) + Gamma * Max[Q(next\ state, all\ actions)]$$

Where:

- **Q (state, action):** The estimated value of taking action in state.
- **R (state, action):** The immediate reward received for taking action in state.
- **γ**: The discount factor (a value between 0 and 1) that determines the importance of future rewards.
- **Max (Q (next_state, all_actions)):** The maximum Q-value for all possible actions in the next state.

This update rule is used to iteratively improve the Q-values, which ultimately lead to the agent learning the optimal policy.

we put the values in the above formula we get this:

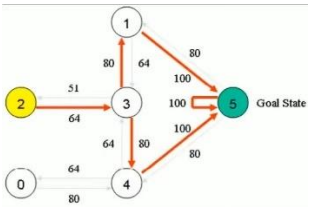
$$Q(1, 5) = R(1, 5) + 0.8 * Max[Q(5, 1), Q(5, 4), Q(5, 5)] = 100 + 0.8 * 0 = 100$$

Now for the values Q(1, 5) is 0 (look for values in the Q matrix) and for R(1, 5) the value will be 100 (look for values in the R matrix) In the max part it takes maximum of all the n values inside the max part here Q(5, 1) , Q(5, 4) and Q(5, 5) all these values are 0 so maximum will be 0 and then multiplied with 0.8 which is 0. So, the value of Q (1, 5) is 100 update the value in Q matrix and now current state is state 5 which is the goal state now one episode is finished this will continue for all other states like the next initial state could be 0, 2, 3, 4, 5 this will continue until all states are initialized as initial states.

After multiple episodes the agent finally reaches convergence values in matrix Q like shown below:

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 80 & 0 \\ 0 & 0 & 0 & 64 & 0 & 100 \\ 0 & 0 & 0 & 64 & 0 & 0 \\ 0 & 80 & 51 & 0 & 80 & 0 \\ 64 & 0 & 0 & 64 & 0 & 100 \\ 0 & 80 & 0 & 0 & 80 & 100 \end{bmatrix} \end{array}$$

Tracing the best sequences of states is as simple as following the links with highest values at each state by drawing the state diagram below or using the Q matrix for optimal path.



In the matrix we can find optimal path by choosing the best value for that particular row like in state 2 row we find that column 3 gives best value so it moves to state 3 From state 3 it again checks for best which is 80 so it can move to either state 1 (column 1) or state 4 (column 4) this will go on until the goal state is reached.

Snake Game (Explanation)

```
import pygame
import random
from enum import Enum
from collections import namedtuple

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

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

class SnakeGame:

    def __init__(self, w=640, h=480):
        self.w = w
        self.h = h
        # init display
        self.display = pygame.display.set_mode((self.w, self.h))
        pygame.display.set_caption('Snake')
        self.clock = pygame.time.Clock()

        # init game state
        self.direction = Direction.RIGHT

        self.head = Point(self.w/2, self.h/2)
        self.snake = [self.head,
                      Point(self.head.x-BLOCK_SIZE, self.head.y),
                      Point(self.head.x-(2*BLOCK_SIZE), self.head.y)]

        self.score = 0
        self.food = None
        self._place_food()

    def _place_food(self):
        x = random.randint(0, (self.w-BLOCK_SIZE )//BLOCK_SIZE )*BLOCK_SIZE
        y = random.randint(0, (self.h-BLOCK_SIZE )//BLOCK_SIZE )*BLOCK_SIZE
        self.food = Point(x, y)
        if self.food in self.snake:
            self._place_food()

    def play_step(self):
        # 1. collect user input
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                quit()
            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_LEFT:
                    self.direction = Direction.LEFT
                elif event.key == pygame.K_RIGHT:
                    self.direction = Direction.RIGHT
                elif event.key == pygame.K_UP:
                    self.direction = Direction.UP
                elif event.key == pygame.K_DOWN:
                    self.direction = Direction.DOWN

        # 2. move
        self._move(self.direction) # update the head
        self.snake.insert(0, self.head)

        # 3. check if game over
        game_over = False
        if self._is_collision():
            game_over = True
            return game_over, self.score

        # 4. place new food or just move
        if self.head == self.food:
            self.score += 1
            self._place_food()
        else:
            self.snake.pop()
```

```
        # 5. update ui and clock
        self._update_ui()
        self.clock.tick(SPEED)
        # 6. return game over and score
        return game_over, self.score

    def _is_collision(self):
        # hits boundary
        if self.head.x > self.w - BLOCK_SIZE or self.head.x < 0 or self.head.y > self.h - BLOCK_SIZE or self.head.y < 0:
            return True
        # hits itself
        if self.head in self.snake[1:]:
            return True

        return False

    def _update_ui(self):
        self.display.fill(BLACK)

        for pt in self.snake:
            pygame.draw.rect(self.display, BLUE1, pygame.Rect(pt.x, pt.y, BLOCK_SIZE, BLOCK_SIZE))
            pygame.draw.rect(self.display, BLUE2, pygame.Rect(pt.x+4, pt.y+4, 12, 12))

        pygame.draw.rect(self.display, RED, pygame.Rect(self.food.x, self.food.y, BLOCK_SIZE, BLOCK_SIZE))

        text = font.render("Score: " + str(self.score), True, WHITE)
        self.display.blit(text, [0, 0])
        pygame.display.flip()

    def _move(self, direction):
        x = self.head.x
        y = self.head.y
        if direction == Direction.RIGHT:
            x += BLOCK_SIZE
        elif direction == Direction.LEFT:
            x -= BLOCK_SIZE
        elif direction == Direction.DOWN:
            y += BLOCK_SIZE
        elif direction == Direction.UP:
            y -= BLOCK_SIZE

        self.head = Point(x, y)

if __name__ == '__main__':
    game = SnakeGame()

    # game loop
    while True:
        game_over, score = game.play_step()

        if game_over == True:
            break
    print('Final Score', score)

    pygame.quit()
```

- **init**: Initializes the game window, sets the initial game state (direction, snake position, score, food position).
- **_place_food**: Randomly places food on the screen, ensuring it doesn't overlap with the snake's body.
- **play_step**: This is the main game loop function. It performs the following steps:
- Collect user input (checks for QUIT event and allows changing direction with arrow keys).
- Moves the snake based on the current direction.
- Checks for game over conditions (hitting the wall or itself).
- Places new food or removes the tail if the snake eats the food.
- Updates the game display (draws the snake, food, and score).
- Sets the game speed and returns game over status and score.
- **_is_collision**: Checks if the snake hits the boundary of the screen or collides with itself.
- **_update_ui:** Updates the game display by drawing the snake, food, score text, and filling the background.
- **_move**: Updates the snake's head position based on the current direction.
- Main Loop:
- Creates a SnakeGame object.
    - Runs a loop that keeps the game running until the game over condition is met. Inside the loop:
    - Calls the play_step function to handle user input, move the snake, and update the game state.
    - Checks if the game is over and breaks the loop if true.

Flowchart

Basic Flow of Actions:

AGENT

- game
- model
- training
  - state = get_state(game)
  - action = get_move(state):
    - model.predict()
  - reward, game_over, score=game.play_step(action)
  - new_state = get_state(game)
  - remember
  - model.train()

GAME (Pygame)

- play_step(action)
  - reward, game_over, score

Model (PyTorch)

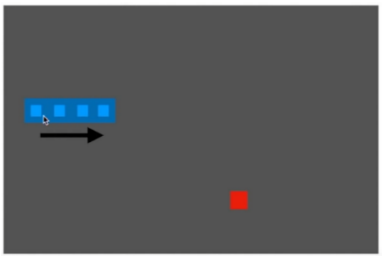
- Linear_QNet(DQN)
  - model.predict(state)
    - action

Explanation:

- Starting with the GAME, we have a game loop, and we will start with a play_step that gets an action done that is it helps the snake move, and returns the reward associated with it, if the game is over or not and the score that the move has resulted in.
- Then we have the AGENT, the AGENT basically puts everything together, it should know about the GAME and the MODEL (we store both of them in our agent).
- Then we implement the training loop, so based on the game we need to calculate the state and based on the state we need to calculate the action involving the model prediction.
- Then we get the snake its move through the game.play_step() function applying the action to it and thus getting the reward, if the game is over or not and the score. Now with these information we calculate the new state.
- Now we need to remember the new state, the older state , the reward, if the game is over or not and the score. So with these information, we will train our model.
- Now for the MODEL, we need to call the Linear_QNet, its basically a feed forward neural network with a few linear layers and it needs few information like the new state, the old state and then we can train the MODEL and then call the model. predict giving it the state and finally it gets us the action to be performed.

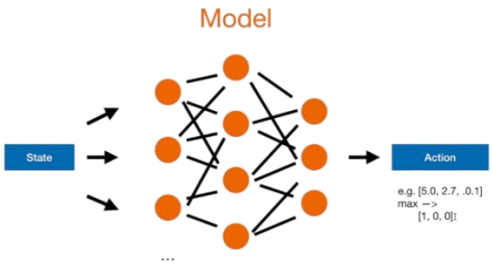
Variables Explanation:

- REWARD
  - For Example: Whenever the snake eats the food, we assign the reward with +10. Whenever the game is over, we assign the reward with -10 and finally for every other action we assign the reward with 0.
- ACTION
  - The action determines our next move, we have 4 different actions, LEFT, RIGHT, UP and DOWN. But 180 degree moves are not allowed. So we will allow only 3 different STRAIGHT [1,0,0] (We stay in the current direction), RIGHT TURN [0,1,0] and LEFT TURN [0,0,1] and the move is dependent on the current direction.
- STATE
  - For State we have 11 values
  - We need to help the snake to learn about the information about the game that it knows about. Basically the environment.
    - Danger straight, Danger right, Danger Left
    - Direction left, Direction right, Direction up, Direction down
    - Food left, Food right, Food up, Food down
  - These can be demonstrated using Boolean values such as [0,0,0,0,1,0,0,0,1,0,1] means direction right, food right, food down ,this is the environment.



Training the Model Explanation:

- So now with the states and the action we can decide the model.
- This is just a feed forward neural Network with an input Layer, a hidden layer and an output layer , for the input it gets the state(11 different numbers Boolean values(input layer size=11)), then we can choose a hidden layer and for the output we need 3 outputs because then we predict the actions that's we get an array of numbers and we can assign the numbers with the maximum as 1 and others as 0, thus getting the direction to move as an action.



- Now we need to train the model, for that we will use the DEEP Q Learning.
    - Q value = Quality of action
        0. init Q value (=init model)
        1. choose action (model.predict(state))
        2. perform action
        3. measure reward
        4. update Q value (+ train model) [repeat again from 1]
- Q value represents the quality of the action, and we need to improve the quality. So each action should improve the quality of the snake.
- We start by initialising the Q value, in this case we initialize our model with some random parameters.
- Then we choose an action by calling the model.predict with an state or sometimes we do a random move too, specifically in the beginning when we don't know more about the game yet, and later we have to do a trade off when we don't want to make a random move anymore and only call model predict (basically a trade off between exploration and exploitation).
- Then with this action we perform the next action and measure the reward and with this information we update our Q value and then train the model and then repeat the steps from choosing an action in a iterative way.
- Now to train the model , we always need to have some loss function, through which we can optimize or minimize , we will use the Bellman Equation.

$$NewQ(s,a) = Q(s,a) + \alpha[R(s,a) + \gamma max\, Q'(s',a') - Q(s,a)]$$

New Q value for that state and that action — Current Q value — Reward for taking that action at that state — Learning Rate — Discount rate — Maximum expected future reward **given the new s' and all possible actions at that new state**

    - NewQ(s,a) :- New Q value for the state and the action
    - Q(s,a) :- Current Q value
    - α :- Learning Rate
    - R(s,a) :- Reward for taking the action at that state
    - γ :- Discount Rate
    - MaxQ'(s',a') :-Maximum expected future reward given the new s' and all possible actions at the new state.
- The new Q value is calculated by taking the current Q value plus the learning rate with the reward for taking that action at that state plus discount rate with the maximum expected future reward.
- Basically updating the Q value
    - Q = model.predict(state0)
    - Qnew = R + γ . max(Q(State1))
- Loss function :
    - loss= $(Qnew - Q)^2$

Environment Set Up and Implementing the Snake Game

- Create a environment using conda and a version of python
  - `conda create -n pygame_env python=3.7`
- Now activate the environment
  - `conda activate pygame_env`
- Now we will install some libraries
  - `pip install pygame`
  - `pip3 install torch torchvision`
  - `pip install matplotlib ipython`

- We need to have a reset function so that after each game our agent should be able to reset the game and start with a new game.

```
def reset(self):
        # init game state
        self.direction = Direction.RIGHT

        self.head = Point(self.w/2, self.h/2)
        self.snake = [self.head,
                      Point(self.head.x-BLOCK_SIZE, self.head.y),
                      Point(self.head.x-(2*BLOCK_SIZE), self.head.y)]

        self.score = 0
        self.food = None
        self._place_food()
        self.frame_iteration=0
```

- Then we need to implement the reward that out agent gets.
- Then we need to change the play function so that it takes an action and computes the direction.

```
def play_step(self,action):
        self.frame_iteration += 1
        # 1. collect user input
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                quit()

        # 2. move
        self._move(action) # update the head
        self.snake.insert(0, self.head)

        # 3. check if game over
        reward=0
        game_over = False
        if self.is_collision() or self.frame_iteration > 100*len(self.snake):
            game_over = True
            reward=-10
            return reward, game_over, self.score

        # 4. place new food or just move
        if self.head == self.food:
            self.score += 1
            reward = 10
            self._place_food()
        else:
            self.snake.pop()

        # 5. update ui and clock
        self._update_ui()
        self.clock.tick(SPEED)
        # 6. return game over and score
        return reward, game_over, self.score
```

- We also need to keep track of current game-iteration(frame) .
- Also need to make changes in the is_collision function to check if it is a collision.

```
def is_collision(self, pt=None):
        if pt is None:
            pt=self.head
        # hits boundary
        if pt.x > self.w - BLOCK_SIZE or pt.x < 0 or pt.y > self.h - BLOCK_SIZE or pt.y < 0:
            return True
        # hits itself
        if pt in self.snake[1:]:
            return True

        return False
```

Implementing the Agent to Control the Game

- Create a agent.py file.
- First we will import the required classes.

```
import torch
import random
import numpy as np
from collections import deque
from game import SnakeGameAI, Direction, Point
```

The deque is a data structure that we will use to store our memory.

- We will declare some constants.

```
MAX_MEMORY =100_000
BATCH_SIZE=1000
LR = 0.001
```

#MAX_MEMORY is the size of the deque that we will use
#BATCH_SIZE the maximum number of tuples we will allow for training set
#LR is basically the Learning Rate

- Now we need to create the following functions
- **The init function**
  - In the init function we will initialise the regular parameters those are **np of games**, **epsilon**(randomness), **gamma**(discount rate), **memory** (using the deque with the length as equal to MAX_MEMORY). We also need a **model** and a **trainer** which we will declare later but for now we will declare them as None (will be useful in training the short and long memory).
  -
```
def __intit__(self):
        self.n_games=0
        self.epsilon=0
        self.gamma=0
        self.memory=deque(maxlen=MAX_MEMORY)
          self.model=None
          self.trainer=None
```

- **The get_state function**
  - We will declare the 11 states that we have declared earlier.

```
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

        state = [
            # Danger straight
            (dir_r and game.is_collision(point_r)) or
            (dir_l and game.is_collision(point_l)) or
            (dir_u and game.is_collision(point_u)) or
            (dir_d and game.is_collision(point_d)),

            # Danger right
            (dir_u and game.is_collision(point_r)) or
            (dir_d and game.is_collision(point_l)) or
            (dir_l and game.is_collision(point_u)) or
            (dir_r and game.is_collision(point_d)),

            # Danger left
            (dir_d and game.is_collision(point_r)) or
            (dir_u and game.is_collision(point_l)) or
            (dir_r and game.is_collision(point_u)) or
            (dir_l and game.is_collision(point_d)),

            # Move direction
            dir_l,
            dir_r,
            dir_u,
            dir_d,
```

```
            # Food location
            game.food.x < game.head.x,  # food left
            game.food.x > game.head.x,  # food right
            game.food.y < game.head.y,  # food up
            game.food.y > game.head.y  # food down
            ]

        return np.array(state, dtype=int)
```

- **The remember function**
  - It takes the **state, action, reward, next_state, done** as parameters and stores them in the memory deque, if the MAX_MEMORY is exceeded then it will automatically perform the popleft function which will help us have the memory of very immediate past.

```
def remember(self, state, action, reward, next_state, done);
        self.memory.append((state,action,reward,next_state,done))
```

- **The train long memory and the train short memory function**
  - In **train_short_memory** we will basically use the use the trainer from the init function and a function train_step which we will declare later and pass the parameters state, action, reward, next state, if the game is over or not and will train the model using the memory.
  - In the **train_long_memory** basically we need to handle 2 forms of data, a single tuple and a group of tuples and train the model similarly in the above step.

```
def train_long_memory(self):
        if len(self.memory) > BATCH_SIZE:
            mini_sample=random.sample(self.memory, BATCH_SIZE) #return a list of tuples
        else:
            mini_sample=self.memory

        states, actions, rewards, next_states, dones=zip(*mini_sample)
        self.trainer.train_step(states, actions, rewards, next_states, dones)

 def train_short_memory(self, state, action, reward, next_state, done):
        self.trainer.train_step(state, action, reward, next_state, done)
```

- **The get_action function**
  - We will perform random moves according to the trade-off exploration / exploitation.
  - Set the **epsilon** to 80- no of games (hardcoded), get the final move in the beginning we set it to [0,0,0].
  - Now check if a random number is between 0 and 200 and if its smaller than epsilon then get a move as a number between 0 and 2 and then finally set that final move[move] to 1.
  - Else we will move based on our model, we will make a prediction using the model class and predict function taking a state as parameter(the state need to be converted to a tensor). Thus getting a raw value and we will convert its max element to 1 and others 0 and finally setting the final move[move] to 1 and returning the final move.

```
def get_action(self, state):
        # random moves: tradeoff exploration / exploitation
        self.epsilon = 80 - self.n_games
        final_move = [0,0,0]
        if random.randint(0, 200) < self.epsilon:
            move = random.randint(0, 2)
            final_move[move] = 1
        else:
            state0 = torch.tensor(state, dtype=torch.float)
            prediction = self.model(state0)
            move = torch.argmax(prediction).item()
            final_move[move] = 1

        return final_move
```

- **A global train function**
  - A **plot_scores** list to help us plot the result later, **plot_mean_scores** is also a list to store the average scores, **total_scores** to store the score for each iteration, record to be used in the memory for future iterations, **agent** as an object of the **Agent class** and **game** as an object of the **SnakeGameAI class.**
  - Next we will have our training loop which will run until we quit the process.
  - We will start with getting the old state using the **get_state** function of the Agent class.

- Now we will go ahead with the move according to the state that we got using the **get_action** function of the Agent class.
- Now after we have the old state and the move according to the **old state** we now need to perform the move using the **play_step** function of the **Game class** and then the **new state** using the **get_state** function of the **Agent class** thus receiving the reward, the game is over or not, score and new state.
- Now we will do a short memory training using only one state, the parameters to be used are the **old state, the move that we performed on the old state, reward that we received, the new state and if the game is over or not** using the **train_short_memory** function of the **Agent class.**
- Finally we need to **remember** that is storing the results in the memory, the parameters to me stored are old state, reward that we received, the new state and if the game is over or not using the remember function of the Agent class.
- If the game is over then we will train the long memory (replaying memory/experience replay).It will train again on all the previous moves and games that it played and this tremendously helps in improving itself.
- Reset the game, increase the number of games, use the train_long_memory function of the Agent class, check if we got a new high score and update accordingly.

```
def train():
    plot_scores=[]
    plot_mean_scores=[]
    total_score=0
    record=0
    agent=Agent()
    game=SnakeGameAI()

    while True:
        state_old= agent.get_state(game)

        final_move=agent.get_action(state_old)

        reward, done, score=game.play_step(final_move)
        state_new=agent.get_state(game)

        agent.train_short_memory(state_old,final_move,reward, state_new, done)

        agent.remember(state_old,final_move,reward,state_new,done)


        if done:
            game.reset()
            agent.n_games +=1
            agent.train_long_memory()

            if score> record:
                record=score

            print('Game',agent.n_games,'Score',score,'Record:', record)
```

Creating and Training the Neural Network

- Create a model.py file
- Import the libraries

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

- Now we will implement a Linear_QNet function, define init with input size, hidden size and a output size as parameters. Call the super initialiser.
- Now implement the feed forward neural network with 2 linear layer one from input to hidden and one from hidden to output.
- Implement the Forward function with applying the first linear layer using a actuation function RELU and then normally the second linear layer.
- Implement a helper function to save the model for further use, using a file name and update in the global train function of the agent.py file and after updating the score save the file as (agent.model.save()).
- Create a new folder in current directory if not present make a directory or else create the file name and save using torch.

```
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)
```

- Now for the actual training and optimization, we will create a QTrainer class.
- Define the init function with parameters like model, LR, gamma , for a pytorch optimization step we need a optimizer like Adam from the optim module with model parameters and LR and finnaly a criterion or a loss function (MEAN SQUARED ERROR) as nn.MSELoss.

```
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()
```

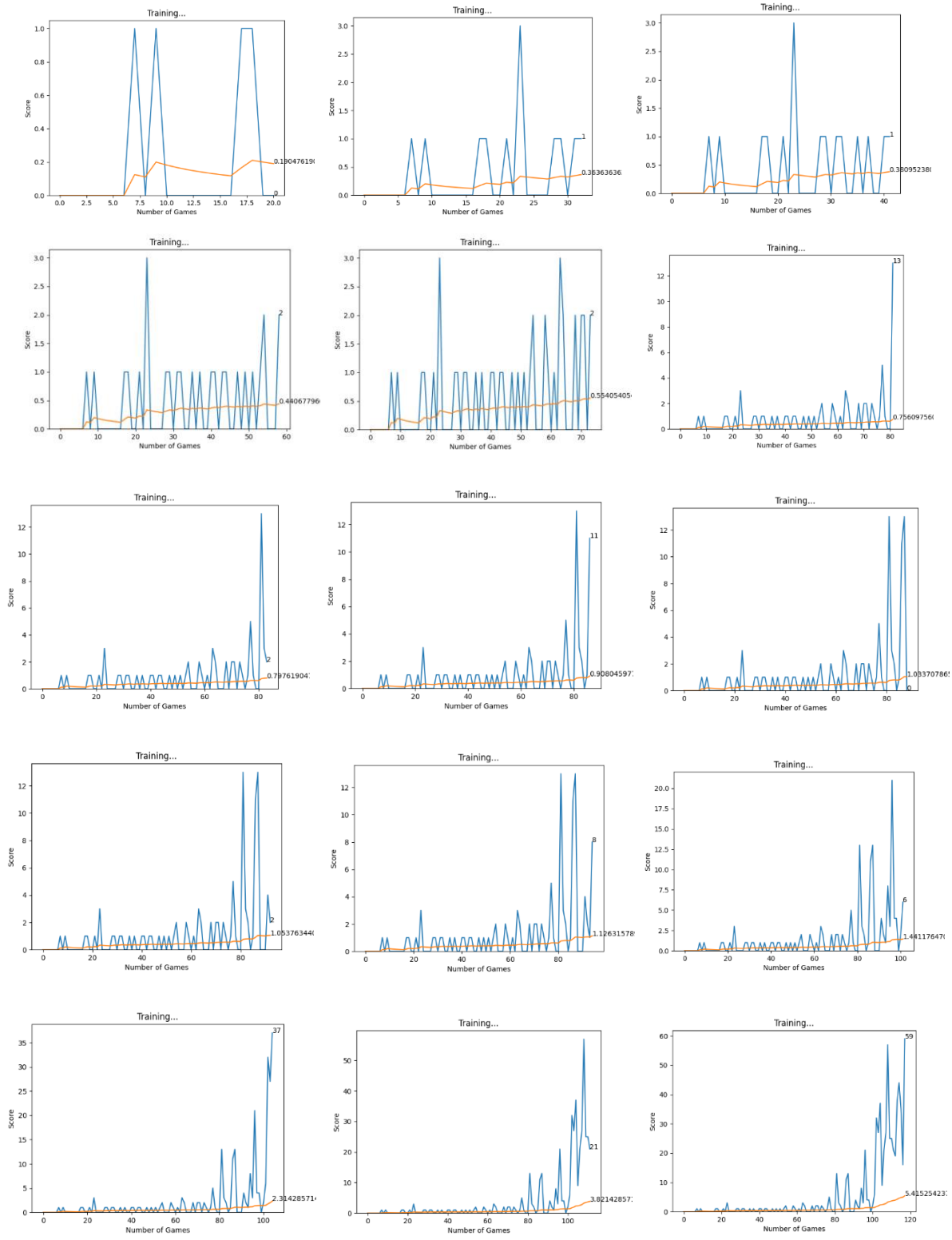
- Now we will define the train_step function with state, action, reward, next state, done as parameters.
- Then we will go to the agent.py file and import the 2 classes Linear_QNet and QTrainer and initialise them in the init function.

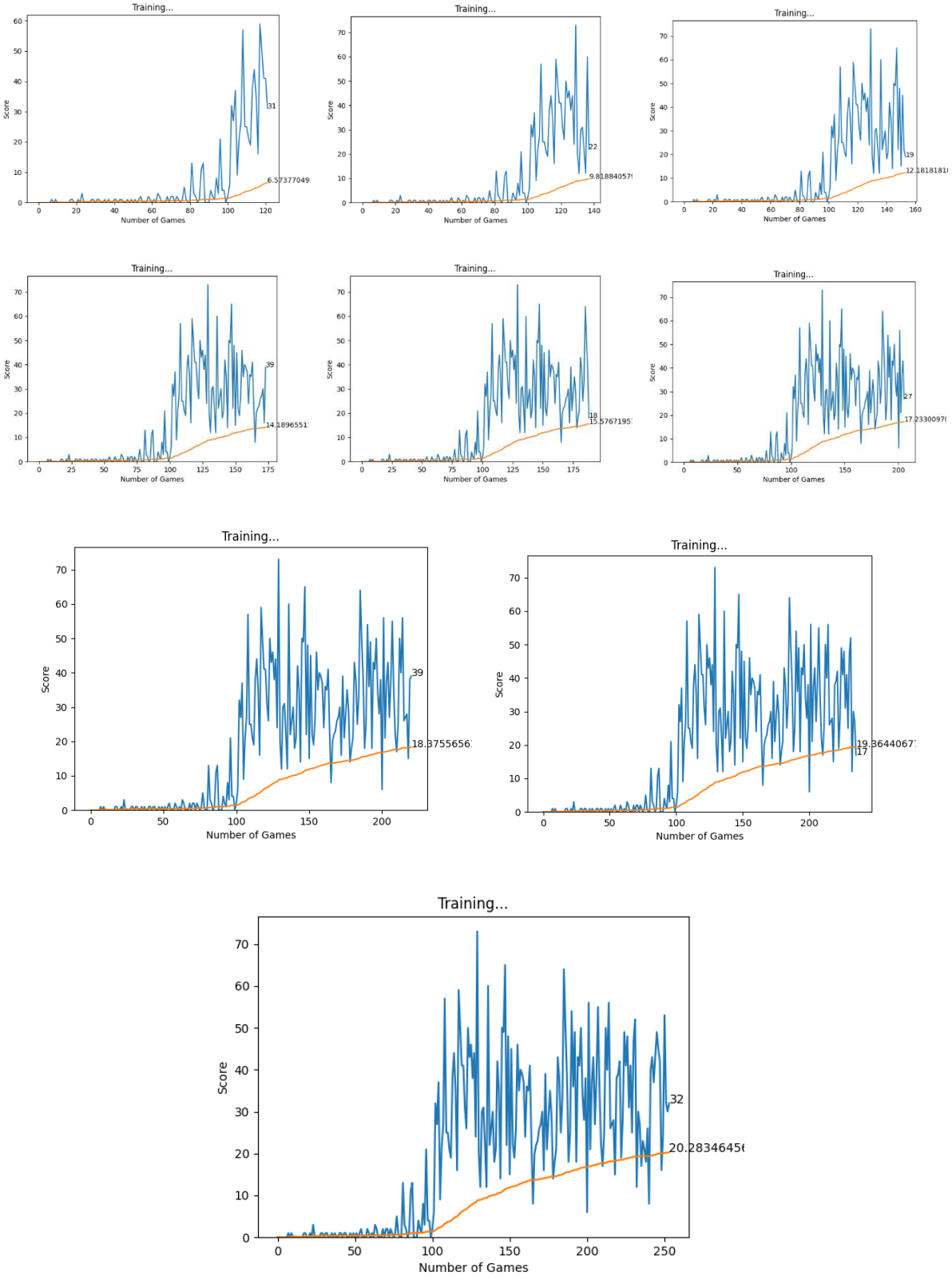
```
from model import Linear_QNet, QTrainer

self.model=Linear_QNet(11,256,3)
self.trainer=QTrainer(self.model, lr=LR, gamma=self.gamma)
```

- Now back to the model.py we will implement the train_step function.
- We will convert the tuple into pytorch tensor using the torch.tensor and the variable and the datatype as parameters.
- We need to handle multiple sizes too, for state's length is 1 that means its only a 1 dimension tuple and we need to reshape the tuple into the form (1,x) where x is the variable using the torch.unsqueeze function and the variable and dimension as parameters.
- Then we need to update the Q value i.e. first predict the Q value with the current state and then generate the new Q value.
- Prediction of Q value with current state can be done using the model(state).
- The new Q value can be calculated as *Q_new = r + y * max(next_predicted Q value)* only if not done.
- Now we apply the loss function using the optimizer with zero_grad to empty the gradient then calculate the loss using the criterion with target and pred as parameters.
- Then we apply the backpropagation to update our gradients.

```
def train_step(self, state, action, reward, next_state, done):
        state = torch.tensor(state, dtype=torch.float)
        next_state = torch.tensor(next_state, dtype=torch.float)
        action = torch.tensor(action, dtype=torch.long)
        reward = torch.tensor(reward, dtype=torch.float)

        if len(state.shape) == 1:
            # (1, x)
            state = torch.unsqueeze(state, 0)
            next_state = torch.unsqueeze(next_state, 0)
            action = torch.unsqueeze(action, 0)
            reward = torch.unsqueeze(reward, 0)
            done = (done, )

        # 1: predicted Q values with current state
        pred = self.model(state)

        target = pred.clone()
        for idx in range(len(done)):
            Q_new = reward[idx]
            if not done[idx]:
                Q_new = reward[idx] + self.gamma * torch.max(self.model(next_state[idx]))

            target[idx][torch.argmax(action[idx]).item()] = Q_new

        # 2: Q_new = r + y * max(next_predicted Q value)
        self.optimizer.zero_grad()
        loss = self.criterion(target, pred)
        loss.backward()

        self.optimizer.step()
```

Results





- **Total Number of Games Played**: 254
- **Time**: 60 minutes
- **Highest Score**: 73
- **Average Score**: 20.2835

Conclusion

In conclusion, this research effectively applies Deep Q-Learning (DQL) and Reinforcement Learning to create an AI agent that can play the Snake Game on its own. Deep neural networks and Q-Learning are combined to teach the agent efficient ways to move through the game environment, avoid obstacles, and increase its score. The implementation of strategies including target networks, epsilon-greedy exploration, and experience replay helped to ensure stable and effective training. In addition to providing a strong basis for applying comparable techniques to more complicated real-world scenarios that call for sequential decision-making, the results show the potential of deep reinforcement learning for game AI.

Future Scope

- Investigating advanced neural network topologies, such as Convolutional Neural Networks (CNNs) for improved feature extraction or Long Short-Term Memory (LSTM) networks to manage temporal dependencies in the game, can improve the performance of the Snake AI.

- Transfer Learning: To cut down on overall training time and increase learning efficiency, the agent could be trained on easier conditions before being adapted to more challenging Snake Game variations.

- Multi-Agent Reinforcement Learning: By extending the project to a multi-agent environment in which several AI agents play the Snake Game at the same time, competitive or cooperative dynamics may be introduced, offering a more expansive experimental environment.

- Applications in the Real World: The concepts employed in this research can be modified to address pathfinding, navigation, and resource optimization issues in the real world, including logistics, robotic motion planning, and driverless cars.

- Improved Exploration Techniques: By incorporating more complex exploration techniques, such as curiosity-driven exploration or Noisy Networks, the AI can learn more effectively and find more varied states and best practices.

- Enhancing Reward Function: Especially in intricate game circumstances, experimenting with various reward structures, such as dynamic or adaptive incentives, may aid the agent's learning by giving it more insightful feedback.

- Future research could examine the integration of Deep Q-Learning with other reinforcement learning algorithms, such as Actor-Critic techniques or Proximal Policy Optimization (PPO), in order to compare stability and performance.

- These extensions could make the AI agent more robust, adaptable, and capable of handling increasingly complex environments and tasks beyond the classic Snake Game.

References

1. Jia, J. and Wang, W., 2020, October. Review of reinforcement learning research. In *2020 35th Youth Academic Annual Conference of Chinese Association of Automation (YAC)* (pp. 186-191). IEEE.

2. Almalki, A.J. and Wocjan, P., 2019, December. Exploration of reinforcement learning to play snake game. In *2019 International Conference on Computational Science and Computational Intelligence (CSCI)* (pp. 377-381). IEEE.

3. Sebastianelli, A., Tipaldi, M., Ullo, S.L. and Glielmo, L., 2021, June. A Deep Q-Learning based approach applied to the Snake game. In *2021 29th Mediterranean Conference on Control and Automation (MED)* (pp. 348-353). IEEE.

4. Singal, H., Aggarwal, P. and Dutt, V., 2017, December. Modeling decisions in games using reinforcement learning. In *2017 International Conference on Machine Learning and Data Science (MLDS)* (pp. 98-105). IEEE.

5. https://docs.python.org/3/

6. https://pytorch.org/docs/stable/index.html

7. https://www.pygame.org/docs/

8. https://www.mathworks.com/help/reinforcement-learning/

9. https://huggingface.co/learn/deep-rl-course/en/unit3/deep-q-algorithm

10. https://towardsdatascience.com/reinforcement-learning-implement-tictactoe-189582bea542

INDIVIDUAL CONTRIBUTION REPORT

A Reinforcement Learning & Deep Q Learning based approach to the Snake Game & Building a Snake AI

Ashutosh Das
21051884

Abstract: In this project, Deep Q-Learning (DQL), a reinforcement learning method that blends Q-Learning with neural networks, is used to create an AI agent for the traditional Snake Game. By eating, the agent gains the ability to move across the game's environment, avoid collisions, and increase its score. By approximating Q-values for every action, the DQL model enables the agent to discover successful tactics via experimentation. Training is stabilized by methods such as target networks, epsilon-greedy exploration, and experience replay. The outcomes show that the agent can play the game on its own, underscoring deep reinforcement learning's promise for challenging decision-making tasks.

Individual contribution and findings:

- Understanding the Research paper: “Modelling Decisions in Games Using Reinforcement Learning” and creating its Literature Review.
- Written and implemented the Agent Code to control the Game.

Individual contribution to project report preparation:

- Code Documentation
- Flowchart
- Literature Review

Full Signature of Supervisor Full Signature of Student

INDIVIDUAL CONTRIBUTION REPORT

A Reinforcement Learning & Deep Q Learning based approach to the Snake Game & Building a Snake AI

Pritisha Giri
21052439

Abstract: In this project, Deep Q-Learning (DQL), a reinforcement learning method that blends Q-Learning with neural networks, is used to create an AI agent for the traditional Snake Game. By eating, the agent gains the ability to move across the game's environment, avoid collisions, and increase its score. By approximating Q-values for every action, the DQL model enables the agent to discover successful tactics via experimentation. Training is stabilized by methods such as target networks, epsilon-greedy exploration, and experience replay. The outcomes show that the agent can play the game on its own, underscoring deep reinforcement learning's promise for challenging decision-making tasks.

Individual contribution and findings:

- Understanding the Research paper: “A Deep Q-Learning based approach applied to the Snake game” and creating its Literature Review.
- Environment Set up and written the SNAKE AI code with implementing it.

Individual contribution to project report preparation:

- Reinforcement Learning Explanation.
- Example of Reinforcement Learning.
- Literature Review

Full Signature of Supervisor Full Signature of Student

INDIVIDUAL CONTRIBUTION REPORT

A Reinforcement Learning & Deep Q Learning based approach to the Snake Game & Building a Snake AI

Bhagyashree Samantsinghar
21052451

Abstract: In this project, Deep Q-Learning (DQL), a reinforcement learning method that blends Q-Learning with neural networks, is used to create an AI agent for the traditional Snake Game. By eating, the agent gains the ability to move across the game's environment, avoid collisions, and increase its score. By approximating Q-values for every action, the DQL model enables the agent to discover successful tactics via experimentation. Training is stabilized by methods such as target networks, epsilon-greedy exploration, and experience replay. The outcomes show that the agent can play the game on its own, underscoring deep reinforcement learning's promise for challenging decision-making tasks.

Individual contribution and findings:

- Understanding the Research paper: “Review of reinforcement learning research” and creating its Literature Review.
- Written the Code for the manual Snake Game and implemented it.

Individual contribution to project report preparation:

- Deep Q Learning Explanation.
- Literature Review
- Snake Game (Manual Game Explanation)

Full Signature of Supervisor Full Signature of Student

INDIVIDUAL CONTRIBUTION REPORT

A Reinforcement Learning & Deep Q Learning based approach to the Snake Game & Building a Snake AI

Debarka Mandal
21051888

Abstract: In this project, Deep Q-Learning (DQL), a reinforcement learning method that blends Q-Learning with neural networks, is used to create an AI agent for the traditional Snake Game. By eating, the agent gains the ability to move across the game's environment, avoid collisions, and increase its score. By approximating Q-values for every action, the DQL model enables the agent to discover successful tactics via experimentation. Training is stabilized by methods such as target networks, epsilon-greedy exploration, and experience replay. The outcomes show that the agent can play the game on its own, underscoring deep reinforcement learning's promise for challenging decision-making tasks.

Individual contribution and findings:

- Understanding the Research paper: “Exploration of Reinforcement Learning to Play Snake Game” and creating its Literature Review.
- Written the Code for the Creating and training the neural network and implemented it.

Individual contribution to project report preparation:

- Deep Q Learning Example Explanation.
- Literature Review
- Results

Full Signature of Supervisor Full Signature of Student