MiniProject Report

on

Implementation of Different Games using Neural Networks and Reinforcement Learning

Submitted by

Group id

Project Members

1. Darshan Gramopadhye 1032221044
2. Ojas Ninawe 1032221850
3. Rushil Pandhare 1032220863
4. Dheeraj Sharma 1032222102
5. Gaurang Patil 1032221535

Under the Guidance of

Dr Yogita Hande

Department of Computer Engineering and Technology
MIT World Peace University, Kothrud,
Pune 411 038, Maharashtra - India
2024-2025

Abstract

In this project, we applied Neural Networks and Reinforcement Learning to develop AI agents capable of playing three classic games: Car Racing, Mario, and Snake. Reinforcement Learning (RL) enables agents to learn optimal strategies by interacting with their environments and maximizing cumulative rewards. The three algorithms implemented in this project are **Proximal Policy Optimization (PPO)**, **Q-Learning**, and **Deep Q-Network (DQN)**.

Neural Networks

Neural Networks (NN) are computing systems inspired by the human brain, designed to recognize patterns in large datasets. Composed of layers of interconnected nodes (neurons), NNs are powerful tools for approximating complex functions and are used as function approximators in reinforcement learning. In RL tasks, NNs predict the best actions based on the agent's state.

Reinforcement Learning

Reinforcement Learning (RL) is a branch of machine learning where an agent learns to take actions in an environment to maximize a cumulative reward. The RL framework consists of:

- **Agent:** The entity making decisions (e.g., snake, Mario, or car).
- **Environment:** The system with which the agent interacts.
- **State:** The current representation of the environment.
- **Action:** The decision made by the agent at each state.
- **Reward:** The feedback the agent receives based on its action.

The goal is to develop a policy that maximizes long-term rewards through interaction with the environment.

Proximal Policy Optimization (PPO)

PPO is a policy-based RL algorithm. It optimizes the policy by updating it in a way that limits large deviations from the current policy, ensuring stable learning. PPO uses the **objective function**:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)]$$

PPO's objective ensures that the updates are both effective and stable by clipping changes that are too large.

Q-Learning

Q-Learning is a value-based algorithm that aims to learn the optimal action-value function Q(s,a)Q(s, a)Q(s,a), which represents the expected cumulative reward of taking action aaa in state sss. The agent updates the Q-values based on the Bellman Equation:

$$Q(s_t, a_t) = (1 - \alpha)Q(s_t, a_t) + \alpha(R(s_t, a_t, s_{t+1}) + \gamma \max_{a'} Q(s_{t+1}, a'))$$

The **Bellman Equation** forms the basis of Q-Learning by updating the current estimate of the Q-value using the reward and the best future Q-value.

Deep Q-Network (DQN)

DQN extends Q-Learning by approximating the Q-values with a **neural network** rather than a table. This is especially useful in environments with large state spaces, where a table-based approach would be impractical.

The key innovation in DQN is the use of **experience replay** and a **target network**:

1. **Experience Replay:** Stores past experiences in a replay buffer and randomly samples them to break the correlation between consecutive experiences.
2. **Target Network:** A second neural network, updated less frequently, is used to stabilize training by providing more consistent target Q-values.

The agent uses this loss function to minimize the difference between the predicted and actual Q-values.

Conclusion

By applying these algorithms, our project trains agents to successfully navigate through different environments (Car Racing, Mario, and Snake), maximizing long-term rewards. Each algorithm (PPO, Q-Learning, and DQN) brings unique strengths suited to the specific nature of each game environment, enabling the agent to learn and perform tasks efficiently.

Chapter 1
Introduction

With the rise of artificial intelligence (AI) and machine learning (ML), neural networks and reinforcement learning (RL) have gained widespread use in various domains. In this project, we aim to demonstrate how these techniques can be integrated into game development to create intelligent agents capable of learning through trial and error. By applying algorithms like PPO, DQN and Q-learning, we can develop AI models that mimic human decision-making, allowing for more dynamic and engaging gameplay. This project covers the implementation of three games that utilize these methods to train AI agents in real-time environments.

1.1 Problem Statement

The goal is to train an AI agent to autonomously play games using reinforcement learning algorithms, with a focus on optimizing performance through trial and error. We aim to demonstrate this with multiple games, each posing unique challenges to the AI agent.

1.2.1 Objectives

- Implement a car racing simulation using Unity's ML-Agent's framework.
- Develop a platformer game inspired by Mario, integrating reinforcement learning.
- Develop a Classic Snake Game using python and an AI Trainer integrating all the algorithms.
- Use Neural Networks and RL algorithms to train AI agents in each game environment.

Chapter 2
Literature Survey

The following research and studies informed our implementation choices:

2.1 AI in Game Development

AI has been a core element in game development for years, but with the advent of machine learning, the capacity to create agents that learn from experience has evolved. We utilized reinforcement learning techniques to enable our AI agents to make decisions based on environmental stimuli.

2.2 Reinforcement Learning Algorithms

We explored various RL algorithms, focusing on Proximal Policy Optimization (PPO), Q-learning (Bellman's Equation), and Deep Q-Networking Algorithms. PPO is widely used for continuous action spaces, while Q-learning works well with discrete actions, making it ideal for games like Mario and Snake.

2.3 Unity ML-Agents

The Unity Machine Learning Agents Toolkit (ML-Agents) is an open-source Unity plugin that enables games and simulations to serve as environments for training intelligent agents. Agents can be trained using reinforcement learning, imitation learning, or other machine learning methods through a simple-to-use Python API.
We also provide implementations (based on TensorFlow) of state-of-the-art algorithms to enable game developers and hobbyists to easily train intelligent agents for 2D, 3D and VR/AR games. These trained agents can be used for multiple purposes, including controlling NPC behavior (in a variety of settings such as multi-agent and adversarial).

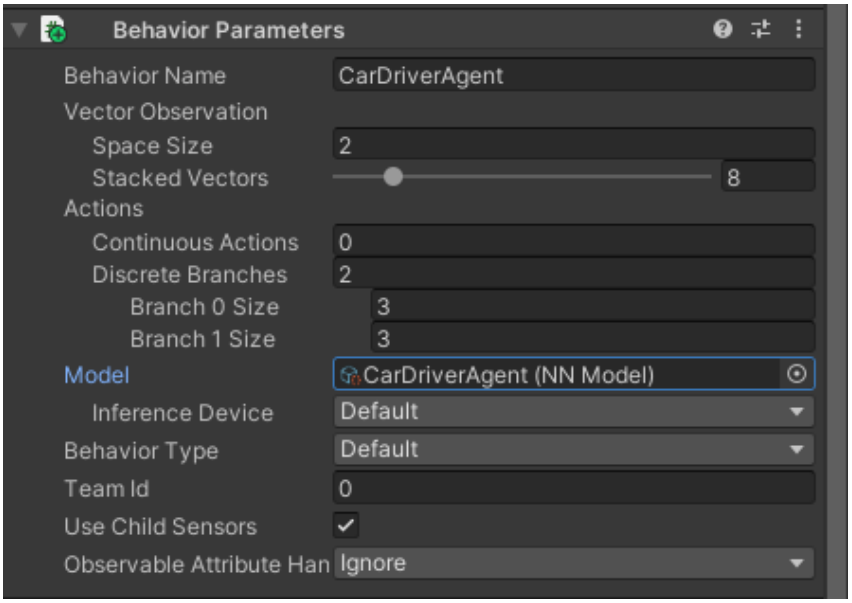

Fig: Race Car Trainer

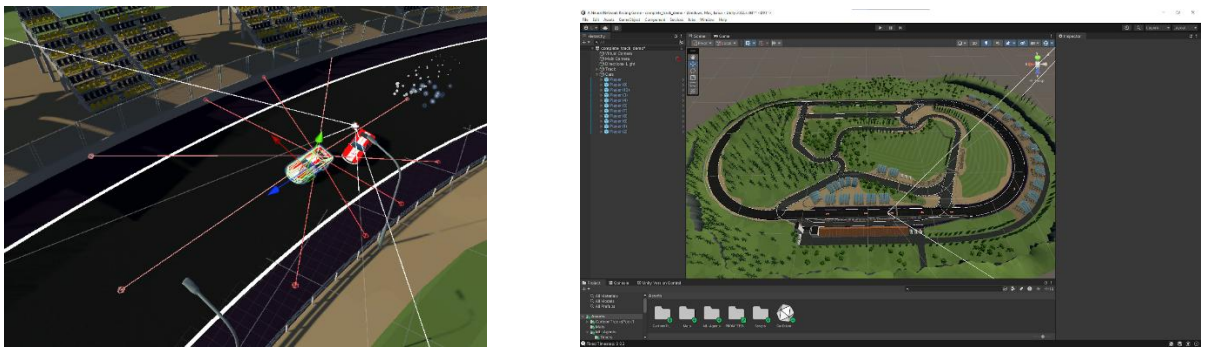

Figure 2.1: Race Car Raycast & Track Development

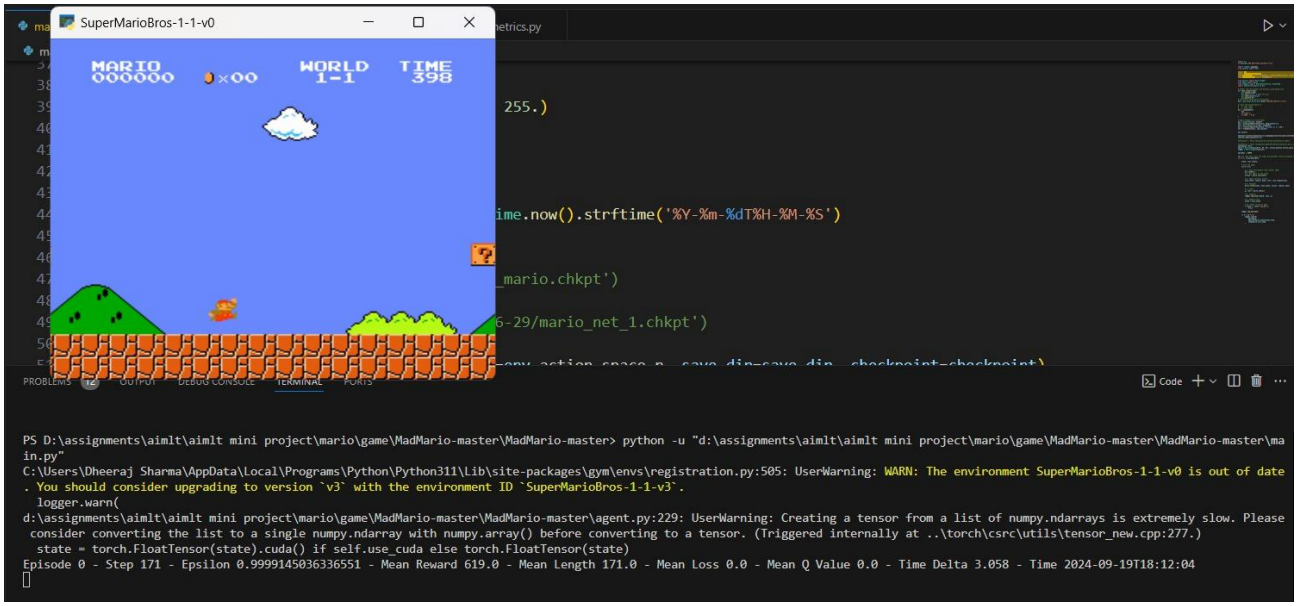

Figure 2.2: Mario Platformer Trainer

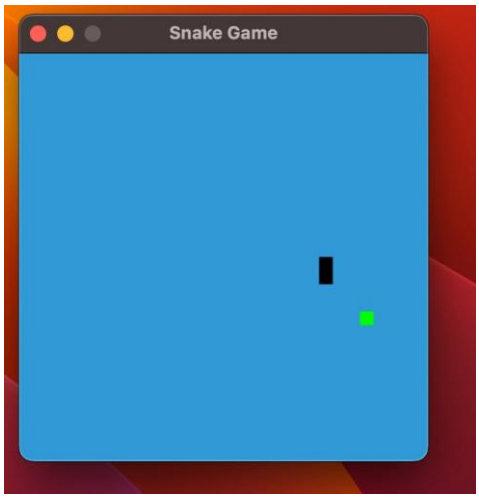

Figure 2.3: Classic Snake Trainer

Q-Learning:
Q-learning finds the best course of action given the current state of the agent through trial and error. It does so by randomizing its actions, and repeating what works.

Deep Q-Learning Networks:
Sometimes, a table isn't very efficient. That's where **deep Q-learning** comes in. It is basically the same as Q-learning, except with a neural network instead of a table. It too uses the Belleman equation. This is what I am using to train the snake game!

Proximal Policy Optimisation:
Algorithms based on policy functions, such as PPO, can learn a policy function, which can select the best action in each state. In the Snake game, PPO can directly learn a strategy function to select the best action to maximize the return. Because the PPO algorithm can achieve faster training speed and a more stable training process while ensuring convergence, the PPO algorithm can obtain higher scores and better game performance in the Snake game.

Chapter 3
Methodology

3.1 Car Racing Game

The car racing game was developed using Unity's ML-Agent's toolkit. The cars were set up with ray perception sensors to detect the track's boundaries, and the AI was trained using PPO, a reinforcement learning algorithm suitable for continuous decision-making.

Key Components:

- **Discrete Branches:** Two input actions – forward-backward movement and turning – each with three states.
- **Raycast Sensors:** These sensors help the car perceive its environment by casting rays around it, ensuring it avoids obstacles and navigates the track effectively.

Training Process:

The training process for the car racing game involved several steps to prepare the Unity environment and the ML-Agents framework, set up the behaviour parameters, and initiate the learning algorithm. The following is an outline of the procedure used to train the car agent to navigate the track successfully:

Step 1: Setting Up Unity Environment

- **Game Scene:**
  The Unity environment was prepared with a racing track and 10 cars placed at various positions on the track. Both the track and the car models were sourced from the Unity Asset Store.
- **Agent Scripts:**
  We already had a script to control the car's movement, which included handling inputs like forward-backward motion and steering. The next step was to create the **Agent Script**, which allowed the car to learn from its environment using reinforcement learning.

Step 2: Adding Behavior Parameters

- **Discrete Action Space:**
  The agent required two discrete action branches, corresponding to the input actions:

    - **Forward-Backward Movement:** Three discrete states: no movement, forward movement, and backward movement.
    - **Turning:** Three discrete states: no turn, turn left, and turn right.

Since the input for our car is discrete (1 or 0), discrete actions were selected over continuous ones (such as a game controller).

- **Branch Sizes:**
  For each action branch, the branch sizes were set to 3. This allowed the agent to select between the three possible states for both movement and turning.

Step 3: Decision Requester and Perception Setup

- **Decision Requester:**
  The **Decision Requester** was set up to dictate how often the agent should take decisions. The default value of 3 was lowered to 1, ensuring the car could make decisions on every frame, which is crucial for a fast-paced game like racing where the state of the car changes rapidly.
- **Ray Perception Sensors:**
  The **Ray Perception Sensor** was added to the car to provide it with information about its surroundings. The sensor emitted rays in multiple directions (with an increased number of rays and an angle expanded to 180 degrees to cover a wider field of view). This helped the car detect objects in all directions, including behind, which was important to prevent the car from moving backward unnecessarily.

Step 4: Reward System Setup

- **Positive Rewards:**
  The car was rewarded for:
    - **Forward Movement:** A reward was given for moving forward, with the reward value being multiplied by the car's current velocity. This incentivized the agent to move forward faster.
    - **Checkpoint Completion:** Reaching checkpoints on the track provided additional rewards, encouraging the agent to complete laps efficiently.
- **Negative Rewards:**
  The car was penalized for:
    - **Backward Movement:** A negative reward was applied when the car moved backward, deterring the agent from reversing.
    - **Collisions:** The agent received a penalty for colliding with objects or going off-track, teaching it to avoid obstacles.

- **Steering Adjustment:**
  To improve the car's turning behavior, a **minimum steering time** was introduced. This ensured that the AI held the steering action long enough to make meaningful turns rather than making quick, inefficient adjustments.

Step 5: Training with ML-Agents

- **Starting the Training:**
  The training was initiated by starting the **ml-agents** Python program. This program interfaced with the Unity environment, enabling the agent to start interacting with the game and learning from its actions.
- **OnActionReceived Function:**
  The agent's decisions were handled by the **OnActionReceived** function. This function used simple switch cases to decide the agent's movement based on the input received:
  - Three cases were implemented for the forward-backward movement (do nothing, go forward, go backward).
  - Similar logic was applied for steering (turn left, turn right, no turn).

During each frame, the agent's actions were based on the observations it made about the track and its surrounding environment.

Step 6: Monitoring Training Progress

- **Training Data Output:**
  Periodically, the system output basic training data, such as the agent's success in navigating the track or its collisions. The data was saved and could be analyzed further to track the AI's learning progress.
- **Tensorboard for Data Visualization:**
  By navigating to the directory where the training results were saved and running **Tensorboard**, we visualized the training progress. Tensorboard provided graphs that showed key metrics such as:
  - **Cumulative Reward:** The total reward earned by the agent over each training episode, which increased as the agent learned to navigate the track more effectively.
  - **Episode Length:** The number of steps taken by the agent to complete a lap, which decreased as the agent became more efficient.

Step 7: Applying the Trained Model

- **Exporting the Model:**
  Once the model had completed training, the final trained model was exported as an **onnx** file. This file contained the learned behavior of the agent.
- **Inference-Only Mode:**
  The onnx file was added to the car model in Unity under the **Behavior Parameters**. The behavior type was changed to **Inference Only**, ensuring that the car no longer learned from its environment and only relied on the trained model to make decisions.

Step 8: Testing the Trained Agent

- **Running the Scene:**
  After the model was applied, the scene was run in Unity, and the trained car was observed driving autonomously, navigating the track using the behaviors it had learned during training. The agent could now complete laps efficiently without any manual intervention.

Fig: Graph Results of Car Race Trainer

3.2 Mario Platformer Game

For the Mario game, we utilized Q-learning to teach the AI agent how to collect coins and avoid enemies. The agent learns from positive reinforcement (collecting coins) and negative reinforcement (colliding with enemies or falling off platforms).

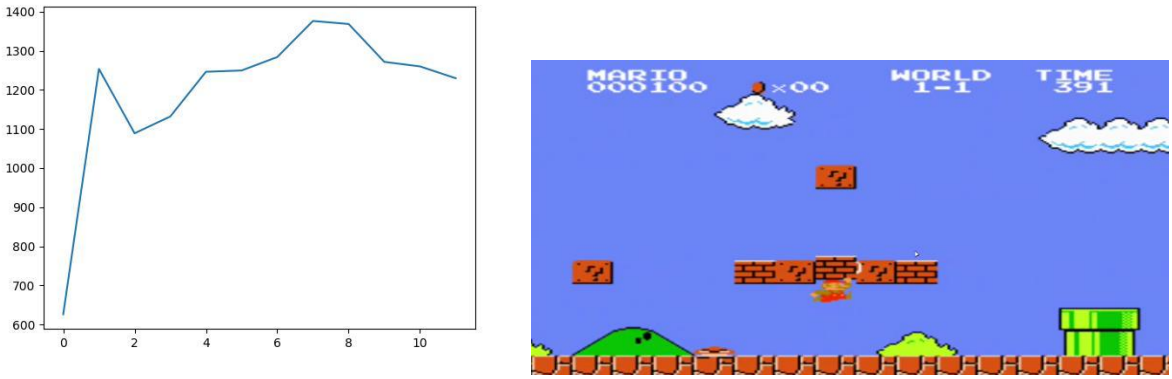

Fig: Reward Plot with GUI

Step 1: Initialize Environment

- **Game Scene:**
  The environment consists of typical Mario world elements like tubes, mushrooms, enemies, and gaps. Each action taken by Mario (such as jumping, moving left or right) results in a response from the environment with a new state, reward, and additional info about the game.
- **Agent Interaction:**
  When Mario makes an action, the environment responds by transitioning to the next state and providing the agent with feedback in the form of a reward based on its behavior.

Step 2: Preprocess Environment

- **Environment Data:**
  The environment's state is represented by a [3, 240, 256] size array, which includes RGB values that may contain more information than the agent needs. Since Mario's actions do not rely on elements like color, preprocessing the environment data is necessary to simplify the input to the agent.
- **Preprocessing Techniques:**
  To streamline the input data, several wrappers are applied to the environment:
  - **GrayScaleObservation:**
    This wrapper transforms the RGB image into grayscale, reducing the state size from [3, 240, 256] to [1, 240, 256]. This process decreases computational complexity without sacrificing key information for decision-making.
  - **ResizeObservation:**
    The observation is further downsampled to a square image to reduce dimensionality. The new state size becomes [1, 84, 84], making the data easier

for the model to process while retaining key spatial information about Mario's position and surroundings.

Step 3: Frame Handling with Wrappers

- **SkipFrame:**
  A custom wrapper, **SkipFrame**, is implemented to skip multiple intermediate frames. This helps reduce the amount of redundant data, as consecutive frames often do not vary much. The agent only processes every nth frame, with accumulated rewards from the skipped frames being passed along.
- **FrameStack:**
  The **FrameStack** wrapper is used to combine consecutive frames into a single observation point. This allows the agent to detect changes in Mario's movement (e.g., whether he is landing or jumping) based on data from multiple previous frames, providing context for the agent's actions.

Step 4: Reward System Setup

- **Positive Rewards:**
  The agent is rewarded for:
  - Collecting coins.
  - Reaching new areas in the level (progression).
  - Defeating enemies or completing objectives.
- **Negative Rewards:**
  The agent is penalized for:
  - Falling into gaps.
  - Colliding with enemies or obstacles.

  Rewards are accumulated over each frame and are influenced by the agent's overall performance during the episode.

Step 5: Training with Q-Learning

- **Initializing Training:**
  The training process is initiated, and the agent starts interacting with the game environment using Q-learning, which allows the agent to learn optimal policies by maximizing rewards. The agent takes actions, observes the results, and updates its Q-values based on the received feedback.
- **Action Handling:**
  The **OnActionReceived** function determines the agent's decisions, handling movements like jumping, moving left or right. The Q-learning algorithm updates the expected rewards for these actions based on the outcomes observed by the agent.

Step 6: Monitoring Training Progress

- **Training Data Output:**
  Throughout the training process, data about the agent's performance is collected. This includes information on how many coins were collected, how many levels were completed, and how frequently Mario encountered obstacles or fell into gaps.

- **Tensorboard for Data Visualization:**
  The training data can be visualized using Tensorboard, where key metrics such as **Cumulative Reward** and **Episode Length** are tracked:
  - **Cumulative Reward:** Shows the total reward accumulated over each episode.
  - **Episode Length:** Indicates the number of steps taken by the agent to complete a level or progress, showing improvements in efficiency as training progresses.

Step 7: Applying the Trained Model

- **Exporting the Model:**
  Once the training is complete, the final model is saved. The model contains the learned behavior for Mario based on all of the interactions and training episodes.
- **Inference Mode:**
  In the final game implementation, the model is set to inference mode, where the agent no longer learns from the environment but relies on the pre-trained model to make decisions.

Step 8: Testing the Trained Agent

- **Running the Scene:**
  After training, the Mario agent runs autonomously in the environment, navigating obstacles, avoiding enemies, and collecting rewards using the knowledge it gained during training. The agent consistently completes levels without manual intervention, showcasing the effectiveness of the trained model.

3.3 Classic Snake Game

Here we have used all three Reinforcement Learning Algorithms like PPO, Q – Learning and DNQ with Snake Game.

The environment is discrete and limited, because the Snake can only move in a limited game area, and can only take limited actions (up, down, left, right);

The state of each time step is completely observable because the greedy snake can see its surrounding environment and its own state
The goal is to maximize the game score, that is, get the maximum return. Therefore, reinforcement learning algorithms are very suitable for training the Snake game.
Reinforcement learning involves a machine learning algorithm that interacts with the environment through agents and obtains the maximum return by learning the optimal strategy.
The agent in reinforcement learning will make an action according to the state of the environment at each time step and get a reward from the environment. The agent maximizes the total long-term returns through learning, which requires the agent to find an optimal strategy to decide when to take which action.

Fig: Snake Game using Q Learning and Bellman Equation

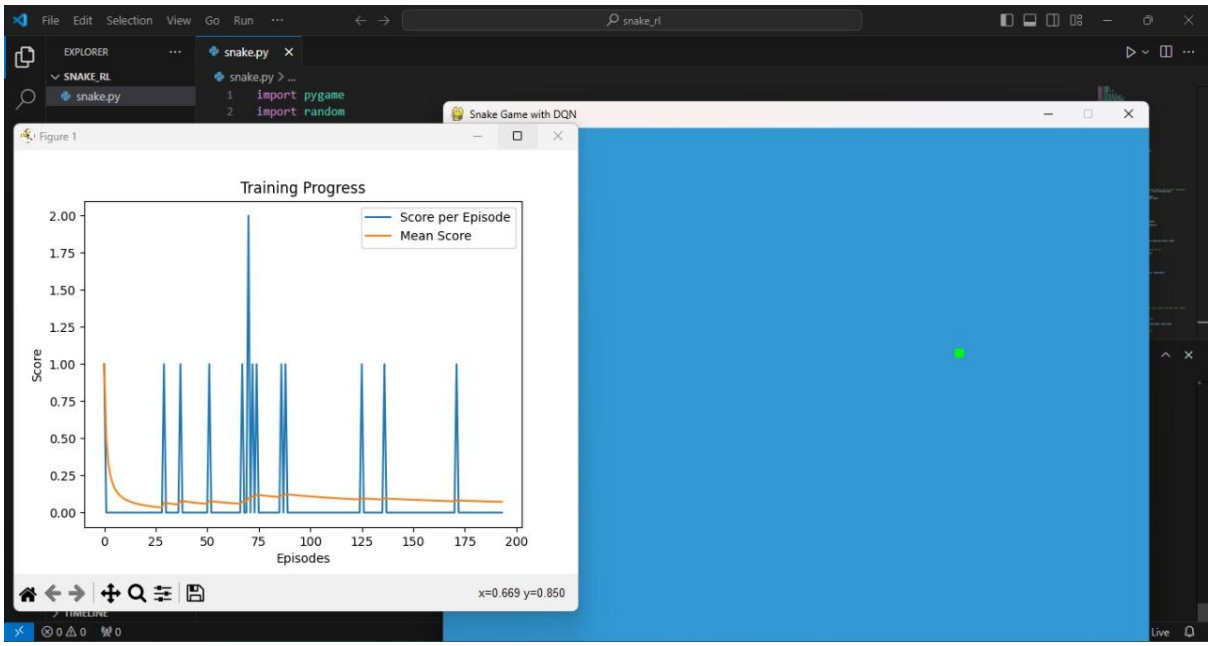

Fig: Snake Game using DQN RL

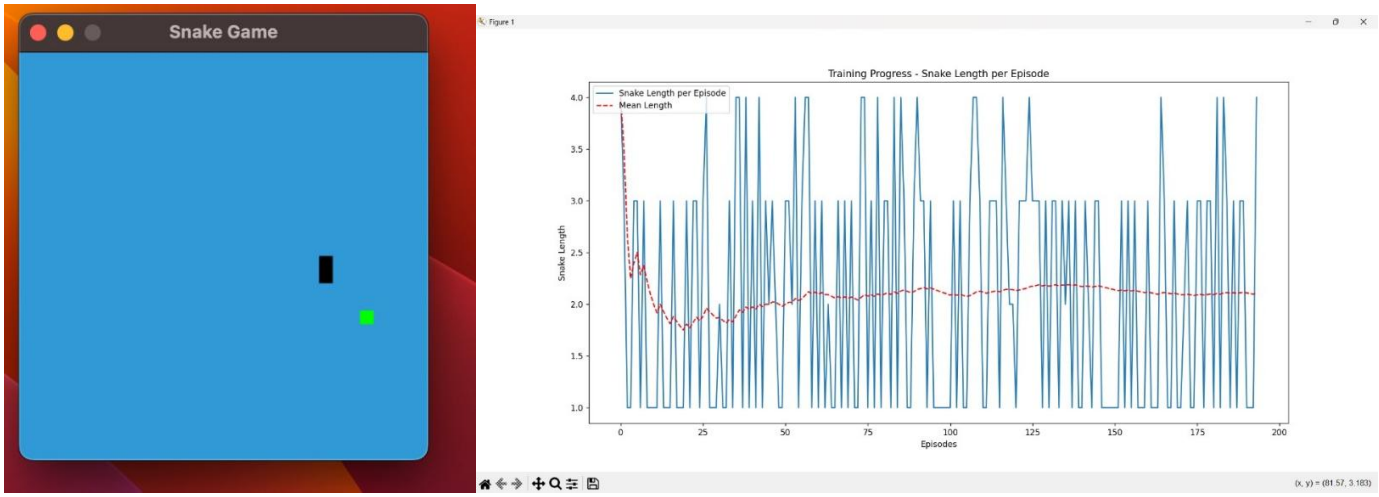

Fig: Snake Game using PPO Algorithm RL

Step 1: Initialize Environment

- **Game Scene:**
  The Snake game is a grid-based environment where the snake moves within a fixed, bounded area. The agent (snake) can take discrete actions: move **up**, **down**, **left**, or **right**. The environment is fully observable, meaning the snake has complete knowledge of its surroundings at every time step.
- **Agent Interaction:**
  At each step, the snake observes the current state (its position, the food's position, and obstacles such as walls and its body), takes an action, and receives feedback in the form of rewards.

Step 2: Discrete Action Space and Observations

- **Discrete Action Space:**
  The snake's movement is restricted to four discrete actions:
  - **Up**, **Down**, **Left**, and **Right**.
- **Observation Space:**
  The snake's observation includes its position relative to the food, the location of the walls, and its own body parts. Each state is fully observable, meaning the snake can see the entire grid and make decisions based on the current layout of the game.

Step 3: Reinforcement Learning Algorithms

Three algorithms were used to train the agent:

- **Proximal Policy Optimization (PPO):**
  A policy gradient method that is well-suited for environments with continuous or discrete action spaces. PPO was used to improve the agent's decision-making over time by maximizing cumulative rewards.
- **Q-Learning:**
  A value-based algorithm that updates a Q-table where each entry corresponds to the expected future reward for taking a specific action in a given state. The agent uses this table to choose the action with the highest expected reward.

- **Deep Q-Network (DQN):**
  DQN extends Q-learning by using a neural network to approximate the Q-value function. This allows the agent to handle more complex state spaces by learning from past experiences (replay memory) and adjusting its strategy based on the predicted rewards.

Step 4: Reward System Setup

- **Positive Rewards:**
  The agent is rewarded for:
  - **Eating Food:** A positive reward is given each time the snake consumes food, which increases the game score.
- **Negative Rewards:**
  The agent is penalized for:
  - **Colliding with the Wall:** The game ends when the snake hits the boundary, resulting in a negative reward.
  - **Colliding with Itself:** The snake also receives a penalty for hitting its own body, ending the game.
- **Long-Term Strategy:**
  The algorithms are designed to optimize the snake's performance over the long term, learning when to take certain actions (such as turning towards food or avoiding dangerous areas) to maximize total returns.

Step 5: Training with PPO, Q-Learning, and DQN

- **PPO Training Process:**
  The snake was trained using **Proximal Policy Optimization (PPO)**, a reinforcement learning algorithm designed for environments with discrete actions. Over time, PPO helped the agent learn which movements maximize rewards by balancing exploration and exploitation.
- **Q-Learning Training Process:**
  In **Q-learning**, the agent learned by updating a Q-table, which assigns expected rewards to each state-action pair. The agent continuously refined its table by interacting with the environment, improving its strategy with every episode.
- **DQN Training Process:**
  DQN used a neural network to approximate the Q-values for each action. This allowed the snake to generalize over states, making it more effective in larger or more complex versions of the Snake game. Replay memory was used to store past experiences, helping the agent learn from previous mistakes and successes.

Step 6: Monitoring Training Progress

- **Training Data Output:**
  Throughout training, data was collected on the snake's performance, including the number of food items consumed, the number of steps taken, and the number of collisions.
- **Tensorboard for Data Visualization:**
  As with the other games, **Tensorboard** was used to monitor the training progress. Key metrics such as **Cumulative Reward** and **Episode Length** were visualized:

- **Cumulative Reward:** The total rewards earned during each episode, showing how well the snake learned to maximize its score.
- **Episode Length:** The number of steps the snake took before the game ended, with longer episodes indicating better performance as the snake learned to avoid collisions.

Step 7: Applying the Trained Models

- **Exporting the Models:**
  Once training was complete, the models for each algorithm (PPO, Q-Learning, and DQN) were saved. These models contained the strategies the agent learned through interaction with the environment.
- **Inference-Only Mode:**
  The saved models were then used in inference mode, where the agent no longer learned from the environment but followed the pre-trained policy to play the game autonomously.

Step 8: Testing the Trained Agent

- **Running the Snake Game:**
  After applying the trained models, the Snake game was run with the agent controlling the snake. The agent demonstrated its ability to navigate the grid, avoid obstacles, and maximize the game score by collecting food and surviving for as long as possible without hitting the walls or itself.

Chapter 4
Results and Discussion

The Reinforcement Learning developed by various games each of the group members:
Dheeraj Sharma: DQN - Super Mario
Ojas Ninawe: PPO - 3D Car Racing
Darshan Gramopadhye: Q Learning + DQN - Snake Game
Rushil Pandhare: Q Learning - Snake Game
Gaurang Patil: PPO - Snake Game

The AI agents in all the games showed progressive improvement as training episodes increased. In the car racing game, the agent initially struggled with basic navigation, but after several training episodes, it learned to complete the track efficiently without colliding with obstacles. The Mario game's agent also showed significant learning, as it eventually mastered jumping over gaps and enemies to collect rewards.

Snake Game: DQN vs PPO vs Q-Learning

In the Snake Game, we implemented and compared three algorithms—**DQN**, **PPO**, and **Q-Learning**. Each algorithm handles decision-making and learning in unique ways. Here's how they compare:

1. DQN (Deep Q-Network)

- **Strengths:**
  DQN is particularly effective in environments with discrete action spaces like the Snake game. It uses a neural network to approximate the Q-values, which allows it to handle larger state spaces and generalize better than table-based Q-Learning.
- **Training:**
  DQN benefits from the use of **experience replay**, which helps break the correlation between consecutive steps by randomly sampling previous experiences, leading to more stable learning. The **target network** further helps to stabilize training by reducing the rapid fluctuations in the Q-values.
- **Performance in Snake Game:**
  DQN performs well in the Snake game due to its ability to approximate complex state-action mappings using neural networks. It learns to avoid collisions with the snake's own body and navigate toward the food, optimizing long-term rewards. However, it may require more computational power and longer training time compared to Q-Learning due to the neural network component.

2. PPO (Proximal Policy Optimization)

- **Strengths:**
  PPO is a policy gradient method that is known for its stability during training. In Snake, PPO ensures that large updates to the policy do not result in drastic changes, making learning smoother and more controlled.
- **Training:**
  PPO directly optimizes the policy by interacting with the environment and adjusting its actions based on the advantage function. Its clipping mechanism prevents large, destabilizing updates, which can be an issue in environments like Snake with sudden, terminal states (such as hitting a wall).
- **Performance in Snake Game:**
  PPO performs effectively in the Snake game, particularly when there is a need for long-term decision-making. The algorithm balances exploration and exploitation and adapts to changes in the environment as the snake grows longer, requiring more complex navigation.

3. Q-Learning

- **Strengths:**
  Q-Learning is a simple and effective algorithm for discrete environments like Snake. It works by maintaining a Q-table, which is updated using the Bellman Equation. Although it is computationally less intensive compared to DQN, it struggles in environments with large or continuous state spaces.
- **Training:**
  Q-Learning updates the Q-table after every action using the immediate reward and the expected future reward. While this approach works well in small, discrete environments, it becomes inefficient as the state space grows (e.g., as the snake becomes longer and the game area expands).
- **Performance in Snake Game:**
  In the early stages of the Snake game, Q-Learning performs quite well. However, as the game progresses and the state space become larger and more complex, Q-Learning may struggle to find optimal strategies compared to DQN or PPO.

PPO in Snake Game vs PPO in 3D Racing Car Game

Although PPO is used in both the Snake game and the 3D Racing Car game, the environments differ significantly, leading to distinct behaviours and performance:

1. PPO in Snake Game

- **Discrete Action Space:**
  In Snake, the action space is discrete (up, down, left, right). PPO handles this well, as it is designed to work with both discrete and continuous actions. The grid-based environment in Snake is simpler compared to the 3D Racing Car game.
- **State Representation:**
  The state space in Snake is relatively smaller and easier to observe. The snake can see the entire grid, which makes it fully observable, allowing PPO to learn effective strategies without having to infer missing information.
- **Performance:**
  PPO in Snake focuses on maximizing long-term rewards by avoiding walls and self-collisions while heading toward the food. The simplicity of the environment means that PPO can converge relatively quickly to an optimal strategy.

2. PPO in 3D Racing Car Game

- **Continuous Action Space:**
  In the 3D Racing Car game, the action space is continuous (e.g., forward acceleration, turning angles), making PPO's flexibility a key advantage. It excels at handling the continuous control required for smooth navigation and precise steering.
- **State Representation:**
  The state space in the 3D Racing Car game is more complex due to the partially observable nature of the environment. The car uses raycast sensors to detect obstacles, making it necessary for PPO to infer hidden states based on sensory inputs.
- **Performance:**
  PPO performs well in the 3D Racing Car game by adjusting the car's steering and speed based on the current sensor inputs. The continuous action space requires PPO to make subtle adjustments, and its ability to limit large updates to the policy ensures smooth and stable learning.

DQN in Super Mario vs DQN in Snake Game

The two games differ greatly in terms of complexity and state space, leading to different implementations and performance of the DQN algorithm.

1. DQN in Super Mario

- **State Representation:**
  In Super Mario, the state is represented as a series of frames (images), which the DQN model processes. The game requires the agent to handle more complex visual data and infer the current position of enemies, gaps, and platforms. The use of **frame stacking** allows the agent to have temporal context, helping it understand Mario's velocity and actions across multiple frames.
- **Action Space:**
  The action space in Super Mario is slightly larger than Snake. Mario can move left, right, jump, and sometimes take special actions. This makes the decision-making process more complex compared to the Snake game.
- **Training and Performance:**
  DQN in Super Mario has to handle a vast, partially observable environment with dynamic elements like enemies and moving platforms. The training is more computationally intensive and requires more episodes to converge compared to the simpler Snake environment. The visual complexity also means that DQN must process raw pixel data efficiently.

2. DQN in Snake Game

- **State Representation:**
  In Snake, the state is simpler, involving the snake's position relative to obstacles (walls, itself) and the food. This discrete environment is easier for DQN to handle, requiring less computational power and fewer frames to be processed.
- **Action Space:**
  The action space is also smaller in Snake (only four actions: up, down, left, right), making it less complex compared to Super Mario.
- **Training and Performance:**
  DQN in Snake converges faster due to the simpler state space and action space. The algorithm can quickly learn optimal strategies for avoiding walls and self-collisions while moving toward food. Unlike Super Mario, there are no moving enemies or dynamic elements to consider.

Chapter 5
Conclusion

Our project successfully demonstrates the application of Neural Networks and Reinforcement Learning in game environments. By leveraging reinforcement learning, we were able to create AI agents that learned from their interactions with the game environment and improved over time. The techniques used in this project could be expanded to more complex games and scenarios, opening the door for further research and development in AI-driven game design.

In this project, we explored the implementation of three reinforcement learning algorithms—Q-Learning, Proximal Policy Optimization (PPO), and Deep Q-Network (DQN)—across three different games: Snake, Mario, and Racing. Each game presented unique challenges in terms of environment complexity and decision-making requirements.

The complexity of the games increases in the following order:

1. **Snake Game** – A simple, fully observable, discrete environment with a limited action space.
2. **Mario Game** – A moderately complex, partially observable environment with dynamic elements like enemies and moving platforms.
3. **Racing Game** – The most complex, continuous environment requiring precise control and navigation using raycast sensors.

Q-Learning was found to be effective in simple environments like the Snake game, where the state and action spaces are discrete and small. However, due to its limitations in handling large or continuous state spaces, Q-Learning was not implemented in more complex games like Mario and Racing, which require more sophisticated algorithms like **PPO** and **DQN**.

Overall, PPO and DQN proved to be robust algorithms for handling both discrete and continuous environments, with PPO excelling in scenarios that required stability in policy updates and DQN performing well in environments with visual inputs and complex state-action relationships.

This project demonstrates the importance of choosing the right reinforcement learning algorithm based on the complexity of the environment and the nature of the tasks the agent needs to perform.

References

[1] D. Vernon, G. Metta, and G. Sandini, "A survey of artificial cognitive systems," IEEE Transactions on Evolutionary A., vol. 247, pp. 529-551, April 2014.
[2] Unity Technologies, "ML-Agents Toolkit," available online at: https://unity.com/ML-Agents.
[3] Sutton, R. S., and Barto, A. G., Reinforcement Learning: An Introduction, MIT Press, 1998.
[4] N. Choe et al., "Emergent Behaviors in Reinforcement Learning," available online at: https://arxiv.org/pdf/2301.05300.
[5] D. Brown, "Q-learning Solution," available online at: https://users.cs.utah.edu/~dsbrown/classes/cs6300/practice/Q-learn_solution.pdf.
[6] M. J. Powers, "Solving Car Racing with Reinforcement Learning," available online at: https://notanymike.github.io/Solving-CarRacing/.
[7] A. Paszke et al., "Reinforcement Learning with PyTorch for Super Mario," available online at: https://pytorch.org/tutorials/intermediate/mario_rl_tutorial.html#.
[8] N. Zhou, "Teaching an AI to Play the Snake Game Using Reinforcement Learning," available online at: https://medium.com/@nancy.q.zhou/teaching-an-ai-to-play-the-snake-game-using-reinforcement-learning-6d2a6e8f3b1c.