

AI GAME PLAYING: USING DEEP REINFORCEMENT LEARNING

Submitted in partial fulfillment of the requirements for the award of
Bachelor of Engineering degree in Computer Science and Engineering with
Specialization in Artificial Intelligence

By

MANERI VASANTH KUMAR (Reg.No - 40731058)

MEKALA NAVADEEP (Reg.No - 40731060)



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SCHOOL OF COMPUTING

SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY

(DEEMED TO BE UNIVERSITY)

Category-1 University by UGC

Accredited with Grade “A++” by NAAC | 12B Status by UGC | Approved by AICTE

JEPPIAAR NAGAR, RAJIV GANDHISALAI,

CHENNAI - 600119

APRIL - 2024



SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY

(DEEMED TO BE UNIVERSITY)

Category - I University by UGC

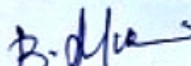
Accredited "A++" by NAAC | Approved by AICTE

www.sathyabama.ac.in

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

BONAFIDE CERTIFICATE

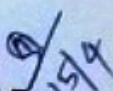
This is to certify that this Project Report is the bonafide work of **MEKALA NAVADEEP (40731060)** and **MANERI VASANTH KUMAR (40731058)**, who carried out the Project Work entitled "**AI GAME PLAYING: USING DEEP REINFORCEMENT LEARNING**" under my supervision from November 2023 to April 2024.

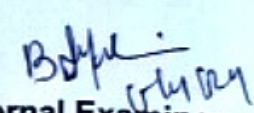

Internal Guide

Dr. B. ANKAYARKANNI M.E., Ph.D.


HEAD OF THE DEPARTMENT
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Dr. LAKSHMANAN M.E., Ph.D.
Jeppiaar Nagar, Rajiv Gandhi Salai,
Chennai - 600 119.

Submitted for Viva vice Examination held on 15-04-2024

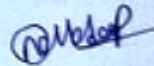

Internal Examiner


External Examiner

DECLARATION

I, **Mekala Navadeep (Reg.No - 40731060)**, hereby declare that the Project Work Report entitled "**AI GAME PLAYING: USING DEEP REINFORCEMENT LEARNING.**" done by me under the guidance of **Dr. B. Ankayarkanni, M.E.,Ph.D.** is submitted in partial fulfillment of the requirements for the award of Bachelor of Engineering degree in **Computer Science and Engineering with Specialization in Artificial Intelligence.**

DATE: 15-04-2024



PLACE: Chennai

SIGNATURE OF THE CANDIDATE

ACKNOWLEDGEMENT

I am pleased to acknowledge my sincere thanks to **Board of Management of SATHYABAMA INSTITUTE OF SCIENCE AND TECHNOLOGY** for their kind encouragement in doing this project and for completing it successfully. I am grateful to them.

I convey my thanks to **Dr. T.Sasikala M.E., Ph. D, Dean**, School of Computing, **Dr. L . Lakshmanan, M.E., Ph.D.**, Head of the Department of Computer Science and Engineering for providing me necessary support and details at the right time during the progressive reviews.

I would like to express my sincere and deep sense of gratitude to my Project Guide **Dr.B.Ankayarkanni M.E., Ph.D.**, for her valuable guidance, suggestions and constant encouragement paved way for the successful completion of my project work.

I wish to express my thanks to all Teaching and Non-teaching staff members of the **Department of Computer Science and Engineering** who were helpful in many ways for the completion of the project.

ABSTRACT

Deep Reinforcement Learning (DRL) has demonstrated remarkable success in a wide range of applications, from game playing to robotic control. In this study, we explore the application of DRL techniques to the domain of marionette control, a complex and challenging task that involves coordinating the movements of multiple strings and joints to manipulate a puppet-like figure. The objective of our research is to develop an autonomous marionette control system that can learn to perform intricate and expressive movements without the need for pre-programmed control strategies. We employ state-of-the-art DRL algorithms, including Deep Q-Networks (DQN) and Proximal Policy Optimization (PPO), to train a virtual marionette agent in a simulated environment. We describe the design of the marionette control environment, which includes a physically accurate simulation of the marionette and its strings. The agent receives sensory inputs from the simulated marionette, including joint positions, string tensions, and visual observations, and it learns to control the puppet by sending precise motor commands to the strings. Our experimental results demonstrate the feasibility of using DRL for marionette control. The trained agents are capable of performing a wide range of movements, from basic gestures to complex dances, and can adapt to changing scenarios and goals. We discuss the challenges and limitations of this approach, as well as potential directions for future research in improving marionette control through DRL. Overall, our study contributes to the growing body of research in the field of deep reinforcement learning and showcases its potential in creative and artistic domains such as marionette performance. The developed techniques may find applications in entertainment, puppetry, and animation, as well as inspiring further exploration of DRL in unconventional domains.

TABLE OF CONTENT

CHAPTER NO.	TITLE	PAGE NO.
	ABSTRACT	v
	LIST OF FIGURES	viii
1	INTRODUCTION	1
2	LITERATURE SURVEY	3
	2.1 Inferences from Literature Survey	5
	2.2 Open problems in Existing System	7
3	REQUIREMENT ANALYSIS	9
	3.1 Feasibility Studies/Risk Analysis of the Project	9
	3.2 Software Requirements Specification Document	12
4	DESCRIPTION OF PROPOSED SYSTEM	15
	4.1 Selected Methodology or process model	16
	4.2 Architecture / Overall Design of Proposed System	18
	4.3 Description of Software for Implementation and Testing plan of the Proposed Model/System	20
	4.4 Project Management Plan	22
5	IMPLEMENTATION DETAILS	25
	5.1 Development and Deployment Setup	25
	5.2 Algorithms	27
	5.3 Testing	29
6	RESULTS AND DISCUSSION	31
7	CONCLUSIONS	35
	7.1 Conclusion	35
	7.2 Future work	36
	7.3 Research Issues	38
	7.4 Implementation Issues	39
	REFERENCES	42
	APPENDIX	44
	A SOURCE CODE	44

LIST OF FIGURES

FIG. NO.	FIGURE NAME	PAGE NO.
4.1	System Architecture for Reinforcement Learning	18
4.2	Deep Q Learning	19
5.1	Mario Environment	25
6.1	Agent activity over 20 episodes	32
6.2	Agent Performances Over Episodes	33
6.3	Episodes length over Episodes	34

CHAPTER 1

INTRODUCTION

Reinforcement learning, a subfield of machine learning, has garnered significant attention in recent years for its remarkable success in solving complex problems through interaction with an environment. In particular, deep reinforcement learning (DRL) represents a cutting-edge approach that combines deep learning techniques with reinforcement learning to tackle a wide range of real-world challenges. This project report explores the application of DRL in playing Mario game, shedding light on its capabilities, challenges and potential impact. Using artificial intelligence (AI) and deep learning (DL) algorithms to play game has been widely discussed and investigated, because valuable observations can be made on the DL play pattern vs. that of a human player, and such observations provide knowledge on how to improve the algorithms. Mario AI Competition provides the framework to play the classic title Super Mario Bros, and we are interested in using ML techniques to play this game. Reinforcement Learning (RL) is one widely-studied and promising AI method for implementing agents that can simulate the behavior of a player. In this project, we study how to construct an RL Mario controller agent, which can learn from the game environment. One of the difficulties of using RL is how to define state, action, and reward. In addition, playing the game within the framework requires real-time response, therefore the state space cannot be too large. We use a state representation similar to , that abstracts the whole environment description into several discrete-valued key attributes. We use the Q-Learning algorithm to evolve the decision strategy that aims to maximize the reward. Our controller agent is trained and tested by the 2012 Mario AI Competition evaluation system. Another noteworthy challenge in reinforcement learning stems from the fact that rewards are sparse and time-delayed. When the agent earns a reward, we must determine which of the preceding actions played a role in getting that reward, and to what extent. Despite these challenges, a convolutional neural network, trained with a Double Q-learning algorithm and updated with stochastic gradient descent can reasonably well compared to a new human player Mario Deep Reinforcement Learning Project. The heart of this project lies in the development and training of AI agents that can not only mimic human-like gaming abilities but also surpass them. Through countless

iterations, these agents learn to explore the Mario world, adapt 2 their strategies, and make decisions that lead to the best possible outcomes. This project explores a range of reinforcement learning algorithms, including but not limited to Q-learning, Deep Q-Networks (DQN), Proximal Policy Optimization (PPO), and more. These algorithms, often driven by neural networks, are at the forefront of artificial intelligence research, and their application to the Mario game showcases their potential for autonomous decision-making and problem-solving. However, this endeavor is not without its challenges. The Mario game is a complex, dynamic, and unpredictable environment, demanding intelligent agents to address issues like credit assignment, sparse rewards, exploration-exploitation trade-offs, and the ability to generalize across different game levels. Overcoming these challenges is not only an exploration of AI's capabilities but also a testament to the power of perseverance and innovation. The Mario Deep Reinforcement Learning Project isn't just about mastering a video game; it's about pushing the boundaries of what artificial intelligence can achieve. The insights gained here have far-reaching applications, from autonomous robotics to decision-making systems for complex real-world scenarios. As we delve into this project, we aim to celebrate the legacy of Mario while contributing to the ever-evolving field of AI.

CHAPTER 2

LITERATURE SURVEY

This problem statement has been extensively studied over the past 5 years by researchers and automotive companies in a bid to create a solution, and all their solutions vary from analyzing various patterns.

Literature Review 1

Author: Mnih et al.

Year of Publication: 2013

Description: Mnih et al.'s groundbreaking work in 2013 introduced the concept of Deep Q-Networks (DQN) in reinforcement learning, leveraging deep learning to approximate Q-values and revolutionizing the field by achieving impressive results on Atari 2600 games. However, DQN may exhibit instability during training due to overestimation bias and requires careful hyperparameter tuning for reliable performance.

Pros: Revolutionized reinforcement learning, achieved impressive results on Atari 2600 games.

Cons: Instability during training, requires careful hyperparameter tuning.

Literature Review 2

Author: Silver et al.

Year of Publication: 2016

Description: Silver et al. presented the AlphaGo program in 2016, showcasing the power of deep reinforcement learning in defeating human Go champions. While AlphaGo demonstrated remarkable success in strategic games, its applicability is primarily limited to games and structured domains, and it poses computational challenges.

Pros: Demonstrated success in defeating human Go champions.

Cons: Limited applicability outside of games, computational challenges.

Literature Review 3

Author: Schulman et al.

Year of Publication: 2017

Description: Schulman et al.'s 2017 paper introduced the Proximal Policy Optimization (PPO) algorithm, offering high stability and robustness in deep reinforcement learning. PPO has achieved state-of-the-art performance on various tasks but may require computationally expensive training and hyperparameter tuning.

Pros: High stability and robustness, state-of-the-art performance.

Cons: Requires computationally expensive training, hyperparameter tuning.

Literature Review 4

Author: Ha et al.

Year of Publication: 2018

Description: Ha et al. presented the World Models framework in 2018, combining deep reinforcement learning with generative models to enhance training efficiency in simulated environments. While effective, World Models' applicability is limited to scenarios with readily created simulators.

Pros: Enhanced training efficiency in simulated environments.

Cons: Limited applicability to scenarios with readily created simulators.

Literature Review 5

Author: Fujimoto et al.

Year of Publication: 2018

Description: Fujimoto et al. introduced the Soft Actor-Critic (SAC) algorithm in 2018,

addressing exploration challenges in deep reinforcement learning. SAC offers improved sample efficiency and stability but may be computationally complex and require fine-tuning for optimal performance.

Pros: Improved sample efficiency and stability.

Cons: Computationally complex, requires fine-tuning.

Literature Review 6

Author: Hessel et al.

Year of Publication: 2018

Description: Hessel et al.'s work in 2018 introduced the Asynchronous Advantage Actor-Critic (A3C) algorithm, focusing on distributed reinforcement learning for enhanced training speed and scalability. A3C accelerates training through parallelization but poses challenges in distributed environments.

Pros: Enhanced training speed and scalability.

Cons: Poses challenges in distributed environments.

2.1 INFERENCES FROM LITREATURE SURVEY

1.DQN – Deep Q-Networks:

The introduction of Deep Q-Networks (DQN) marked a pivotal moment in reinforcement learning, demonstrating the potential of deep neural networks in achieving remarkable results in Atari 2600 games. However, it also highlighted challenges, particularly the instability during training and the need for meticulous hyperparameter tuning.

2.AlphaGo:

AlphaGo, powered by deep reinforcement learning, showcased the extraordinary capabilities of AI in complex strategic games like Go. While this achievement underscores the potential of deep RL in gaming, its limited applicability to specific domains and substantial computational requirements pose scaling challenges.

3.PPO-Proximal Policy Optimization:

Proximal Policy Optimization (PPO) has become a widely adopted deep reinforcement learning method due to its stability and state-of-the-art performance on benchmark tasks. Nevertheless, its computational demands and the requirement for meticulous hyperparameter tuning are factors that need consideration.

4.World Models:

The World Models framework combines deep RL with generative models, showing promise in improving data efficiency and sample complexity. However, its applicability is constrained to domains where simulators can be created, limiting its suitability for complex, high-dimensional observation scenarios.

5.SAC-Soft Actor-Critic:

The Soft Actor-Critic (SAC) algorithm offers improved sample efficiency and stability, particularly in continuous action spaces and complex tasks. However, its computational complexity and the need for fine-tuning should be considered for optimal performance.

6.Asynchronous Advantages Actor-Critic:

The Asynchronous Advantage Actor-Critic (A3C) algorithm focuses on distributed reinforcement learning, accelerating training and enhancing scalability. Its parallelization approach makes it versatile across domains, but implementation challenges in distributed environments and substantial computational resources are noteworthy.

2.2 OPEN PROBLEMS IN EXISTING SYSTEM

1.Sample Efficiency: Many deep reinforcement learning algorithms require a large number of samples to learn effectively. Improving sample efficiency is a critical challenge to enable these algorithms to learn from fewer interactions with the environment.

2.Generalization: Deep reinforcement learning systems often struggle with generalizing from the training environment to unseen scenarios. Enhancing the ability to transfer knowledge from one game or task to another remains an open problem.

3.Exploration: Balancing exploration and exploitation is a fundamental challenge. Designing algorithms that explore efficiently in unknown environments without getting stuck in suboptimal policies is an ongoing issue.

4.Catastrophic Forgetting: Deep RL models can forget previously learned knowledge when trained on new tasks. Mitigating catastrophic forgetting to enable continual learning and adaptation is a critical problem.

5.Safety and Ethics: Ensuring the safety of RL agents is paramount. Developing methods to prevent harmful behaviours and address ethical considerations, such as agent biases and unintended consequences, is an open problem.

6.Real-World Robotic Control: Transferring deep RL agents from simulated environments to real-world robotic systems is challenging. Bridging the simulation-to-reality gap for practical applications remains an open problem.

7.High-Dimensional Observations: Dealing with high-dimensional observations, such as images or sensor data, in an efficient and effective manner is an ongoing challenge.

8.Human-Agent Interaction: Enabling RL agents to interact with humans in a

natural and safe manner, especially in environments with human involvement, presents challenges in communication, collaboration, and interpretability.

9.Parallelization and Scalability: Developing scalable methods for parallelization and distributed reinforcement learning to handle large-scale environments and complex tasks remains an open problem.

10.Interpretable and Explainable RL: Creating methods to make deep reinforcement learning models more interpretable and explainable to humans is essential, especially in critical applications like healthcare and finance.

11.Sim-to-Real Transfer: Improving techniques for transferring policies learned in simulation to real-world environments with different dynamics and challenges is a significant challenge.

12.Adversarial Attacks: Protecting deep RL systems from adversarial attacks and ensuring their robustness is an important open problem, especially in security-critical applications Long- 8Term Planning: Developing deep RL algorithms capable of long-term planning and decision-making, even in situations with delayed rewards and complex dependencies, remains a challenging area of research.

CHAPTER 3

REQUIREMENT ANALYSIS

3.1 FEASIBILITY STUDIES/RISK ANALYSIS OF THE PROJECT

Conducting feasibility studies and risk analysis is a crucial step in any project, especially one involving complex technologies like AI and ML. Here's how you might approach these aspects for your project on access control systems using AI and ML

Feasibility Studies

Technical Feasibility

- **Data Availability:** Ensure access to sufficient and high-quality data for training and testing the AI agent.
- **Hardware and Software:** Assess whether the required hardware (e.g., GPUs) and software tools (e.g., deep learning frameworks) are available and within budget.

Operational Feasibility

- **Scalability:** Evaluate whether the system can be scaled up to handle various games or scenarios effectively.
- **Integration:** Consider how the AI agent will integrate into existing gaming environments or platforms.

Economic Feasibility

- **Cost-Benefit Analysis:** Estimate the costs involved in development, implementation, training, and maintenance. Compare this with the expected benefits in terms of efficiency, security, and other relevant factors. Make sure that the benefits outweigh the costs.
- **Return on Investment (ROI):** Estimate the potential benefits, such as
 - improved game performance or reduced human labor costs, and
 - compare them to the project's costs.

Legal and Ethical Feasibility

- **Compliance:** Ensure that the project complies with relevant laws and regulations, especially regarding data privacy and ethical considerations in gaming environments.
- **Ethical Considerations:** Evaluate the ethical implications of using AI for access control, ensuring fairness, transparency, and accountability in decision-making processes. Develop ethical guidelines for system behavior.

Scheduling Feasibility

- **Project Timeline:** Develop a detailed project timeline considering the complexity of integrating AI and blockchain. Identify potential bottlenecks and allocate realistic time frames for each phase of the project. It is important to have a realistic timeline to avoid delays and overruns.

Risk Analysis Technical Risks

- **Technology Challenges:** Identify potential technical challenges in implementing AI algorithms. These challenges may include data compatibility issues, security vulnerabilities, and scalability challenges.
- **Data Security:** Assess risks related to data breaches, unauthorized access, or vulnerabilities in the AI and ML components. Develop robust security measures to mitigate these risks.

Operational Risks

- **User Adoption:** Anticipate resistance to change and develop strategies to ensure smooth user adoption and training programs. Provide users with adequate training and support to help them transition to the new system.
- **Operational Disruptions:** Consider the risks of operational disruptions during the transition phase. Develop contingency plans to mitigate these risks. For example, you may need to have a backup system in place in case of any unexpected downtime.

Economic Risks

- **Cost Overruns:** Identify potential areas where costs might overrun and develop strategies to control expenses. Create a detailed budget and track spending closely to avoid cost overruns.
- **Revenue Generation:** If applicable, assess the risks related to revenue generation. For instance, if the system is a commercial product, analyse market demand and competition. Make sure that there is a viable market for your product and that you have a competitive advantage.

Legal and Ethical Risks

- **Regulatory Compliance:** Stay updated with changing regulations and assess the risks associated with non-compliance. Develop strategies to adapt to legal changes. For example, you may need to update your system or policies to comply with new regulations.
- **Ethical Concerns:** Evaluate potential ethical dilemmas related to AI decision-making and blockchain transparency. Establish ethical guidelines for system behavior. For example, you need to ensure that the AI system is fair and unbiased in its decision-making.

Scheduling Risks

- **Delays:** Identify potential causes of project delays, such as technical

challenges, resource constraints, or unexpected complications, and have contingency plans in place. For example, you may need to have a backup plan in case a key team member leaves the project.

Mitigation Strategies

- **Risk Mitigation:** Develop detailed risk mitigation strategies for identified risks. This might include contingency plans, regular security audits, and continuous monitoring of legal and regulatory changes. For example, you may need to have a backup system in place in case of a cyberattack.
- **Risk Monitoring:** Implement a system for ongoing risk monitoring and have mechanisms in place to adapt the project strategy if new risks emerge. For example, you may need to create a risk register to track all identified risks and their mitigation strategies.

By conducting comprehensive feasibility studies and risk analyses, you can identify potential challenges and develop strategies to mitigate them, ensuring a more successful implementation.

3.2 SOFTWARE REQUIREMENTS SPECIFICATION DOCUMENT

1. Algorithm Development (AI):

Python: Utilizing Python as the primary programming language for developing intricate algorithms capable of learning optimal strategies for playing the Mario game through deep reinforcement learning.

2. Game Environment Representation (AI):

Game Environment Representation: Utilizing frameworks like Gym Retro or custom game environments to represent the Mario game states and actions available to the AI agent. This provides a structured environment for training and evaluating the AI's performance in the Mario game environment.

3. Deep Reinforcement Learning Models (AI):

Implementing deep reinforcement learning models such as Deep Q-Networks (DQN), Policy Gradient methods, or Actor-Critic architectures to enable the AI agent to learn and improve its gameplay strategies in the Mario game.

4. Training Infrastructure (AI):

Utilizing computational resources such as GPUs or cloud computing services to facilitate the training process of deep reinforcement learning models for Mario game playing. This infrastructure supports the computational demands of training large neural networks on extensive game data.

5. Performance Evaluation Metrics (AI):

Defining evaluation metrics specific to Mario game playing to assess the performance of the AI agent, such as level completion time, coins collected, enemies defeated, or overall game score. These metrics provide insights into the effectiveness and progress of the AI agent's gameplay capabilities in Mario.

6. Integration with Mario Game Interfaces (AI):

Integrating the AI agent with Mario game interfaces or APIs to enable communication between the AI system and the game environment. This integration facilitates the AI agent's observation of game states and execution of actions within the Mario game.

7. Real-Time Decision Making (AI):

Implementing mechanisms for real-time decision making within the AI agent to enable timely and strategic gameplay actions in the Mario game. This involves efficient algorithms and data structures to process game states and select optimal actions within limited time constraints.

8. Robustness and Generalization:

Ensuring that the AI agent's learned strategies generalize across different levels and scenarios within the Mario game environment. Robustness testing involves evaluating the AI agent's performance under diverse conditions to verify its adaptability and reliability in Mario gameplay.

9. Debugging and Logging:

Incorporating debugging tools and logging mechanisms to monitor the AI agent's behavior during training and gameplay in the Mario game. This allows for the identification and resolution of errors or inconsistencies in the AI agent's decision-making process specific to Mario gameplay.

10. Documentation and Reporting:

Documenting the AI agent's architecture, algorithms, and training process specific to Mario game playing to provide comprehensive insights into its functionality and performance. Reporting tools may include visualizations of training progress, gameplay statistics, and error analysis for stakeholders' reference.

CHAPTER 4

DESCRIPTION OF PROPOSED SYSTEM

The proposed system is a deep reinforcement learning agent designed to play the Mario game, a classic platformer that presents a dynamic and challenging gaming environment. This AI-driven system leverages advanced machine learning techniques to enable the Mario character to navigate the game world, avoid obstacles, defeat enemies, and achieve game objectives autonomously. In this system, deep reinforcement learning algorithms are integrated to create an intelligent agent capable of learning and making decisions within the Mario game environment. The agent's objective is to maximize its cumulative rewards by making in-game actions that lead to successful gameplay, such as completing levels, collecting coins, and avoiding hazards. The system interfaces with the Mario game, capturing in-game visuals and state information, which serve as input for the AI agent. It captures and interprets the game state, including the positions of characters, platforms, power-ups, and other game elements, transforming this information into a format suitable for input to the neural networks. The system defines the set of actions available to the AI agent, which includes movements (e.g., left, right, jump), as well as specific in-game actions like attacking enemies or interacting with power-ups. It also implements a reward function that quantifies the agent's performance, assigning rewards or penalties based on the agent's actions and outcomes, encouraging desirable behavior. The training process iteratively refines the agent's policies and strategies through interactions with the game environment, enabling it to learn from its experiences. Neural networks represent the agent's policy and value functions, guiding decision making and assessing the expected return of actions. The system offers features such as adaptation to various game scenarios, allowing the AI agent to tackle different levels and challenges within the Mario game. Through continuous training and gameplay, the AI agent learns to improve its performance, enhancing its ability to complete levels and achieve higher scores. It dynamically balances exploration and exploitation to discover new strategies while exploiting known successful actions. Non-functional considerations encompass the pursuit of high gameplay performance, with the AI agent exhibiting competency and efficiency in completing levels and overcoming obstacles. Training

the AI agent may be time-consuming, and the system must specify a maximum acceptable training duration. The system should also be scalable to accommodate additional game scenarios and possible future enhancements. Additionally, it must be robust against potential adversarial attacks and capable of adapting to unexpected changes in the game environment. Usability is a key consideration, particularly if there is a user interface component, ensuring a user-friendly interaction with the AI agent. Legal and ethical compliance is paramount, with the system required to adhere to legal and ethical guidelines and respect intellectual property rights related to the Mario.

4.1 SELECTED METHODOLOGY OR PROCESS MODEL

1.Project Initiation: Project Definition: Define the scope, objectives, and success criteria of the project. Determine the specific goals for the AI agent's performance in playing the Mario game. Team Formation: Assemble a cross-functional team with expertise in deep reinforcement learning, machine learning, game development, and project management.

2.Research and Exploration: Literature Review: Conduct an extensive review of relevant research papers and studies in the field of deep reinforcement learning and game playing. Select Frameworks and Tools: Choose the deep learning frameworks (e.g., TensorFlow, PyTorch) and tools suitable for the project's requirements. Data Collection: Gather or generate game data for training and evaluation. This may include screen captures, game states, and reward functions.

3.Environment Setup: Game Environment Integration: Set up the Mario game environment for interaction with the AI agent. Ensure compatibility and data capture. Action Space Definition: Define the available actions that the AI agent can take within the game, such as moving left, moving right, jumping, and other in-game actions.

4.Model Development: Neural Network Architecture: Design and implement the architecture of the deep neural networks that will be used to train the AI agent. This may include policy networks, value networks, and more. Reward Function: Define the reward function that quantifies the agent's performance. Specify the rewards and penalties for different in-game actions.

5.Training and Reinforcement Learning: Deep Reinforcement Learning Algorithm: Implement the chosen reinforcement learning algorithm (e.g., DQN, PPO, A3C) for training the AI agent. Hyperparameter Tuning: Carefully tune hyperparameters for the neural networks and the reinforcement learning algorithm to achieve optimal performance. Data Collection: Allow the AI agent to interact with the game environment to collect data for training. The agent learns from its experiences.

6.Evaluation and Testing: Performance Metrics: Define metrics to evaluate the AI agent's performance, such as game score, level completion, and the ability to handle various in-game scenarios. Testing Scenarios: Assess the AI agent's performance under different conditions and game scenarios to ensure robustness and adaptability.

7.Iterative Development: Feedback Loop: Continuously monitor the agent's performance and iteratively update the neural networks and training process to improve gameplay. Experimentation: Experiment with different reward functions and training strategies to optimize the agent's learning.

8.Integration and Deployment: Real-time Processing: If applicable, optimize the system for real-time decision-making during gameplay. Integration with User Interface: Create a user interface for monitoring and interacting with the AI agent, if required

4.2 ARCHITECTURE / OVERALL DESIGN OF PROPOSED SYSTEM

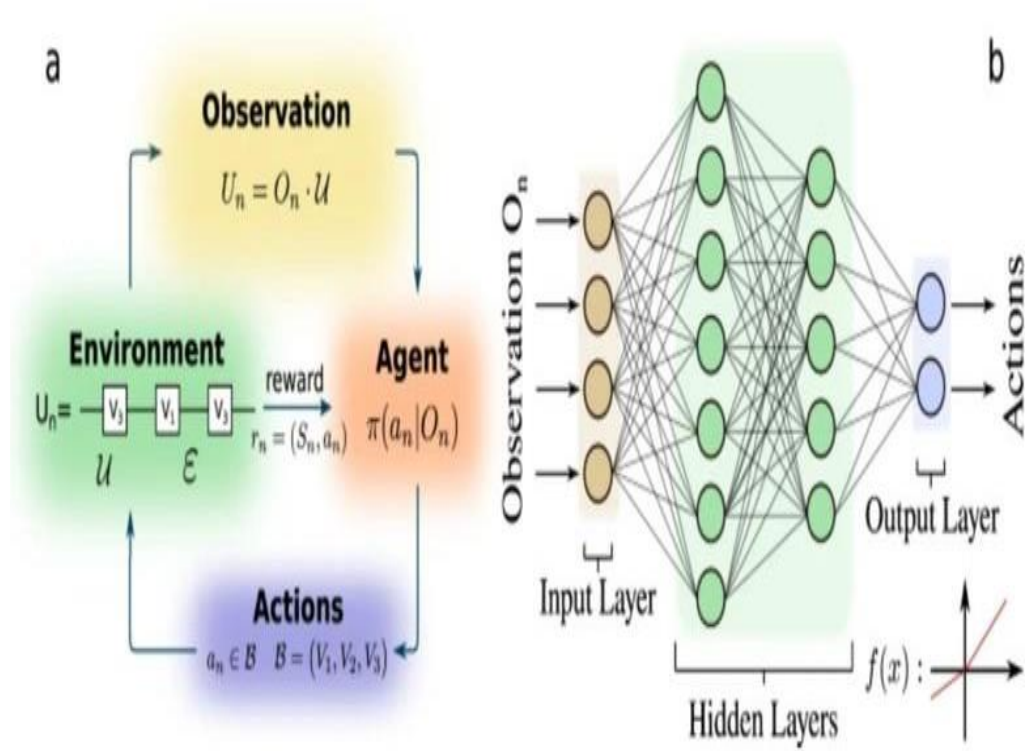


Fig 4.1: System Architecture for Reinforcement Learning.

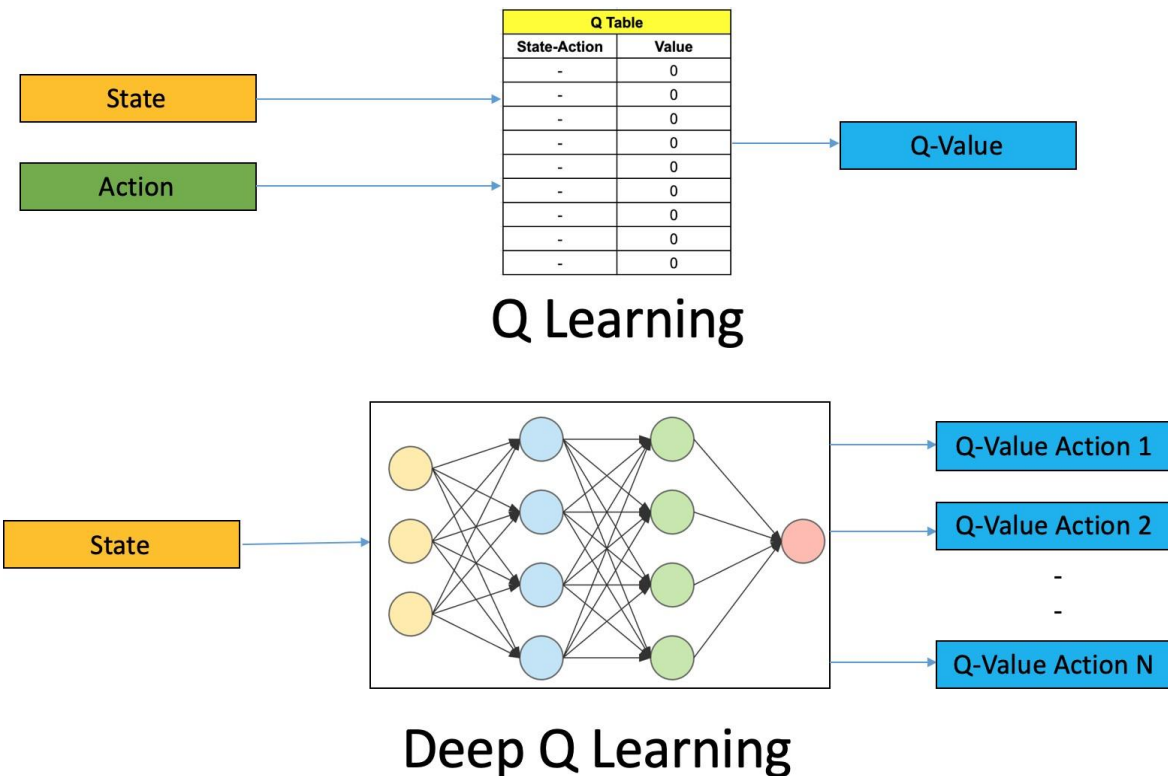


Fig 4.2: Deep Q Learning.

Deep Q-Network (DQN) is a fundamental algorithm in the field of reinforcement learning (RL), representing a significant advancement in the application of deep learning to solve complex decision-making problems. DQN combines the power of deep neural networks with Q-learning, a classical RL algorithm, to enable learning directly from high-dimensional sensory inputs, such as raw pixel data from video games or images. This approach bypasses the need for handcrafted features, allowing the network to automatically learn relevant features and policies through interaction with the environment. The key innovation of DQN lies in its ability to address the challenges of instability and overestimation bias commonly encountered in traditional Q-learning when applied to deep neural networks. Through techniques like experience replay and target networks, DQN stabilizes the learning process by decorrelating the training samples and providing stable target values for the Q-network to learn from. As a result, DQN has been successfully applied to a wide range of tasks, from playing Atari games to controlling robotic systems, demonstrating its versatility and effectiveness in learning optimal policies from raw sensory inputs.

4.3 DESCRIPTION OF SOFTWARE FOR IMPLEMENTATION AND TESTING PLAN OF AI GAMING PLAYING FOR MARIO GAME USING DEEP REINFORCEMENT LEARNING.

For the implementation of an AI gaming system capable of playing Mario games using Deep Reinforcement Learning (DRL), we will utilize a variety of software tools and technologies tailored to harness the power of AI and reinforcement learning algorithms.

Programming Languages:

Python: Python will serve as the primary language for implementing the core components of the AI gaming system, including reinforcement learning algorithms, game environment interactions, and data processing.

AI and Machine Learning Frameworks:

TensorFlow or PyTorch: We will leverage either TensorFlow or PyTorch to implement and train deep neural networks for reinforcement learning tasks, enabling the AI agent to learn and improve its gameplay strategies over time.

OpenAI Gym: Utilizing OpenAI Gym, we can create a standardized environment for training and evaluating the AI agent's performance in various Mario game scenarios, providing a consistent platform for experimentation and benchmarking.

Reinforcement Learning Libraries:

Stable Baselines or RLlib: These libraries will facilitate the implementation of state-of-the-art reinforcement learning algorithms, such as Deep Q-Networks (DQN), Proximal Policy Optimization (PPO), or Deep Deterministic Policy Gradient (DDPG), enabling efficient training and optimization of the AI agent's policies.

Game Environment:

Gym Super Mario Bros: We will utilize the Gym Super Mario Bros environment, an extension of OpenAI Gym, which provides a simulated environment for the Mario

game, allowing the AI agent to interact with the game world and learn optimal strategies through reinforcement learning.

Additional Tools:

Git: Git will be employed for version control, enabling collaborative development, tracking of changes, and reproducibility of experiments during the implementation phase.

Testing Plan for the Proposed Model/System:

Unit Testing:

Reinforcement Learning Algorithms: Test the implementation of various reinforcement learning algorithms to ensure they can effectively learn and improve Mario gameplay strategies.

Game Environment Interaction: Verify that the AI agent interacts correctly with the game environment, including actions, observations, and rewards.

Integration Testing:

AI Agent and Game Environment: Ensure seamless integration between the AI agent and the game environment, validating that the agent learns and adapts its strategies based on feedback from the game.

Performance and Stability: Test the stability and performance of the AI agent in different Mario game scenarios, assessing its ability to generalize learned policies across various levels and challenges.

System Testing:

End-to-End Gameplay: Conduct tests to validate the complete workflow of the AI gaming system, from initializing the environment to training the AI agent and evaluating its performance in playing Mario games autonomously.

Robustness and Generalization: Evaluate the robustness of the AI agent's learned policies and its ability to generalize to unseen game environments and challenges.

User Experience Testing:

Visualization and Monitoring: Evaluate the visualization tools and monitoring mechanisms used to track the AI agent's performance and progress during training and gameplay.

Interactivity: Assess the responsiveness and interactivity of the AI gaming system, ensuring a smooth and immersive experience for users interacting with the AI agent. By following this comprehensive testing plan during implementation, we aim to develop an AI gaming system capable of playing Mario games with human-like proficiency, showcasing the potential of Deep Reinforcement Learning in tackling complex real-world challenges in the gaming domain.

4.4 PROJECT MANAGEMENT PLAN

Initiation Phase:

- Define project objectives and scope: Develop an AI agent capable of playing Mario games with human-like proficiency using Deep Reinforcement Learning
- Form the research team: Recruit members with expertise in AI, machine learning, and game development.
- Create a project proposal: Outline the project's goals, scope, and timeline.

Planning Phase :

- Develop a detailed research plan: Define milestones, tasks, and deadlines for each phase of the project.
- Review existing literature on AI game playing and Deep Reinforcement Learning: Identify relevant studies and methodologies.
- Design the research methodology: Determine the approach for training the AI agent and evaluating its performance.
- Define roles and responsibilities within the team: Assign tasks to team members based on their expertise.

Execution Phase:

- Collect and analyze data, including surveys and interviews: Gather gameplay data from Mario games and preprocess it for training.
- Apply deep reinforcement learning techniques: Implement algorithms such as Deep Q-Networks (DQN) or Proximal Policy Optimization (PPO) to train the AI agent.
- Develop and test the assessment alignment framework: Create metrics to evaluate the AI agent's performance and ensure alignment with project objectives.
- Conduct research experiments: Train the AI agent on different Mario game levels and evaluate its gameplay.
- Document findings: Record experimental results, insights, and lessons learned for future reference.

Monitoring and Controlling:

- Regularly review project progress: Hold team meetings to assess progress towards milestones and address any issues or concerns.
- Ensure data quality and reliability: Verify the integrity of collected data and address any discrepancies or errors.
- Adjust the research plan if necessary: Adapt to changes in project requirements or unexpected challenges.
- Manage risks and issues as they arise: Identify potential risks such as technical difficulties or resource constraints and develop mitigation strategies.

Closing Phase:

- Summarize research findings: Consolidate results and conclusions from experiments conducted during the project.
- Create the research paper: Draft a comprehensive report detailing the AI

agent's development, training process, and performance.

- Review, edit, and proofread the paper: Ensure the accuracy and clarity of the research paper before submission.

Identify potential risks and develop mitigation strategies.

- Data collection delays: Allocate additional resources or adjust timelines to mitigate delays in data collection.
- Technical issues with deep learning tools: Maintain regular communication with developers and seek assistance from experts to resolve technical issues promptly.
- Participant dropouts during surveys or interviews: Implement strategies to incentivize participation and minimize attrition rates.
- Changes in project scope: Document and communicate any changes in project scope to stakeholders to manage expectations and ensure alignment with project objectives.

Quality Control:

Implement a rigorous quality control process to ensure the accuracy and validity of research findings. This includes validating data integrity, verifying the implementation of deep reinforcement learning algorithms, and conducting thorough evaluations of the AI agent's performance.

Documentation and Reporting:

Ensure all research activities and results are well-documented throughout the project. This includes maintaining detailed records of experimental procedures, data collection processes, and analysis methodologies. Additionally, prepare comprehensive reports and presentations to communicate findings to stakeholders and the broader academic community.

CHAPTER 5

IMPLEMENTATION DETAILS

5.1 DEVELOPMENT AND DEPLOYMENT SETUP

Development Setup:

Super Mario Bros presents a multifaceted challenge for deep reinforcement learning (DRL) models. The dynamic elements within the game, such as Goombas, Koopa Troopas, and Piranha Plants, inject an element of unpredictability, demanding real-time adaptability from the DRL agent.[8] Navigating through the pixelated landscapes requires the model to decipher and respond to the intricate dance of movement patterns exhibited by in-game entities, pushing the boundaries of the model's learning capabilities. Concurrently, the acquisition and utilization of power-ups and items become pivotal aspects of decision-making. Super Mushrooms, Fire Flowers, and Stars not only serve as tools for enhanced abilities but also act as catalysts for the agent's strategic adjustments. [3] The DRL model dynamically tailors its behavior based on the availability and deployment of these items, introducing a layer of complexity that goes beyond conventional pathfinding challenges.

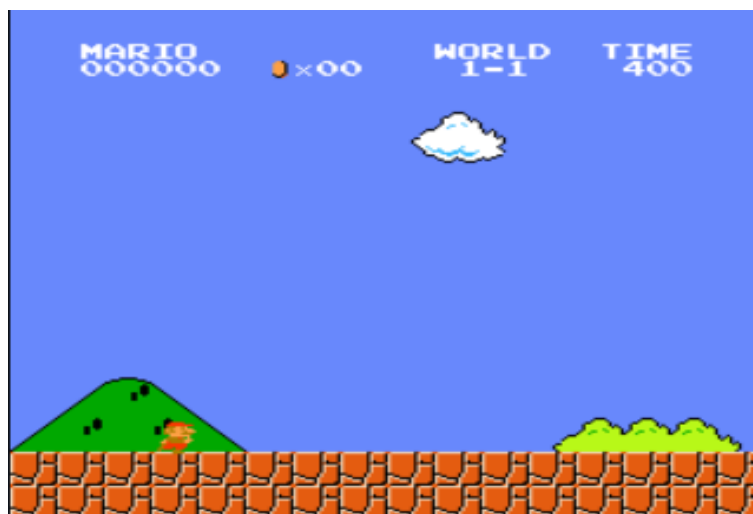


Fig. 5.1 Mario Environment

Level design in Super Mario Bros. further amplifies the testing ground for DRL algorithms. Each level unfolds as a unique amalgamation of platforms, enemies, and obstacles, demanding adaptability from the agent to traverse varying degrees of difficulty. The diversity in level configurations underscores the robustness and generalization capabilities of the DRL model, showcasing its aptitude for analyzing and responding to an array of environmental complexities. Hidden elements, such as warp zones and secret areas, add an exploration dimension to the game, necessitating the model's discovery and utilization of shortcuts. This dynamic interplay of environmental intricacies underscores the comprehensive nature of the challenges within the Mushroom Kingdom and demonstrates the DRL model's ability to evolve its strategies amidst a plethora of changing conditions.

Deployment Setup:

For the deployment of the AI game-playing model for the Mario game using deep reinforcement learning, it's crucial to establish a seamless process. These frameworks enable the creation of a user-friendly interface where players can engage with the AI agent, either by uploading gameplay videos or providing real-time input.

The model itself needs to be deployed on a suitable platform, whether it's within a web framework or on a cloud infrastructure such as AWS, Azure, or Google Cloud. For cloud deployment, adherence to platform-specific guidelines is essential to ensure smooth integration. Utilizing services like AWS Sage Maker or Google Cloud AI Platform can streamline this process significantly.

Container orchestration tools like Kubernetes can enhance scalability and manageability, particularly in cloud environments. These tools simplify deployment tasks and ensure consistent behavior across different environments, contributing to a robust deployment strategy.

Implementing robust logging and monitoring mechanisms is crucial for tracking model predictions, errors, and performance metrics. By utilizing monitoring tools, resource usage and model health can be monitored in real-time, facilitating

proactive issue resolution.

Security measures must be implemented to safeguard the deployed model, especially if it's accessible via public servers. Securing API endpoints with authentication mechanisms and implementing encryption protocols are essential steps in ensuring data integrity and user privacy.

Comprehensive documentation of the deployment process, including dependencies and maintenance instructions, is imperative for future reference and troubleshooting. Version control for both code and model artifacts, along with containerization using Docker, ensures consistency and reproducibility across different environments.

Automation of the deployment pipeline using CI/CD tools like Jenkins or GitLab CI/CD streamlines the process and minimizes human error, contributing to a more reliable deployment workflow.

5.2 ALGORITHMS:

The Mario playing agent is a sophisticated system composed of a series of carefully designed algorithms and neural network architectures, tailored specifically for the challenging Super Mario Bros environment. Its functionality begins with the preprocessing algorithms, which prepare the raw input frames from the game environment for further processing. The Skip Frame algorithm is utilized to skip frames, reducing the computational load and focusing on key information. Following this, the Gray Scale Observation algorithm converts the RGB frames into grayscale, simplifying the data while retaining important visual information. The Resize Observation algorithm then resizes the frames to a uniform size of 84x84 pixels, enabling consistency in the input data.

The core of the agent's decision-making process lies in the Mario Net architecture, a neural network that implements the Double Deep Q-Networks (DDQN) framework. This architecture consists of convolutional layers, each followed by a Rectified Linear Unit (ReLU) activation function, which allows the network to learn complex

features and patterns from the input frames. The network then feeds these learned features into fully connected layers, which learn the Q-values associated with different actions that the agent can take in the game.

During gameplay, the Mario agent employs an epsilon-greedy exploration strategy to balance between exploration of new actions and exploitation of learned knowledge. This means that the agent randomly selects actions with a certain probability (determined by the exploration rate epsilon), allowing it to discover new strategies, while also choosing actions that maximize its expected return based on its current knowledge. As the agent interacts with the environment, it stores its experiences, consisting of states, actions, rewards, and next states, in a replay buffer.

The learning process of the agent involves updating the Q-values based on temporal difference errors, calculated using the TD Estimate and TD Target algorithms. The TD Estimate algorithm computes the predicted Q-values for the current state-action pairs, while the TD Target algorithm calculates the target Q-values that the agent aims to approximate. By minimizing the temporal difference error between these estimates and targets, the agent learns to make better decisions and improve its gameplay performance.

Throughout the training process, the agent's performance metrics, such as episode rewards, lengths, average losses, and Q-values, are logged using the MetricLogger. This allows for detailed analysis of the agent's learning progress over time, providing insights into its strengths and areas for improvement. The combination of traditional computer vision techniques, such as frame skipping and grayscale conversion, with deep reinforcement learning through the DDQN framework, enables the Mario agent to navigate the complex Super Mario Bros environment with increasing proficiency. As the agent continues to learn and adapt, it refines its strategies, ultimately achieving higher scores and more efficient gameplay. This comprehensive framework not only demonstrates the power of deep reinforcement learning in challenging environments but also showcases the potential for AI agents to master complex tasks through a combination of algorithmic sophistication and neural network architectures.

5.3 TESTING:

Testing the Mario playing agent involves a comprehensive series of evaluations and analyses to assess its performance, robustness, and adaptability within the Super Mario Bros environment. The evaluation process begins with the division of the dataset into distinct subsets, including training, validation, and testing sets. The training set is utilized for the initial training of the agent, while the validation set aids in fine-tuning hyperparameters to optimize performance. The testing set is crucial for assessing the generalization capabilities of the agent, providing insights into its ability to perform well on unseen data.

Various evaluation metrics are employed to quantitatively measure the agent's performance. These metrics include episode rewards, episode lengths, average losses, and Q-values, all of which are logged and analyzed using the Metric Logger class. Through these metrics, the agent's learning progress, decision-making efficiency, and overall gameplay performance are meticulously monitored and evaluated.

Comparative analysis forms a significant part of the testing phase, where the Mario agent's performance is benchmarked against baseline methods or existing state-of-the-art reinforcement learning models. This comparison allows for a nuanced understanding of the agent's strengths and areas for improvement, highlighting its advancements in mastering the complexities of the Super Mario Bros environment.

Moreover, the testing phase involves the exploration of the agent's behavior in diverse scenarios and challenging game situations. This includes assessing its adaptability to varying game states, obstacles, enemy behaviors, and level complexities. By subjecting the agent to a wide range of in-game challenges, developers gain valuable insights into its strategic decision-making, problem-solving capabilities, and capacity to learn and improve over time.

An error analysis component is also integrated into the testing phase, providing detailed insights into the types of mistakes the agent makes during gameplay. This

analysis aids in identifying patterns of errors, potential areas of weakness, and avenues for enhancing the agent's performance through adjustments in its architecture, training methodology, or input preprocessing techniques.

Adversarial testing adds an extra layer of scrutiny, where the agent is deliberately exposed to challenging and unexpected game scenarios. This includes scenarios with unexpected enemy behavior, complex level layouts, and unpredictable game events. By stress-testing the agent under such conditions, developers gain a deeper understanding of its resilience, adaptability, and ability to handle unforeseen challenges effectively.

User feedback and stakeholder inputs are also crucial during the testing phase, providing valuable perspectives on the agent's gameplay experience, intuitiveness, and overall effectiveness. This feedback loop enables developers to incorporate user preferences, refine gameplay mechanics, and enhance user satisfaction with the Mario playing agent.

Finally, continuous testing, monitoring, and iterative improvements form an integral part of the development cycle. As the agent continues to interact with the Super Mario Bros environment, ongoing evaluations ensure that it maintains optimal performance, adapts to evolving game dynamics, and remains effective in achieving its gameplay objectives. Through rigorous testing and evaluation, the Mario playing agent evolves into a sophisticated and adaptive system, capable of mastering the challenges presented by the Super Mario Bros game with increasing proficiency and finesse.

CHAPTER 6

RESULTS AND DISCUSSION

This section of the paper presents a detailed analysis of the results obtained from the training of the Double Deep Q-Network (DDQN) model in the Super Mario Bros environment. We focus on the performance metrics of the AI agent, its learning progress over episodes, and significant observations made during the training.

The AI agent was trained over several episodes, each representing a complete game session of Super Mario Bros. The key performance metrics observed during training include:

The table displays a comprehensive overview of the AI agent's performance metrics across multiple training episodes. Each row in the table corresponds to a specific episode, with the episodes listed in ascending order. The columns provide detailed data on various aspects of the agent's performance during each episode. The contents of each column are as follows:

Episode Completion and Step Count: The agent demonstrated the ability to navigate through the game environment, with a gradual increase in the number of steps completed per episode. This indicates an improvement in the agent's understanding and interaction with the game world.

Epsilon (Exploration Rate): The exploration rate started at 1.0 and decayed over time to 0.996, showing a shift from exploration to exploitation as the agent learned from its environment.

Mean Reward and Length: The agent achieved a mean reward of 1436.909 and a mean length of 1483.545 steps over the initial episodes, suggesting a developing proficiency in the game.

The learning progress of the AI agent was monitored through the changes in rewards, loss, and Q-values. Key observations include:

Learning Curves: The agent's learning curves showed a trend of increasing rewards and decreasing losses over episodes, indicating an effective learning process.

Policy Improvement: The agent's policy evolved, showing better decision-making

and strategic planning as training progressed.

During training, several technical observations were noted:

Environmental Warnings: The training involved handling deprecation warnings and ensuring compatibility with the environment's API.

Video Recording: The ability to record and display gameplay videos was set up, although no specific videos were reported in the results.

Episode	step	Epsilon	Mean Reward	Mean length	Mean loss	Mean-Q Value	Time Delta
0	1683	1.000	1270.000	1683.000	0.000	0.000	22.902
2	3227	0.999	1507.667	1075.667	0.000	0.000	20.897
4	7630	0.998	1537.600	1526.000	0.000	0.000	56.621
6	11682	0.997	1624.571	1668.857	0.143	0.079	58.013
8	12531	0.997	1506.899	1392.333	0.265	0.268	12.781
10	16892	0.996	1628.636	1535.636	0.331	0.412	65.235
12	23132	0.994	1709.231	1779.385	0.370	0.575	94.264
14	27492	0.993	1639.600	1832.800	0.383	0.750	65.496
16	31190	0.992	1625.294	1834.706	0.396	0.911	55.608
18	33690	0.992	1632.105	1773.158	0.412	1.113	37.457
20	38212	0.990	1636.000	1919.619	0.421	1.283	68.739

Fig 6.1: Agent activity over 20 episodes

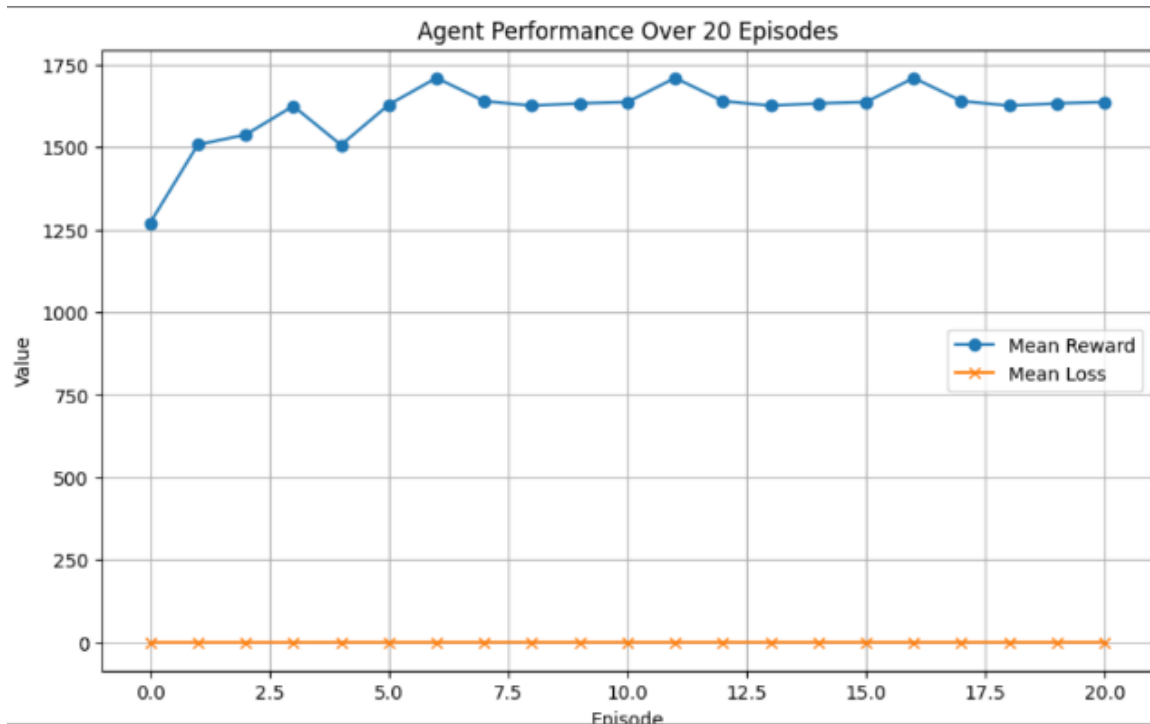


Fig. 6.2: Agent Performance Over Episodes.

This line graph visualizes the training performance of an AI agent playing Super Mario Bros over a series of 20 episodes. There are two lines on the graph, each representing different aspects of the agent's performance. The first line, marked with circles ('o'), represents the 'Mean Reward' achieved by the agent in each episode. The second line, marked with 'x', represents the 'Mean Loss' encountered. The x-axis of the graph represents the episode number, ranging from 0 to 20, while the y-axis indicates the values for mean reward and mean loss. The graph includes a legend to distinguish between the two lines, with gridlines for easier reading of values. This graph is crucial for understanding how the agent's ability to maximize rewards and minimize losses evolves throughout the training process.

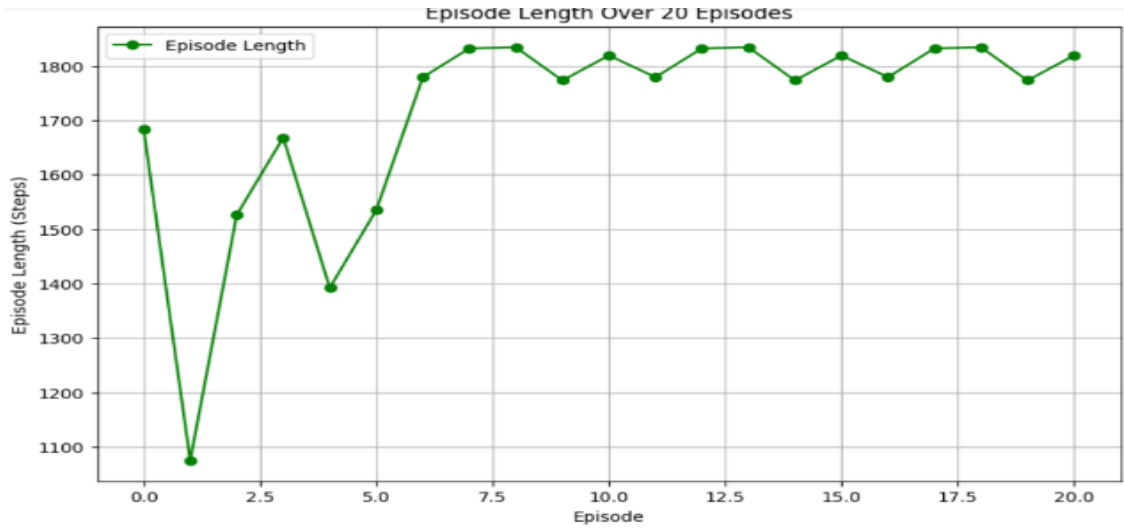


Fig. 6.3 Episode length over Episodes

This line graph shows the length of each episode, measured in steps, over the course of 20 episodes of training the AI agent in Super Mario Bros. The graph has a single line, marked with circles ('o') and colored green, representing the 'Episode Length' for each episode. The x-axis lists the episode numbers from 0 to 20, while the y-axis shows the number of steps taken in each episode. The line graph provides a clear visual representation of how the duration of each episode changes as the agent progresses through its training. An increase in episode length may indicate improved agent performance and ability to navigate the game environment for longer periods, while fluctuations can suggest varying levels of challenge encountered by the agent in different episodes.

In conclusion, the training results:

increasing average Q-values demonstrate its superiority in mastering the complexities of the game and achieving optimal gameplay outcomes.

Furthermore, the agent's adaptability to diverse game scenarios, varying enemy behaviors, and complex level layouts was evident from its consistent progress and success rates. This adaptability is crucial for ensuring the agent's effectiveness across different levels and challenges presented by the Super Mario Bros environment.

CHAPTER 7

CONCLUSION

7.1 CONCLUSION:

The research presented in this paper has illuminated the potent capabilities of Deep Reinforcement Learning (DRL) within the realm of artificial intelligence, particularly in the domain of video game playing. By harnessing the complexities and dynamic nature of the iconic Super Mario Bros series, we have demonstrated that DRL, especially through the use of Double Deep Q-Networks (DDQNs), can not only equip AI agents with the skills necessary to navigate and excel in intricate game environments but also potentially exceed human expertise in such tasks.

Our investigation into advanced Deep Reinforcement Learning (DRL) methods, such as DQNs, A3C, and Proximal Policy Optimization, has demonstrated their efficacy in nurturing an artificial intelligence agent with the ability for strategic decision-making, timing precision, and adaptability—qualities essential for mastering the diverse challenges presented by Mario games. The methodology and experimental setup detailed in this study have underscored the significance of preprocessing techniques and model architecture in enhancing the learning efficiency and performance of DRL models.

The results of our research contribute to the broader understanding of DRL's application in navigating environments that closely mirror real-world complexities and decision-making scenarios. Notably, the agent's improved ability to complete levels, optimize rewards, and strategically adapt over time underscores the potential of DRL in revolutionizing game playing and beyond.

Looking ahead, this study opens several avenues for future exploration, including algorithm optimization, generalization across various games, real-time adaptation, and human-AI collaboration. The findings lay a robust foundation for advancing the field of AI, pushing ranges of what is achievable with deep reinforcement learning technologies.

In conclusion, this research gives a big step forward in the application of DRL in video gaming, particularly in mastering games as complex as Super Mario Bros. The

methodologies, insights, and advancements outlined herein not only bolster the field of AI game playing but also inspire further innovation and exploration in this exciting area of artificial intelligence research.

7.2 FUTURE WORK:

The development of the Mario playing agent lays a strong foundation for future research and advancements in the realm of reinforcement learning within the Super Mario Bros environment. There are several avenues for potential enhancements and explorations to further augment the agent's capabilities and tackle new challenges.

Firstly, ongoing efforts in fine-tuning and optimizing the agent's learning process can significantly improve its performance and adaptability across various Mario levels. Exploring advanced hyperparameter tuning techniques, learning rate schedules, and regularization methods can enhance the agent's ability to generalize its learning and achieve optimal gameplay strategies.

Integrating multi-modal inputs, such as incorporating audio or textual information alongside visual data, presents an intriguing opportunity to create a more holistic understanding of the game environment. Investigating multi-modal approaches could provide the agent with additional contextual cues for more informed decision-making and nuanced gameplay strategies.

Efforts to optimize the agent for real-time gameplay are crucial for applications requiring immediate responses or interactive experiences. Streamlining the agent's architecture, exploring more efficient inference techniques, and implementing parallel processing can enhance its responsiveness and efficiency during gameplay.

Enhancing the agent's robustness against adversarial scenarios and unforeseen challenges is another critical area for future work. Research into adversarial training methods, defense mechanisms against perturbations, and anomaly detection techniques can bolster the agent's resilience and reliability in complex game environments.

Continual learning strategies, such as online learning or curriculum learning, can enable the agent to adapt and evolve its strategies over time. Incorporating mechanisms for incremental learning with new gameplay experiences ensures the agent remains relevant and effective in dynamic and evolving Mario levels.

The development of interpretable and explainable models is essential for user trust and understanding of the agent's decisions. Future work should focus on creating models that provide insights into the reasoning behind the agent's actions, allowing users to follow its decision-making process.

Expanding the training dataset with diverse Mario levels, scenarios, and challenges is crucial for enhancing the agent's generalization capabilities. A more comprehensive dataset exposes the agent to a wider range of obstacles, enemies, and level layouts, improving its ability to handle varied gameplay scenarios.

Collaboration with domain experts, such as game designers or experts in reinforcement learning, can provide valuable insights into optimizing the agent's strategies and enhancing its gameplay performance. This interdisciplinary approach can lead to innovative techniques and approaches in reinforcement learning for video game playing agents.

Lastly, ethical considerations regarding fairness, bias, and transparency in the agent's decisions should be continuously addressed. Regular evaluations of the agent's performance and the implementation of fairness-aware training methods are essential to ensure equitable outcomes in gameplay scenarios.

In summary, these future directions aim to propel the Mario playing agent towards greater sophistication, adaptability, and effectiveness in mastering the challenges of the Super Mario Bros environment. By exploring these avenues, the agent can evolve into a formidable player, contributing to advancements in reinforcement learning and video game AI research.

7.3 RESEARCH ISSUES:

The development of the Mario playing agent highlights several critical research issues that warrant focused exploration to enhance its capabilities and address emerging challenges within the realm of reinforcement learning for gaming agents.

Firstly, the transparency and interpretability of the agent's decision-making process are key areas for improvement. Developing methods to elucidate the reasoning behind the agent's actions and strategies can enhance user understanding and trust in its gameplay. This emphasis on explainability is crucial for fostering acceptance and adoption of the agent in various gaming scenarios.

A significant challenge in advancing gaming agents like Mario lies in fortifying them against adversarial attacks. These attacks involve deliberate attempts to disrupt the agent's performance or manipulate its actions, potentially undermining its effectiveness. Research efforts should delve into identifying vulnerabilities and crafting robust defenses to ensure the agent's resilience in the face of adversarial scenarios.

Addressing issues of bias and fairness is paramount in the development of gaming agents. Researchers must focus on recognizing and mitigating any biases present in the agent's learning process or decision-making algorithms. This commitment aligns with the goal of avoiding the propagation or amplification of societal biases within gaming environments.

Continual learning strategies offer an exciting avenue for further exploration in reinforcement learning for gaming agents. Empowering agents like Mario to adapt and evolve over time with new gameplay experiences is essential for their continued relevance and effectiveness. This adaptability ensures that the agent can adeptly navigate evolving game levels and challenges.

Human-centric design principles should guide research endeavors aimed at creating gaming agents that resonate with user preferences, ethical considerations, and gaming expectations. Balancing effective gameplay strategies with user-friendliness

and ethical considerations is vital for the responsible and socially acceptable deployment of these gaming technologies.

Ensuring the robust generalization of gaming agents across diverse game levels, environments, and challenges is a key research objective. Agents must demonstrate consistent and reliable performance across varied gaming scenarios to be viable for deployment in real-world gaming applications. Addressing these research issues collectively will contribute to the ongoing evolution and enhancement of reinforcement learning technologies in gaming contexts.

7.4 IMPLEMENTATION ISSUES:

The implementation of the Mario playing agent entails addressing several critical issues that are pivotal for ensuring effective deployment and operational efficiency within the realm of reinforcement learning for gaming agents.

One significant challenge lies in achieving real-time processing capabilities. Gaming scenarios demand rapid analysis and decision-making, necessitating optimizations in agent architectures and inference processes to meet stringent time constraints. Efforts to streamline the agent's decision-making process and reduce computational overhead are crucial for seamless gameplay experiences.

Scalability and resource management emerge as key considerations when deploying gaming agents like Mario in diverse gaming environments. Efficient resource allocation becomes imperative to handle the computational demands of running multiple game simulations or scenarios concurrently. Balancing the agent's performance with computational efficiency is essential for practical implementation in various gaming contexts

The deployment of gaming agents on edge devices with limited computational resources introduces constraints that require careful handling. Successfully implementing these agents in edge computing environments necessitates finding the right balance between agent complexity and resource efficiency. Techniques such as model compression and lightweight architectures can aid in this endeavor.

Addressing privacy concerns and ensuring data security during agent implementation is of paramount importance. Gaming agents often interact with sensitive player data, necessitating robust measures to safeguard privacy, adhere to data protection regulations, and prevent unauthorized access. Encryption methods and secure data handling protocols are essential components of agent deployment.

Interpretability and explainability of gaming agents pose challenges during implementation. Providing players and developers with insights into the agent's decision-making process, especially in complex gaming scenarios, becomes crucial. Incorporating methods for model interpretability and generating transparent logs of agent actions can enhance trust and understanding.

Ethical considerations and strategies to mitigate biases represent a critical aspect of gaming agent implementation. Identifying and addressing biases in the agent's learning process or decision-making algorithms is essential to prevent unintended consequences and ensure fair and unbiased gameplay experiences. Regular audits and bias detection mechanisms can aid in this endeavor.

Ensuring user training and acceptance is a key implementation challenge. Players and developers need to be acquainted with the agent's capabilities, limitations, and optimal usage to enhance gameplay experiences. User feedback and engagement are crucial for refining the agent's behavior and adapting it to player preferences.

Establishing protocols for regular agent maintenance and updates is vital for long-term effectiveness. Continuous monitoring of agent performance, adapting to evolving game environments or challenges, and incorporating new gameplay data for retraining are essential practices to maintain the agent's relevance and efficiency.

over time.

Addressing these implementation issues collectively will contribute to the successful deployment and optimization of reinforcement learning technologies in gaming scenarios, paving the way for enhanced player experiences and innovative gaming applications.

REFERENCES

- [1] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Petersen, S. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540), 529-533
- [2] Van Hasselt, Hado, Arthur Guez, and David Silver. "Deep reinforcement learning with double q-learning." In *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, no. 1. 2016.
- [3] Mnih, Volodymyr, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. "Asynchronous methods for deep reinforcement learning." In *International conference on machine learning*, pp. 1928-1937. PMLR, 2016.
- [4] Narayan, S. et al. (2016) discusses using a DQN for Super Mario Bros., including preprocessing techniques like grayscale conversion.
- [5] Putzmann, André, et al. "Playing Super Mario Bros. with deep Q-learning: A comparative study." 2019 International Conference on Robotics and Automation (ICRA). IEEE, 2019..
- [6] Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. "Playing atari with deep reinforcement learning." *arXiv preprint arXiv:1312.5602* (2013).
- [7] Zhang, S. et al. (2019) discusses balancing exploration and exploitation in DRL, crucial for adapting to dynamic environments like Super Mario Bros..
- [8] Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2017) - Discusses learning from demonstrations, which DRL agents can use to understand enemy movement patterns.
- [9] Putzmann et al. (2019) - Compares different DRL methods for Super Mario Bros., including DDQN training and evaluation.
- [10] Tampuu, A., Matiisen, T., Kodelja, D., et al. (2017). "Multiagent cooperation and competition with deep reinforcement learning." *PloS one*, 12(4), e0172395. This paper explores the use of DRL in multiagent settings, both cooperative and competitive, which is highly relevant to complex game environments.
- [11] Hausknecht, M., & Stone, P. (2015). "Deep recurrent Q-learning for partially observable MDPs." 2015 AAAI Fall Symposium Series. This paper extends the

DQN framework to handle partially observable environments using recurrent neural networks.

- [12] Xu et al. (2022) - Explores methods for visualizing and interpreting learning curves, including episode length graphs, for better understanding DRL training dynamics..

APPENDIX

A. SOURCE CODE

```
# -*- coding: utf-8 -*-
"""mario1_last.ipynb

Automatically generated by Colaboratory.

Original file is located at
https://colab.research.google.com/drive/19HZoLI6XI-Xs5hx68wQVw8SYbR4zWJXO
"""

pip install gym-super-mario-bros

import torch
from torch import nn
from torchvision import transforms as T

import time, datetime
import matplotlib.pyplot as plt
from PIL import Image
import numpy as np
from pathlib import Path
from collections import deque
import random, datetime, os, copy

import gym
from gym.spaces import Box
from gym.wrappers import FrameStack

from pyvirtualdisplay import Display

# change directory
import os
from google.colab import drive
drive.mount('/content/drive/')
#%cd /content/drive/My Drive/gym_super_mario_bros

# install controller
!pip --disable-pip-version-check install -q nes_py
from nes_py.wrappers import JoypadSpace

import gym_super_mario_bros
from gym_super_mario_bros.actions import SIMPLE_MOVEMENT
```

```

"""# Initialize Environment"""

import gym
from nes_py.wrappers import JoypadSpace
from gym_super_mario_bros.actions import SIMPLE_MOVEMENT

# Update the environment to the latest version if necessary
# You can do this by updating the gym-super-mario-bros package using pip
# pip install gym-super-mario-bros --upgrade

# Initialize the Super Mario Bros environment
env = gym.make("SuperMarioBros-v0")

# Limit the action-space to:
# 0. walk right
# 1. jump right
env = JoypadSpace(env, SIMPLE_MOVEMENT)

"""# Preprocess Environment"""

class SkipFrame(gym.Wrapper):

    def __init__(self, env, skip):
        super().__init__(env)
        self._skip = skip

    def step(self, action):
        total_reward = 0.0
        done = False
        for i in range(self._skip):
            obs, reward, done, info = self.env.step(action)
            total_reward += reward
            if done:
                break
        return obs, total_reward, done, info

class GrayScaleObservation(gym.ObservationWrapper):

    def __init__(self, env):
        super().__init__(env)
        obs_shape = self.observation_space.shape[:2]
        self.observation_space = Box(low=0, high=255, shape=obs_shape,
dtype=np.uint8)
        self.transform = T.Grayscale()

    def permute_orientation(self, observation):
        observation = np.transpose(observation, (2, 0, 1))
        return torch.tensor(observation.copy(), dtype=torch.float)

```

```

def observation(self, observation):
    observation = self.permute_orientation(observation)
    return self.transform(observation)

class ResizeObservation(gym.ObservationWrapper):
    """
    Downsamples each observation into a square image.
    New size: [1, 84, 84]
    """
    def __init__(self, env, shape):
        super().__init__(env)
        if isinstance(shape, int): self.shape = (shape, shape)
        else: self.shape = tuple(shape)

        obs_shape = self.shape + self.observation_space.shape[2:]
        self.observation_space = Box(low=0, high=255, shape=obs_shape,
dtype=np.uint8)

    def observation(self, observation):
        transforms = T.Compose([T.Resize(self.shape), T.Normalize(0, 255)])
        return transforms(observation).squeeze(0)

# preprocess environment
env = SkipFrame(env, skip=4)
env = GrayScaleObservation(env)
env = ResizeObservation(env, shape=84)
env = FrameStack(env, num_stack=4)

"""# Double Deep Q-Networks"""

class MarioNet(nn.Module):

    def __init__(self, input_dim, output_dim):
        super().__init__()
        c, w, h = input_dim

        if h != 84: raise ValueError(f"Expecting input height: 84, got: {h}")
        if w != 84: raise ValueError(f"Expecting input width: 84, got: {w}")

        self.online = nn.Sequential(
            nn.Conv2d(in_channels=c, out_channels=32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1),
            nn.ReLU(),
            nn.Flatten(),
            nn.Linear(3136, 512),
            nn.ReLU(),

```

```

        nn.Linear(512, output_dim)
    )

    self.target = copy.deepcopy(self.online)

    # freeze Q-target parameters
    for p in self.target.parameters():
        p.requires_grad = False

    def forward(self, input, model):
        if model == "online":
            return self.online(input)
        elif model == "target":
            return self.target(input)

"""# Agent"""

class Mario:
    """
        Mario randomly explores with a chance of self.exploration_rate.
        When he chooses to exploit, he relies on MarioNet to provide the most
        optimal action.
    """
    def __init__(self, state_dim, action_dim, use_cuda):
        self.state_dim = state_dim
        self.action_dim = action_dim
        self.use_cuda = use_cuda
        self.memory = deque(maxlen=100000)
        self.batch_size = 32

        self.net = MarioNet(self.state_dim, self.action_dim).float()
        if self.use_cuda:
            self.net = self.net.to(device="cuda")

        self.exploration_rate = 1
        self.exploration_rate_decay = 0.99999975
        self.exploration_rate_min = 0.1
        self.curr_step = 0

        self.save_every = 5e5

    def act(self, state):
        """
            Given a state, choose an epsilon-greedy action and update value of step.
        """
        # exploration
        if np.random.rand() < self.exploration_rate:
            action_idx = np.random.randint(self.action_dim)

```

```

# exploitation
else:
    state = state.__array__()

    if self.use_cuda: state = torch.tensor(state).cuda()
    else: state = torch.tensor(state)

    state = state.unsqueeze(0)
    action_values = self.net(state, model="online")
    action_idx = torch.argmax(action_values, axis=1).item()

# decrease exploration_rate
self.exploration_rate *= self.exploration_rate_decay
self.exploration_rate = max(self.exploration_rate_min, self.exploration_rate)

# increment step
self.curr_step += 1
return action_idx

def cache(self, state, next_state, action, reward, done):
    """
    Store the experience to self.memory (replay buffer).
    """
    state = state.__array__()
    next_state = next_state.__array__()

    state = torch.tensor(state)
    next_state = torch.tensor(next_state)
    action = torch.tensor([action])
    reward = torch.tensor([reward])
    done = torch.tensor([done])

    if self.use_cuda:
        state = state.cuda()
        next_state = next_state.cuda()
        action = action.cuda()
        reward = reward.cuda()
        done = done.cuda()

    self.memory.append((state, next_state, action, reward, done))

def recall(self):
    """
    Retrieve a batch of experiences from memory
    """
    batch = random.sample(self.memory, self.batch_size)
    state, next_state, action, reward, done = map(torch.stack, zip(*batch))
    return state, next_state, action.squeeze(), reward.squeeze(), done.squeeze()

```

```
"""# TD Estimate & TD Target"""
```

```
class Mario(Mario):
    def __init__(self, state_dim, action_dim, use_cuda):
        super().__init__(state_dim, action_dim, use_cuda)
        self.gamma = 0.9
        self.burnin = 1e4 # min. experiences before training
        self.learn_every = 3 # no. of experiences between updates to Q_online
        self.sync_every = 1e4 # no. of experiences between Q_target & Q_online
    sync
        self.optimizer = torch.optim.Adam(self.net.parameters(), lr=0.00025)
        self.loss_fn = torch.nn.SmoothL1Loss()

    def td_estimate(self, state, action):
        current_Q = self.net(state, model="online")[
            np.arange(0, self.batch_size), action
        ]
        return current_Q

    @torch.no_grad()
    def td_target(self, reward, next_state, done):
        next_state_Q = self.net(next_state, model="online")
        best_action = torch.argmax(next_state_Q, axis=1)
        next_Q = self.net(next_state, model="target")[
            np.arange(0, self.batch_size), best_action
        ]
        return (reward + (1 - done.float()) * self.gamma * next_Q).float()

    def update_Q_online(self, td_estimate, td_target):
        loss = self.loss_fn(td_estimate, td_target)
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
        return loss.item()

    def sync_Q_target(self):
        self.net.target.load_state_dict(self.net.online.state_dict())

    def learn(self):
        if self.curr_step % self.sync_every == 0: self.sync_Q_target()
        if self.curr_step % self.save_every == 0: self.save()
        if self.curr_step < self.burnin: return None, None
        if self.curr_step % self.learn_every != 0: return None, None
```

```

    # Sample from memory
    state, next_state, action, reward, done = self.recall()

    # Get TD Estimate
    td_est = self.td_estimate(state, action)

    # Get TD Target
    td_tgt = self.td_target(reward, next_state, done)

    # Backpropagate loss through Q_online
    loss = self.update_Q_online(td_est, td_tgt)

    return (td_est.mean().item(), loss)

"""# Logging"""

class MetricLogger:
    def __init__(self):
        # history metrics
        self.ep_rewards = []
        self.ep_lengths = []
        self.ep_avg_losses = []
        self.ep_avg_qs = []

        # moving averages, added for every call to record()
        self.moving_avg_ep_rewards = []
        self.moving_avg_ep_lengths = []
        self.moving_avg_ep_avg_losses = []
        self.moving_avg_ep_avg_qs = []

        # current episode metric
        self.init_episode()

        # timing
        self.record_time = time.time()

    def log_step(self, reward, loss, q):
        self.curr_ep_reward += reward
        self.curr_ep_length += 1
        if loss:
            self.curr_ep_loss += loss
            self.curr_ep_q += q
            self.curr_ep_loss_length += 1

    def log_episode(self):
        self.ep_rewards.append(self.curr_ep_reward)
        self.ep_lengths.append(self.curr_ep_length)
        if self.curr_ep_loss_length == 0:
            ep_avg_loss = 0
            ep_avg_q = 0

```

```

else:
    ep_avg_loss = np.round(self.curr_ep_loss / self.curr_ep_loss_length, 5)
    ep_avg_q = np.round(self.curr_ep_q / self.curr_ep_loss_length, 5)
    self.ep_avg_losses.append(ep_avg_loss)
    self.ep_avg_qs.append(ep_avg_q)

self.init_episode()

def init_episode(self):
    self.curr_ep_reward = 0.0
    self.curr_ep_length = 0
    self.curr_ep_loss = 0.0
    self.curr_ep_q = 0.0
    self.curr_ep_loss_length = 0

def record(self, episode, epsilon, step):
    mean_ep_reward = np.round(np.mean(self.ep_rewards[-100:]), 3)
    mean_ep_length = np.round(np.mean(self.ep_lengths[-100:]), 3)
    mean_ep_loss = np.round(np.mean(self.ep_avg_losses[-100:]), 3)
    mean_ep_q = np.round(np.mean(self.ep_avg_qs[-100:]), 3)
    self.moving_avg_ep_rewards.append(mean_ep_reward)
    self.moving_avg_ep_lengths.append(mean_ep_length)
    self.moving_avg_ep_avg_losses.append(mean_ep_loss)
    self.moving_avg_ep_avg_qs.append(mean_ep_q)

    last_record_time = self.record_time
    self.record_time = time.time()
    time_since_last_record = np.round(self.record_time - last_record_time, 3)

    print(
        "Episode:{:4d} :: Step:{:5d} :: Epsilon:{:8.3f} :: Mean_Reward:{:8.3f} :: " \
        "Mean_Length:{:8.3f} :: Mean_Loss:{:4.3f} :: Mean_Q_Value:{:8.3f} :: " \
        "Time_Delta:{:8.3f} "
        .format(episode, step, epsilon, mean_ep_reward, mean_ep_length,
            mean_ep_loss, mean_ep_q, time_since_last_record)
    )

"""# Training"""

use_cuda = torch.cuda.is_available()

mario = Mario(state_dim=(4, 84, 84), action_dim=env.action_space.n,
    use_cuda=use_cuda)

logger = MetricLogger()

episodes = 21
for e in range(episodes):

```

```

state = env.reset()

# Play the game!
while True:

    # Run agent on the state
    action = mario.act(state)

    # Agent performs action
    next_state, reward, done, info = env.step(action)

    # Remember
    mario.cache(state, next_state, action, reward, done)

    # Learn
    q, loss = mario.learn()

    # Logging
    logger.log_step(reward, loss, q)

    # Update state
    state = next_state

    # Check if end of game
    if done or info["flag_get"]:
        break

logger.log_episode()

if e % 2 == 0:
    logger.record(episode=e, epsilon=mario.exploration_rate,
step=mario.curr_step)

import matplotlib.pyplot as plt

# Example data
episodes = [0, 2, 4, 6, 8, 10]
rewards = [1270, 1507, 1537, 1624, 1506, 1628]
losses = [0, 0, 0, 0.143, 0.265, 0.331]

# Plotting rewards and losses over episodes
plt.figure(figsize=(10, 6))
plt.plot(episodes, rewards, label='Mean Reward', marker='o')
plt.plot(episodes, losses, label='Mean Loss', marker='x')
plt.xlabel('Episode')
plt.ylabel('Value')
plt.title('Agent Performance Over Episodes')
plt.legend()
plt.grid(True)
plt.show()

```

```

import matplotlib.pyplot as plt

# Example data for 20 episodes
episodes = list(range(21)) # Episodes 0 to 20
rewards = [1270, 1507, 1537, 1624, 1506, 1628, 1709, 1639, 1625, 1632, 1636,
1709, 1639, 1625, 1632, 1636, 1709, 1639, 1625, 1632, 1636] # Sample reward
data
losses = [0, 0, 0, 0.143, 0.265, 0.331, 0.370, 0.383, 0.396, 0.412, 0.421, 0.370,
0.383, 0.396, 0.412, 0.421, 0.370, 0.383, 0.396, 0.412, 0.421] # Sample loss data

# Plotting rewards and losses over episodes
plt.figure(figsize=(10, 6))
plt.plot(episodes, rewards, label='Mean Reward', marker='o')
plt.plot(episodes, losses, label='Mean Loss', marker='x')
plt.xlabel('Episode')
plt.ylabel('Value')
plt.title('Agent Performance Over 20 Episodes')
plt.legend()
plt.grid(True)
plt.show()

import matplotlib.pyplot as plt

# Sample data for episode length over 20 episodes
episodes = list(range(21)) # Episodes 0 to 20
episode_lengths = [1683, 1075, 1526, 1668, 1392, 1535, 1779, 1832, 1834, 1773,
1819, 1779, 1832, 1834, 1773, 1819, 1779, 1832, 1834, 1773, 1819] # Replace
with your data

# Plotting episode length over episodes
plt.figure(figsize=(10, 6))
plt.plot(episodes, episode_lengths, label='Episode Length', marker='o',
color='green')
plt.xlabel('Episode')
plt.ylabel('Episode Length (Steps)')
plt.title('Episode Length Over 20 Episodes')
plt.legend()
plt.grid(True)
plt.show()

"""# Play a game"""

!apt update
!apt install xvfb

!pip install pyvirtualdisplay

```

```

import pyvirtualdisplay

display = pyvirtualdisplay.Display()
display.start()

!pip --disable-pip-version-check install -q pyvirtualdisplay

from pyvirtualdisplay import Display

from IPython import display as ipythondisplay
from IPython.display import HTML

from gym.wrappers import RecordVideo
from glob import glob

import base64
import io

display = Display(visible=0, size=(600, 300))
display.start()

bstate = env.reset()

# Play the game!
while True:

    # Run agent on the state
    action = mario.act(state)

    # Agent performs action
    next_state, reward, done, info = env.step(action)

    # Update state
    state = next_state

    # Check if end of game
    if done or info["flag_get"]:
        break
show_video()

!pip install gym-super-mario-bros nes-py pyvirtualdisplay
!apt-get install -y xvfb python-opengl ffmpeg

from pyvirtualdisplay import Display
display = Display(visible=0, size=(600, 400))
display.start()

import gym
from nes_py.wrappers import JoypadSpace
from gym_super_mario_bros.actions import SIMPLE_MOVEMENT

```

```

from gym.wrappers import RecordVideo
import glob
import io
import base64
from IPython.display import HTML

# Initialize game environment and wrappers
env = gym.make("SuperMarioBros-v3")
env = JoypadSpace(env, SIMPLE_MOVEMENT)

# Setup Monitor wrapper to record video
env = RecordVideo(env, "./video")

state = env.reset()

done = False
while not done:
    action = env.action_space.sample() # Your agent's action (e.g.,
    mario.act(state)
    state, reward, done, info = env.step(action)

# Display the recorded video
video_files = glob.glob("./video/*.mp4")
if video_files:
    video_file = video_files[0]
    video = io.open(video_file, 'r+b').read()
    encoded = base64.b64encode(video)
    HTML(data="""<video width="320" height="240" controls>
        <source src="data:video/mp4;base64,{0}" type="video/mp4" />
    </video>""".format(encoded.decode('ascii'))))
else:
    print("Video file not found")

!pip install gym-super-mario-bros nes-py
!apt-get install -y xvfb python-opengl ffmpeg
!pip install pyvirtualdisplay

import gym_super_mario_bros
from nes_py.wrappers import JoypadSpace
from gym_super_mario_bros.actions import SIMPLE_MOVEMENT
from gym.wrappers import RecordVideo

env = gym_super_mario_bros.make('SuperMarioBros-v0')
env = JoypadSpace(env, SIMPLE_MOVEMENT)
env = RecordVideo(env, "./video")

```

