GENERALIZABLE ARTIFICIAL INTELLIGENCE AGENT IN GAME DEVELOPMENT

MR. TANATANEE PONARK
MR. INTOUCH YUSOH

A PROJECT SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR
THE DEGREE OF BACHELOR OF ENGINEERING (COMPUTER ENGINEERING)
FACULTY OF ENGINEERING
KING MONGKUT'S UNIVERSITY OF TECHNOLOGY THONBURI
2024

Generalizable Artificial Intelligence Agent in Game Development

Mr. Tanatanee Ponark
Mr. Intouch Yusoh

A Project Submitted in Partial Fulfillment
of the Requirements for
the Degree of Bachelor of Engineering (Computer Engineering)
Faculty of Engineering
King Mongkut's University of Technology Thonburi
2024

Project Committee

..... (Assoc.Prof. Natasha Dejdumrong, D.Tech.Sci.)	Project Advisor
..... (Asst.Prof. Nuttanart Muansuwan, Ph.D.)	Committee Member
..... (Asst.Prof. Phond Phunchongharn, Ph.D.)	Committee Member
..... (Jaturon Harnsomburana, Ph.D.)	Committee Member

Project Title	Generalizable Artificial Intelligence Agent in Game Development
Credits	3
Member(s)	Mr. Tanatanee Ponark
	Mr. Intouch Yusoh
Project Advisor	Assoc.Prof. Natasha Dejdumrong, D.Tech.Sci.
Program	Bachelor of Engineering
Field of Study	Computer Engineering
Department	Computer Engineering
Faculty	Engineering
Academic Year	2024

Abstract

This project focuses on the development of adaptive artificial intelligence (AI) for NPCs in video games, with an emphasis on creating reusable AI assets for indie developers. The goal is to design a scalable and efficient reinforcement learning (RL) system that can be applied across similar game environments, reducing the complexity and cost of integrating AI in game development. The project explores the use of deep reinforcement learning algorithms, specifically Proximal Policy Optimization (PPO) and Deep Q-Networks (DQN), to train AI agents capable of learning dynamic behaviors through interaction with the game environment.

The primary objective of this project is to create a 2D platformer game in Unity, where two RL agents—the ”player” and the ”opponent”—learn to interact with each other and the environment. Both agents undergo training in a reinforcement learning setup using the Unity ML-Agents framework, with agents receiving rewards and penalties based on their performance. The project compares PPO and DQN algorithms in terms of training efficiency, stability, and performance to determine the most suitable approach for different game scenarios. A variety of evaluation metrics, both quantitative and qualitative, are used to assess the agents' learning progress, behavior, and generalization ability.

The outcome of this research will provide valuable insights into the practical application of adaptive AI in games, offering reusable and scalable AI solutions for developers, particularly those with limited resources. The project also contributes to the broader field of AI in game development by addressing challenges related to agent generalization, learning speed, and performance evaluation. Ultimately, the developed AI assets aim to streamline the integration of adaptive NPCs, enabling indie developers to create more engaging and intelligent game experiences.

Keywords: Game Development / Artificial Intelligence / Enemy AI / Reusable Assets / Non-Player Characters / Adaptive AI / Reinforcement Learning / Procedural Learning / Game AI Training / AI Assets

หัวข้อปริญญานิพนธ์	ปัญญาประดิษฐ์ที่ปรับตัวได้สำหรับการพัฒนาเกม
หน่วยกิต	3
ผู้เขียน	นายธนธานี โพธิ์นาค
	นายอินทัช ยูโซะ
อาจารย์ที่ปรึกษา	รศ.ดร.ณัฐชา เดชดำรง
หลักสูตร	วิศวกรรมศาสตรบัณฑิต
สาขาวิชา	วิศวกรรมคอมพิวเตอร์
ภาควิชา	วิศวกรรมคอมพิวเตอร์
คณะ	วิศวกรรมศาสตร์
ปีการศึกษา	2567

บทคัดย่อ

Thai translation coming soon

คำสำคัญ: Game Development / Artificial Intelligence / Enemy AI / Reusable Assets / Non-Player Characters / Adaptive AI / Reinforcement Learning / Procedural Learning / Game AI Training / AI Assets

ACKNOWLEDGMENTS

Acknowledge your advisors and thanks your friends here..

CONTENTS

PAGE

LIST OF TABLES

TABLE **PAGE**

LIST OF FIGURES

FIGURE **PAGE**

LIST OF SYMBOLS

SYMBOL		UNIT
α	Test variable	m^2
λ	Interarival rate	jobs/ second
μ	Service rate	jobs/ second

LIST OF TECHNICAL VOCABULARY AND ABBREVATIONS

AI	=	Artificial Intelligence
NPC	=	Non Player Characters
RL	=	Reinforcement Learning

CHAPTER 1 INTRODUCTION

1.1 Background

Artificial intelligence (AI) has become integral to modern game development, particularly in controlling non-player characters (NPCs) that provide dynamic, interactive gameplay experiences. NPCs, whether enemies or allies, play a vital role in enhancing immersion by reacting to player actions and shaping the overall game dynamics. Traditionally, NPC behavior has been managed using systems like finite state machines or behavior trees. While effective for basic functionality, these systems often result in predictable behaviors that lack the depth and variability expected by contemporary players. As player expectations grow, there is an increasing need for NPCs that exhibit more sophisticated, adaptive behaviors—behaviors that can dynamically evolve based on player interactions, making the game world feel more reactive and alive.

Machine learning, particularly reinforcement learning (RL), has emerged as a promising method to create more intelligent and adaptable NPCs. RL allows agents to learn from their environment by interacting with it and receiving feedback, enabling NPCs to adapt and evolve their strategies in response to the player's actions. This represents a significant advancement over traditional AI methods and offers the potential for more complex, unpredictable behavior that enhances the overall gaming experience.

However, as AI becomes more sophisticated, it also becomes more resource-intensive to develop. Building and training intelligent NPCs requires significant time, computational power, and specialized knowledge. This can be particularly challenging for small, indie game developers with limited resources, who often rely on simpler AI models like state machines or behavior trees due to budget constraints. Furthermore, in traditional game development workflows, NPC AI models are often tailored to the specific mechanics and environment of a single game. As a result, each new game requires a new AI model, making it inefficient and costly to integrate advanced AI into multiple projects.

In addition to enhancing NPC behavior, AI systems can also play a crucial role in the game testing phase. With enough training in a game environment, AI agents can be used as automated players, playing through the game to identify bugs, balance gameplay, and improve overall quality. This reduces the need for manual testing and speeds up the development cycle.

1.2 Problem Statements

While the gaming industry continues to push the boundaries of interactive and immersive experiences, there remains a significant gap in the accessibility and scalability of advanced AI systems, especially for indie developers. Many small studios and developers face the challenge of limited resources, which makes the integration of complex, adaptive AI systems difficult. Traditional NPC AI models, such as finite state machines or behavior trees, are easy to implement but result in static, predictable behaviors that fail to meet modern expectations for dynamic, responsive gameplay. Moreover, these AI systems are often specific to a single game and need to be redesigned and retrained for every new project, leading to redundancy and increased development costs.

The lack of reusable, adaptable AI assets across multiple game titles is a major bottleneck in the development process. Each game requires developers to start from scratch in designing NPC behaviors, resulting in significant time and resource expenditures that could otherwise be dedicated to more creative aspects of game development, such as narrative design and gameplay mechanics.

Additionally, the process of testing a game is often lengthy and repetitive, requiring developers to manually test various aspects of gameplay, identify bugs, and adjust balance. The integration of AI-driven automated testing could streamline this process, allowing for faster iteration and a more polished final product.
This research initiative seeks to address these issues by developing reusable and adaptable AI assets that can be integrated into multiple games, reducing the need for redundant AI development. By leveraging machine learning techniques, particularly reinforcement learning, the project aims to create AI systems that can adapt to a variety of game environments and player behaviors. This would not only allow developers to save time and resources but also provide a more dynamic and engaging experience for players. Furthermore, the project explores the potential of using AI agents for automated game testing, further improving efficiency in the game development lifecycle.

1.3 Objectives

The main objectives of this project are to develop adaptive AI systems that are reusable across multiple game titles, enhancing both gameplay and testing processes. These objectives will be pursued through the following key goals:

1.3.1 Create a Generalized Game Environment for AI Training

Design and develop a game that provides a balanced training environment adaptable to various AI learning scenarios. The game will be structured to avoid overfitting, ensuring that the trained AI models can generalize well across different game mechanics and environments. This flexibility will maximize the adaptability of the AI models to new challenges and scenarios.

1.3.2 Develop Two Adaptive AI Agents

- **Opponent AI:** Create an AI model to function as a challenging non-player character (NPC) in the game. This AI will observe the player's actions and adjust its strategy accordingly, providing dynamic and unpredictable gameplay.
- **Player AI:** Develop a second AI model that learns to play the game efficiently through reinforcement learning. This AI will adapt its strategy based on the environment, improving its performance over time by learning from both successes and failures.

1.3.3 Create Reusable AI Assets

Both the opponent and player AI models will be trained and developed as reusable assets. These assets will be designed for easy integration into future games within similar genres, requiring minimal adjustments. The AI models will be tunable to accommodate game-specific features, such as unique level mechanics and new actions, ensuring their applicability across a variety of titles.

1.3.4 Ensure AI Model Adaptability and Scalability

The AI models will be designed to generalize across different game levels and mechanics, enabling scalability for future game expansions. Both the opposition and player AI will be adaptable to new environments, additional features, and evolving gameplay mechanics, ensuring their continued relevance as games grow and evolve.

1.4 Scope of Work

This project focuses on the design, development, and implementation of adaptive AI agents for use in game development. The scope of the work encompasses the following key areas:

1.4.1 Game Design and Development

- **Game Environment Creation:** Develop a generalized 2D game environment that serves as a training ground for the AI models. The game will be designed to provide diverse challenges that allow the AI to adapt and improve over time.
- **Game Balancing:** Ensure that the game mechanics are balanced to prevent overfitting during AI training while providing a variety of scenarios for both player and opponent AI to learn from.

1.4.2 AI Model Development

- **Player AI Model:** Develop a reinforcement learning model that will learn to play the game efficiently by adapting its strategies based on the game environment and its interactions with the opposition AI.
- **Opponent AI Model:** Create an adaptive AI model that will serve as a challenging non-player character (NPC), adjusting its behavior dynamically based on the player's actions.

1.4.3 Training and Evaluation

- **Reinforcement Learning:** Train both the player and opponent AI models using reinforcement learning techniques, ensuring the models improve over time through trial and error.
- **Evaluation Metrics:** Implement evaluation mechanisms to track the progress and performance of the AI agents during training. This includes assessing their ability to adapt to different challenges within the game environment.

1.4.4 Reusable AI Assets Creation

- **Modular Design:** Both AI models will be designed as reusable assets, allowing them to be easily integrated into future games with minimal adjustments. This includes creating a modular framework for actions, behaviors, and decision-making processes that can be adapted to new game environments.

1.4.5 Scalability and Adaptability

- **Adaptable AI Models::** The AI models will be tested and refined to ensure they can scale across multiple levels and adapt to different game mechanics. This will allow future games to incorporate these AI agents without the need for extensive retraining.
- **Customization:** Provide flexibility in AI behavior customization, enabling future developers to adjust the agents' actions to fit the specific needs of new games.

1.4.6 AI for Automated Game Testing

- **Automated Testing Integration:** Develop the AI models to also serve as automated game testers. This will involve enabling the AI to play through the game from start to finish, identifying potential bugs and providing insights for gameplay balancing, which will reduce time and costs associated with manual testing.

1.5 Project Schedule

The Gantt chart below (Figure 1.1) illustrates the planned timeline for the project.

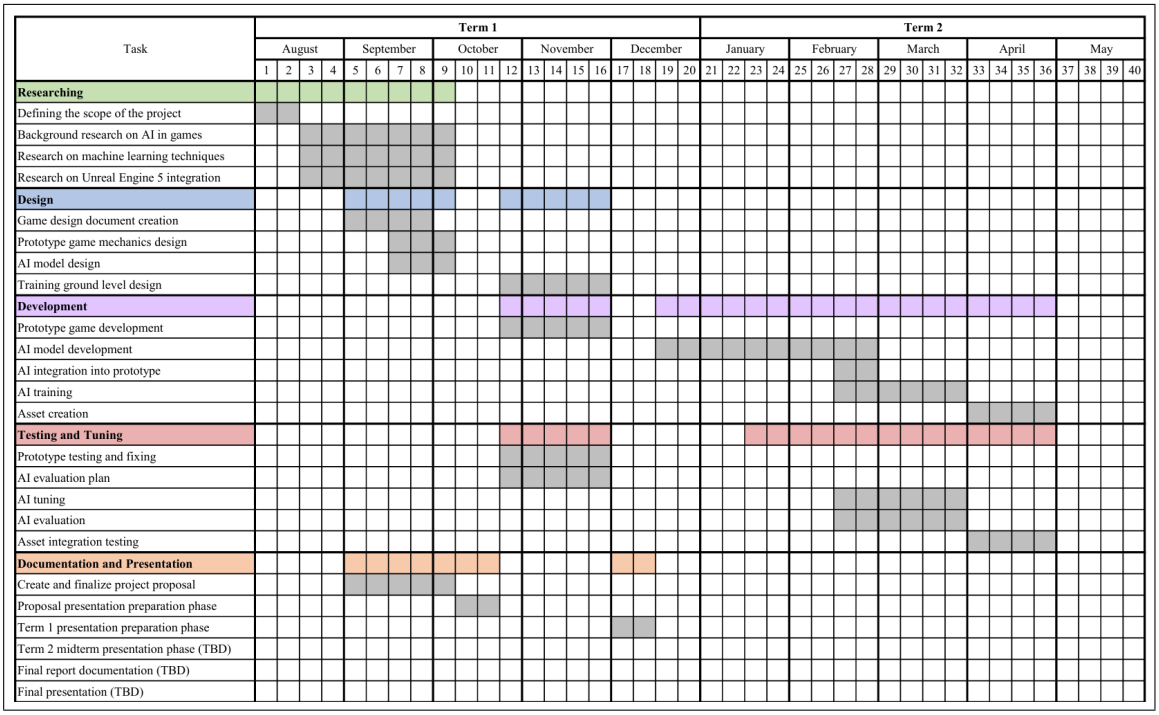

Figure 1.1 Gantt Chart

CHAPTER 2 BACKGROUND THEORY AND RELATED WORK

2.1 Introduction

This chapter provides the essential background knowledge and theoretical foundations necessary to understand the development of adaptive artificial intelligence (AI) in game environments. It covers key concepts, technologies, and programming languages utilized in AI development, with a focus on machine learning techniques such as reinforcement learning and relevant level design principles. Additionally, this chapter will explore related research and existing solutions, offering insights into the current state of AI in game development.

2.2 Theories and Core Concepts

2.2.1 Game Development

2.2.1.1 Game Development Life Cycle

Game development is a complex and multifaceted process that involves various stages, from initial concept to the final product. The main stages include:

- **Pre-production:** Conceptualization, storyboarding, and planning of the game. This stage defines the genre, mechanics, platform, and overall vision for the game.
- **Production:** The actual creation of the game, which includes coding, asset creation (art, sound, and animations), level design, and initial testing.
- **Post-production:** Refining the game through bug fixes, balancing, optimization, and quality assurance. This is also when the game is released to the public, followed by ongoing support such as updates and patches.

Throughout all stages, collaboration among developers, designers, artists, and sound engineers is essential to creating a cohesive experience. Game development today often involves teams of varying sizes, from small indie developers to large studios with hundreds of team members.

2.2.1.2 Game Design

Game design encompasses various elements that contribute to a game's playability, engagement, and overall success:

- **Game Mechanics:** The rules and systems that govern how the game works. These include the physics, player controls, objectives, and win/lose conditions.
- **Story and Narrative:** The storyline or plot that guides the player through the game. This can range from minimal or abstract narratives to fully immersive stories with complex characters and world-building.
- **Level Design:** The creation of game environments and how players progress through them. Effective level design includes balancing challenge, pacing, and exploration, often introducing new elements at each stage to maintain player interest.
- **Assets:** Game assets refer to the components used to build the visual, auditory, and interactive elements of a game. These include:

  - **Visual Assets:** Textures, 3D models, 2D sprites, animations, and visual effects that define the game's appearance.
  - **Audio Assets:** Sound effects, background music, and character voices that create the auditory experience.
  - **Code Assets:** Scripts and algorithms that govern game mechanics, AI behavior, and interactive elements.
  - **UI Assets:** Icons, buttons, menus, and other user interface components.

  Assets are the building blocks of any game, providing the necessary materials to bring concepts to life. Their quality and integration play a significant role in defining the overall aesthetic and functionality of the game.

- **User Interface (UI) and User Experience (UX):** The design of menus, HUDs (heads-up displays), and the interaction flow. A clean and intuitive UI is critical for keeping players immersed in the game.

2.2.1.3 Challenges Faced by Developers

- **Resource Limitations:** Small teams often lack the budget and manpower to implement complex systems like advanced AI, detailed art, or high-end graphics
- **Complexity of Game Design:** The more complex the game, the harder it is to manage. Creating a balanced and fun experience while ensuring that the game performs well across different platforms is a constant challenge.
- **Market Saturation:** With the rise of digital storefronts and indie game platforms, the market is flooded with games, making it hard for any individual title to stand out.
- **Time Constraints:** Many indie developers work on tight deadlines, often needing to finish a game within a limited period to avoid budget overruns or loss of momentum.

2.2.1.4 The Increasing Demand for Adaptive and Dynamic Gameplay

Players today expect more than static, predictable game experiences. They want games to react to their choices in real-time, creating a more dynamic and immersive experience. This demand has led to the integration of more advanced AI systems that can adapt and respond to player behavior in meaningful ways.
Games like *The Witcher 3* and *Horizon Zero Dawn* offer NPCs that respond realistically to the player's actions, creating a sense of a living world. This trend toward dynamic and reactive gameplay is challenging for developers, especially smaller teams with limited resources, as it often requires implementing sophisticated AI and adaptive systems that can learn and evolve.

2.2.2 Artificial Intelligence

Artificial Intelligence (AI) in gaming refers to the simulation of human-like intelligence and decision-making processes in game entities. It plays a vital role in enhancing the interactivity and immersion of games by enabling non-player characters (NPCs) to exhibit responsive and believable behaviors. AI in games can range from simple rule-based systems to sophisticated machine learning models that adapt and evolve over time.

2.2.2.1 Types of AI in Game Development

- **Finite State Machines (FSMs):** FSMs are one of the simplest AI systems used in games. They consist of a finite set of states (e.g., idle, attack, flee) and predefined transitions between these states based on specific conditions. FSMs are easy to implement but may lead to predictable and repetitive behaviors.
- **Behavior Trees:** Behavior trees are hierarchical structures used to manage decision-making processes. They allow for more modular and reusable AI behaviors compared to FSMs, making them popular in modern game development for controlling NPCs.
- **Utility-Based AI:** This system evaluates different actions based on a utility score, selecting the action with the highest score. Utility-based AI allows NPCs to make decisions based on dynamic priorities, leading to more flexible and context-aware behaviors.
- **Pathfinding and Navigation:** AI-controlled entities often need to move through complex environments. Pathfinding algorithms like A* (A-star) are widely used to calculate the shortest or most efficient paths, avoiding obstacles and optimizing movement.
- **Machine Learning in Games:** Machine learning (ML) introduces adaptability and dynamic behaviors that traditional systems cannot achieve. Techniques like reinforcement learning enable AI agents to learn from their interactions with the environment, improving their performance over time. These systems can produce more realistic, challenging, and unpredictable NPCs.

2.2.2.2 Applications of AI in Game Development

- **NPC Behavior and Opponent AI:** AI is commonly used to control NPCs, making them act as enemies, allies, or neutral characters. The goal is to create behaviors that challenge and engage players without being overly predictable or frustrating.
- **Procedural Content Generation (PCG):** AI can generate game assets such as levels, characters, and items dynamically. This enhances replayability and reduces development time by automating repetitive tasks.
- **Game Testing and QA:** Adaptive AI agents can be trained to simulate player behaviors, identifying bugs, balancing gameplay, and stress-testing game mechanics during development.
- **Player Profiling and Personalization** AI systems can analyze player data to adapt the gameplay experience, tailoring difficulty, content, and narratives to individual player preferences.

2.2.2.3 Advantages of AI in Game Development

- **Dynamic and Immersive Gameplay:** Adaptive AI creates experiences that feel more responsive and personalized.
- **Enhanced Development Efficiency:** Automating tasks like testing, content generation, and debugging reduces time and costs.
- **Scalability:** AI systems can be designed to adapt across different game genres and mechanics, enabling reusable assets.s

2.2.2.4 Challenges of AI in Game Development

- **Development Costs:** Advanced AI systems require significant computational resources and expertise, which can be a challenge for indie developers.
- **Complexity in Balancing:** Creating an AI that is both challenging and fair to players requires careful design and testing.
- **Unpredictable Outcomes:** In adaptive systems, emergent behaviors may lead to unintended gameplay consequences.

AI in game development is an ever-evolving field, driven by advances in computational power and machine learning algorithms. It holds tremendous potential for creating smarter, more immersive games while addressing development bottlenecks like scalability and automation.

2.2.3 Machine Learning

Machine learning involves programming computers to optimize a performance criterion based on example data or past experiences. It entails constructing models capable of recognizing patterns or regularities in data, enabling systems to make predictions, adapt, or gain insights. Grounded in statistical theory, machine learning employs efficient algorithms to manage large datasets and perform complex computations. It has widespread applications in fields such as retail, finance, healthcare, and telecommunications, addressing challenges like prediction, optimization, and pattern recognition.
Machine learning algorithms are commonly categorized into three types based on their learning paradigm: supervised learning, unsupervised learning, and reinforcement learning.

2.2.3.1 Supervised Learning

Supervised learning relies on labeled datasets, where the correct outcome is predefined, to train models. These models predict outcomes based on input data and refine predictions by learning from errors. Supervised learning is frequently applied in classification and regression tasks.

- **Naive Bayes:** Naive Bayes is a classification method based on Bayes Theorem, assuming that features are independent of each other. There are three types: Multinomial, Bernoulli, and Gaussian Naive Bayes. It's commonly used in text classification, spam detection, and recommendation systems.
- **K-nearest neighbor:** KNN is a non-parametric algorithm that classifies data points based on proximity, usually using Euclidean distance. It's simple and efficient for small datasets but becomes slower as the dataset grows. KNN is often used in recommendation engines and image recognition.
- **Random forest:** Random forest is a flexible supervised algorithm for classification and regression, combining multiple decision trees to reduce variance and improve accuracy.

2.2.3.2 Unsupervised Learning

Unsupervised learning models analyze unlabeled data to discover hidden patterns or structures. This approach is typically applied in clustering, dimensionality reduction, and anomaly detection.

- **K-means clustering:** K-means clustering groups data points into K clusters based on their distance from the centroids. Larger K values create smaller, more granular groups, while smaller K values result in larger clusters. It's widely used in market segmentation, document clustering, and image compression.

- **Principal component analysis (PCA):** PCA is a dimensionality reduction method that transforms data into new components, maximizing variance while reducing redundancy. Each successive component is uncorrelated and orthogonal to the previous one, capturing the most variance in fewer dimensions.
- **Autoencoders:** Autoencoders use neural networks to compress and reconstruct data, with the hidden layer acting as a bottleneck. The process is split into *encoding* (input to hidden layer) and *decoding* (hidden layer to output).

2.2.3.3 Reinforcement Learning

Reinforcement learning allows an agent to learn by interacting with an environment to achieve a specific goal. Mimicking trial-and-error learning, RL optimizes actions using rewards for goal-oriented behaviors and penalties for undesirable actions. This iterative process helps the agent refine its strategy to maximize cumulative rewards.

Key Components of RL:

- **Agent:** The learner or decision-maker.
- **Environment:** The system with which the agent interacts.
- **Actions:** Choices available to the agent.
- **Rewards:** Feedback to guide the agent's learning process.
- **Policy:** The strategy determining the agent's actions.

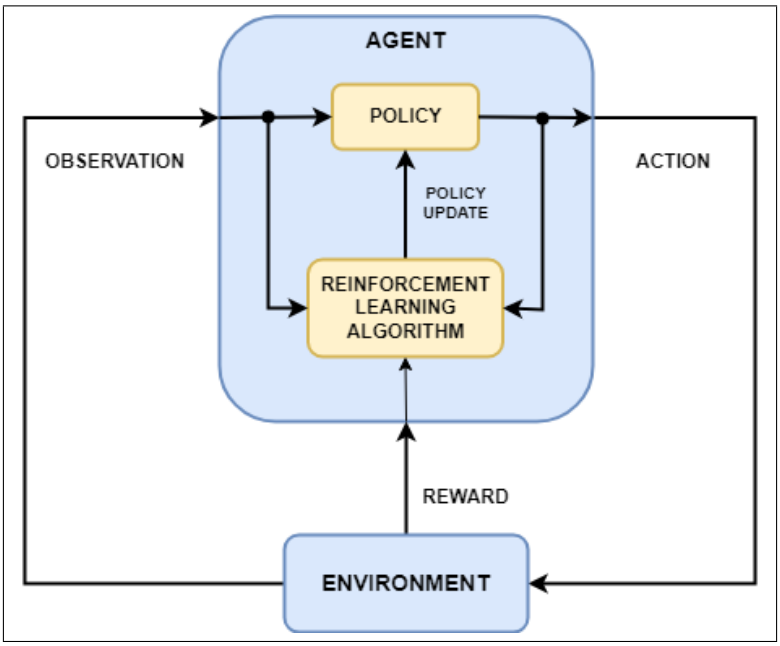

Figure 2.1 Reinforce Learning Model Diagram

Source: https://www.mathworks.com/help/reinforcement-learning/ug/what-is-reinforcement-learning.html

In game development, RL is particularly useful for training adaptive AI that can learn and evolve based on player interactions. This enables the creation of NPCs with more dynamic and challenging behaviors, as well as automated game testing agents that evaluate game mechanics and balance.

2.2.4 Reinforcement Learning

Reinforcement learning (RL) focuses on training agents to make sequences of decisions by interacting with an environment. The agent learns to optimize a long-term reward by observing states, taking actions, and receiving feedback in the form of rewards or penalties. This section explores key RL algorithms relevant to game development and their applications.

2.2.4.1 Q-Learning

Q-Learning is one of the simplest RL algorithms that builds a Q-value table to represent the expected future rewards for every state-action pair.

Mathematical Foundation

The Bellman Equation updates the Q-value for a given state-action pair iteratively

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma max_a(Q(s',a')) - Q(s,a)) \tag{2.1}$$

- $Q(s,a)$: Current Q-value for state s and action a.
- s: Current state
- a: Current action
- r: Reward received after taking action a in state s
- a': Next action
- s': Next state
- α: Learning rate, controlling the size of the update step.
- γ: Discount factor, determining the importance of future rewards.

Key Features

- Off-policy: It learns the optimal policy independently of the agent's actions.
- Suitable for discrete and small state-action spaces.

Use Case in Games

Training AI for grid-based games like Snake or Pac-Man.

Psuedocode

Algorithm 1 Q-Learning Algorithm

```
Initialize Q-table with zeros
for each episode do
    Initialize state s
    while not done do
        Choose action a using epsilon-greedy policy
        Take action a, observe reward r and next state s'
        Update Q-value:
        Q(s,a) ← Q(s,a) + α * (r + γ * max(Q(s',a')) − Q(s,a))
        s = s'
    end while
end for
```

2.2.4.2 Deep Q-Learning (DQN)

DQN scales Q-Learning to environments with continuous or high-dimensional state spaces by approximating the Q-function using a deep neural network.

Enhancements over Q-Learning:

- **Experience Replay:** Stores past transitions (s, a, r, s') in a replay buffer, sampling mini-batches for training to reduce correlation.
- **Target Network:** A separate network for computing target Q-values, updated less frequently to stabilize training.

Key Equations

Update the neural network weights θ by minimizing the loss:

$$L(\theta) = \mathbb{E}_{(s,a,r,s')}[(r + \gamma max_a Q(s', a; \theta^-) - Q(s, a; \theta))^2] \tag{2.2}$$

- $L(\theta)$: The loss function, a measure of how far the model's predictions are from the true or desired values.
- $Q(s, a; \theta)$: $Q(s, a)$ approximate using neural network θ.
- θ: Parameters of a neural network used to approximate a function.
- $\mathbb{E}$: The average value of the loss across sampled data points.

Use Case in Games

Training agents in platformers, complex strategy games, or visual navigation tasks.

Psuedocode

Algorithm 2 Deep Q-Learning Algorithm

```
Initialize replay buffer and Q-network
for each episode do
    Initialize state s
    while not done do
        Choose action a using epsilon-greedy policy
        Take action a, observe reward r and next state s'
        Store transition (s, a, r, s') in replay buffer
        Sample mini-batch from replay buffer
        Compute target: y = r + γmax(Q_target(s', a'))
        Update Q-network using gradient descent
        Periodically update target network
    end while
end for
```

2.2.4.3 Policy Gradient Methods

Policy gradient methods optimize the policy directly by maximizing the expected reward.

Policy Representation

The policy is parameterized as $\pi_\theta(a|s)$, where θ are the weights of the model.

Objective Function

$$J(\theta) = \mathbb{E}_{\pi_\theta}[\sum_t \gamma^t r_t] \tag{2.3}$$

Gradient Update:

Using the policy gradient theorem:

$$\nabla_\theta J(\theta) = \mathbb{E}[\nabla_\theta log \pi_\theta(a|s) R] \tag{2.4}$$

where R is the cumulative reward.

Use Case in Games

Training agents in games requiring smooth control, such as racing or flight simulators.

Psuedocode

Algorithm 3 Policy Gradient Methods

```
Initialize policy network
for each episode do
    Collect trajectory of states, actions, and rewards
    Compute discounted rewards for each state
    Update policy network to maximize log-probability of actions weighted by rewards
end for
```

2.2.4.4 Proximal Policy Optimization (PPO)

PPO improves policy gradient methods by introducing a clipped objective to ensure stable updates.

Objective Function

$$L^{CLIP}(\theta) = \mathbb{E}[\min(r_t(\theta)A_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t] \tag{2.5}$$

where $r_t(\theta)$ is the probability ratio, A_t is the advantage estimate, and ϵ is a hyperparameter for clipping.

Key Features

- Prevents large policy updates.
- Balances performance and training stability.

Use Case in Games

Robust AI for large-scale multi-agent systems or highly variable environments.

Psuedocode

Algorithm 4 Proximal Policy Optimization

```
for each iteration do
    Collect trajectories using current policy
    Compute advantages using value function
    Update policy network using clipped surrogate objective
    Update value network to reduce value estimation error
end for
```

2.2.4.5 Actor-Critic Methods

Actor-Critic combines policy optimization (actor) with value function estimation (critic).

Advantages

- Prevents large policy updates.
- Balances performance and training stability.

Variants

- **A2C:** Uses synchronous updates with multiple workers.
- **A3C:** Parallelizes training by running multiple agents asynchronously.

Use Case in Games

Real-time strategy games requiring fast adaptation.

Psuedocode

Algorithm 5 Actor-Critic Methods

```
for each iteration do
    Collect trajectories from actor network
    Compute value targets using critic network
    Update actor to maximize policy objective
    Update critic to minimize value estimation error
end for
```

2.2.4.6 Multi-Agent Reinforcement Learning (MARL)

In MARL, multiple agents learn and interact in a shared environment, adapting strategies collaboratively or competitively.

Key Challenges

- Coordination among agents.
- Balancing exploration and exploitation in dynamic environments.

Applications in Games

- Multiplayer strategy games.
- Cooperative tasks in simulation-based training.

Table 2.1 Comparison of RL Algorithms

Algorithm	Advantages	Limitations	Suitable Games
Q-Learning	Simple, effective for small problems.	Struggles with large state spaces.	Grid-based games, Puzzles
Deep Q-Learning (DQN)	Handles complex state spaces.	Computationally intensive.	Platformers, Visual games
Policy Gradient	Works well in continuous action spaces.	Susceptible to instability.	Racing, Flight simulators
PPO	Stable, efficient in diverse scenarios.	Trade-off between stability and speed.	Scalable multi-agent games
Actor-Critic	Fast convergence, suitable for real-time games.	Requires careful tuning.	Real-time strategy games, Simulators
MARL	Enables collaborative/ competitive AI.	Complexity increases with agent count.	Multiplayer games, Co-op challenges

2.3 Development Tools

2.3.1 Unity

Unity is a versatile and widely-used game development engine renowned for its ease of use, flexibility, and strong support for both 2D and 3D projects. It was chosen for this project due to its compatibility with machine learning tools, efficient workflow, and robust asset pipeline. Unity provides a range of tools for AI development, including the Unity ML-Agents Toolkit, NavMesh systems, and animation controllers, which enable the creation of intelligent and adaptive behaviors. These tools allow AI agents to interact with their environment, learn from rewards, and optimize performance effectively.

- **ML-Agents Capabilities** Unity's ML-Agents Toolkit is a robust feature for integrating machine learning into game development. It supports training AI agents using techniques like reinforcement learning (RL), imitation learning, and other advanced approaches. Key features include:
  - **Reinforcement Learning:** ML-Agents supports deep reinforcement learning (DRL), enabling agents to learn through interactions with the environment by maximizing cumulative rewards. This is particularly suitable for developing adaptive AI behaviors in platformer games.
  - **Training Environments:** It simplifies the creation of training environments, allowing agents to interact with dynamic game worlds, respond to stimuli (e.g., obstacles or goals), and improve their performance iteratively.
  - **Training with Multiple Agents:** The toolkit supports simultaneous training of multiple agents in the same environment, ideal for creating coordinated enemy behaviors or competitive agents.
  - **Sensor Integration:** ML-Agents accommodates various sensors (e.g., visual, ray-casting, and custom inputs), enabling agents to gather feedback from the game world for informed decision-making.
  - **Model Export:** Trained models can be exported in the ONNX (Open Neural Network Exchange) format for seamless deployment in Unity, integrating advanced AI behavior into gameplay.
- **2D Game Support:** Unity's native 2D tools, including Tilemaps, physics, and animation systems, streamline the creation of platformer environments. This aligns perfectly with the project's goals by facilitating a smooth workflow for designing levels and mechanics.

2.3.2 C#

C# is Unity's primary programming language, offering an optimal balance of performance, ease of use, and flexibility. It is instrumental in implementing game mechanics and AI behaviors.

- **Performance Optimization:** C# enables efficient scripting for real-time calculations, game events, and AI logic. Its clean syntax and extensive libraries ensure that resource-intensive tasks, such as agent movement and obstacle detection, run efficiently.
- **Custom Gameplay Features:** Using C#, developers can create custom gameplay mechanics, such as pathfinding, player interactions, and dynamic environmental elements. Unity's scripting API provides precise control over agents and game objects, offering flexibility that extends beyond built-in tools.

2.3.3 Python

Python is a widely-used programming language in machine learning and AI development, known for its simplicity, flexibility, and extensive ecosystem of libraries. In this project, Python plays a critical role in developing and training AI models.

- **Machine Learning Development:** Python, in combination with PyTorch, is used to implement reinforcement learning algorithms (e.g., Deep Q-Networks, Proximal Policy Optimization) for training AI agents. Its high-level syntax accelerates experimentation and prototyping, making it ideal for AI development.
- **Data Processing:** Libraries like NumPy and Pandas are used for pre-processing and analyzing training data, enabling efficient monitoring of agent performance and model behavior.
- **Integration with Unity:** Trained models developed in Python can be exported (e.g., using ONNX) and integrated into Unity, where they control agent behavior within the game environment.

2.3.4 PyTorch

PyTorch is an open-source machine learning library widely used for deep learning and reinforcement learning. It plays a crucial role in this project for developing, training, and fine-tuning AI models to operate within the Unity environment.

- **Reinforcement Learning Compatibility:** PyTorch supports key reinforcement learning algorithms such as Deep Q-Networks (DQN) and Proximal Policy Optimization (PPO). These algorithms are integral to training AI agents, enabling them to interact with and learn from the game environment by adapting strategies based on feedback and rewards.
- **GPU Acceleration:** PyTorch's GPU acceleration significantly reduces training time for neural networks. This allows for faster iteration and experimentation, facilitating efficient model development and parameter tuning.
- **Integration with Unity:** Although PyTorch is Python-based, trained models can be exported in ONNX or other compatible formats for integration into Unity. This ensures that machine learning models seamlessly control in-game agents during gameplay, bridging the gap between AI development and deployment.

2.3.5 GitHub

GitHub is a version control and collaboration platform that facilitates the management of source code and assets throughout the project's development.

- **Version Control:** GitHub enables efficient version tracking, allowing developers to monitor changes to the codebase and revert to previous versions when necessary. This ensures stability and helps maintain control over the evolving project.
- **Collaboration:** GitHub supports team collaboration by allowing multiple developers to work simultaneously on different parts of the project. The platform's branching and pull request features ensure an organized and efficient development workflow, enabling seamless merging of contributions.
- **Backup and Deployment:** The repository serves as a secure backup for all project files and assets. Additionally, GitHub Actions can automate tasks such as deployment and testing, streamlining development and ensuring smooth continuous integration.

CHAPTER 3 METHODOLOGY AND DESIGN

3.1 Introduction

This chapter presents the methodological approach and design principles employed in developing the adaptive artificial intelligence for game development. Building upon the theoretical foundation established in Chapter 2, we will explore the practical implementation of machine learning techniques, particularly reinforcement learning, within our project. Key areas of focus include the overall system architecture, specific algorithms and models selected, game environment design, data handling processes, and our iterative development approach. By examining these aspects, we aim to provide a comprehensive understanding of how we've translated theoretical concepts into a functional adaptive AI system, bridging the gap between background knowledge and practical results.

3.2 System Architecture

Figure 3.1 below shows the system architecture diagram of our project which shows the relation of each component.

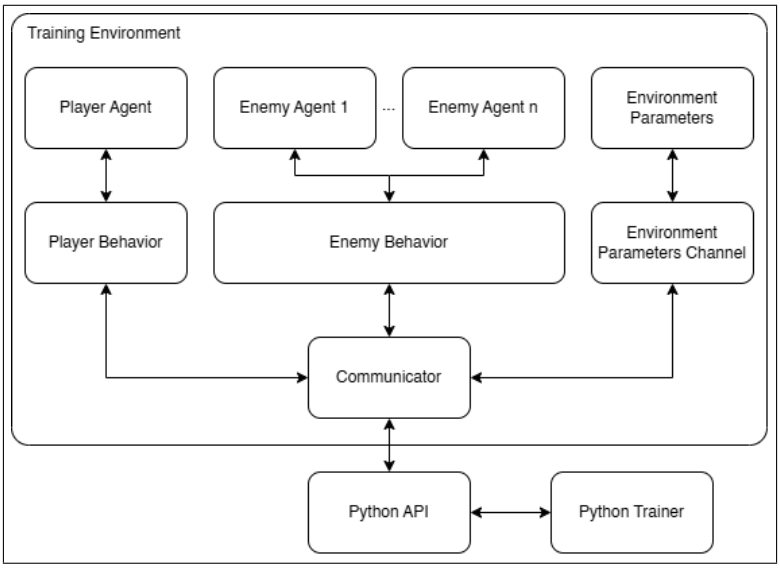

Figure 3.1 System Architecture Diagram, drawn on draw.io

3.2.1 Training Environment

The training environment consists of the Unity scene and the game characters involved. The Unity scene serves as the dynamic environment where agents observe, make decisions, and take actions to learn and improve their behavior.

3.2.1.1 Agents

Agents are entities in the environment that actively interact with their surroundings. Each agent observes the current state of the game environment, make decisions based on the information, and perform actions accordingly. These actions influence the environment and the agent's progression toward its assigned goals.

Agents are also subject to a reward system, where positive or negative feedback is assigned based on the outcomes of their actions. This feedback mechanism enables agents to learn and adapt their behavior over time, aligning their actions with the objectives of the game.
In this project, agents are categorized into two types: the Player Agent, which focuses on achieving level completion and optimizing performance, and the Enemy Agents, which create challenges and obstacles for the Player Agent.

3.2.1.2 Behaviors

Behaviors define how an agent reacts to the environment. A Behavior functions by receiving observations and rewards from the agent, processing this information, and returning appropriate actions. Behaviors can be adjusted and refined through learning, enabling agents to adapt and improve. Each Behavior is uniquely identified by its name and can be dynamic, evolving based on feedback.

3.2.1.3 Environment Parameters

Environment Parameters are configurable settings within the game that can be adjusted during runtime. These parameters can control elements like the difficulty level, enemy behavior, or platform properties. Through Side Channels, Python can send updates to Unity to modify these parameters, while Unity can also send data back to Python for analysis. This two-way communication enables dynamic adjustments, precise control over agent training, and scalability for creating varied scenarios.

3.2.1.4 Communicator

The Communicator acts as the bridge between Unity and external systems, facilitating the exchange of observations, actions, and rewards necessary for reinforcement learning. It enables real-time communication, where Unity sends environment data to Python, which processes it and sends back actions for the agents to execute.

3.2.2 Python API

The Python API allows Python processes to control and interact with the Unity environment through the Communicator. This API enables tasks like training AI models, running simulations, and adjusting game settings during agent learning, offering a flexible way to manage and manipulate the Unity environment.

3.2.3 Python Trainer

The Python Trainer contains machine learning algorithms used to train agents within Unity environments. By interacting with the Python API, the Trainer coordinates the learning process, enabling agents to improve their performance in the Unity simulation. It supports various algorithms, such as reinforcement learning, to allow agents to learn optimal strategies for achieving their goals.

3.3 Game Environment Design

3.3.1 Level Design

To create a robust environment for the AI agent, we conducted a detailed survey of popular 2D platformer games, analyzing their core mechanics, player progression systems, and environmental challenges. This analysis revealed design patterns and gameplay elements that could be adapted for training adaptive AI. Key elements included:

3.3.1.1 Platforms

- Static platforms serve as foundational structures for navigation.
- Dynamic platforms (e.g., moving, disappearing, or rotating platforms) add temporal challenges requiring precise timing and planning.
- These variations ensure that the AI learns to navigate diverse terrain effectively.

3.3.1.2 Obstacles

- Environmental hazards like spikes and moving traps create scenarios that penalize poor decision-making and reward cautious or strategic actions.
- Gaps test the AI's jumping precision, while moving traps challenge its timing and adaptability.

3.3.1.3 Goals

- Reaching an endpoint or collecting specific items provides explicit objectives, simulating real-world gameplay scenarios.
- Goals encourage exploration, decision-making, and risk assessment in pursuit of rewards.

3.3.1.4 Enemies

- These non-playable characters exhibit scripted behaviors, such as patrolling or chasing.
- Their presence requires the AI to decide between engagement (combat) or evasion, depending on its objectives and available resources.

The level progression is carefully crafted to gradually introduce these challenges, ensuring that the AI starts with basic navigation tasks and progresses to solving complex, multi-step problems as it learns and adapts.

3.3.2 Agent Actions

The actions table shown in Table 3.1 below outlines the possible interactions the AI agents have with the environment, divided into two roles: Player Agent and Enemy Agent. These actions simulate the gameplay mechanics and ensure that the AI can engage with the environment and other agents meaningfully.

Table 3.1 Agent Actions Table

	Player Agent	**Enemy Agent**
Actions	Walk and run in all directions Jump Attack (Melee) Crouch Push object Press button/Interact	Walk and run in all direction Jump Attack player (Melee) Crouch

3.3.2.1 Player Agent

- **Walk and run in all directions:** Allows navigation of horizontal and vertical spaces.
- **Jump:** Essential for crossing gaps and reaching elevated platforms.
- **Attack (Melee):** Simulates combat mechanics where the player can engage with enemies or break objects.

- **Crouch:** Enables traversal through tight spaces or avoidance of overhead obstacles.
- **Push objects:** Allows interaction with movable objects, such as boxes or switches.
- **Press button/Interact:** Adds versatility to the environment, enabling the player to trigger events or access locked areas.

3.3.2.2 Enemy Agent

- **Walk and run in all directions:** Supports patrol or chase behaviors.
- **Jump:** Ensures enemies can traverse complex terrains.
- **Attack (Melee):** Adds combat mechanics, making the environment more dynamic.
- **Crouch:** Enables traversal through tight spaces or avoidance of overhead obstacles.
- Reaching an endpoint or collecting specific items provides explicit objectives, simulating real-world gameplay scenarios.
- Goals encourage exploration, decision-making, and risk assessment in pursuit of rewards.

3.3.3 Environment Setup in Unity

The Unity game engine was selected for its versatility, comprehensive development tools, and compatibility with machine learning frameworks like Unity ML-Agents. The environment setup process involved the following steps:

3.3.3.1 Asset Integration

- **Textures and Sprites:** 2D assets were imported to create visually comprehensible platforms, obstacles, and characters, ensuring the game environment is both functional and aesthetically cohesive.
- **Physics Implementation:** Unity's built-in physics engine was utilized to handle gravity, collision detection, and movement mechanics, providing realistic interactions between game elements.

3.3.3.2 Level Design

Levels were crafted using Unity's tilemap system for platforms and prefabs for obstacles and NPCs. This modular design approach ensures flexibility, allowing for efficient testing and rapid iteration.

3.3.3.3 AI Training Environment

- Levels were optimized to function as AI training grounds, balancing accessibility and challenge.
- Unity ML-Agents toolkit was integrated to simulate agent behavior, facilitate training, and monitor performance metrics.
- Metrics were collected to assess the environment's complexity, ensuring it provided sufficient variety to support effective machine learning.

3.3.4 Environment Testing

Thorough testing was conducted to verify the suitability of the designed levels for AI training, with the process divided into the following stages:

3.3.4.1 Initial Testing

Basic AI agents were introduced to evaluate fundamental navigability and identify any immediate design issues.

3.3.4.2 Complexity Evaluation

Environments were iteratively refined to achieve a balance between challenge and learning potential, ensuring that the AI faced progressively difficult scenarios to encourage adaptation and improvement.

3.3.4.3 Performance Metrics Analysis

- Key metrics, such as training success rates, agent decision accuracy, and failure cases, were collected and analyzed.
- Insights from these metrics informed further adjustments to the level design, optimizing the environment for efficient and adaptable AI training.

This testing process ensured that the game environment aligned with the project's objective of creating robust and reusable AI assets for game development.

3.4 AI Algorithms and Models

In this section, we discuss the AI algorithms and models used to train the agents in the game environment. The core objective is to allow agents, both the Player and Enemy, to learn optimal strategies and adapt their actions based on environmental observations. For this project, we are considering two popular reinforcement learning algorithms: Deep Q-Networks (DQN) and Proximal Policy Optimization (PPO).

3.4.1 Rewards and Penalties

In reinforcement learning, agents learn through interactions with the environment, receiving feedback in the form of rewards or penalties based on the actions they take. The rewards and penalties drive the agent's behavior, encouraging actions that lead to positive outcomes while discouraging those that lead to negative consequences.

3.4.1.1 Player Agent Rewards and Penalties

The Player Agent is primarily focused on completing levels and maximizing performance. The rewards and penalties assigned to the Player Agent are designed to incentivize behaviors that contribute to level progression, combat efficiency, and overall success.

Table 3.2 Player Agent Reward Table

Action	Rewards
Level Completion	+1000
Enemy Defeated	+100
Collecting Objective Items	+50
Interact with Objectives	+20
Exploring	+1 per new point reached
Falling or Hitting Hazards	-50
Taking Damage from Enemy	-200
Eliminated	-1000

3.4.1.2 Enemy Agent Rewards and Penalties

The Enemy Agents are designed to create challenges for the Player Agent. Their behavior is shaped by rewards and penalties that encourage actions which counter the Player Agent's progress.

Table 3.3 Enemy Agent Reward Table

Action	Rewards
Player Damaged	+100
Chasing Player	+5 for every period of time chasing player
Survival Time	+1 for every period of time survived
Falling or Hitting Hazards	-50
Eliminated	-200

3.4.2 Training Algorithms

Reinforcement learning algorithms serve as the backbone of the adaptive AI system, enabling agents to learn and optimize their behaviors in the game environment. For this project, we consider two popular algorithms: Deep Q-Networks (DQN) and Proximal Policy Optimization (PPO). Each algorithm is examined in terms of its strengths, weaknesses, and suitability for different aspects of the game environment.

3.4.2.1 Deep Q-Networks (DQN)

DQN is a value-based reinforcement learning algorithm that employs a neural network to approximate the Q-function, which predicts the future rewards of actions in a given state. It is particularly effective for environments with discrete action spaces.

Strengths

- **Efficiency in Discrete Action Spaces:** DQN excels in environments where the actions available to the agent are clearly defined and limited in number.
- **Simplicity and Effectiveness:** The algorithm is relatively straightforward to implement and has been proven effective in various applications, such as Atari game environments.
- **Experience Replay:** DQN employs a replay buffer to store past experiences, reducing correlations between training samples and improving stability.

Weakness

- **Limited in Continuous Action Spaces:** DQN struggles in environments requiring fine-grained control, such as precise movements or timing.
- **Instability in Complex Environments:** Training DQN in environments with dynamic or highly variable states can be challenging.

3.4.2.2 Proximal Policy Optimization (PPO)

PPO is a policy-gradient-based algorithm that directly optimizes the policy by minimizing a surrogate loss function. It is versatile and suitable for both discrete and continuous action spaces.

Strengths

- **Stability and Efficiency:** PPO incorporates a clipping mechanism to prevent large updates to the policy, improving training stability.

- **Simplicity and Effectiveness:** The algorithm works well in both discrete and continuous environments, making it highly adaptable.
- **Experience Replay:** PPO ensures that policy changes remain within a trust region, avoiding drastic shifts that could destabilize learning.

Weakness

- **Higher Computational Cost:** PPO generally requires more computational resources than DQN due to its policy gradient approach and frequent model updates.
- **Hyperparameter Sensitivity:** Effective training with PPO depends heavily on fine-tuning hyperparameters such as the learning rate and clipping range.

3.4.2.3 Planned Experiments

To determine the most suitable algorithm for this project, both DQN and PPO will be implemented and evaluated in identical game environments. The following criteria will guide the comparison:

- **Training Speed:** The time required for each algorithm to reach a baseline performance level.
- **Performance:** The effectiveness of the learned policy, measured by the agents' success in completing levels and achieving objectives.
- **Scalability:** The ability of the algorithm to adapt to increasingly complex environments and dynamic scenarios.
- **Robustness:** The consistency of training results across multiple runs, accounting for random variations and initialization.

3.4.2.4 Anticipated Outcomes

DQN is expected to perform well in early-stage training, particularly in environments with fewer variables and well-defined states. However, its limitations may become apparent as the environment complexity increases. In contrast, PPO is anticipated to excel in handling the dynamic challenges of the platformer environment, thanks to its versatility and stability.

The insights gained from these experiments will guide the selection of the final algorithm for use in this project, ensuring optimal performance for the adaptive AI system.

Table 3.4 Comparison Between DQN and PPO

Aspect	**Deep Q-Networks (DQN)**	**Proximal Policy Optimization (PPO)**
Approach	Value-based (Q-function approximation)	Policy-gradient-based (direct policy optimization)
Action Space	Best suited for discrete action spaces	Works well for both discrete and continuous action spaces
Stability	May experience instability in complex environments	Incorporates clipping mechanisms for stable training
Learning Mechanism	Uses experience replay and target networks to stabilize learning	Smooth policy updates via surrogate loss and trust region optimization
Performance	Effective in simpler environments with clearly defined states	Excels in dynamic and complex environments
Scalability	Struggles with scalability in environments requiring precise control	Adapts well to environments with increasing complexity
Computational Cost	Relatively low, computationally efficient	Higher computational requirements
Hyperparameter Sensitivity	Moderate sensitivity	High sensitivity; requires careful tuning
Implementation Complexity	Straightforward to implement	More complex due to policy gradient calculations
Training Speed	Generally faster to converge in simple scenarios	May require longer training due to iterative updates
Robustness	Can exhibit variability across training runs	Consistent performance across multiple training iterations

3.5 Data Handling Processes

In this section, we describe the data handling processes used in conjunction with ML-Agents and PyTorch to facilitate the training of reinforcement learning (RL) agents. Effective data handling is critical for training efficient models, and it involves the collection, storage, preprocessing, and usage of data throughout the learning process.

3.5.1 Data Collection

During training, a variety of data is collected from the Unity environment. This data includes both agent-specific information and environment-level metrics. The primary data collected involves:

- **Agent Observations:** Information about the agent's state in the environment, including positions, velocities, and other relevant variables such as whether the agent is touching platforms or near obstacles.
- **Actions Taken:** The actions performed by the agents (e.g., moving, jumping, attacking), as selected by the RL algorithms.

- **Rewards:** Feedback provided based on agent actions, which helps to reinforce learning through positive or negative reinforcement.
- **Environment States:** Metrics related to the environment, including the current level, any active obstacles, and enemy positions.

Data is gathered at each step of the training loop and serves as the foundation for training the RL models. The Unity ML-Agents toolkit is used to capture this data during each simulation frame, and it is then sent to the Python trainer for processing using PyTorch.

3.5.2 Data Storage and Management

The data collected during training needs to be managed effectively to ensure its accessibility and optimal usage. In ML-Agents, data is typically stored in a structured format, such as:

- **PyTorch-Compatible Tensors:** Data collected from agents is formatted into PyTorch tensors, which are then used to update the RL models. This allows us to leverage PyTorch's GPU acceleration and other tools for efficient training.
- **Observation Buffers:** Each agent's observations are stored in buffers to ensure that temporal information is retained. These buffers are used for training the models with methods like experience replay, which is crucial for off-policy algorithms like DQN.
- **Action Logs:** Actions taken by agents are logged and paired with the corresponding rewards and observations. This data is vital for understanding the relationship between agent behavior and the environment's response.

Additionally, as training proceeds, large amounts of data are generated. To handle this, periodic checkpoints are taken, where the agent's state, observations, and progress are saved to disk. This allows for efficient memory management and the ability to resume training from previous states.

3.5.3 Data Preprocessing and Normalization

Before feeding data into the RL models, preprocessing steps are essential for improving the efficiency and stability of the training process. These include:

- **Observation Normalization:** Observations are normalized to ensure that all input features are on the same scale, improving the stability and convergence of the RL algorithms. PyTorch provides easy-to-implement methods for scaling and normalization.
- **Reward Scaling:** Rewards are often scaled to ensure that they are within a manageable range. This is particularly important for preventing gradient explosion or vanishing problems in the learning process.
- **Action Space Representation:** Actions are converted into a representation suitable for RL models. For example, discrete actions (e.g., move left, jump, attack) are encoded into numerical values.

These preprocessing steps help in aligning the input data with the requirements of the reinforcement learning models and ensure consistent training progress.

3.5.4 Data Usage for Training

The data collected, preprocessed, and stored in previous steps is then used for the actual training of the RL agents. ML-Agents integrates seamlessly with PyTorch and provides an interface to train reinforcement learning models. The data is used to:

- **Train the Model:** The preprocessed data is fed into the RL algorithms, where agents learn optimal policies through trial and error. The collected observations and rewards guide the agents toward making better decisions in future episodes.
- **Monitor Training Progress:** Training metrics such as reward over time, agent success rates, and failure cases are monitored throughout the learning process. This data provides insights into the agent's performance and helps to adjust parameters or environments for better learning outcomes.
- **Evaluate the Model:** Once the model is trained, it is evaluated using new episodes to test its generalization and ability to perform under unseen conditions. The evaluation data is analyzed to determine whether the agent has achieved the desired performance metrics.

By integrating ML-Agents with PyTorch in the data pipeline, we ensure that the collected data directly feeds into the learning process, enabling the creation of adaptive, intelligent agents.

3.6 Evaluation Metrics

In this section, we describe the evaluation metrics used to assess the performance of the reinforcement learning (RL) agents in the developed environment. The evaluation process is crucial to determine whether the agents have learned the desired behaviors and can perform tasks effectively in the game environment.

3.6.1 Quantitative Metrics

To objectively measure the performance of RL agents, we focus on several quantitative metrics that capture different aspects of the agent's learning progress and behavior. These include:

- **Cumulative Reward:** The total sum of rewards accumulated by the agent over an episode or a series of episodes. This metric is commonly used to assess how well the agent is achieving its goals, with higher cumulative rewards indicating better performance.
- **Episode Length:** The number of time steps an agent survives or performs actions within an episode before either completing the task or reaching the maximum allowed steps. Shorter episode lengths may indicate inefficient behavior, while longer episode lengths can suggest that the agent is learning to navigate its environment effectively.
- **Success Rate:** The percentage of episodes where the agent successfully achieves the predefined goal or objective (e.g., completing a level, reaching a target). This metric is particularly important for tasks with clear goals.
- **Average Return per Episode:** The average reward per episode over a number of episodes. This metric provides insight into the consistency of the agent's performance and can be useful in tracking improvement over time.
- **Action Efficiency:** The ratio of optimal actions (actions that contribute positively to achieving the task) to total actions taken. A higher action efficiency suggests that the agent is learning to make decisions more effectively.

- **Learning Speed:** The rate at which the agent improves its performance, typically measured by the rate of increase in cumulative reward over episodes or the reduction in the number of episodes required to achieve a target reward.

These metrics provide a comprehensive understanding of the agent's learning progress and performance in various aspects of the task.

Table 3.5 Quantitative Evaluation Metrics

Metric	Description	Purpose
Cumulative Reward	Total sum of rewards accumulated over an episode or series.	Measures the agent's overall success in achieving its goals.
Episode Length	Number of time steps the agent survives or performs actions.	Indicates task completion efficiency and potential learning speed.
Success Rate	Percentage of episodes where the agent achieves the goal.	Assesses the agent's ability to accomplish the task.
Average Return per Episode	Average reward per episode over multiple episodes.	Tracks consistency and performance over time.
Action Efficiency	Ratio of optimal actions taken vs. total actions performed.	Measures the agent's decision-making efficiency.
Learning Speed	Rate at which the agent's performance improves.	Evaluates how quickly the agent learns from experiences.

3.6.2 Qualitative Metrics

While quantitative metrics provide valuable insights into the agent's performance, qualitative analysis is also essential for assessing how well the agent performs in more complex, subjective aspects of the environment. These include:

- **Behavior Analysis:** A qualitative assessment of the agent's behavior in different situations. This involves analyzing whether the agent demonstrates intelligent decision-making, adapts to changes in the environment, and successfully interacts with game elements like enemies, platforms, and obstacles.
- **Policy Interpretability:** The ability to interpret the learned policy, i.e., understanding the agent's decision-making process. Although deep reinforcement learning models can sometimes be viewed as black boxes, examining how the agent behaves in specific situations can provide insights into whether it is following a reasonable strategy.
- **Generalization to New Scenarios:** Testing the agent's ability to generalize learned behaviors to new, unseen scenarios or levels. For example, evaluating whether the agent can still perform well after being exposed to different enemy configurations, environmental changes, or altered reward structures.
- **Player Feedback:** If applicable, gathering feedback from human players who observe or interact with the agent. This can provide subjective insights into how well the agent performs in terms of user experience, entertainment, or realism.

Qualitative metrics provide valuable information that complements the quantitative analysis, especially for tasks that involve complex decision-making or human-like behavior.

Table 3.6 Qualitative Evaluation Metrics

Metric	Description	Purpose
Behavior Analysis	Observes the agent's behavior in different scenarios.	Provides insight into the agent's decision-making and adaptability.
Policy Interpretability	Ability to understand the agent's decision-making process.	Assesses how easy it is to interpret the learned policy.
Generalization to New Scenarios	Ability to perform in unseen environments or with changed conditions.	Measures the agent's adaptability to new situations.
Player Feedback	Subjective evaluations from human testers or players.	Assesses how the agent's behavior aligns with user expectations and experience.

3.6.3 Evaluation Process

The evaluation of the RL agents will take place through a series of tests conducted under controlled conditions. These tests aim to evaluate both the generalization of the learned policies and the robustness of the agent under varying conditions. The following procedures are used for evaluation:

- **Training Evaluation:** During training, periodic evaluations are performed to track the agent's progress. This includes monitoring cumulative reward, episode length, and other relevant metrics over time. These evaluations help to identify any issues such as underfitting, overfitting, or slow convergence.
- **Cross-Scenario Evaluation:** The agent is tested in different game environments, each with varying levels of difficulty or changes in environmental conditions. This helps assess the agent's ability to adapt and generalize to unseen situations.
- **A/B Testing:** To compare different RL algorithms (e.g., PPO vs. DQN), A/B testing is conducted. The agents trained with different algorithms are subjected to identical testing conditions, and their performance metrics are compared to determine which algorithm produces better results for the specific task.
- **Human Evaluation (if applicable):** If the agent's behavior is part of a player-facing experience, user studies or feedback from human testers are used to assess the agent's effectiveness in achieving the desired user experience, such as its intelligence, entertainment value, or realism.

The combination of these evaluation techniques ensures a thorough analysis of the agent's performance across various dimensions.

3.6.4 Metric Interpretation and Decision Making

Once the evaluation metrics are gathered, they are analyzed to determine if the agent has met the project's goals and whether further iterations or improvements are necessary. The analysis involves:

- Comparing performance across different training algorithms to select the most effective method.

- Identifying areas for improvement, such as behavior inconsistencies, failure to generalize, or inefficiencies in task completion.
- Making adjustments to the training process, such as modifying hyperparameters, changing the environment setup, or revisiting the reward structure to enhance learning outcomes.

The evaluation results guide future development steps, ensuring that the RL agents continue to improve and adapt to the task requirements.

CHAPTER 4 IMPLEMENTATION RESULTS

You can title this chapter as **Preliminary Results** or **Work Progress** for the progress reports. Present implementation or experimental results here and discuss them.

ALL SECTIONS IN THIS CHAPTER ARE OPTIONAL. PLEASE CONSULT YOU ADVISOR AND DESIGN YOUR OWN SECTION

หัวข้อต่าง ๆ ในแต่ละบทเป็นเพียงตัวอย่างเท่านั้น หัวข้อที่จะใส่ในแต่ละบทขึ้นอยู่กับโปรเจคของนักศึกษาและอาจารย์ที่ปรึกษา

CHAPTER 5 CONCLUSIONS

Figure 5.1 This is how you mention when figure come from internet https://www.google.com

This chapter is optional for proposal and progress reports but is required for the final report.
THIS IS AN EXAMPLE. ALL SECTIONS BELOW ARE OPTIONAL. PLEASE CONSULT YOU ADVISOR AND DESIGN YOUR OWN SECTION
หัวข้อต่าง ๆ ในแต่ละบทเป็นเพียงตัวอย่างเท่านั้น หัวข้อที่จะใส่ในแต่ละบทขึ้นอยู่กับโปรเจคของนักศึกษาและอาจารย์ที่ปรึกษา

5.1 Problems and Solutions

State your problems and how you fixed them.

5.2 Future Works

What could be done in the future to make your projects better.

REFERENCES

1. I. Norros, 1995, "On the use of Fractional Brownian Motion in the Theory of Connectionless Networks," **IEEE J. Select. Areas Commun.**, vol. 13, no. 6, pp. 953–962, Aug. 1995.

2. H.S. Kim and N.B. Shroff, 2001, "Loss Probability Calculations and Asymptotic Analysis for Finite Buffer Multiplexers," **IEEE/ACM Trans. Networking**, vol. 9, no. 6, pp. 755–768, Dec. 2001.

3. D.Y. Eun and N.B. Shroff, 2001, "A Measurement-Analytic Framework for QoS Estimation Based on the Dominant Time Scale," in **Proc. IEEE INFOCOM'01**, Anchorage, AK, Apr. 2001.

APPENDIX A

FIRST APPENDIX TITLE

Put appropriate topic here

This is where you put hardware circuit diagrams, detailed experimental data in tables or source codes, etc..

Figure A.1 This is the figure x11 https://www.google.com

This appendix describes two static allocation methods for fGn (or fBm) traffic. Here, λ and C are respectively the traffic arrival rate and the service rate per dimensionless time step. Their unit are converted to a physical time unit by multiplying the step size Δ. For a fBm self-similar traffic source, Norros [1] provides its EB as

$$C = \lambda + (\kappa(H)\sqrt{-2\ln\epsilon})^{1/H} a^{1/(2H)} x^{-(1-H)/H} \lambda^{1/(2H)} \tag{A.1}$$

where $\kappa(H) = H^H(1-H)^{(1-H)}$. Simplicity in the calculation is the attractive feature of (A.1).
The MVA technique developed in [2] so far provides the most accurate estimation of the loss probability compared to previous bandwidth allocation techniques according to simulation results. Consider a discrete-time queueing system with constant service rate C and input process λ_n with $\mathbb{E}\{\lambda_n\} = \lambda$ and $\mathrm{Var}\{\lambda_n\} = \sigma^2$. Define $X_n \equiv \sum_{k=1}^{n} \lambda_k - Cn$. The loss probability due to the MVA approach is given by

$$\varepsilon \approx \alpha e^{-m_x/2} \tag{A.2}$$

where

$$m_x = \min_{n \geq 0} \frac{((C-\lambda)n + B)^2}{\mathrm{Var}\{X_n\}} = \frac{((C-\lambda)n^* + B)^2}{\mathrm{Var}\{X_{n^*}\}} \tag{A.3}$$

and

$$\alpha = \frac{1}{\lambda\sqrt{2\pi\sigma^2}} \exp\left(\frac{(C-\lambda)^2}{2\sigma^2}\right) \int_C^\infty (r-C) \exp\left(\frac{(r-\lambda)^2}{2\sigma^2}\right) dr \tag{A.4}$$

For a given ε, we numerically solve for C that satisfies (A.2). Any search algorithm can be used to do the task. Here, the bisection method is used.
Next, we show how $\mathrm{Var}\{X_n\}$ can be determined. Let $C_\lambda(l)$ be the autocovariance function of λ_n. The MVA technique basically approximates the input process λ_n with a Gaussian process, which allows $\mathrm{Var}\{X_n\}$ to be represented by the autocovariance function. In particular, the variance of X_n can be expressed in terms of $C_\lambda(l)$ as

$$\mathrm{Var}\{X_n\} = nC_\lambda(0) + 2\sum_{l=1}^{n-1}(n-l)C_\lambda(l) \tag{A.5}$$

Therefore, $C_\lambda(l)$ must be known in the MVA technique, either by assuming specific traffic models or by off-line analysis in case of traces. In most practical situations, $C_\lambda(l)$ will not be known in advance, and an on-line measurement algorithm developed in [3] is required to jointly determine both n^* and m_x. For fGn traffic, $\mathrm{Var}\{X_n\}$ is equal to $\sigma^2 n^{2H}$, where $\sigma^2 = \mathrm{Var}\{\lambda_n\}$, and we can find the n^* that minimizes (A.3) directly. Although λ can be easily measured, it is not the case for σ^2 and H. Consequently, the MVA technique suffers from the need of prior knowledge traffic parameters.

APPENDIX B

SECOND APPENDIX TITLE

Put appropriate topic here

Figure B.1 This is the figure x11 https://www.google.com

Next, we show how $\mathrm{Var}\{X_n\}$ can be determined. Let $C_\lambda(l)$ be the autocovariance function of λ_n. The MVA technique basically approximates the input process λ_n with a Gaussian process, which allows $\mathrm{Var}\{X_n\}$ to be represented by the autocovariance function. In particular, the variance of X_n can be expressed in terms of $C_\lambda(l)$ as

$$\mathrm{Var}\{X_n\} = nC_\lambda(0) + 2\sum_{l=1}^{n-1}(n-l)C_\lambda(l) \tag{B.1}$$

Add more topic as you need

Therefore, $C_\lambda(l)$ must be known in the MVA technique, either by assuming specific traffic models or by off-line analysis in case of traces. In most practical situations, $C_\lambda(l)$ will not be known in advance, and an on-line measurement algorithm developed in [3] is required to jointly determine both n^* and m_x. For fGn traffic, $\mathrm{Var}\{X_n\}$ is equal to $\sigma^2 n^{2H}$, where $\sigma^2 = \mathrm{Var}\{\lambda_n\}$, and we can find the n^* that minimizes (A.3) directly. Although λ can be easily measured, it is not the case for σ^2 and H. Consequently, the MVA technique suffers from the need of prior knowledge traffic parameters.