Final Project - Initial Report

Pusan National University

Computer Science and Engineering

2025-1

RLBoss

Team: BYTE

Nafikova Sofi 202270109

Oripov Mirshod 202255572

Karimova Savrelina 202155630

지도교수: 손준영

I. Project goals

The primary goal of this project is to develop an adaptive AI boss system for a game environment that dynamically adjusts its behavior and strategy based on player actions, using reinforcement learning. This system aims to improve player engagement through intelligent, real-time difficulty adjustment while trying to maintain balance between unpredictability and fairness.

Specifics:

a. Implement Reinforcement Learning Algorithms.
   Integrate algorithms such as Proximal Policy Optimization (PPO) and Deep Q-Networks (DQN) within the Unity environment to enable the AI boss to learn from player interactions and adapt its strategy accordingly.
b. Utilize Long Short-Term Memory (LSTM) Networks
   Incorporate LSTM to enable the AI to analyze sequential player behavior and detect long-term patterns for more sophisticated adaptive responses.
c. Integrate Monte Carlo Tree Search (MCTS)
   Use MCTS to support real-time decision-making in combat scenarios, allowing the boss to plan multi-step strategies and introduce unpredictability in gameplay.
d. Enable Real-Time and Historical Data Analysis
   Design the system to process both live gameplay data and past session data to inform and fine-tune the AI's behavior dynamically.
e. Achieve Dynamic Difficulty Adjustment (DDA)
   Ensure the boss can scale its difficulty based on the player's skill level, learning curve, and combat behavior to maintain an engaging and challenging experience.
f. Implement input reading.
   Bosses react to specific player actions-such as healing or casting spells-by responding aggressively, a technique often called "input reading
g. Deploying in Unity the ML-Agents Toolkit
   Develop and train the AI boss using Unity's ML-Agents Toolkit, creating a fully interactive prototype that demonstrates the adaptive system in action.
h. Develop an engaging game in Unity
   Using Unity engine, chosen for its good integration with machine learning frameworks. The goal is to create a fighting style game, the player plays against a boss and tries to kill it through different attack types

II. Target Problem and Requirements Analysis

1. Target Problem

Gaming often has enemies and NPCs that are rigid, have predictable behaviors predefined by rules and static scripts. These bosses can be very challenging by implementing techniques like input reading, behavior trees, and finite state machines allowing bosses to make different actions, but it still can result in repetitive encounters and do not adapt over time to the player which may lead to lost excitement and difficulty. This lack of adaptability can reduce player engagement, especially skilled players who quickly learn boss's patterns and exploit them.

The problem this project aims to solve is the lack of dynamic, adaptive enemy behavior in usual boss fight systems. The goal is to develop boss AI that can analyze player behavior through real time and analyzing past fights, and adjust its strategy accordingly, providing a more personalized, unpredictable, and challenging gameplay experience.

This is particularly relevant in game design, where player retention and immersion are critical. By applying reinforcement learning, this project investigates whether AI can learn and adapt within a gaming context in real-time, thus improving gameplay balance.

2. Requirements Analysis

Functional requirements

Player control system	The player must be able to move freely in the arena space using directional input - WASD or arrow keys for up, down, left, right movement - Collision detection with the arena bounds must prevent the player from exiting the play area - Health & Damage Response, players must lose health and receive visual/auditory feedback. If player health reaches zero, a game over condition is triggered.
Player actions system	Players must be able to: - Attack, attack must trigger a hitbox that can register damage to the boss - Dodge / Evade, execute a dodge roll or quick step move to evade boss attacks - Heal and change gear

	- Have an inventory
AI Boss behavior	Similar to the player system, the boss must be able to perform attacks, evade the player, and change its strategy based on observed player actions. - Basic attacks, Melee swipes, punches, or close-range attacks. Includes attack wind-up and cooldown phases to allow counterplay - Ranged Attacks, projectiles, energy blasts, or area-of-effect spells. Triggered when the player maintains distance or uses ranged attacks frequently - Evasion and Defense, Dodge-rolls, shield blocks, or phase-shifts to avoid incoming damage.
AI boss Behavior Selection System - Reinforcement Learning Integration	- Implemented using Unity ML-Agents inference mode. - The RL agent (trained via PPO or DQN) takes as input: Player position and velocity, Recent actions, Boss's current health and cooldown timers, Relative distance to the player - LSTM layer: enables short-term memory to track sequences and avoid repetitive behavior. - Outputs a decision vector representing the next action to perform.
Adaptive Difficulty	Dynamic Difficulty Adjustment (DDA) - The RL agent uses player performance metrics to assign a "difficulty score" - Based on this score, the boss dynamically adjusts parameters like: Attack frequency and combo depth, Movement speed and dodge chance, Aggression level, Timing windows, etc. - The AI alters move frequency, damage, or combos based on difficulty zone (e.g., beginner, intermediate, expert)
Data Collection and Analysis	The system must continuously log and analyze player actions. - Tracked Data Points: Hit Rate, Evasion Success, Damage Taken, Combat Style, Time per Phase/Battle, etc. - Long-term logs are stored as JSON or sent to a analytics database for retraining - Analysis modules compute trends - Player input data analysis: keyboard combinations

Training and Inference Pipeline	A complete ML workflow must be in place: one for training agents in simulation, and one for using trained models inside the actual game environment. - multiple simulated players → exhibit varied behaviors → Reward functions → Agents are trained with PPO or DQN - In the Unity game, the model is loaded by an inference brain → During real gameplay, the model receives observations → The model outputs action decisions → A fallback state machine is provided in case of model failure or corruption
Game Loop and Win/Loss Conditions	The game must define clear victory/defeat states, provide health UI, and offer feedback throughout the match. Start Phase → Combat Phase (Player and boss can attack, dodge, and use abilities) → victory Condition(Player wins when boss health reaches 0) || Defeat Condition (Player loses if health reaches 0 or a timeout occurs) → Post-Match (Shows battle summary)

Non-Functional Requirements

Performance	The game must run at a nice frame rate and inference must maintain fluidity
Scalability	RL agents should be retainable with new player behavior data
Robustness	Fallback behavior ensures gameplay continuity if ML interference fails
Reusability	The trained AI components and Unity environment should be reusable for other enemy types or similar gameplay scenarios
Modularity	The architecture should separate game logic, AI logic, and data logging to ensure maintainability

III. Analysis of Real-World Constraints and Countermeasures

a. Training Time and Computational Resources

Constraint:

RL requires intensive computing power and long training time to produce meaningful behaviors. This can become a problem, especially when iterating frequently during development.

Countermeasures:

- Use Unity ML agents in parallel environments to simulate multiple instances at once and leverage cloud-based GPUs
- Start training with simplified reward functions and state spaces, and progressively add complexity
- Leverage pre-trained models or curriculum learning to speed up convergence

b. Unpredictable Player Behavior

Constraint:

Players can adopt highly variable or unconventional playstyles that the boss may not have seen during training, leading to poor or non-reactive AI responses

Countermeasures:

- Implement a hybrid control system: fall back to a finite state machine (FSM) or rule-based behavior if the model produces erratic outputs

c. Balancing Challenge and Fairness

Constraint:

Reinforcement learning agents may exploit loopholes in reward functions and develop overpowered or unfair strategies.

Countermeasures:

- Define strict boundaries in the action space (e.g., cooldowns, stamina costs)
- Regularly observe agent behavior during training to spot and fix reward shaping issues early

d. Player Experience and Usability

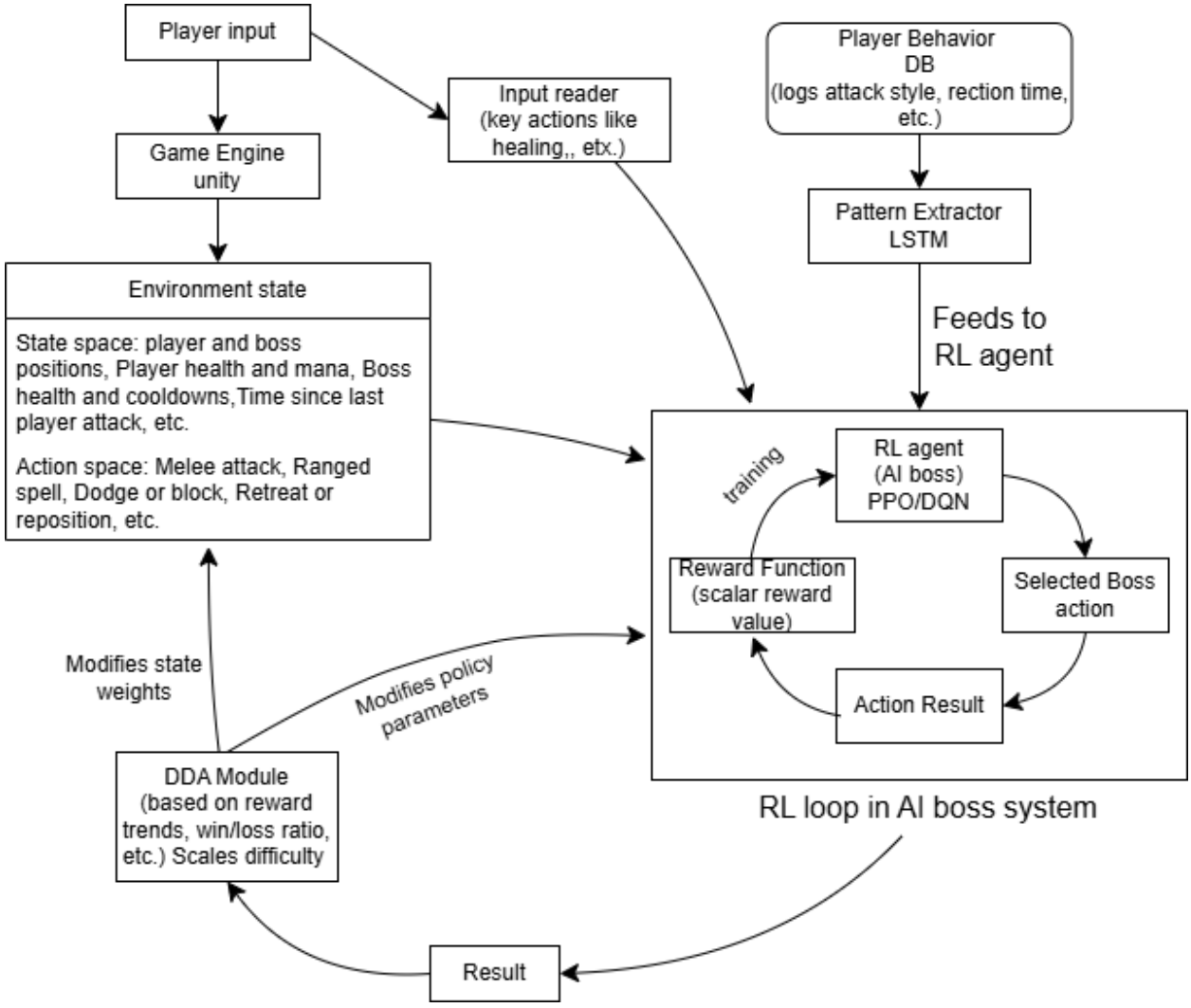
Constraint:

A technically advanced AI may still fail if the gameplay experience is confusing, overly punishing, or lacks proper feedback for the player

Countermeasures:

- Include adaptive tutorials or in-game tips that respond to the player's struggle

IV. Design Documentation



Game Engine	Interface between player input and environment
Environment State	Tracks real-time data
Input Reader	Detects key player actions
Player Behavior DB & Pattern Extractor	Analyzes past player behavior
RL Agent	Learns optimal actions using PPO/DQN
Reward Function	How behavior is evaluated
DDA Module	Adjusts difficulty dynamically

V. Timeline

Phase	Timeframe	Key Activities	Deliverables
1. Planning & Research	May 1 – May 16	- Finalize game concept and goals - Define scope and architecture - Research RL techniques	Initial Report + Advisor Confirmation
2. Prototype Development	May 17 – June 10	- Build Unity game loop prototype - Implement FSM fallback AI	Core game prototype with placeholder boss
3. AI Design & Training	June 11 – July 10	- Create training environment in Unity ML-Agents - Design reward functions - Train PPO & DQN	Trained RL model prototypes
4. Integration & Testing	July 11 – July 18	- Integrate trained model into Unity - Perform initial playtesting - Tweak inference logic	Midterm Report + Evaluation Form
5. Refinement & Expansion	July 19 – Aug 20	- Add LSTM for temporal memory - Add UI elements - Polish game feel	Expanded game build with smarter boss
6. Final Testing & QA	Aug 21 – Sept 10	- Conduct external playtesting - Analyze logs and adjust parameters - Fix bugs and optimize	Final game version (pre-presentation)
7. Final Report & Delivery	Sept 11 – Sept 19	- Write final report - Prepare evaluation forms - Record demo if needed	Final Report + Evaluation Form
8. Presentation Prep	Sept 20 – Oct 1	- Build slides, script and visuals	Final Presentation (Tech Week)
9. Deliverables Submission	Oct 2 – Oct 15	Package Unity project, source code	Final Deliverables Upload

VI. Team Member Role Allocation

Nafikova Sofi	RL development / game dev
Oripov Mirshod	Database and data management
Karimova Savrelina	Game design / game dev