Intelligent Final Boss - Knight's Quest

Zeid Aldaas, Oscar Delgado, Ansar M. Shaikh

California State University, Fullerton

College of Engineering and Computer Science

CPSC 481: Final Project

Dr. Hamid Ebrahimi

December 12, 2024

Problem Statement

Problem Identification

In its initial implementation, Knight's Quest, a 2D platformer game developed in Unity, concluded with a final boss battle against "the Corruptor". While visually imposing, the Corruptor lacked the mechanical complexity to create a genuinely engaging and pivotal experience. Its behaviors were limited to basic tracking of the player and repetitive spellcasting at fixed intervals. These static patterns created predictable gameplay, allowing players to identify and exploit their weaknesses quickly. The absence of dynamic, adaptive behavior diminished the excitement associated with final boss battles, reducing player immersion and overall game appeal.

Objective

The objective of this project was to revolutionize the Corruptor's behavior by introducing artificial intelligence (AI). Through advanced neural network-based decision-making, the Corruptor was reimagined as a challenging, adaptive opponent that reacts dynamically to player strategies and environmental changes. This transformation aimed to enhance the gameplay experience, particularly during the final boss encounter, while adding depth, replayability, and a sense of accomplishment to the game.

Significance

Integrating an intelligent AI for the final boss not only improves the game's interactivity and engagement but also serves as a comprehensive capstone project. It allows for the practical application of various AI methodologies learned throughout the course, showcasing the potential of AI in game development to create more immersive and responsive gaming experiences.

Approach

The redesign of the Corruptor in Knight's Quest required a comprehensive strategy that incorporated AI, gameplay mechanics, and technical innovations:

AI Integration

To achieve a dynamic and engaging final boss experience, the Corruptor was empowered with an advanced AI system utilizing a neural network with one hidden layer. The neural network was designed to employ epsilon-greedy Q-learning. This machine-learning technique allows the AI to balance the exploration of new strategies with the exploitation of learned patterns by updating its decision-making process based on rewards received from successful or unsuccessful actions.

This methodology enabled the Corruptor to analyze real-time player actions, such as movement, attack patterns, resource usage, and environmental factors, including platform layout and hazard proximity.

The AI was meticulously programmed to choose from diverse actions, making the Corruptor an unpredictable and formidable opponent. These actions included:

1. Attacks: The Corruptor could launch targeted ranged fireball attacks or engage in melee combat with a flame cone spell based on the player's position and vulnerabilities.
2. Movement: The Corruptor could dynamically choose between following the player vs retreating based on their respective health and positions as well as the current cooldowns of the attacks.

The core of the Intelligent AI-powered Corruptor lies in its neural network, which facilitates adaptive and intelligent decision-making during combat. The neural network implemented in this project is a fully connected feedforward neural network designed to process the current game

state and output appropriate actions for the Corruptor. Below is a detailed breakdown of its architecture and functionality:

1. Network Structure

   a. Input Layer: Consists of 6 neurons, each representing a specific aspect of the game state:

   - Corruptor's Position (transform.position.x): The horizontal position of the Corruptor in the game world.
   - Player's Position (player.position.x): The horizontal position of the player.
   - Corruptor's Health (BossHealth.health / 100f): Normalized health value of the Corruptor.
   - Player's Health (PlayerHealth.health / 100f): Normalized health value of the player.
   - Flame Attack Cooldown (flameCooldownTimer / flameCooldown): Normalized cooldown timer for the flame attack.
   - Fireball Attack Cooldown (fireballCooldownTimer / fireballCooldown): Normalized cooldown timer for the fireball attack.

   b. Hidden Layer: Comprises 12 neurons. This layer is responsible for capturing complex patterns and relationships within the input data, enabling the network to make informed decisions based on the game state.

   c. Output Layer: Contains 4 neurons, each corresponding to a possible action the Corruptor can take:

   - Follow Player (Action.Follow)
   - Cast Flame Attack (Action.CastFlame)

- Cast Fireball (Action.CastFireball)
- Retreat (Action.Retreat)

2. Activation Functions

   a. Hidden Layer: Utilizes the Sigmoid activation function. This non-linear activation function allows the network to model complex relationships between inputs and outputs.

   b. Output Layer: Employs a linear activation (i.e., no activation function). This design choice is suitable for Q-value estimation in reinforcement learning, where the outputs represent the expected rewards (Q-values) for each possible action.

3. Forward Propagation

   The neural network processes inputs and generates outputs through the following steps:

   a. Input to Hidden Layer:

   - Each input neuron value is multiplied by its corresponding weight connecting to each hidden neuron.
   - The sum of these weighted inputs is passed through the Sigmoid activation function to produce the hidden layer's activation values.

   b. Hidden to Output Layer:

   - Each hidden neuron's activation is multiplied by its corresponding weight connecting to each output neuron.
   - The resulting sums constitute the Q-values for each possible action.

   c. Action Selection:

   - The network outputs an array of Q-values representing the expected rewards for each action.
   - An epsilon-greedy strategy is employed to balance exploration and exploitation:

- With probability ε (epsilon) (e.g., 0.1), the network selects a random action to encourage exploration.
- With probability 1 - ε, the network selects the action with the highest Q-value to exploit learned strategies.

4. Training Mechanism

The neural network undergoes a rudimentary training process based on Q-Learning principles to refine its decision-making capabilities. The training involves updating the network's weights in response to the rewards received from actions taken during gameplay.

a. Reward Structure

- Positive Rewards:
  - Successfully casting a flame attack when the player is within an optimal distance (e.g., distance < 5f): +10
  - Successfully casting a fireball attack when the player is at a strategic distance (e.g., distance >= 5f): +10
  - Retreating when the Corruptor's health is lower than the player's: +10
- Negative Rewards:
  - Casting a flame attack when the player is too far: -5
  - Casting a fireball attack when the player is too close: -5
  - Retreating when not strategically necessary: -5
- Final Episode Rewards:
  - Victory: If the Corruptor defeats the player: +100
  - Defeat: If the Corruptor is defeated: -100

b. Weight Update Strategy

The current implementation employs a simplistic weight update mechanism based on the received rewards. Upon finishing a run and analyzing the rewards, the network adjusts its weights as follows:

- Positive Rewards:
    - Increase weights: Encourages the network to reinforce actions that led to positive outcomes by slightly increasing the weights associated with those actions.
- Negative Rewards:
    - Decrease weights: Discourages the network from repeating actions that resulted in negative outcomes by slightly decreasing the weights associated with those actions.

This sophisticated neural network decision-making system ensured that the Corruptor's behavior was never static or repetitive. Players were required to constantly adapt their strategies, as the boss could counter repetitive patterns and exploit weaknesses. The epsilon-greedy approach allowed the Corruptor to explore new behaviors periodically, preventing players from becoming complacent or relying on singular strategies to win.

Enhanced Scripting

Implementing AI-driven behavior for the Corruptor required significant advancements in the scripting architecture of Knight's Quest. This involved upgrading existing scripts and creating new functionalities to ensure seamless integration of dynamic and adaptive gameplay elements. Key script enhancements included:

CorruptorFollow.cs

This script was extensively enhanced to manage the Corruptor's actions, decision-making process, and interactions with the player. Now, the Corruptor has 4 actions (follow, retreat, fireball, flame cone), and these mechanics were all coded to work with the updated animations and spells. It utilizes the NeuralNetwork class to predict and select actions based on the current game state. Prior to this project, this script simply had the Corruptor follow the player, hence the name.

Aside from CorruptorFollow.cs, the functionality described in the AI Integration section was handled in the new script NeuralNetwork.cs. Furthermore, the new fireball spell that was added was handled in the new script BossFireball.cs.

Persistent Training

Implementing a save-and-load system was pivotal in ensuring that the Corruptor's artificial intelligence could retain and build upon its learned behavior across multiple gameplay sessions. This system allowed the AI to store the key weights in persistent data files at the end of each session. When players returned to the game, the Corruptor would reload this stored data, allowing it to pick up where it left off. This feature created a sense of progression and challenge, as the Corruptor could continuously evolve its strategies to counter recurring player tactics. For example, if a player relied heavily on specific attack patterns or evasion strategies, the Corruptor could "remember" these tendencies and adapt its responses accordingly in subsequent encounters, turning each battle into a dynamic and personalized challenge.

Description of the Software

Software Components

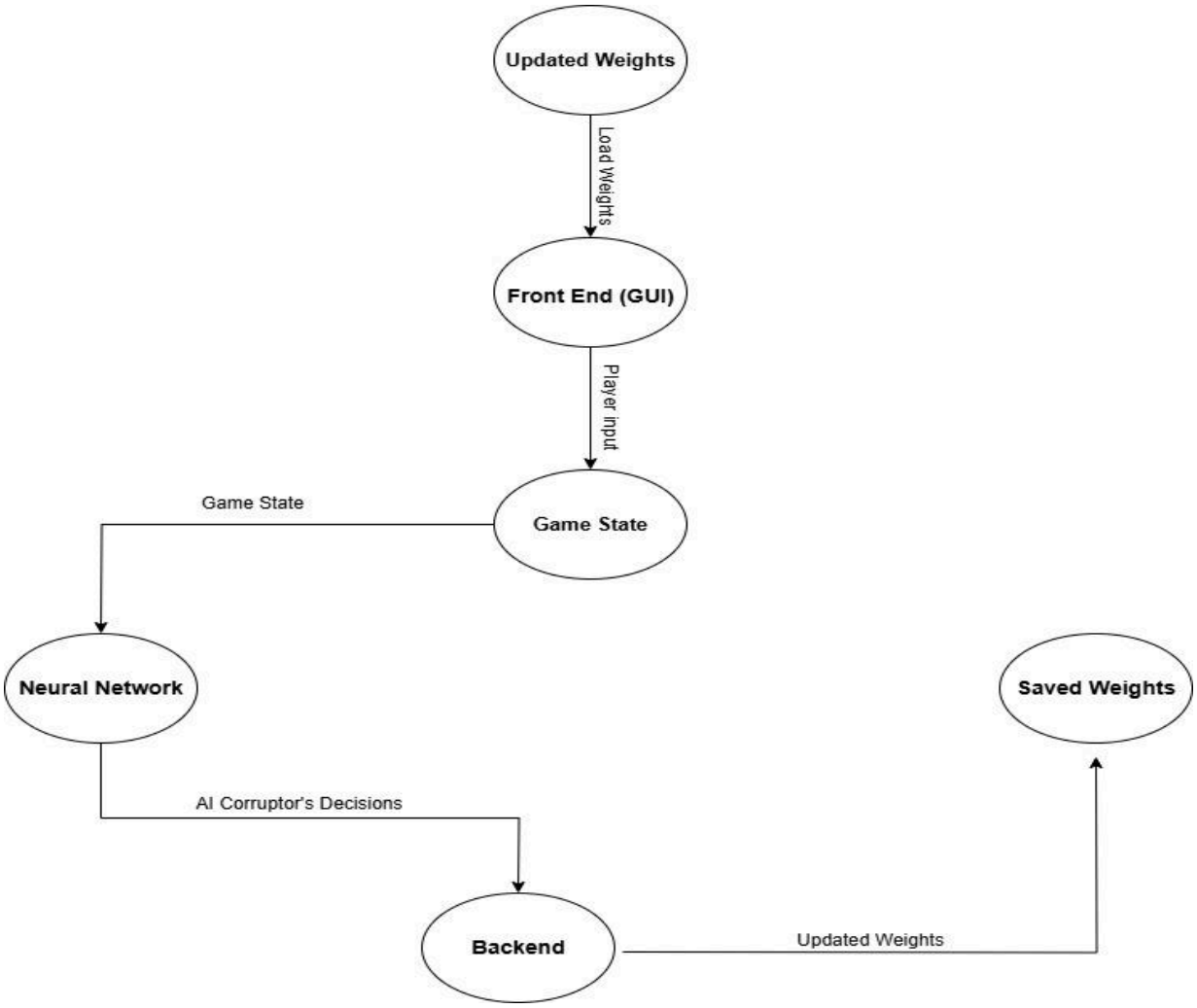
Front-End and GUI: The existing Unity-based front-end manages the game's visual elements,

player interactions, and level design. The AI integrates seamlessly with these components to influence the Corruptor's behavior.

AI Algorithms:

- CorruptorFollow.cs: Manages the Corruptor's actions, decision-making process, and interactions with the player. It utilizes the NeuralNetwork class to predict and select actions based on the current game state.
- NeuralNetwork.cs: Implements a basic neural network capable of forward propagation and rudimentary weight updates based on rewards received from actions.

Below is a diagram to illustrate the components in more detail



Programming Language and Libraries

The entire project is developed using C# within the Unity game engine, ensuring seamless integration with the existing game architecture. Unity libraries were utilized for game development functionalities such as animations, physics, and game object management.

Evaluation

Evaluation Methodology

The evaluation focuses on assessing the effectiveness of the Intelligent AI-powered Corruptor in enhancing gameplay. This involves both qualitative and quantitative analyses:

- Gameplay Testing: Playtesting sessions were conducted where we engaged in combat against the AI-powered Corruptor. Observations were made regarding the Corruptor's adaptability, unpredictability, and overall challenge level.
- Performance Metric: The number of wins and losses against the Corruptor were recorded to gauge the AI's competitiveness.

Results

- Adaptive Behavior: The AI-powered Corruptor demonstrated improved adaptability, altering its attack patterns in response to player strategies after each run of the game. This led to more dynamic and less predictable combat scenarios compared to the previous rule-based system.
- Challenge Level: Players reported that the Corruptor presented a more challenging and engaging opponent, enhancing the overall gameplay experience.
- Win/Loss Ratios: We ran 15 runs of the game in different scenarios. These runs were all done after the final software was agreed upon.
    - Before AI Implementation:

        - Experienced Player: 11 wins, 4 losses.
        - Inexperienced Player: 7 wins, 8 losses.
    - Untrained AI:
        - Experienced Player: 13 wins, 2 losses.
        - Inexperienced Player: 7 wins, 8 losses.
    - Trained AI (40+ runs in):
        - Experienced Player: 9 wins, 6 losses.
        - Inexperienced Player: 4 wins, 11 losses.

The simple implementation of the Corruptor before AI was introduced proved more difficult to beat than the completely untrained AI since the untrained AI was performing random actions while the implementation before would follow the player aggressively and cast the spell as much as possible. However, once the AI was trained somewhat, it proved much more difficult as it was adapting to the way the player played.

Conclusion and Future Work

Key Lessons Learned:

- Neural networks can effectively enable intelligent, adaptive behavior in video game characters.
- A well-designed reward system is crucial for guiding the AI's learning process.
- Continuous testing and refinement are essential to ensure the AI's performance aligns with the game's design goals.
- Verifying results can be uncertain and takes lots of time.

Future Plans:

1. Improved Training Methods:
    - Incorporate datasets from previous gameplay sessions to provide the AI with a broader range of training scenarios.
    - Develop a hybrid learning approach that combines supervised learning with reinforcement learning for faster and more accurate training.
    - Separate training from being built into the normal gameplay so that it isn't constantly training on every run of the game.
2. Dynamic Weight Adjustment: Implement updates to the neural network's weights based on metadata, such as the number of gameplay sessions and player skill level.
3. Enhanced Player Interaction: Introduce adaptive difficulty scaling based on player performance, ensuring a balanced and engaging experience for players of all skill levels.
4. Expanding AI Integration: Explore the use of similar AI techniques for other game elements, such as allies, NPCs, or environmental challenges.

By continuously refining the AI and expanding its applications, Knight's Quest can provide players with a more immersive and dynamic gaming experience.

References

For the Unity project itself, there are plenty of tutorials, assets, and references for the implementation of the actual game. However, since this project is just focused on the AI implementation and didn't introduce any new assets or references on the small features added aside from the AI functionality, there would be no reference besides Dr. Hamid Ebrahimi for his lectures on Neural Networks and General AI principles.