Adaptive Enemy AI Using Q-Learning in Unreal Engine

CSE 4633

George Kotti, ghk24

04/10/2025

Introduction

This project explores the integration of Q-Learning, a model-free reinforcement learning algorithm, into an asynchronous enemy combat AI system built within Unreal Engine 5 (UE5) using C++. Rather than relying on traditional finite state machines or behavior trees to drive enemy actions, I implemented a dynamic and data-driven system through a custom enemy class, AQLearningEnemy. The goal was to develop a modular, adaptive AI that learns from experience during combat, making real-time decisions based on the game state—such as health, distance to the player, and player actions.

By encoding game state into discrete values and mapping them to possible combat actions (e.g., attack, dodge, guard, heal, wait), the system is capable of evaluating past outcomes and adjusting its strategy over time using a Q-table. This approach not only enhances enemy behavior realism but also allows for persistent learning across sessions, enabling AI agents to refine their decision-making with each encounter.

Background and Setup

The foundation of the project was built using UE5 and C++. To effectively evaluate and train the Q-Learning AI system, a complete gameplay loop had to be established. This included not only AI agents and player control, but also the core gameplay mechanics required for meaningful reinforcement learning feedback. These systems included combat interaction (melee, guard, dodge), health and attributes, animation montages, and a modular actor-component architecture.

The environment also needed to provide structured state transitions, visual feedback, and rewards for actions taken, all integrated into real-time UE5 systems such as animation blueprints, perception, and pathfinding. A functioning navmesh, controller logic, and animation notify bindings were crucial for synchronizing Q-learning state updates with visual execution.

Below is a breakdown of the essential C++ classes and associated Blueprint assets used in the core setup:

Category	Assets / Classes
Breakables	BreakableActor
Characters	BaseCharacter, CharacterTypes, KnightCharacter, KnightAnimInstance, SlashCharacter, SlashAnimInstance
Components	AttributeComponent
Enemies	Enemy, AQLearningEnemy
HUD	HealthBar, HealthBarComponent
Interfaces	HitInterface
Items	Item, Treasure, Weapon

(Though, much is done inside the editor itself, such as each's AnimBPs)

What is Q-Learning?

Q-Learning is a model-free reinforcement learning algorithm used to determine the optimal action-selection policy in a given environment. It relies on the concept of a Q-table, whichs maps state-action pairs to expected long-term rewards. The agents learns by interacting with the environment, taking actions, and updating the Q-values based on the observed rewards and next states using the Temporal Difference (TD) update rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma a' \max Q(s',a') - Q(s,a)]$$

To balance exploration and exploitation, the system uses an ε-greedy policy when selecting actions. This means that with probability ε (i.e. 0.25 or 25%), the enemy chooses a random action to explore new strategies. The remaining 1- ε of the time, it selects the action with the highest known Q-value for the current state. This approach prevents the agent from getting stuck in suboptimal behaviors and ensures that is has the change to discover more rewarding actions over time.

Implementation

The Q-Learning System is implemented in a modular, event-driven architecture to support Unreal Engine's asynchronous nature. The main class AQLearningEnemy contains a Q-table (TMap<FQState, TMap<EQAction, float>>) that maps combat states to possible actions and their associated rewards. States are represented using a custom FQState struct which includes useful features about the game's current environment.

The decision-making process is broken into two phases. UpdateFunction_Phase1() captures the current state, selects an action using the ε-greedy policy, and executes it via PerformAction(). Each action may take time to complete (e.g., playing an attack animation), so UpdateFunction_Phase2() is only triggered after the action has finished. At that point, rewards are gathered and applied using the learning rule in UpdateQValue(), updating the Q-table to reflect whether the action was beneficial.

Actions include: Attack, Guard, Dodge, Heal, and Wait. Each is linked to a reward system to reinforce smart decisions and discourage poor timing. Rewards are managed in a PendingRewards buffer and added during events such as landing a hit, being blocked, missing, or dying. The Q-table is serialized to disk using JSON so that agents can retain their learned strategies between game sessions. A shared Q-learning manager (AQLearningManager) optionally merges Q-tables from multiple enemies for collective learning.

Here is a list of the Q-Learning specific classes:

Category	Assets / Classes
Q-Learning	QLearningManager, QLearningTypes, QLearningEnemy
Referenced	BaseCharacter, KnightCharacter, Enemy, Weapon

Challenges

One of the biggest challenges was managing asynchronous timing between combat animations, state updates, and reward assignments. If rewards were applied too early—before an animation finished or before the enemy received feedback—the Q-table would be updated incorrectly, leading to ineffective learning. This required careful coordination using timers, animation notifies, and phase-separated updates to ensure accurate state transitions and learning outcomes.

Another challenge, unrelated to Q-Learning, was managing bugs and crashes within Unreal Engine. The debugging process for issues relating to nullptr values was a chaotic mess. I found that a system of logs to be most helpful along with turning off related functionality to narrow down the possible causes. An unset value in the editor itself, well that is another story! I recommend tedious amounts of logging and documentation for anyone using UE5.

Results

The results of this project demonstrated that Q-Learning could be successfully integrated into a real-time enemy AI system in UE5, allowing an agent to learn meaningful combat behavior over time. After several iterations of debugging and tuning, the AQLearningEnemy was able to explore various combat strategies and adapt based on reward feedback.

Initially, the enemy AI defaulted to specific actions, such as repeated attacks, due to an unbalanced reward system. However, after refining this along with adjusting the update loop timing to adhere to asynchronous events coming from multiple different classes (each on their own tick loops) the agent began to show more dynamic behavior. It started experimenting with defensive actions and only attacking when in range.

Through JSON serialization, the enemy's Q-table was saved between sessions. This persistence allowed the AI to retain learned behavior, resulting in faster adaptation when reloaded. Additionally, with the introduction of a AQLearningManager class, multiple Q-learning enemies could share their experiences by merging Q-tables upon death. This enabled collective learning and helped generate a more balanced behavior model when training multiple agents simultaneously.

Deliverables

- Lists of the current Rewards, State values, and Actions (also found in QLearningTypes.h)
- Source code for the current project
- Current Q-Tables
- Short readme and notes
- Presentation PowerPoint
- Demo videos
- Report