Mississippi State University

Department of Computer Science & Engineering

George Kotti IV

Intro to AI

Dr. Eric Hanson

May 06, 2025

Project Report:

Integrating Deep Q-Learning with Real-Time Combat AI in Unreal Engine

Project Overview

In this project, we explore the use of Deep Q-Networks (DQNs) in controlling the mechanics of an enemy combat AI inside an Unreal Engine 5 environment.

The goal was to integrate reinforcement learning into a real-time, game environment by allowing an enemy character to acquire combat tactics through repeated interaction and reward-based learning.

Unlike traditional behavior trees and finite-state machines, which rely on hard-coded, static rules, DQNs enable adaptive behavior learned over time through experience. We can further evolve this setup to include additional hyper parameters or neurons which focus on separate game settings such as difficulty, timing, and squad tactics.

The core components of this project include the UE5 environment and game mechanics, along with the external DQN server/agent Python application. These two systems communicate over a custom TCP-based network layer, allowing the Unreal Engine enemy to serialize its state and reward information and send it to the Python agent. The agent, in turn, processes the data, updates its policy via deep learning, and responds with an action to execute.

Throughout training, the AI agent learns to perform contextually appropriate behaviors—such as attacking when in range, guarding during enemy strikes, and healing when low on health. This setup not only showcases the feasibility of integrating deep reinforcement learning into a real-time game environment but also opens the door to more complex extensions, such as asynchronous multi-agent learning, adaptive difficulty tuning, and procedurally generated behavior sets based on learned experience.

Ultimately, this project demonstrates how modern machine learning techniques can be brought into game development pipelines to create more dynamic, intelligent, and responsive in-game agents.

What was Learned

After successfully integrating a Deep Q-Network in our UE5 environment, we learned a few key insights:

Understanding DQN Behavior and Parameters

The project showed firsthand experience with how DQN parameters – such as learning rate, discount factor, epsilon, batch size, and target network update frequency – directly affect training efficiency. For instance,

- **Learning rate** – ***Low***: learns very slowly; may not converge. ***High***: becomes unstable; overshoots or diverges

- **Discount factor** – ***Low***: Focuses only on immediate rewards. ***High***: becomes too farsighted, hard to converge.
- **Exploration rate** (**epsilon**) – ***Low***: Agent becomes too deterministic too early. ***High***: Agent explores too much, unpredictable, ignores learning.
- **Batch size** – ***Low***: Noisy updates; slower training. ***High***: smoother updates but slower responsiveness.
- **Replay buffer size** – ***Low***: Not enough diversity; prone to overfitting. ***High***: May use outdated/stale transitions.
- **Target update rate** – ***Low***: Target is too outdated; slow learning. ***High***: Target is too similar; destabilizes training.

Applications and Strengths of DQNs in Combat AI

Like the previous Q-learning project, the DQN was able to learn multiple patterns and strategies such as guarding attacks, healing at low health, and attacking within range or more aggressively when player is at low health. This demonstrated that DQNs can replace or complement finite-state machines and behavior trees with more flexible, data-driven behavior, especially for one-on-one combat encounters. By expanding the action space of the DQN to action sequences instead of individual actions, we can help to prevent the overfitting of single actions and create a more diverse experience for the player.

Limitations of DQNs in Games

DQNs are data-hungry and require many episodes to learn stable behaviors; along with performance limitations when training directly within a game engine – which can cause frame locking or delay, blocking sockets, and other bottlenecks without careful consideration and following safe, efficient programming practices.

Results

The DQN enemy AI was able to learn simple behaviors within a few dozen episodes, like guarding instead of waiting, healing when not at full health, and prioritizing attacking when within range. This expanded to some smarter action planning such as occasionally dodging an attack or blocking at the moment of attack, but these required perfect synchronization between the attack events and the DQN.

Challenges

Implementing a DQN-based system for real-time combat AI introduced several technical and design challenges, especially at the intersection of machine learning and game engine logic. As covered earlier, areas like reward engineering, training stability, model performance, and practical integration required careful tuning and iteration.

A particularly subtle but critical issue involved the synchronization between Unreal Engine's asynchronous combat system and the DQN's discrete update phases. In our architecture, UpdateFunction_Phase1() initiates the decision process by requesting an action from the DQN, while UpdateFunction_Phase2() sends a JSON-formatted transition packet (containing state, reward, and done flag) back to the Python training server. However, in-game actions such as dodging, attacking, or guarding are not immediate—they occur over time, and can be interrupted or modified by external stimuli, like being hit by the player.

The challenge arose when player actions occurred mid-animation or mid-decision cycle, resulting in temporal misalignment between the game state used to calculate rewards and the actual state experienced by the agent. For example, if the player's attack animation completed *before* the DQN sent a response back to UE, the reward applied to that state might no longer reflect the actual outcome (e.g., taking a hit and losing health). This could lead to the agent inaccurately assigning positive or negative rewards to the wrong state-action pair.

Interestingly, the system proved robust enough to tolerate these missed assignments, as the frequency of them was estimated to be at a low enough level to not cause major inconsistencies. Therefore, rather than further complicating the synchronization system by introducing event-locking or flagging systems, we accepted this limitation. Nonetheless, it remains a valuable consideration for future work and expansions.

Deliverables

The project submission includes:

- Project report (this document)
- Presentation
- Demo recordings
- Zipped project directory
    - Source code
    - Logs
    - Model .pt data files
    - .txt DQN rewards and types
    - Plots & data gathered

References

Amin, S. (2025, January 30). Deep Q-Learning (DQN) - Samina Amin - Medium. *Medium*. https://medium.com/@samina.amin/deep-q-learning-dqn-71c109586bae.

Rafatirad, S., Homayoun, H., Chen, Z., & Dinakarrao, S. M. P. (2022). Machine learning for computer scientists and data analysts. In *Springer eBooks*. https://doi.org/10.1007/978-3-030-96756-7.

Reinforcement Learning (DQN) Tutorial — PyTorch Tutorials 2.7.0+cu126 documentation. (n.d.). https://docs.pytorch.org/tutorials/intermediate/reinforcement_q_learning.html.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013, December 19). *Playing Atari with Deep Reinforcement Learning*. arXiv.org. https://arxiv.org/abs/1312.5602.

The Deep Q-Network (DQN) - Hugging face Deep RL course. (n.d.). https://huggingface.co/learn/deep-rl-course/en/unit3/deep-q-network.