

# UE5 Enemy Combat AI using Deep Q-Networks

By George Kotti - CSE 4633 - April 2025

# Introduction

- ▶ This project explores the use of Deep Q-Learning (DQN) to control enemy AI in a UE5-based combat system, using a custom Python implementation
- ▶ The Python program acts as the learning brain, communicating with the Unreal Engine game loop via sockets (127.0.0.1, 5555)
- ▶ Rather than rely on built-in behavior trees or blackboards, the AI learns from experience and adapts over time, based on rewards
- ▶ This project will build on the previous vanilla Q-learning combat AI implementation

# Project Goals

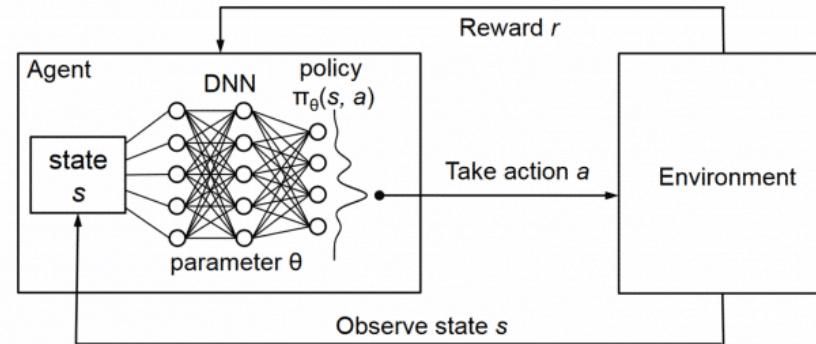
- ▶ Implement a custom, lightweight DQN agent outside UE5
- ▶ Connect it to a real-time UE5 combat environment via a custom socket interface
- ▶ Allow the AI to:
  - ▶ Learn which combat actions to use (i.e. attack, heal, dodge)
  - ▶ Optimize its behavior over episodes
  - ▶ Persist training data across sessions
- ▶ Use Python + PyTorch to handle learning independently of the game engine

# Background Information

- ▶ Deep Q-Networks (DQN)
  - ▶ Combines Q-Learning with a neural network to handle large or continuous state spaces.
  - ▶ Predicts the expected reward for each possible action in a given state
  - ▶ Uses:
    - ▶ Replay Buffer - to break correlation between sequential experiences(TD)/data samples, better sample efficiency
    - ▶ Target Network - to stabilize learning
    - ▶  $\epsilon$ -Greedy Policy - to balance exploration vs. exploitation
    - ▶ Bellman Equation - update rule
    - ▶ Q-Neural-Network - used to approximate the Q-value (outputs value for each action)
    - ▶ Mini-Batch Updates - samples X transitions per learning step (e.g. 32)

# Background Information

- ▶ DQN Training Loop:
  - ▶ Observe Environment (current state)
  - ▶ Select Action (using a policy, e.g.  $\epsilon$ -greedy)
  - ▶ Execute Action
  - ▶ Store Transition
  - ▶ Sample Mini-Batch
  - ▶ Compute Targets
  - ▶ Train Network
  - ▶ (Update Target Network - periodically)



# Background Information

## ► Network Communication

- ▶ Unreal Engine connects to a Python socket server over 127.0.0.1:5555 using the TCP internet protocol.
- ▶ Python's 'socket' module provides low-level access to TCP/IP.
- ▶ In project:
  - ▶ UE sends:
    - ▶ { "state": [...], "reward": float, "done": bool }
  - ▶ Python responds with a 4-byte action ID.

## ► Framing and JSON Messages

- ▶ TCP is a stream protocol - it does not preserve message boundaries.
- ▶ To safely send messages like JSON:
  - ▶ Prefix each message with a fixed-size length header (i.e. 4 bytes).
  - ▶ This tells the receiver how many bytes to expect for the full payload.
  - ▶ Then send/receive the actual UTF-8 serialized JSON string.
- ▶ This prevents issues like partial reads or hanging on recv().

# Setup

- ▶ Same UE5 environment as the previous Q-Learning project.
- ▶ Python:
  - ▶ dqn\_agent.py: agent logic, QNet, training loop
  - ▶ dqn\_server.py: TCP server, game loop
- ▶ Unreal:
  - ▶ Calls UpdateFunction\_Phase2() to send state
  - ▶ Calls UpdateFunction\_Phase1() to receive action
  - ▶ NetComm class to handle low-level network functionality

# DQN Architecture Overview

- ▶ Model Objective
  - ▶ Learn a function  $Q(s, a)$  to predict the expected long-term reward of  $(s, a)$
  - ▶ Use a neural network to approximate this function instead of a table
- ▶ Neural Network Structure
  - ▶ state\_dim=8, action\_dim=5
  - ▶ Layers - 2 hidden layers with 64 neurons each
  - ▶ Activation - ReLU for non-linearity
  - ▶ Output - 5 Q-values, one per action

```
QNet = nn.Sequential(  
    nn.Linear(state_dim, 64),  
    nn.ReLU(),  
    nn.Linear(64, 64),  
    nn.ReLU(),  
    nn.Linear(64, action_dim)  
)
```

Note: each parameter shown here is mutable.

# Tuning and Design Decisions

Setting	Value (avg)	Purpose / Effect
<b>Replay Buffer Size</b>	10,000	Allows learning from long history, reduces variance
<b>Batch Size</b>	32	Balance between generalization and stability
<b>Learning Rate</b>	4e-4	Controls step size in weight updates
<b>Gamma (<math>\gamma</math>)</b>	0.95	Focus on long-term future rewards
<b>Epsilon (<math>\epsilon</math>)</b>	0.25 → 0.05	Exploration decayed over time to favor exploitation
<b>Target Update Rate</b>	Every 50 episodes	Keeps predictions stable while QNet learns
<b>Loss Function</b>	MSE	Compares predicted Q-values vs. target Q-values
<b>State Normalization</b>	Manual	Ensures inputs fall in predictable numeric ranges

# Training Loop

- ▶ Each step:
  - ▶ UE sends state, reward, and done (as JSON string)
  - ▶ Python stores transition in replay memory
  - ▶ Agent selects action using  $\epsilon$ -greedy
  - ▶ Python replies with action\_id
  - ▶ Learning occurs using sampled mini-batches:
    - ▶ Loss:  $L = \text{MSE}(Q(s, a), r + \gamma \max_{a'} Q(s', a'))$
- ▶ When ‘done’ - Save model in Python server directory using `torch.save`

# Results

- ▶ Early episodes: random or sticking to action that worked well that playthrough
- ▶ As training progressed:
  - ▶ AI learns to avoid healing when no heals left
  - ▶ Prioritizes attacking when in range
  - ▶ Favors guarding instead of waiting
  - ▶ Occasionally learns to dodge appropriately
- ▶ Behaviors evolve based on reward shaping and state awareness (rather than specified hard-coded values)
- ▶ Limitations
  - ▶ Behavior remained mostly reactive - no long-term planning
  - ▶ Occasional overfitting to local minima, e.g. guarding excessively in certain states

# Challenges

- ▶ Sample inefficiency
  - ▶ DQNs require hundreds of episodes (depending on the state/action size) to converge to decent policies - especially with sparse or delayed rewards.
- ▶ Network blocking in UE
  - ▶ Unreals synchronous socket calls froze the game when Python wasn't ready.
  - ▶ Required rethinking the update loop and enforcing safe send-before-receive ordering
- ▶ Timing & Synchronization
  - ▶ Unreal and Python must remain perfectly in sync
  - ▶ Timed or additive events must not interrupt or influence current step
  - ▶ Problems in UE Update Loop:
    - ▶ 1 - Update interval not fast enough
    - ▶ 2 - Another, separate asynchronous event, occurs during a timed action window, changing the current reward and state given to that action
- ▶ Reward Shaping
- ▶ Stability & Divergence
- ▶ Input Format

# Training Challenges

## Underfitting

- ▶ Causes:
  - ▶ Too few hidden nodes
  - ▶ State features poorly encoded
  - ▶ Insufficient training episodes

## Overfitting

- ▶ Causes:
  - ▶ Network memorizes transitions
  - ▶ Too many updates without target net sync
- ▶ Solved with:
  - ▶ Smaller model (64 nodes)
  - ▶ More exploration (higher  $\epsilon$ )
  - ▶ More frequent target network updates

# Additional Takeaways

- ▶ Building a DQN **externally** from UE gives flexibility and transparency
- ▶ Importance of **reward shaping** (while action masking or hard-coding can help stabilize early learning, it defeats the purpose of Q-learning when used regularly)
- ▶ **Replay buffer** stabilizes learning
- ▶ **Real-time learning** can be unstable without careful action timing and synchronization between timed events on the main thread
- ▶ **JSON TCP sockets** can be lightweight and robust – but blocking reads must be handled carefully
- ▶ **Limited generalization** in vanilla DQNs

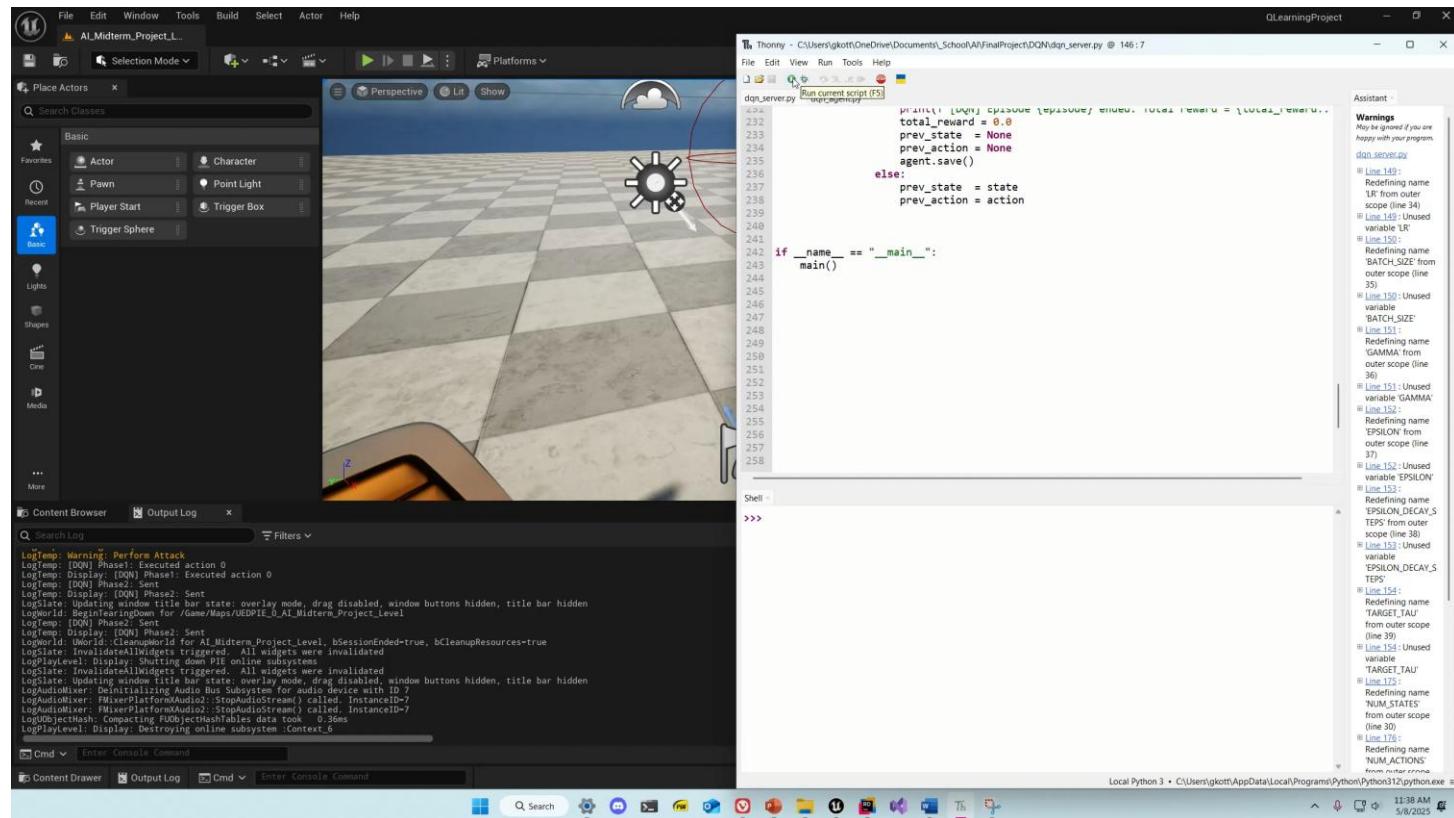
# Comparing DQNs to Traditional Q-Learning in the Unreal Engine Environment

- ▶ Using the same core parameters, the Deep Q-Network (DQN) exhibited behavior similar to the previous Q-Learning implementation, but with several key advantages. Most notably, it introduced the ability to:
  - ▶ Handle large and continuous state and action spaces, thanks to the neural network's generalization capabilities.
  - ▶ Offload processing to an external Python server, minimizing the performance impact on Unreal Engine's main thread and avoiding in-engine bottlenecks.
  - ▶ Allow greater flexibility and modularity, as the standalone Python architecture supports rapid experimentation, integration with machine learning libraries like PyTorch, and performance optimizations that are impractical or unavailable within the UE5 C++ codebase.
- ▶ Additionally, the DQN's separation of concerns makes it easier to prototype advanced features (e.g., prioritized replay, Double DQN, curriculum learning) without needing to modify game engine code, fostering a clean interface between gameplay systems and learning logic.

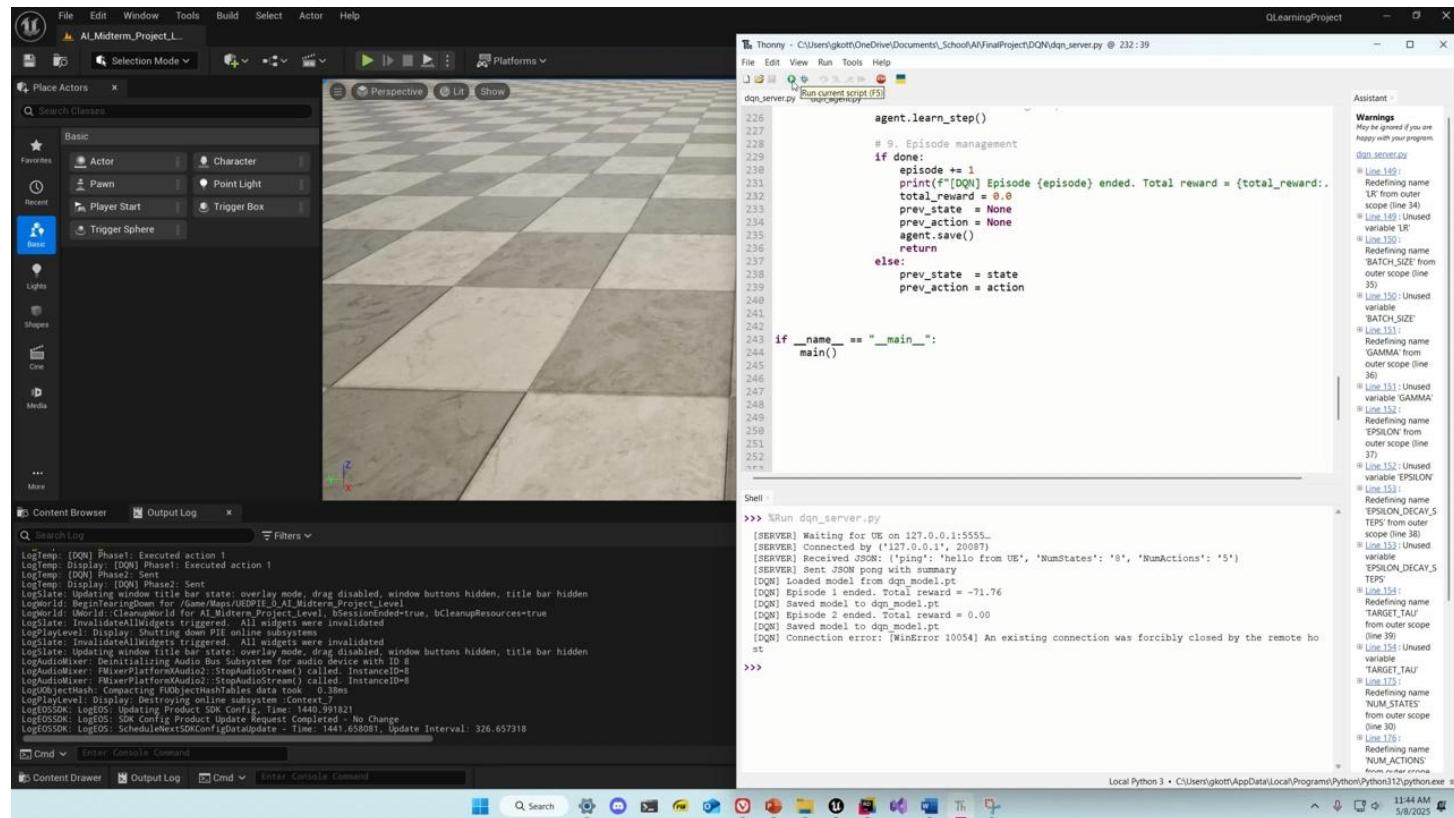
# Future Improvements

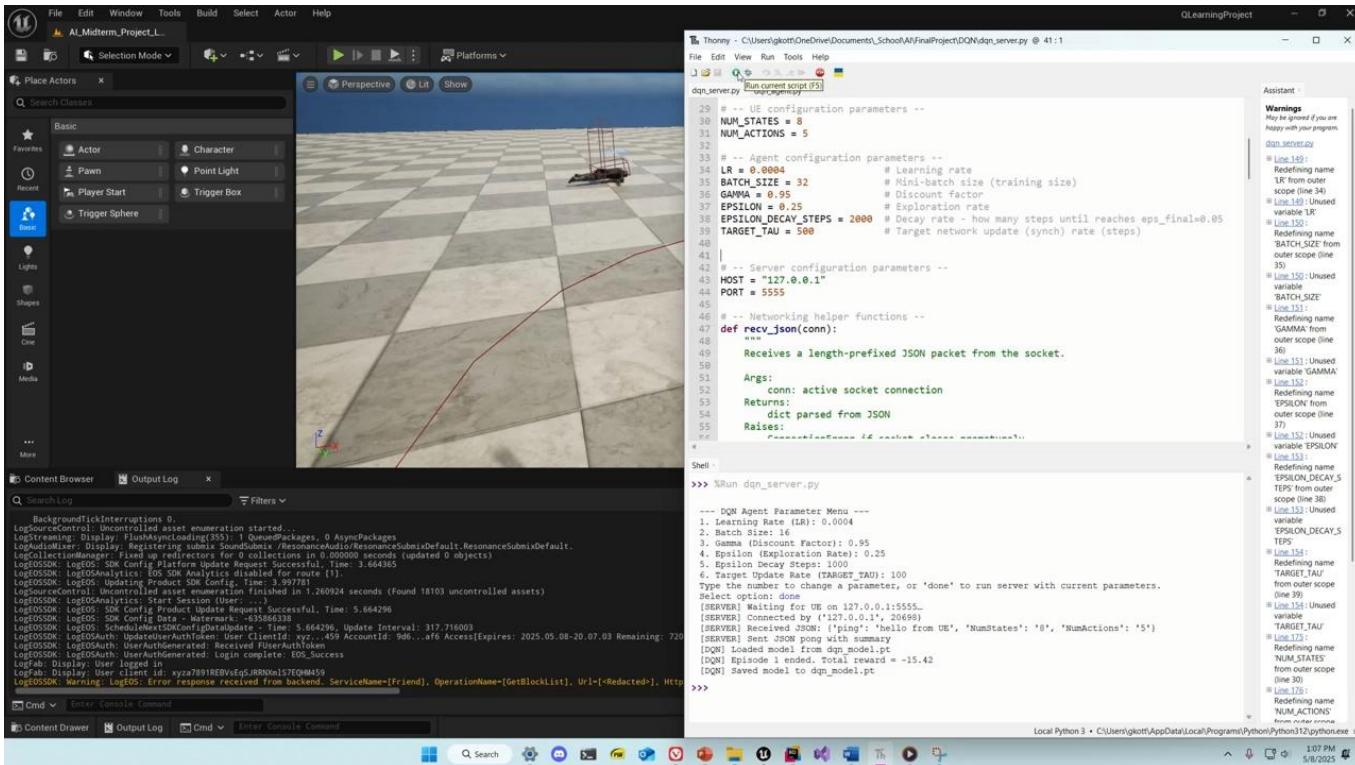
- ▶ Add **asynchronous threading or timers** to avoid blocking in UE (connection to server)
- ▶ Train on multiple agents using a ‘**shared brain**’ (similar to the previous project)
- ▶ Introduce **curriculum learning** - gradually increase difficulty
- ▶ Log and **visualize** (plot):
  - ▶ Episode reward
  - ▶ Q-values
  - ▶ Action distribution over time
- ▶ Optionally transition to **PPO, A3C, or policy gradient methods**

# Demo: Initial Training - Random Behavior



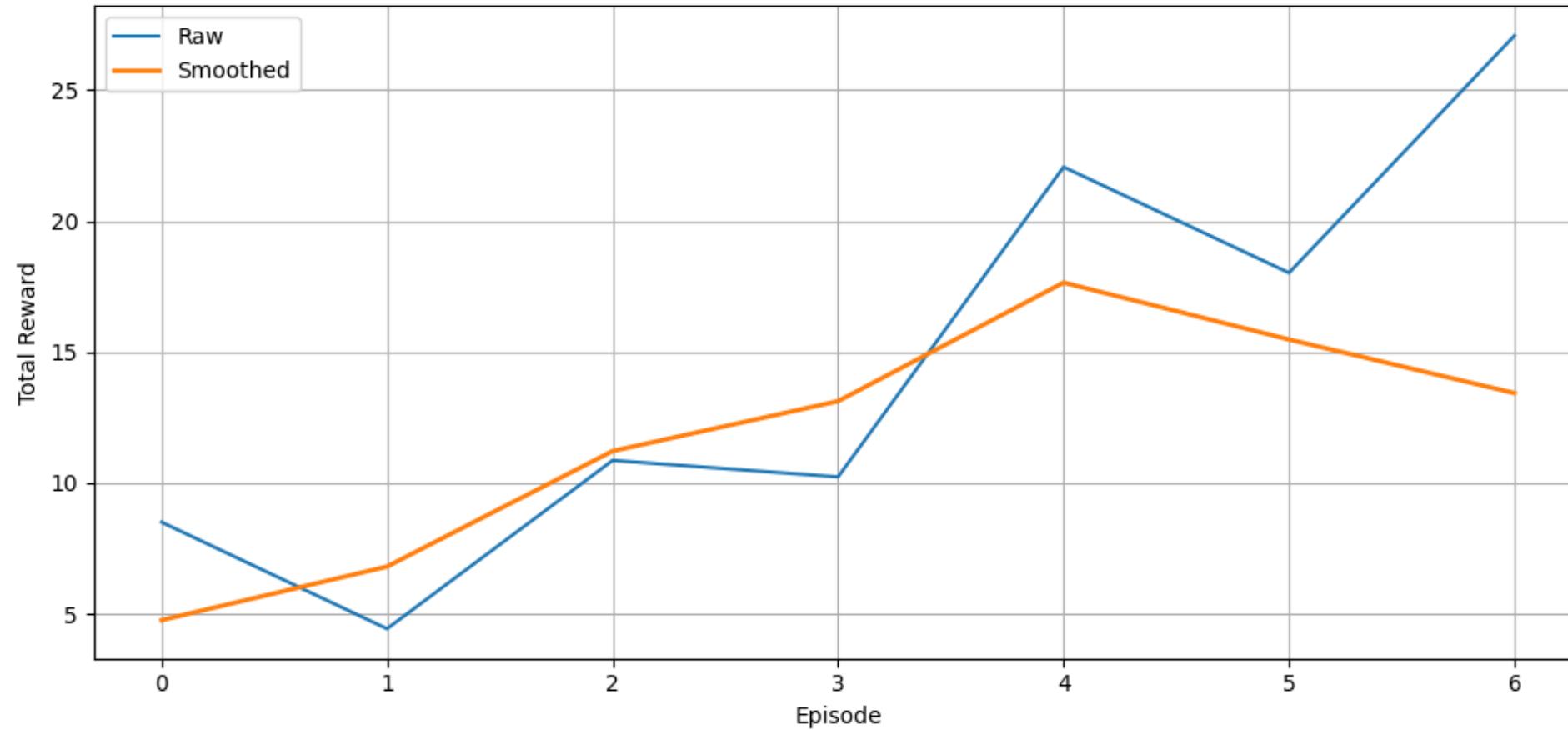
# Demo: Iteration 2 - Stuck Guarding





Demo: Iteration 4 -  
Healing at expected times  
Attacking at better ranges  
Guarding at better ranges

Episode Reward Over Time



Thanks!

# 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., & Dinakarao, 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](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) - Huggingface Deep RL course*. (n.d.). <https://huggingface.co/learn/deep-rl-course/en/unit3/deep-q-network>.