Reinforcement Learning Inside a Closed Game Platform: Combat NPCs Trained and Evaluated Entirely Within Roblox

Anonymous author(s)[1]

[1]*Affiliation withheld for double-blind review*

Abstract

Roblox is one of the largest game platforms in the world, and it is a closed system: creator code runs in sandboxed Luau inside the engine, with no external-trainer path, no GPU access, and, to our knowledge based on public Roblox Creator documentation, no first-party workflow for reinforcement learning (RL) at all. This paper reports what we believe is the first deep RL combat-NPC study trained and evaluated entirely inside that closed loop. The practical problem is blocked line-of-sight combat: an NPC must move around cover, reacquire the opponent, enter weapon range, and actively attack. The main contribution is an empirical deep RL case study showing how action representation, encounter-level labels, and clean data boundaries changed the interpretation of a learned NPC behavior experiment. We compare two deep Q-network (DQN) action representations under the same executable 6×2 movement-attack action space: a direct flat-joint Q output and a factored dual-head value decomposition related to action-branching DQN. The study is not a statistical architecture benchmark: each architecture/time condition is represented by one trained checkpoint evaluated across several evaluation windows. Across those evaluation windows, the factored model showed stronger blocked-start recovery at its 60-minute training checkpoint, while its continued 120-minute checkpoint degraded, mainly by failing to get around the cover. The paper distills an encounter-level evaluation protocol for game-AI developers: score shared encounters, preserve start-condition and failure labels, keep clean/excluded data boundaries visible, match executable action spaces, and re-test continued checkpoints before replacing useful earlier models.

Keywords

reinforcement learning, game AI, Roblox, deep Q-network, action factorization, evaluation methodology, NPC behavior

1. Introduction

Most published game-AI reinforcement learning runs on open stacks: Unity ML-Agents, for example, gives researchers an external Python trainer, GPU acceleration, and parallel copies of the environment [1]. Roblox, one of the largest game platforms in the world, is different. It is a closed system: creator code executes in sandboxed Luau inside the engine, there is no supported real-time external-trainer integration (game scripts can make only rate-limited, asynchronous HTTP calls), scripts have no GPU access, and, to our knowledge based on public Roblox Creator documentation, no first-party workflow exists for training NPC policies with reinforcement learning. A Roblox creator who wants a learning NPC cannot import the standard toolchain; whatever learning happens must happen inside the engine, on the CPU, within the engine's frame budget. This paper reports what we believe is the first deep reinforcement learning (deep RL) combat-NPC study conducted entirely inside that closed loop. It is not a handicapped imitation of the standard workflow; it is the workflow the platform's constraints actually permit, and it matches the position of individual and small-team creators, many of them students, who have no lab cluster.

Deep RL for game NPCs is often presented as a policy-performance problem: train an agent, report a score, and compare the score to a baseline. In a real game-engine combat loop, that framing can miss the behavior that matters to game designers and players. A sword-fighting NPC is not only expected to produce hits. It must respect line of sight, move through geometry, avoid standing still, and convert approach behavior into active combat.

EXAG 2026: Experimental AI in Games Workshop at the AIIDE 2026 Conference, November 9–10, 2026, Belo Horizonte, Brazil

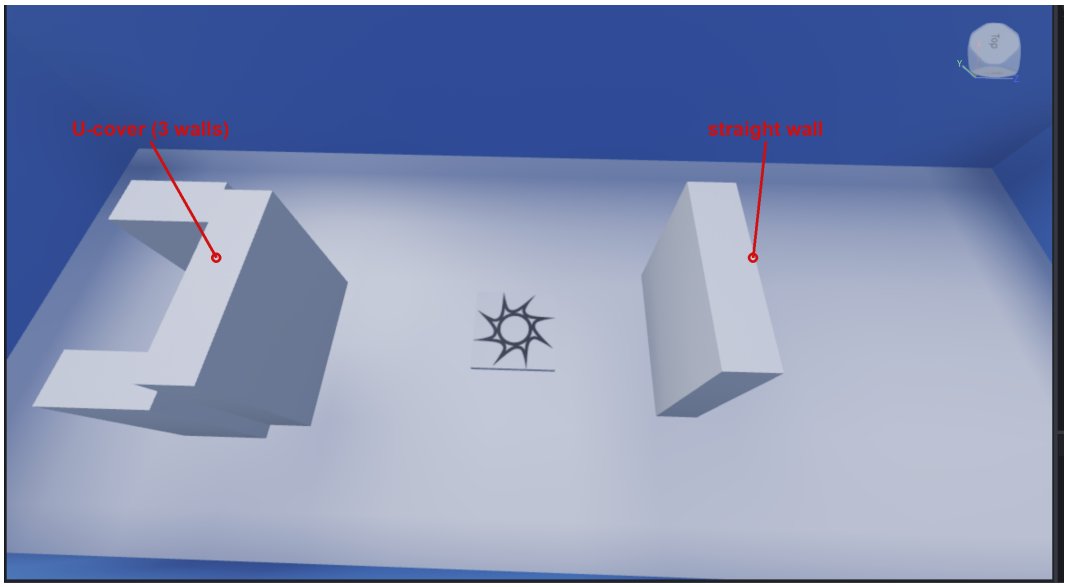

Figure 1: The 1v1 arena in Roblox Studio, with a three-walled U-cover and a separate straight wall. At a blocked start, a cover wall lies on the line of sight between the two NPCs, which must route around it to reacquire sight and attack.

This case study describes a Roblox 1v1 sword-fighting project that exposed this evaluation problem. Figure 1 shows the arena, which includes a U-shaped cover and a separate straight-wall occluder; the blocked-start results below pool episodes from both occluders.

In blocked line-of-sight starts, an NPC cannot immediately attack the opponent. A useful policy must move around cover, regain line of sight, enter sword range, choose an attack, and land a hit. That makes the task both a delayed-credit RL problem and an evaluation-design problem. A raw hit total over a fixed time window can mix the initial encounter with later deaths and respawns. It can also hide whether the agent that moved around cover actually attacked, or whether it simply walked into range and was hit by the opponent.

The lessons below are organized around one evaluation principle: learned game-NPC behavior should be scored at the encounter level rather than only through aggregate hit windows. For this project, the key question became: when line of sight is blocked at the start of an encounter, can either NPC recover from that situation and produce real combat? In the observed trajectory, answering that question exposed a factored model that recovered often at 60 minutes but degraded after continued training, a result a window-level hit count could have obscured.

2. Methods

2.1. Game environment

The setting is a 1v1 sword-fighting scene built in Roblox Studio, Roblox's creation tool. The game is real time, not turn-based: the simulation runs continuously, and each NPC repeatedly reads its observation and picks its next action while movement and combat play out in the world. The arena is a walled rectangle of roughly 60 by 62 studs (Roblox's unit of length) containing the two cover structures shown in Figure 1: a three-walled U-shaped cover and a single straight wall, both solid and collidable. Each NPC carries a short-range sword; a hit lands when the swung sword makes contact with the opponent, which costs the opponent health. When an NPC's health reaches zero it dies and respawns, and fighting continues with fresh health. There is no human player in the loop: the two NPCs fight autonomously, and combat timing, hit detection, and health run on the same engine rules a normal Roblox game uses.

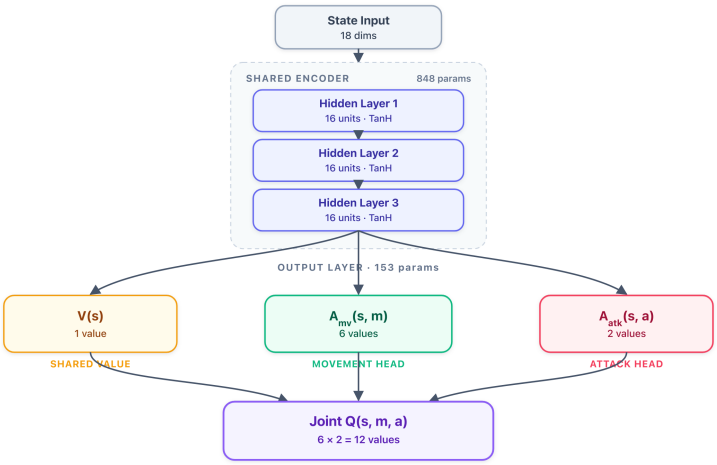

Figure 2: The factored dual-head architecture. A shared 18–16–16–16 encoder feeds a state value and separate movement (6) and attack (2) advantage streams, recombined additively into twelve joint action-values; the flat head outputs the twelve values directly over the same executable actions.

2.2. AI system

The policies were trained and evaluated inside the Roblox game-engine loop. "In-engine" means that combat timing, line-of-sight checks, sword hit detection, death and respawn boundaries, and learning updates run in the same Roblox/Luau environment used to inspect and debug the NPCs. This made the setup less like a clean offline benchmark and more like the workflow a game developer would use while iterating in Roblox Studio.

Each NPC receives an 18-dimensional compact observation of combat and geometry, including opponent distance, line-of-sight state, cover distances, recent damage, and stuck-state signals. Rewards emphasize landed hits and combat outcomes while penalizing damage taken, unnecessary swings, and passive no-engagement behavior.

The comparison focuses on action representation. Both models are deep Q-network (DQN) policies [2] that use the same executable action space: six movement choices crossed with two attack choices, for 12 executable movement-attack actions. The `flat_joint` policy emits one Q-value for each executable movement-attack pair. The `factored_dual_head` policy emits a shared value, six movement advantages, and two attack advantages, then reconstructs each joint action value additively, following the value/advantage decomposition of dueling networks [3]. The two networks are intentionally comparable in size: the flat-joint model has 1,052 trainable parameters, while the factored dual-head model has 1,001. Thus, the comparison changes how Q-values are parameterized over actions, not the overall model scale.

Over the twelve movement-attack pairs (m, a), the flat head outputs a value per pair directly, $Q_{\text{flat}}(s, m, a) = f_\theta(s)_{(m,a)}$, while the factored head shares one state value and adds centered movement and attack advantages:

$$Q_{\text{fac}}(s, m, a) = V(s) + \big(A_{\text{move}}(s, m) - \overline{A}_{\text{move}}\big) + \big(A_{\text{atk}}(s, a) - \overline{A}_{\text{atk}}\big),$$

where $\overline{A}_{\text{move}}$ and $\overline{A}_{\text{atk}}$ are the means over the six movement and two attack advantages. Both heads score the same twelve actions; only the parameterization differs. Figure 2 summarizes this factored head and notes that the flat head scores the same twelve executable actions directly.

The factored model is related to action-branching DQN [4]. Factored heads are attractive in game AI because compositional NPC decisions (movement, attack, target choice, skill choice, and timing) can make flat joint action spaces grow multiplicatively. Here, however, the 6×2 space is small enough to enumerate, so we do not claim a scalability result. Instead, the matched setting lets us ask a controlled question: when scalability pressure is removed, how does factoring movement and attack Q-values change blocked-start recovery and failure modes?

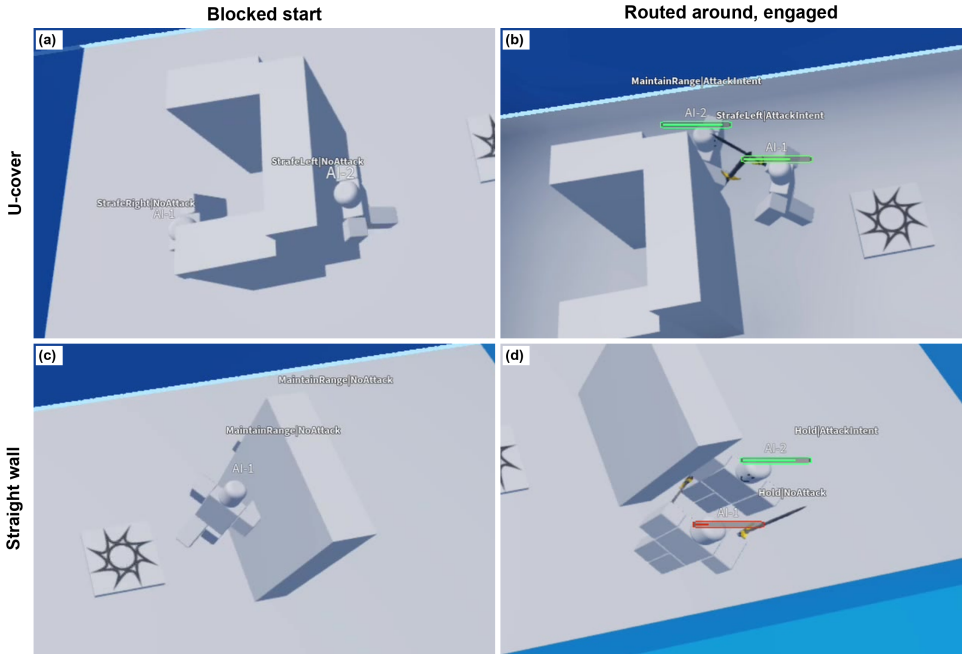

Figure 3: Representative evaluation frames for the U-cover (top) and straight wall (bottom). Left: blocked starts with line of sight occluded. Right: recovered engagements after routing around cover. Colored bars show health.

We do not compare against a hand-scripted NPC. Scripted baselines have no standard strength: a carefully tuned behavior tree and a naive one would give very different bars. This study's question is representation and evaluation, not learned-versus-scripted performance, so we claim nothing about how these policies compare to scripted NPCs.

2.3. Evaluation protocol

We train the two NPCs together for a set amount of in-engine time, then freeze the trained network and evaluate it; we report checkpoints trained for about 60 and 120 minutes. For each frozen checkpoint, we run multiple bounded 600-second frozen-policy evaluation (EVAL) seed windows and pool the clean blocked pair episodes. Inside each window the NPCs fight repeatedly: whenever one dies it respawns and the next fight begins, so a single window contains many separate fights. A raw count of sword hits over the whole window is therefore a poor score, because it sums across all of those fights and respawns.

We instead score one shared fight at a time, which we call a pair episode. A pair episode begins when both NPCs are alive and ends the instant either one dies; that NPC respawns and a new pair episode begins for both. For each pair episode we record its starting condition. A blocked start is one where, at the instant the pair episode begins, neither NPC can see the other, because a cover wall lies on the line of sight between them. These are the hard fights we evaluate. Figure 3 shows a blocked start and a recovered engagement for each of the two occluders.

In a blocked start, an NPC recovers if it completes the whole task: it moves around the cover, regains line of sight to the opponent, closes into sword range, and lands a hit on the opponent. Our primary metric is any-side recovery, where at least one of the two blocked NPCs recovers; we also report the stricter both-side recovery, where both do.

When a blocked fight is not a recovery, we record why. Two focal diagnostic classes are most actionable. The main navigation failure class is range-without-line-of-sight: the NPC reaches sword-range distance while still occluded and never reacquires sight. The second is navigation-without-attack: the NPC reaches clear line of sight and sword range, but records no self swing before the pair ends. Other no-recovery, stall, near-cover, and swing-without-hit cases remain in an Other bucket. This split

Table 1
Clean blocked episode recovery from trained checkpoints. Rows pool different evaluation seed windows and are descriptive, not seed-paired. Any-side means at least one NPC recovered; both-side means both did.

Checkpoint	Episodes	Any-side	Both-side
60 min flat	107	36 (33.6%)	30 (28.0%)
60 min factored	102	70 (68.6%)	51 (50.0%)
120 min flat	101	34 (33.7%)	30 (29.7%)
120 min factored	100	13 (13.0%)	7 (7.0%)

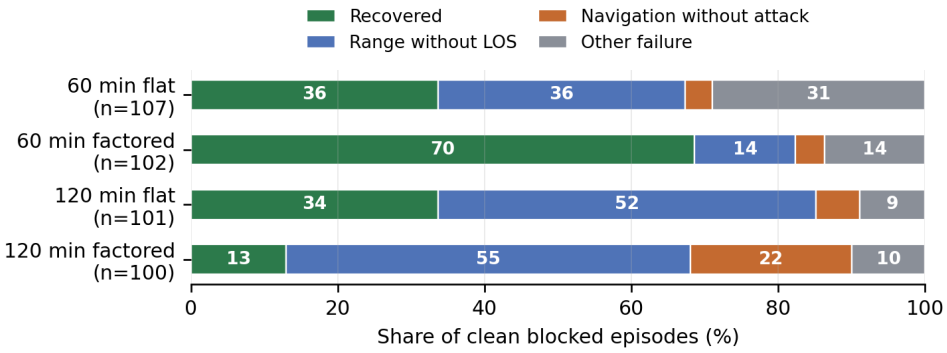

Figure 4: Recovery and failure-mode breakdown for the same clean blocked episodes as Table 1. Range without LOS means sword-range distance while still occluded; navigation without attack means clear LOS and range with no self swing logged. Other includes no-recovery, stall, near-cover, and swing-without-hit cases. Occluders remain pooled because the clean pair ledger lacks reliable per-episode occluder-type labels.

tells you whether a dropped score means the agent stopped getting around cover or stopped swinging, two problems that call for different fixes.

3. Results

Table 1 reports the clean blocked-start pair episodes used for the results below. Clean rows follow the experiment ledger's documented exclusions, keeping invalid, truncated, overrun, restart-contaminated, or explicitly flagged artifacts out of the headline counts. They are descriptive checkpoint evidence, not a seed-paired statistical architecture benchmark. The 60-minute flat and factored rows use overlapping but not identical EVAL seed batches, so the 68.6% versus 33.6% gap should be read as a descriptive checkpoint comparison, not a paired per-seed contrast.

In these clean rows, the 60-minute factored model recovered from blocked starts more often. However, after continuing training to a roughly two-hour total checkpoint, the factored model dropped sharply. Figure 4 shows why. For the factored model, the dominant 120-minute failure class is range-without-line-of-sight (rising from 14 to 55 episodes): the agent reaches sword-range distance while still occluded and never reacquires a clear line of sight. A smaller factored-model class is navigation-without-attack (rising from 4 to 22 episodes), where the agent gets around the cover, regains line of sight, and enters attack range, but then never swings. The flat-joint model had lower recovery than the factored model at 60 minutes. It did not show the same 60-to-120-minute collapse in this set of EVAL windows, but because rows are not seed-paired, that should be read descriptively rather than as robustness evidence.

4. Discussion

4.1. Lessons for game-AI practice

The first lesson is to make the encounter the scoring unit. A fixed EVAL window can contain many deaths and respawns; pair episodes preserve which fight produced each recovery or failure.

The second lesson is to record each encounter's start condition and failure class. Start labels separate blocked-start recovery from ordinary clear-view combat, while failure labels distinguish policies that never get around cover from policies that route successfully but fail to attack.

The third lesson is to keep headline and excluded evidence separate. Truncated runs, overrun windows, restart interruptions, flagged respawn-boundary artifacts, and other edge cases should remain auditable without entering the headline table.

The fourth lesson is to match executable action spaces before interpreting architecture effects. Factored heads are partly motivated by scalability in larger compositional action spaces, but this arena is small enough to enumerate. That makes the comparison a controlled test of behavior, stability, and Q-parameterization, not a comparison of unequal action choices.

The fifth lesson is to re-test continued checkpoints before replacing earlier useful models. In this trajectory, the factored model looked stronger at 60 minutes but drifted after continued training.

4.2. Impact and significance

This study connects academic RL with game development practice by testing learned NPC behavior under the constraints of a live game-engine workflow rather than a clean offline benchmark. It does not present a polished benchmark victory. It presents an empirical in-engine study of what changed our interpretation of an NPC learning experiment in Roblox, a creator platform with different constraints from standard research simulators.

This platform choice is itself part of the significance. Mature stacks such as Unity ML-Agents provide a standard external trainer, parallel environments, and broad research use [1]. By contrast, to our knowledge based on public Roblox Creator documentation, there is no comparably standard first-party Roblox Studio workflow for RL-trained NPC behavior. The value of this small DQN-style setup is therefore not that it replaces those mature stacks, but that it shows a lower-overhead path for individual and small-team Roblox creators: discrete CPU-friendly policies can be inspected with ordinary Studio logs and encounter labels, and can sit beside ordinary scripted NPC logic. Many of these creators are themselves students, and a student who wants to add a learning NPC to their own game typically has exactly the resources this study assumes: an ordinary computer, no GPU, no budget, and no lab. The workflow reported here is one such a creator can follow end to end. We therefore contribute a complementary Roblox-native path, together with an encounter-level protocol that makes learned NPC combat behavior auditable without assuming an external trainer, GPU, or platform-standard RL framework.

The transferable artifact is a compact evaluation protocol: episode scoring, blocked-start labels, active-attack checks, a clean/excluded data ledger, and failure-class labels for line-of-sight combat. These ideas can be reused for other cover-based or visibility-dependent NPC behaviors. Because the policies are small and CPU-friendly, the workflow is also practical for iterative debugging inside Roblox Studio rather than only in offline benchmark code.

Declaration on Generative AI

During the preparation of this work, the authors used Claude (Anthropic) to reword, restructure, and format the manuscript under the authors' direction. The research design, implementation, experiments, data, and conclusions are entirely the authors' own. The authors reviewed and edited all content and take full responsibility for the publication's content.

References

[1] A. Juliani, V. Berges, E. Teng, A. Cohen, J. Harper, C. Elion, C. Goy, Y. Gao, H. Henry, M. Mattar, D. Lange, Unity: a general platform for intelligent agents, arXiv preprint, https://arxiv.org/abs/1809.02627, 2018.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, D. Hassabis, Human-level control through deep reinforcement learning, Nature 518 (2015) 529–533. doi:10.1038/nature14236.

[3] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, N. de Freitas, Dueling network architectures for deep reinforcement learning, in: Proceedings of the International Conference on Machine Learning (ICML), 2016, pp. 1995–2003.

[4] A. Tavakoli, F. Pardo, P. Kormushev, Action branching architectures for deep reinforcement learning, in: Proceedings of the AAAI Conference on Artificial Intelligence, volume 32, 2018. doi:10.1609/aaai.v32i1.11798.