HollowAI: Reinforcement Learning for Hollow Knight Boss Fights

Federico Fantozzi [2047034], Elia Belli [2006305], Giovanni Colasuonno [2046000]

1 Introduction

Hollow Knight is a 2D action-adventure game featuring boss fights of varying difficulty. Our goal was to use reinforcement learning (RL) to train an agent capable of defeating some of these bosses (Figure 1).

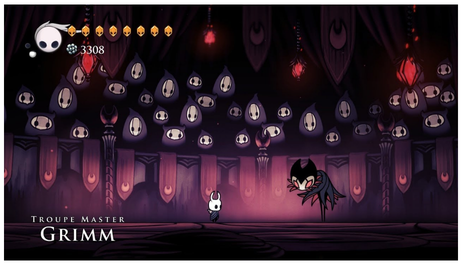

Figure 1. Boss-fight arena in Hollow Knight

While many existing projects attempt this, they typically rely on computer vision to extract data from the game. This method is computationally expensive and often requires a long time for the model to converge, due to the high-dimensional observation space (thousands of features). Given our limited time and hardware resources, we chose a different approach: we modded the game to directly extract the bosses' finite state machines (FSMs) and other relevant information. This allowed us to reduce the observation space to a few hundred features.

Although this approach provides the model with less information compared to image-based methods, we believe it is still sufficient to learn how to fight and win, offering a good compromise between accuracy and performance. Additionally, the reduced model complexity and elimination of rendering requirements enabled us to run multiple instances of the game concurrently, accelerating training while also being significantly more energy-efficient.

1.1 *About Hollow Knight*

For readers unfamiliar with the game, here is an overview of the core combat mechanics.

The player controls a character capable of moving horizontally, jumping, double-jumping, wall-climbing, and performing a dash—a quick burst of speed that grants temporary invincibility frames (Figure 2).

Figure 2. Moveset abilities

Combat primarily revolves around the use of a **melee weapon** to strike enemies at close range: this melee attack can be charged to increase damage and alter its area of effect. Additionally, the player can cast **ranged spells** (Figure 3) that consume a special resource called “Soul”, obtained by inflicting melee damage, which can also be used to **heal**.

Figure 3. Spells: each spell is cast in a specific direction.

The player has a health bar, a Soul meter, and can equip "Charms"—items selected outside of combat that grant various perks (e.g., increased damage, extended attack range). Only a limited number of Charms can be equipped at a time, making their selection a strategic decision that significantly affects the chances of success.

Boss encounters are particularly challenging due to their complex, multi-phase attack patterns. These often blend predictable, telegraphed moves with sudden, rapid strikes, forcing the player to recognize and adapt to evolving sequences. Some bosses feature unique mechanics, such as summoning minions or teleporting, which introduce additional complexity. To avoid handling numerous edge cases, we limited our focus to simpler bosses without such behaviours. Additionally, opportunities to heal during combat are rare, further increasing the challenge.

2 Methodology

2.1 Mod Infrastructure and Environment

Our first step involved developing HollowGym, a custom Hollow Knight mod designed to expose the game as a reinforcement learning environment. HollowGym facilitates direct interaction with the game's internal state and provides the following core functionalities:

- **Data Access**: allows retrieval of player and boss state information;
- **Input Management**: executes agent actions received via WebSocket communication;
- **Data Processing**: locally computes rewards and termination conditions before sending feedback to the agent;
- **Rendering Control**: enables toggling the in-game rendering for performance or debugging purposes.

To coordinate the **training loop**, we developed HollowAI, a Python interface connecting Stable-Baselines3 agents [1] with HollowGym.

The **environment** is built on top of Gymnasium, OpenAI's standard library for reinforcement learning [2], while HollowAI manages agent logic, episode flow, and communication with the game.

Communication Protocol: Interaction between HollowAI (server) and HollowGym (client) occurs via WebSocket, using a simple JSON-encoded message protocol. Before training begins, a **three-way handshake** ensures synchronization between the two components:

1. HollowGym sends a `ready` message;
2. HollowAI responds with configuration details;
3. HollowGym replies with its capabilities, including the defined observation space for the selected boss.

Multi-Environments: Stable-Baselines3 supports parallel training through vectorized environments, their execution is **synchronous**, meaning that all environments have to finish each step before moving on. So, to start the training, we only need to launch multiple game instances and connect them to HollowAI, i.e. the server (Figure 4).

If rendering was enabled, this approach would be very GPU-intensive, but disabling graphical output makes it lightweight enough to shift the bottleneck to RAM usage instead.

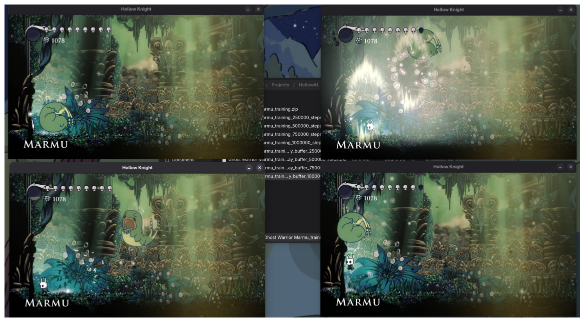

Figure 4. Multi-Environment in action: rendering enabled for demonstration

2.2 Feature Design

In designing the feature set, our goal was to provide the agent with all the critical information that an image-based approach would typically capture. To achieve this, we selected a combination of **spatial** and **game-state** features.

To give the agent spatial awareness, we included the following:

- player and boss positions and velocities;
- the distance between the player and the boss;
- the distance from the player to the centre of the scene;
- the player's current state (e.g., attacking, dashing, airborne).

In addition to spatial data, we incorporated key gameplay variables such as the health of both the player and the boss, as well as the player's soul meter.

To compensate for the lack of visual input, typically used by human players to interpret and anticipate boss attacks, we also incorporated the boss's finite state machine (Figure 5), allowing the agent to anticipate the boss's next move based on its current internal state.

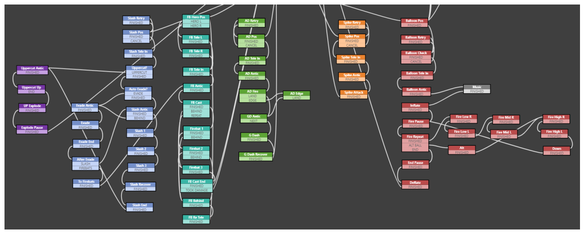

Figure 5. Excerpt from the boss's finite-state machine.

To refine it, we manually grouped semantically similar finite state machine (FSM) states into unified categories: while this process is time-consuming and somewhat subjective, it helps reduce input dimensionality and can improve learning efficiency, if applied carefully so as not to lose critical behavioural information.

We ended up with a relatively small observation space of approximately 100 features, slightly larger for bosses with more complex finite state machines. Before feeding the data to the model, all numerical values are normalized to the range $[-1, 1]$ to facilitate learning, meanwhile categorical data, such as player and boss states, are one-hot encoded.

2.3 *Reward Function*

The main objective for the agent is to defeat the boss, which yields a **base reward** of +1000 upon success. However, to provide more frequent feedback during the episode and support learning stability, we designed a denser reward structure by introducing intermediate **sub-rewards**.

Specifically, the agent is granted access to a reward pool of +500, distributed proportionally as it deals damage to the boss. For example, assuming the boss has 1000 total health, dealing 100 damage awards the agent 10% of the pool, i.e., +50 reward. If the agent defeats the boss, it will have received the full +500 from this pool in addition to the +1000 victory reward.

To discourage reckless behaviour, such as charging the boss without regard for incoming damage, we introduced penalties for taking damage. Each time the agent is hit, it incurs a -50 penalty and if it manages to heal himself he gets +50. Since the agent starts with 9 lives, dying results in a cumulative penalty of -450 : this mechanism ensures that even if the boss barely survives the encounter, the agent still receives a net positive sub-reward, though significantly lower than if it had defeated the boss.

This balance between offensive progress and defensive caution encourages the agent to both defeat the boss and survive efficiently.

We also experimented with **instrumental rewards** to facilitate learning in the early stages of training. For example, the agent was rewarded with +1 for performing an attack action, even if the attack did not deal damage. The intention was to encourage exploration and help the agent learn the basic mechanics of interaction within the environment.

However, instrumental rewards must be used with caution. They can lead to unintended behaviours, as the agent may exploit the reward signal [3]; in our case this risked the agent learning to spam attacks without engaging strategically with the boss. Due to this potential for reward hacking and divergence from the primary objective, we chose to exclude instrumental rewards from the final reward function shown in Table 1.

Table 1: Reward function used during training

Event	Reward	Type
Win	+1000	Base
Damage Inflicted	+500 × DamagePercentage	Sub-Reward
Damage Received	-50	Sub-Reward
Heal	+50	Sub-Reward

2.4 *Models and Hyperparameters*

Based on existing projects in this domain [4, 5], we identified DQN and PPO as the most commonly used reinforcement learning models. We opted to begin with DQN due to its relative simplicity, with the intention of transitioning to PPO if additional performance or stability was required.

To ensure flexibility in experimenting with different models, we utilized Stable-Baselines3 framework (SB3), which provides robust implementations of both algorithms, integrates seamlessly with the Gymnasium environment and supports live visualization through TensorBoard [6], allowing us to effectively monitor training progress.

DQN: Deep Q-Network (DQN) is a value-based reinforcement learning algorithm that uses a neural network to approximate the Q-function, which estimates the expected return for taking a given action in a given state. It learns by storing experiences in a replay buffer and training on randomly sampled batches.

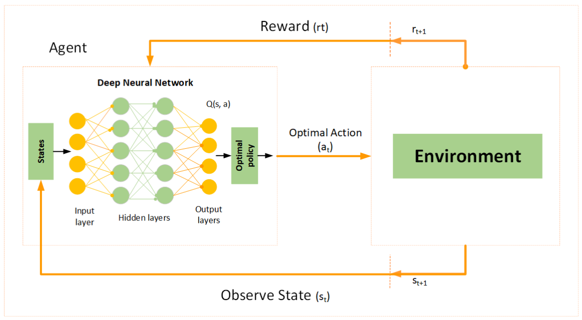

Figure 6. High-Level Diagram of DQN (adapted from [7])

PPO: Proximal Policy Optimization (PPO) is a policy-based reinforcement learning algorithm that directly learns a stochastic policy by optimizing a clipped surrogate objective. It balances exploration and stability by preventing large, destabilizing policy updates through a clipping mechanism that limits how much the new policy can deviate from the old one. PPO uses advantage estimates—often from Generalized Advantage Estimation (GAE)—to evaluate how much better an action is than the expected value.

Regarding hyperparameters, we initially used the default

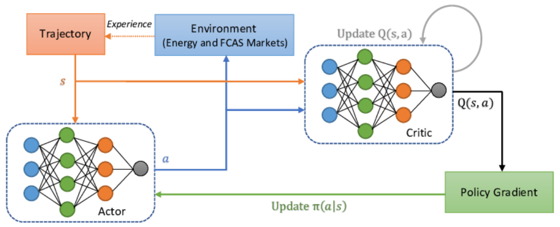

Figure 7. High-Level Diagram of PPO (adapted from [8])

settings from SB3 and varied them to observe their interaction with our observation space and their impact on the model's behavior. The final configurations we settled on are shown in Tables 2-3.

Table 2: DQN Hyperparameters

Hyperparameter	**Value**
Learning Rate	7e-5
Learning Start	5000 steps
Gamma (Discount Factor)	0.95
Replay Buffer Size	100000
Batch Size	64
Exploration Strategy	ϵ-greedy
Initial ϵ	1.0
Final ϵ	0.1
Linear ϵ Decay	0.7
Training Frequency	4 steps

Table 3: PPO Hyperparameters

Hyperparameter	**Value**
Parallel Enviroments	10
Steps per Enviroment	2048
Learning Rate	3e-4
Gamma (Discount Factor)	0.99
Batch Size	64
Epochs	10
GAE Lambda	0.95
Clip Range	0.2
Entropy Coefficient	0.01
Value Function Coefficient	0.5
Max Gradient Clipping Value	0.5

2.5 *Parallel Training*

The use of parallel environments impacts training in two main ways:

1. **Faster Data Collection**: Parallel environments allow for collecting experiences more quickly, increasing the throughput of training data.
2. **Reduced Data Correlation**: Running multiple environments simultaneously produces more diverse and less correlated samples by gathering data from multiple independent trajectories.

The extent of these benefits depends on the type of algorithm. Off-policy methods like DQN primarily gain from faster data collection, since they learn from a replay buffer. In contrast, on-policy methods like PPO significantly benefit from parallelism because they require fresh, uncorrelated data for each policy update.

We experimented with parallel training for both models, but DQN's performance actually worsened. Although DQN in SB3 can interface with vectorized environments, its implementation does not natively support multi-environment training, as a result, using multiple environments may lead to inconsistent or degraded performance. While parallelizing DQN is possible, it requires a redesign of the algorithm [9, 10].

For this reason, we used single-environment training for DQN and multi-environment training for PPO.

3 Conclusion

3.1 *Training Results*

For our training experiments, we selected two bosses as test subjects: a relatively easy one, **Marmu**, and a medium-difficulty opponent, **Hornet**. These bosses were chosen in part because prior work using image-based methods has demonstrated successful training results on them [11, 12].

Charms Setup: The selection of equipped charms is a critical factor in training. Omitting charms significantly increases the difficulty of the encounter, even for skilled human players. In our experiments, training an agent without charms resulted in a dramatic increase in convergence time, taking nearly twice as long to achieve a single win. We opted for a straightforward charm setup focused on enhancing offensive capabilities, specifically increasing melee damage, weapon range, and attack speed.

Marmu: Marmu is essentially a large ball that bounces around the arena, taking knock-back upon being hit, which can be a bit chaotic, but this is basically everything he does: his large hitbox and lack of complex attack patterns make him relatively easy to deal with.

The agents were trained for a total of 2 million steps (1750 episodes) (Figure 8). Both successfully learned to track Marmu's erratic movement and consistently avoid collisions.

Hornet: Hornet's moveset is significantly more diverse than Marmu's, featuring a combination of melee strikes, ranged attacks, and area-of-effect abilities, Hornet also introduces moments of vulnerability, during which the player can heal or inflict additional damage: this encounter is more strategic in nature and provides a more rigorous test of the agent's ability to recognize and adapt to varied attack pat-

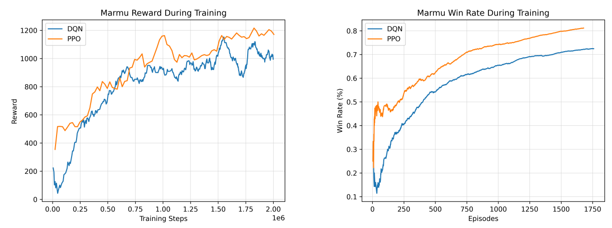

Figure 8. Win rate and reward per episode during Marmu training

terns.

The agents were trained for a total of 4 million steps (1000 episodes) (Figure 9). Throughout training, both learned to consistently track Hornet's position, maintain appropriate distance during attacks, and combine both melee and spell-based offences, including the use of charged attacks when possible, although PPO is less precise on this boss.

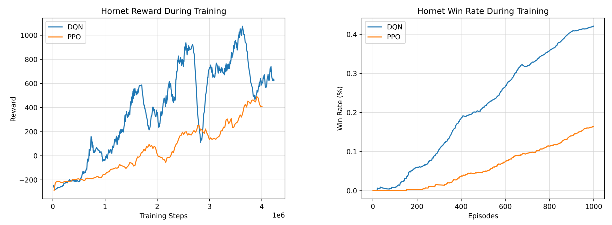

Figure 9. Win rate and reward per episode during Hornet training

Evaluation: After training, the agents were evaluated over 100 matches. DQN and PPO achieved win rates of **90% and 90% against Marmu**, and **96% and 54% against Hornet**, respectively.

You can watch the trained agents in action in the following gameplay videos:

- DQN vs. Marmu
- DQN vs. Hornet
- PPO vs. Marmu
- PPO vs. Hornet

Healing Overlooked: One consistent observation across all training sessions, was that the agent never learned to heal. Probably, it actively chose not to heal, consistently preferring to use the soul it gathered to cast offensive spells instead: this behaviour is understandable given that attacking with spells provides a higher reward.

While this strategy did not negatively impact performance in our specific scenarios, it could become problematic in longer boss fights or against opponents that deal more damage, where survivability becomes more critical: in such cases, the reward structure may require adjustment to incentivize healing behaviour.

Transfer Learning: To evaluate potential transfer learning, we tested Hornet-trained agent against Marmu. Surprisingly, the agent demonstrated strong transferability—it defeated Marmu in 94 out of 100 matches. Although the state-machine portion of the observation space was irrelevant in this context, the agent was still able to perform effectively using only spatial data. This is likely because Marmu is a relatively simple boss, and basic tracking behaviour was sufficient for victory.

3.2 *Limitations and Possible Improvements*

HollowAI does not currently support boss fights involving multiple enemies, but it can be extended to do so with minimal modifications.

The model should also be capable of learning to defeat bosses with multiple phases. However, training in such scenarios tends to be slower, as the agent must first learn to consistently reach later phases before it can begin optimizing behaviour within them. Overfitting to a single phase could become an issue, but this can be mitigated by introducing a feature that explicitly encodes the current phase.

Although HollowAI could support multiple enemies, it faces limitations when dealing with projectiles. Specifically, the system is currently unable to extract the positions of dynamic projectiles. This may impair the agent's ability to respond effectively, particularly in fights where projectile trajectories are non-deterministic. In Hornet's case, projectiles follow a fixed pattern, allowing the agent to infer and adapt to them with relative success. However, in encounters with unpredictable projectile behaviour, the agent may struggle to develop reliable dodging strategies, even if it can detect when projectiles are being launched.

3.3 *Final Remarks*

This work demonstrates that a state-machine-based approach to reinforcement learning in Hollow Knight can yield effective and efficient results. While it sacrifices some of the flexibility and generality of image-based methods, it offers significant advantages in terms of performance, interpretability, and training speed.

The use of direct game state access, combined with manual state abstraction and well-tuned reward shaping, proved sufficient to handle a variety of boss mechanics. Although this approach has some inherent limitations, particularly regarding projectile tracking, it provides a strong foundation for further development.

Overall, this project highlights an alternative path for applying reinforcement learning in games, particularly in cases where full modding access allows simplification of the learning problem without undermining its strategic complexity.

References

[1] Antonin Raffin et al. *Stable-Baselines3: Reliable reinforcement learning implementations.* Journal of Machine Learning Research (JMLR) Open Source Software. 2021. URL: https://github.com/DLR-RM/stable-baselines3.

[2] J. K. Terry, T. Helmuth, B. E. Sullivan, et al. *Gymnasium: A standard API for reinforcement learning environments.* 2023. URL: https://github.com/Farama-Foundation/Gymnasium.

[3] Andrew Y. Ng. "Shaping and policy search in reinforcement learning". In: *Proceedings of the ICML-03 Workshop on the Continuum from Labeled to Unlabeled Data.* 2003. URL: https://dl.acm.org/doi/abs/10.5555/979521.

[4] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *arXiv preprint arXiv:1312.5602* (2013). URL: https://arxiv.org/abs/1312.5602.

[5] Maximilian Schuck. *SoulsAI: Distributed Reinforcement Learning Framework for SoulsGym Environments.* Accessed: 2025-05-18. 2023. URL: https://github.com/amacati/SoulsAI.

[6] TensorFlow Authors. *TensorBoard: Visualizing Learning.* 2023. URL: https://www.tensorflow.org/tensorboard.

[7] Samina Amin. *Deep Q-Learning (DQN).* [Online; accessed 31-May-2025]. 2018. URL: https://medium.com/@samina.amin/deep-q-learning-dqn-71c109586bae.

[8] Muhammad Anwar et al. *Proximal Policy Optimization Based Reinforcement Learning for Joint Bidding in Energy and Frequency Regulation Markets.* Dec. 2022. DOI: 10.48550/arXiv.2212.06551.

[9] Volodymyr Mnih et al. "Asynchronous methods for deep reinforcement learning". In: *arXiv preprint arXiv:1507.04296* (2016). URL: https://arxiv.org/abs/1507.04296.

[10] Zhihong Liu et al. "Acceleration for Deep Reinforcement Learning using Parallel and Distributed Computing: A Survey". In: *arXiv preprint arXiv:2411.05614* (2024). URL: https://arxiv.org/abs/2411.05614.

[11] Ailec0623. *DQN_HollowKnight [Computer software].* 2023. URL: https://github.com/ailec0623/DQN_HollowKnight.

[12] Seermer. *HollowKnight_RL [Computer software].* 2023. URL: https://github.com/seermer/HollowKnight_RL.