Adaptive Game AI using PPO and the Impact of Hyperparameters in training of a PPO Model

Anonymous Authors

Paper under double-blind review

Abstract. Reinforcement learning in video games has become a modern approach for the game development community to integrate interactive AI, with mechanisms and algorithms such as DQN, PPO, and Soft-Actor Critic being the more popular approaches. Our work focuses on applying PPO for a 2-player 1v1 streetfighter-style PvP game developed on Unity. We have further conducted a user survey of the game and its AI to test how adaptive the agent was. The work further discusses the relevance of hyperparameters and training steps in the performance of AI agents.

Keywords: Reinforcement Learning · Game AI · PPO · ML-Agents-Toolkit

1 Introduction

Video game AI has come a long way, from the predictable patterns and rule-based schemes in old arcade games to today's smart, learning-based systems powered by neural networks. Reinforcement learning (RL) plays a key role in this shift by helping agents learn through trial and error.

In this work, we focus on training AI agents for a two-player PvP (player vs. player) game using Unity's ML-Agents Toolkit. The goal is to create agents that can make strategic decisions like closing in on opponents, attacking with good timing, and using shields to block damage. We use Proximal Policy Optimization (PPO) as the training method, allowing the agents to improve steadily through repeated matches. To evaluate the agents, we conducted user tests where players faced off against the trained models. Their feedback gave us insight into how the agents behaved at different training stages and how their difficulty evolved over time.

2 Related Works

There have been many developments in the realm of AI for game development, with [2] using genetic algorithms for AI development and [4] using RL-based methods for enhanced enemy AI to perform equal to or surpass human gameplay.

Park and Lee [5] applied PPO in a Match-3 puzzle game within Unity, achieving a 44% improvement over traditional RL. Their agents learned complex strategies without hand-coded rules, highlighting PPO's practical benefits

in casual games. Yu et al. [8] demonstrated PPO's effectiveness in cooperative multi-agent environments like StarCraft II and Google Research Football. Their results showed PPO performed robustly across tasks without heavy domain-specific tuning, suggesting PPO as a strong general baseline.

Shao et al. [7] surveyed deep RL methods including value-based, policy gradient, and model-based approaches. They also identified challenges such as balancing exploration-exploitation, improving sample efficiency, generalization, transfer learning, multi-agent coordination, and handling sparse rewards. Finally, Mekni et al. [3] compared various RL toolkits used in game development, evaluating factors such as usability, documentation, algorithm support, and integration with engines like Unity, guiding developers in selecting appropriate frameworks for adaptive NPC AI.

3 Hypothesis

Reinforcement learning models enhance their performance with an increased number of training steps. The model refines its decision-making with each incremental training step so that it becomes more interactive and has more adaptive behavior. The model optimally fine tunes its responses. This implies that more training steps let the model *exploit* a greater scope of possibilities. Players would have to engage more deeply and employ advanced strategies themselves in order to defeat the agent.

4 Proposed Methodology

4.1 Proximal Policy Optimization:

Proximal Policy Optimization (PPO)[6] is a policy gradient method designed to improve the stability and efficiency of RL training. The policy π_θ is stochastic and $\hat{A}_t$ represents an estimator of the advantage function at timestep t.

$$L^{\mathrm{PG}}(\theta) = \mathbb{E}_t \left[\log \pi_\theta(a_t \mid s_t) \hat{A}_t \right] \tag{1}$$

The expectation $\mathbb{E}_t$ denotes an empirical average computed over a finite batch of samples, following an iterative process of sampling and optimization. Equation (1) represents the surrogate loss function used in Proximal Policy Optimization (PPO). In Equation (1), $\pi_\theta(a_t \mid s_t)$ denotes the probability of selecting action a_t given state s_t. This probability is then multiplied by the advantage estimator $\hat{A}_t$, which quantifies how *good* the action a_t was compared to other possible actions. A higher value of $\hat{A}_t$ increases the probability of selecting a_t again in similar states, whereas a lower value reduces this probability. Over time, this encourages the policy to improve, leading to better agent performance.
Another improvement discussed about PPO in [6] is the L^{CPI}, where CPI stands for Conservative Policy Iteration, in which it focuses on major updates to policy rather than smaller updates. L^{CPI} is defined as

$$L^{CPI} = \hat{\mathbb{E}}_t[r_t(\theta)\hat{A}_t] \tag{2}$$

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t} \tag{3}$$

PPO also incorporates the Trust Region Policy Optimization (TRPO) concept by limiting the policy update step to prevent drastic changes, ensuring stable learning. TRPO constrains policy updates using the Kullback-Leibler (KL) divergence [1]:

$$\max_\theta \mathbb{E}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \hat{A}_t \right] \tag{4}$$

$$\text{subject to } D_{\text{KL}}\left(\pi_{\theta_{\text{old}}} || \pi_\theta\right) \leq \delta \tag{5}$$

Where D_{KL} is the KL divergence between the old and new policies, and δ is a small threshold. Additionally, PPO introduces a clipped surrogate objective to prevent excessively large updates:

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t \right) \right] \tag{6}$$

Where ϵ is a small hyperparameter that controls how much $r_t(\theta)$ can deviate from 1. This clipping mechanism ensures more stable training and prevents excessively large policy updates.

4.2 Training

Agents were trained using a self-play approach in a two-player environment, where they periodically switched teams. This allowed them to experience both roles, improving adaptability and strategic diversity. Both agents shared the same neural network architecture but followed separate policies. Starting from random strategies, they improved through repeated competition, adapting as their opponents evolved. By alternating between Player 1 and Player 2, agents learned both offensive and defensive tactics. This prevented overfitting to a single role and promoted more balanced, generalizable behaviors.

4.3 Reward Scheme

During training, agents receive rewards that help shape their behavior. Positive rewards encourage certain actions, while negative ones discourage others. As the agent interacts with the environment, these rewards accumulate step by step, gradually guiding it toward more effective strategies. Table 1a shows the various rewards assigned.

Positive rewards promote goal-oriented behavior. Moving toward a weakened enemy, for example, gives +0.1, encouraging the agent to take advantage of the situation. A hit results in +1.5, while defeating an opponent results in +2.0, both of which increase effective offense. Defensive moves, such as shielding, receive

Action/Event	Reward
Successful attack on enemy	+0.75
Defeating the enemy	+1.0
Moving closer to weak enemy (Health ≤ 40)	+0.01
Blocking an attack with shield	+0.5
Attack missed	−0.1
Taking damage	−0.5
Moving away from the enemy	−0.005
Losing an episode (timeout)	−0.5
Getting defeated	−1.0

(a) Reward Scheme for PlayerAgent

Hyperparameter	Group 1	Group 2
Trainer Type	PPO	PPO
Learning Rate	0.001	0.0001
Clipping Range (ϵ)	0.2	0.15
GAE Lambda	0.95	0.97
Epochs per Update	3	6
LR Schedule	Constant	Linear
Hidden Units	128	256
Network Layers	2	3
Entropy Coeff (β)	0.01	0.005
Batch Size	512	4096

(b) Hyperparameter Settings for PPO

Table 1: PlayerAgent Reward Design and PPO Hyperparameters

+1.0, rewarding strategic defense. Possessing more health than the opposition at the end of a match awards +1.0 (or +0.5 in a tie), encouraging survival. Negative rewards assist the agent in avoiding tactics that don't work. The penalty for backing off is -0.2, which deters passivity. Attacks that are blocked (-0.5) or missed (-0.2) indicate poor timing or aim, which motivates better execution. Taking damage or losing (-1.0) emphasizes the need for location and survival. There is no reward for certain acts, such as turning a shield on or off. This prevents the agent from abusing basic maneuvers in an attempt to score points.

5 Experiment

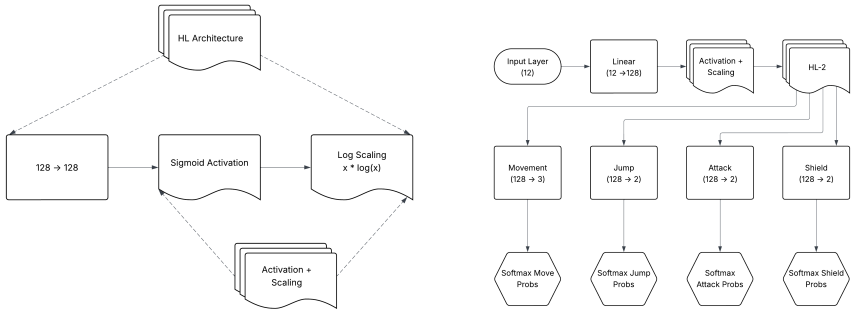
5.1 Training Phase

The agents were trained using Proximal Policy Optimization (PPO) within the Unity ML-Agents Toolkit. A self-play setup was employed, allowing agents to compete against both themselves and previous versions, fostering continual improvement and the development of robust strategies. Two PPO configurations, referred to as Group 1 and Group 2, were utilized during training. Group 1 prioritized faster policy updates by employing simpler neural networks and higher learning rates. In contrast, Group 2 focused on stability and generalization, leveraging larger networks with slower update frequencies. A detailed comparison of the key hyperparameter settings for both groups is provided in Table 1b.

5.2 Neural Network Architectures

Two types of neural network architectures were used: one with 2 hidden layers (Group 1), and another with 3 hidden layers (Group 2). The simpler 2-layer model had 128 hidden units per layer, while the 3-layer model had 256 hidden units.Fig 1b shows us an architecture with 2 hidden layers. It has 12 inputs, and the next hidden layer is of 128. Each Hidden layer has its set of sigmoid activation function and performs logarithmic scaling, which is depicted in fig 1a.

The output of the network is in terms of probabilities with softmax activation functions, in terms of what action to be performed.



(a) Architecture of each Hidden Layer

(b) Architecture of a 2-Layer Neural Network

Fig. 1: Comparison between hidden layer design and overall neural network architecture

5.3 Trained Agent Groups

The final models were grouped into two groups based on their hyperparameters and training durations. Group 1 had a total training step of 750K, and 3 models were generated from this, with checkpoints at 250K (Model B), 500K (Model A), and 750K (Model C). Group 2 similarly was trained for 1.2M steps, with checkpoints at 750K (Model E), 900K (Model D), and 1.2M (Model F).

Group 1 agents were trained using a smaller batch size, simpler visual encoders, and fewer steps. These agents tended to behave more aggressively, often engaging the opponent quickly. Group 2 agents were trained with larger networks and more data. They adopted a more strategic approach, using movement and defensive tactics to engage opponents. Both groups used self-play with different configurations. Group 1 used a swap window of 5 and swap steps of 1000, while Group 2 used a window of 10 and swap steps of 2000.

6 Results

6.1 Results for AI vs AI

The following table 2 provides statistics of the AI vs AI part of the experiment, where two models were taken and were made to fight against each other. The models were given titles B as in blue and R as in Red, and their statistics were collected after making them battle for approximate 100 episodes.

Light model represents a model of group 1, i.e. a 2-hidden-layered neural network and Heavy model represents a model of group 2 i.e. a 3-hidden-layered

neural network. The steps columns represent the amount of steps the models had been training for and are to be understood in 1000s, i.e. 100 represents 100,000 steps. The columns for death represent the number of deaths for the Blue side and the Red side, and on that basis, the win rate had been calculated.

Difficulty_B	Difficulty_R	Steps_B	Steps_R	Death_B	Death_R	Total Episodes	Win Rate B	Win Rate R
Light	Light	100	250	82	28	110	25.45%	74.55%
Light	Light	250	750	102	0	102	0.00%	100.00%
Light	Light	250	500	66	34	100	34.00%	66.00%
Light	Light	100	500	104	10	114	8.77%	91.23%
Light	Light	100	750	102	8	110	7.27%	92.73%
Light	Light	500	750	83	23	106	21.70%	78.30%
Light	Heavy	500	500	4	105	109	96.33%	3.67%
Heavy	Heavy	450	700	73	31	105	29.52%	69.52%
Heavy	Heavy	500	700	69	34	104	32.69%	66.35%
Heavy	Heavy	500	900	72	28	100	28.00%	72.00%
Heavy	Heavy	700	900	47	56	103	54.37%	45.63%
Heavy	Heavy	700	940	78	37	115	32.17%	67.83%
Heavy	Heavy	700	1200	70	50	120	41.67%	58.33%
Heavy	Heavy	500	1200	76	25	103	24.27%	73.79%
Heavy	Heavy	900	1200	76	42	118	35.59%	64.41%

Table 2: Data collected from over 100 episodes for different models fighting against each other.

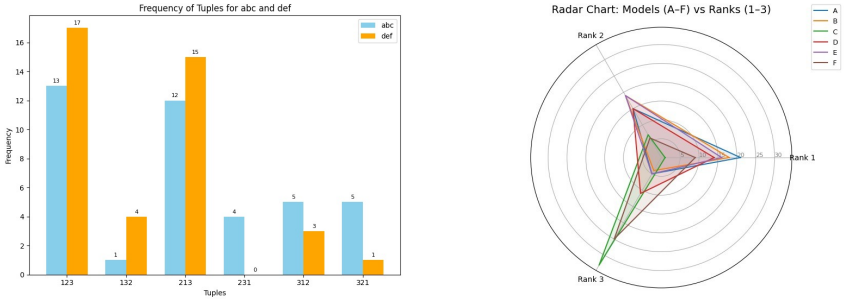
From the Table 2, we can verify our hypothesis that if two such agents were simulated to fight against in an environment, the model with more training steps will outperform the model with fewer training steps. One key observation made here that two models with varying hyperparameters and neural network architecture cannot be compared in terms of training steps, as they have different utilizations and completion of their underlying architecture, and even at same number of steps, a model that has nearly exhausted its network will perform better than a model with a larger architecture.

An anomaly present in the table is when Heavy models, i.e. models of group 2 fought against each other with 700k and 900k steps respectively and the model with 700k performed ever so slightly better than the model with 900k steps. This maybe due to plateauing of the 900k model.

6.2 Results for AI vs Players

All of the subjects were asked to rank the models in both groups, with 1 being the weakest model and 3 being the strongest model. It is evident from the feedback that Model A and Model B were very close in performance. Model A had an average rank of 1.73 and Model B had 1.75, indicating only a slight difference. This marginal gap may have caused ambiguity among participants when ranking the models. While Model B was actually the weakest, its perceived difficulty was rated at 3.73, which is close to Model A's 4.15. This suggests that participants found it challenging to distinguish between them based on performance. Model C stood out as the most difficult with a difficulty rating of 4.66 and also had the highest average rank of 2.88. This aligns with the observation that it was indeed the strongest model in Group 1.

Now coming to models from Group 2, the statistics more clearly reflect the actual performance hierarchy. Model E, with the lowest average rounds won (2.56) and an average rank of 1.73, was correctly perceived as the weakest. Model F was rated moderately difficult at 3.70, had an average rank of 2.25, and achieved higher average rounds won at 3.41, indicating that it was both challenging and effective. Model D had an average rank of 1.93 and a win rate of 3.24, placing it in the middle. Overall, the collected feedback and rankings align with our original hypothesis that models trained for more steps generally performed better and were perceived as stronger.



(a) Tuple Rank Comparison (b) Radar Plot of Feedback

Fig. 2: Visual overview of tuple-based rank evaluation and user feedback.

The distribution of ranks assigned to each model by participants is illustrated in Fig. 2b. Models C and F received Rank 3 most frequently, indicating they were perceived as the most difficult in their respective groups. Among the Rank 1 choices, Model A was more commonly selected than Model B; however, this preference may be influenced by participants' initial assumptions that the models were ordered by difficulty. In Group 2, Model E emerged as the most frequently chosen easiest model, likely due to its minimal training steps, followed closely by Model D.

To analyze the popularity of different rank tuples assigned by participants, Fig. 2a presents the distribution of rank combinations across both model groups. Each tuple, such as [1, 2, 3], corresponds to the ranks given to models [A, B, C] or [D, E, F], respectively. The tuples [1, 2, 3] and [2, 1, 3] emerged as the most common, with [1, 2, 3] slightly more prevalent. This preference may reflect a cognitive bias, where participants assumed the models were presented in increasing order of difficulty. Notably, across both groups, the third model (Model C or F) was most consistently identified with Rank 3. However, some ambiguity remained between Ranks 1 and 2—specifically between Models A and B in Group 1, and D and E in Group 2.

Lastly, the subjects were asked to rate which set was more difficult to play against, and the subjects voted with a majority that the set of models (A, B, C), i.e., Group 1, was difficult to play with. This is because the models under Group

2 were not even near their convergence and might need up to 5-7M training steps to almost completely exhaust their network.

7 Conclusion

We leveraged the properties of PPO and used a reinforcement learning scheme to successfully create an AI that can adapt to player strategies and create strategies of its own. We deployed these models into a game and tested them by conducting a user survey of forty participants, and we proved the hypothesis that we made that if a model had been trained for a greater number of steps, then it will outperform the model trained with fewer steps, given that both the models haven't converged and that the difference in the number of steps is significant.

References

1. Kullback, S., Leibler, R.A.: On information and sufficiency. The Annals of Mathematical Statistics **22**(1), 79–86 (1951). https://doi.org/10.1214/aoms/1177729694
2. Martinez-Arellano, G., Cant, R., Woods, D.: Creating ai characters for fighting games using genetic programming. IEEE Transactions on Computational Intelligence and AI in Games **PP**, 1–1 (12 2016). https://doi.org/10.1109/TCIAIG.2016.2642158
3. Mekni, M., Jayaramireddy, C.S., Naraharisetti, S.V.V.S.S.: Reinforcement learning toolkits for gaming: A comparative qualitative analysis. Journal of Software Engineering and Applications **15**, 417–435 (2022). https://doi.org/10.4236/jsea.2022.1512024, https://doi.org/10.4236/jsea.2022.1512024
4. Oh, I., Rho, S., Moon, S., Son, S., Lee, H., Chung, J.: Creating pro-level ai for a real-time fighting game using deep reinforcement learning (2020), https://arxiv.org/abs/1904.03821
5. Park, D.G., Lee, W.B.: Design and implementation of reinforcement learning agent using ppo algorithm for match 3 gameplay. Journal of Convergence for Information Technology **11**(3), 1–6 (Mar 2021). https://doi.org/10.22156/CS4SMB.2021.11.03.001
6. Schulman, J., Wu, Y., Donahue, J., Dhariwal, P., Ashfordin, A.: Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347 (2017), https://arxiv.org/abs/1707.06347
7. Shao, K., Tang, Z., Zhu, Y., Li, N., Zhao, D.: A survey of deep reinforcement learning in video games. IEEE Transactions on Computational Intelligence and AI in Games **12**(1), 1–26 (2020). https://doi.org/https://doi.org/10.48550/arXiv.1912.10944
8. Yu, C., Velu, A., Vinitsky, E., Gao, J., Wang, Y., Bayen, A., WU, Y.: The surprising effectiveness of ppo in cooperative multi-agent games. In: Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., Oh, A. (eds.) Advances in Neural Information Processing Systems. vol. 35, pp. 24611–24624. Curran Associates, Inc. (2022)