A comparative analysis of different training methods and algorithms for 1v1 RL environments

Wassim Jabbour (260969699) - Enzo Benoit-Jeannin (260969262) - Justin Novick (260965106)

Abstract

This study consisted of exploring the two main methods for training 1v1 agents, expert training and selfplay, on the SlimeVolleyball gym environment. The algorithms implemented are PPO, DDQN, A2C, and a genetic approach. By ranking the agents using their ELO scores, we are able to conclude that selfplay leads to more flexible agents, and that PPO is the easiest to train out of the RL agents. We also observed the evolution of the selfplay strategies across generations to better understand the way agents learn an optimal strategy.

1 Introduction

Player-against-player games are one of the most prominent forms of software entertainment that exist today. Implementing a 1v1 AI agent that generalizes well against any style of play can be a challenging task. In this project, we explore different training methods that can be used for training such agents. We implemented different algorithms to use as comparison points: First, we engineered our own versions of PPO (1) and DDQN (2) from scratch, including multiple optimizations such as entropy regularization and prioritized replay buffers (3). We also used the Stable Baselines 3 RL library to train an A2C (4) agent as a comparison point. For each of these three algorithms, we attempted two training methods: Expert training, which consists of training directly against an expert baseline, as well as self-play (5), which consists of training the model against previous versions of itself. Furthermore, we also implemented a basic genetic algorithm (6) from scratch as a comparison point. Our analysis was threefold: First, we analyzed the stability of the different algorithms and training approaches. Then, we evaluated the trained algorithms against each other. Finally, we studied the evolution of the self-play agents' behavior across generations.

2 Background

2.1 The SlimeVolleyball gym environment

The SlimeVolleyball gym environment (7) consists of a 1v1 game with the objective of landing the ball on the ground of the opponent's side. Each agent starts the game with five lives, and landing the ball on the opponent's side causes them to lose a life. The agent receives a reward of +1 when its opponent loses a life or -1 when it loses a life. The state space consists of a 12-dimensional vector containing the absolute coordinates of the agent, the ball, and the opponent, all concatenated for two consecutive frames so that the agent can determine the direction of movement of the ball. The action space is characterized by 3 binary branches of movement: forward motion, backward motion, and jumping motion. For instance, the combination [1, 0, 1] indicates the simultaneous execution of forward and jump actions.

2.2 Training methods for 1v1 environments

2.2.1 Expert training

The first way to train an agent in a 1v1 game is to load an "expert" baseline policy that can already play the game to a satisfactory degree, and consider it to be part of a stationary environment. Our agent will then have to learn to beat the baseline by heavily exploring until it stumbles upon a way to stop losing every game. This approach presents multiple challenges, including the difficulty of learning to beat an expert opponent when

receiving little to no positive rewards, as well as the issue of overfitting the expert's style of play. Furthermore, finding an expert baseline that is already good at the game isn't always possible.

2.2.2 Self-play

Another way a 1v1 agent can be trained is through self-play (5), which is a dynamic training methodology in which the agent competes against a previous copy of itself. Initially, both models start out as random, with the opponent being frozen during matches. When the agent being trained becomes better than its static opponent, we update the latter to be a more recent version of the former, starting a new "generation". This cycle of training, evaluation, and updating continues, progressively refining the agent's abilities through iterative self-competition. This approach should hypothetically be easier to train as the opponent always matches the agent's skill level, as opposed to training directly against an expert. Another benefit of self-play is that the agent should learn to play against a wide variety of opponents since every subsequent generation should learn play-styles that were the weak points of the previous generation.

2.3 Algorithms & Optimizations

2.3.1 PPO optimizations

Following the advice in (8), we added a batch normalization for the advantages in order to reduce the training variance and converge faster. We also implemented learning rate annealing (9) which consists of linearly decaying the learning rate from its initial value to zero as the training progresses, in order to maximize our learning rate range and make the model more generalizable. We also included entropy regularization by substracting the entropy of the actor's predicted probability distribution from its actor's loss function. This modification should encourage more exploration, as it leads the backpropagation process to increase the entropy of the predicted probabilities. Finally, we also estimated the KL divergence between the new and old policy distributions every iteration using the approach in (10), and stopped the training iteration early if it was larger than a pre-defined hyperparameter to avoid taking steps that are too large in the wrong direction.

2.3.2 DDQN optimizations

The problem with vanilla DQN is that, every training step, the update target shifts when the Q-function changes, leading to a lot of instability during training. This is why we implement Double DQN, using a frozen target network to calculate the targets, and only updating it every K iterations (2). Another problem of DDQN is that sampling the buffer uniformly leads to a very low chance of selecting important transitions. This is why we implemented prioritized experience replay (3), where the sampling probability for a step becomes proportional to its TD-error.

2.3.3 Genetic algorithm

In addition to the three models above, we implemented a very simple genetic algorithm: 128 MLPs are first randomly initialized. Then, for 500000 consecutive iterations, the algorithm randomly picks 2 agents, plays them against each other, and copies the winner's weights into the loser, followed by adding some random Gaussian noise. Incredibly enough, this process, inspired by Charles Darwin's theory of evolution, leads to great players without the need for any hyperparameter tuning.

3 Related Work

We mainly attempted to reproduce the results obtained in the training results of the SlimeVolleyGym github repository (7), using a more diverse set of baselines as well as implementing some of the algorithms from scratch. The baseline used with the package that was used for the expert training is an RNN that was trained through an "arms race" genetic approach (11) where only the top 20% of the population is kept every generation while the rest is mutated.

4 Methodology

4.1 Training methodology

Aside from the genetic approach (which is trained through self-play only by definition), all other algorithms were trained through both expert training and self-play, and for a multitude of hyperparameter combinations.

This dual method enabled us to evaluate the effectiveness and adaptability of each algorithm under varied conditions. We also logged a multitude of metrics, include episode lengths, as well as performance against random and baseline opponents throughout every run.

4.2 Evaluation methodology

After training all our agents, we picked the best one for each algorithm, and ran an evaluation phase during which every pair of agent played 1000 games against each other, including the expert and random baselines. Average scores were then extracted, and ELO ranking scores (12) were computed, starting at 1200 ELO points for all algorithms.

5 Experiments

5.1 Training results

We preface this section by stating that our DDQN agent was very slow to train, which meant its training could not be finished on time. The test return plots for the first 10M steps for that algorithm can be found in Figure 4 in the Appendix, along with the training curves of our genetic algorithm in Figure 5.

When it comes to hyperparameter tuning, training the models required a number of steps ranging from 20M all the way up to 50M, with each being trained through selfplay and expert training. For this reason, we could not try a variety of hyperparameters, and instead had to set most of them to a fixed value after trial and error: For PPO, we used a learning rate of 0.0003 with an ADAM optimizer (13), a loss clipping value of 0.2 as recommended by the paper (1), a KL divergence threshold of 0.03, a total of 20M training steps, and tried 2 different entropy coefficient values: 0 and 0.1. As for A2C, we used a learning rate of 0.0007 with an RMSProp optimizer, a gradient clipping value of 0.5, an entropy coefficient of 0.1, and a total of 50M training steps.

We evaluated all our models against the expert at periodic intervals during training. Figure 1 displays the test returns against the expert on the left, along with the evolution of the training episode lengths on the right.

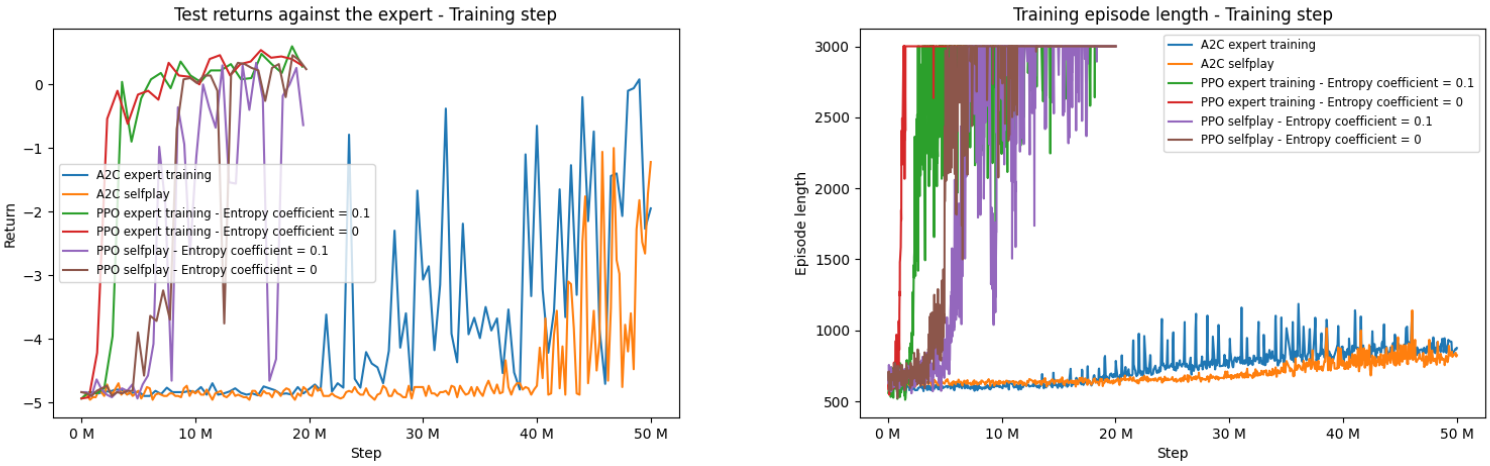

Figure 1: Test returns (Left) and training episode length (Right) as a function of the training step

First, we observe that the PPO agents learned much faster than their A2C counterparts, with the former only requiring around 20M steps to become as good as the expert, while the latter needed around 50M steps. Furthermore, we can see that an entropy coefficient of 0.1 was best for PPO expert training, while a value of 0 was preferable for PPO selfplay. Finally, on the right, we can see that the PPO agents were much stable during training, reaching the maximum episode length of 3000, while the A2C agents only performed well during evaluation. As further proof that A2C learned properly, we Figure 3 in the Appendix proves that A2C test episodes reaching maximum length.

Furthermore, Figure 2 on the left shows the evolution of test returns as a function of the generation number for the self-play algorithms. We again see that the PPO algorithms required a lower number of generations to become as good as the baseline. Furthermore, the figure on the right displays a very interesting result: While both the self-play and expert training variants of PPO improved to be just as good as the expert (See Figure 1 -

left), we see that the expert training versions forgot how to beat a random opponent past 2M steps, showing a sign of overfitting the expert's style of play. This shows the superiority of using self-play when compared to training directly against an expert, as an agent trained through the former needs to be more flexible to counter the multiple versions of itself it faced throughout training.

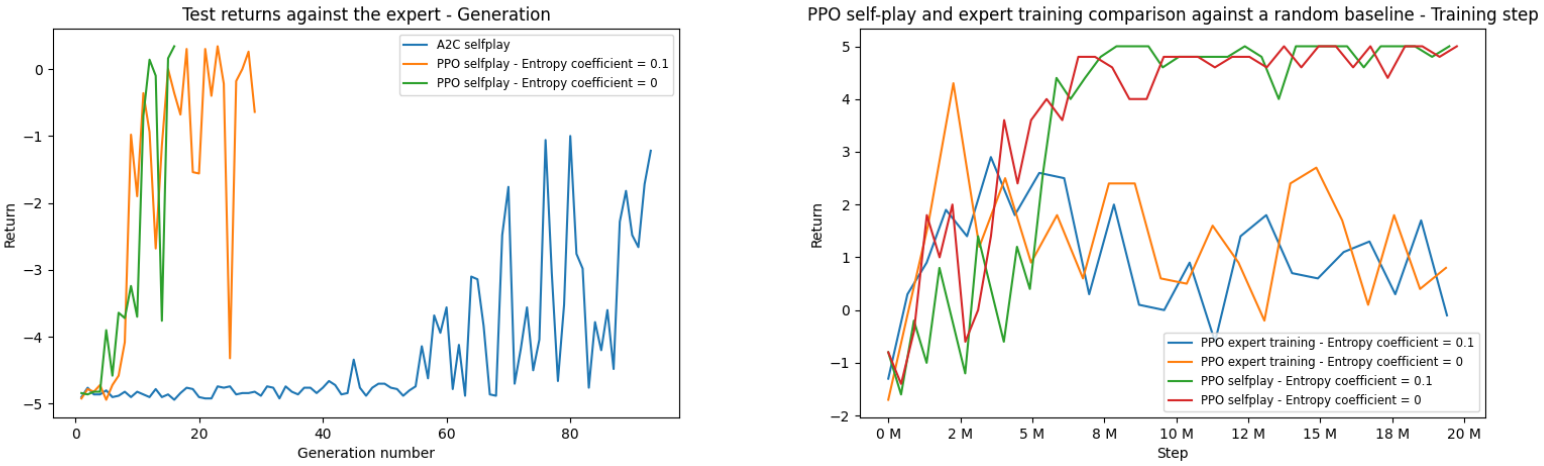

Figure 2: Test returns against the expert as a function of the generation number (Left) and PPO self-play and expert training comparison against a random baseline throughout training (Right)

5.2 Evaluation results

To evaluate the models, we loaded the best agent of each type by finding the training step at which its test returns were maximal. We also included the random and expert baselines as comparison points, and played 1000 episodes across each pair of agents for a total of 21000 episodes. We were then able to derive Table 1 for the average return across every pair of agents, as well as the ELO scores calculated following (14) in the last column.

Table 1: Mean pairwise returns with standard deviation (Row against column), and calculated ELO scores

	Genetic	PPO ET	PPO SP	A2C ET	A2C SP	Random	Expert	ELO
Genetic	-	4.89 +- 0.35	0.08 +- 0.80	4.24 +- 1.09	1.21 +- 1.08	4.93 +- 0.27	0.38 +- 0.89	1493
PPO ET	-4.89 +- 0.35	-	-3.60 +- 1.43	2.63 +- 2.01	0.92 +- 0.93	0.47 +- 2.77	0.44 +- 0.71	1253
PPO SP	-0.08 +- 0.80	3.60 +- 1.43	-	3.82 +- 1.30	0.91 +- 0.95	4.74 +- 0.54	0.39 +- 0.86	1437
A2C ET	-4.24 +- 1.09	-2.63 +- 2.01	-3.82 +- 1.30	-	0.68 +- 0.95	0.96 +- 2.70	0.10 +- 0.89	1054
A2C SP	-1.21 +- 1.08	-0.92 +- 0.93	-0.91 +- 0.95	-0.68 +- 0.95	-	4.53 +- 0.75	-0.84 +- 1.09	1092
Random	-4.93 +- 0.27	-0.47 +- 2.77	-4.74 +- 0.54	-0.96 +- 2.70	-4.53 +- 0.75	-	-4.88 +- 0.34	827
Expert	-0.38 +- 0.89	-0.44 +- 0.71	-0.39 +- 0.86	-0.10 +- 0.89	0.84 +- 1.09	4.88 +- 0.34	-	1243

We observe that, in terms of ELO score, the genetic algorithm performs the best, followed closely by PPO selfplay. Furthermore, both selfplay algorithms outperform their expert training counterparts, as expected. Finally, we can see that the ET algorithms clearly overfit the style of play of the expert, matching the latter but failing to beat a random baseline. All these results reinforce the conclusion that selfplay increases flexibility, and is best when paired with policy gradient methods such as PPO.

5.3 PPO self-play evolution

We observed the evolving behavior of the PPO selfplay agent across generations by rendering and analyzing games where each generation competes against itself, which are displayed in the accompanying video. For instance, by the sixth generation, agents had learned to bounce the ball against the wall. By the fourteenth generation, they demonstrated significantly enhanced skills in tracking the ball and the ability to keep the ball in play by bouncing it repeatedly without it touching the ground, although the time taken to return the ball was still lengthy. By the twenty-second generation, there was a noticeable increase in the pace of play, with agents sending the ball back and forth in a timely manner. By the twenty-ninth generation, some agents began using the walls more strategically, employing them to accelerate the ball and add complexity to their gameplay. These observations highlight a clear progression in tactical depth and responsiveness as the generations advanced.

6 Conclusion and Future Work

Overall, this study demonstrates the effectiveness of self-play coupled with traditional RL algorithms. Self-play drastically outperformed expert training in terms of flexibility, as the latter promoted an overly specific style of play. With respect to the algorithms, the best results were obtained using PPO due to the various optimizations implemented. The second-best performer was A2C, followed by DDQN which was very slow to train. The superior training speeds of PPO and A2C over DDQN highlighted the efficiency of using policy-based approaches. As a follow-up, we would like to try more hyperparameter combinations, and even study how changing the rules of the game affects behavior. As a concluding note, the work was distributed equally across all team members.

7 Appendix

Figure 3: A2C episode length showing the algorithm learned

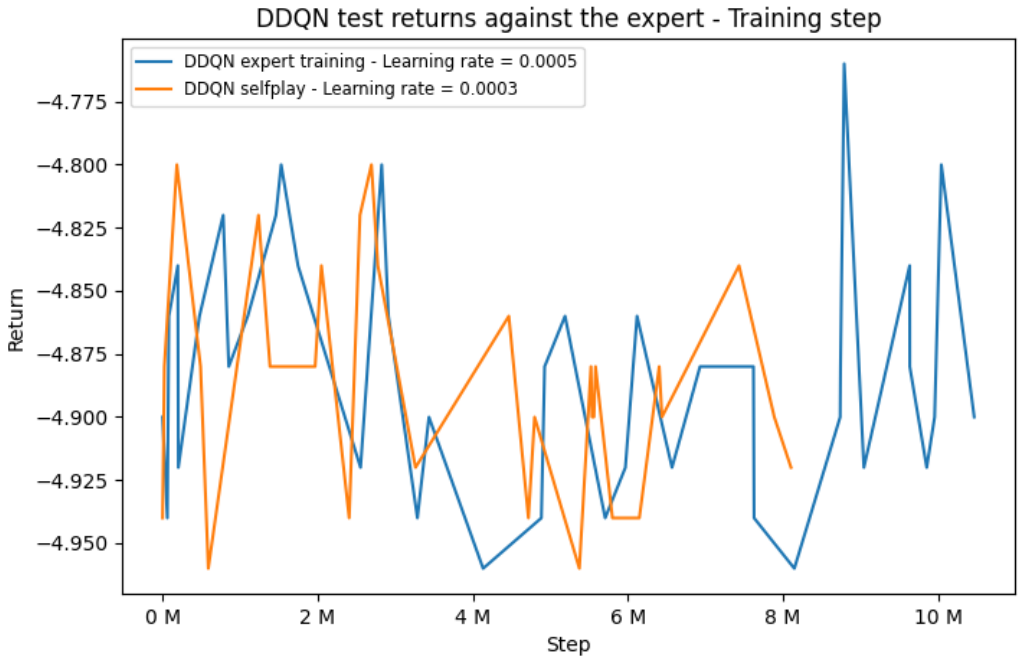

Figure 4: DDQN failed training runs due to executing too slowly

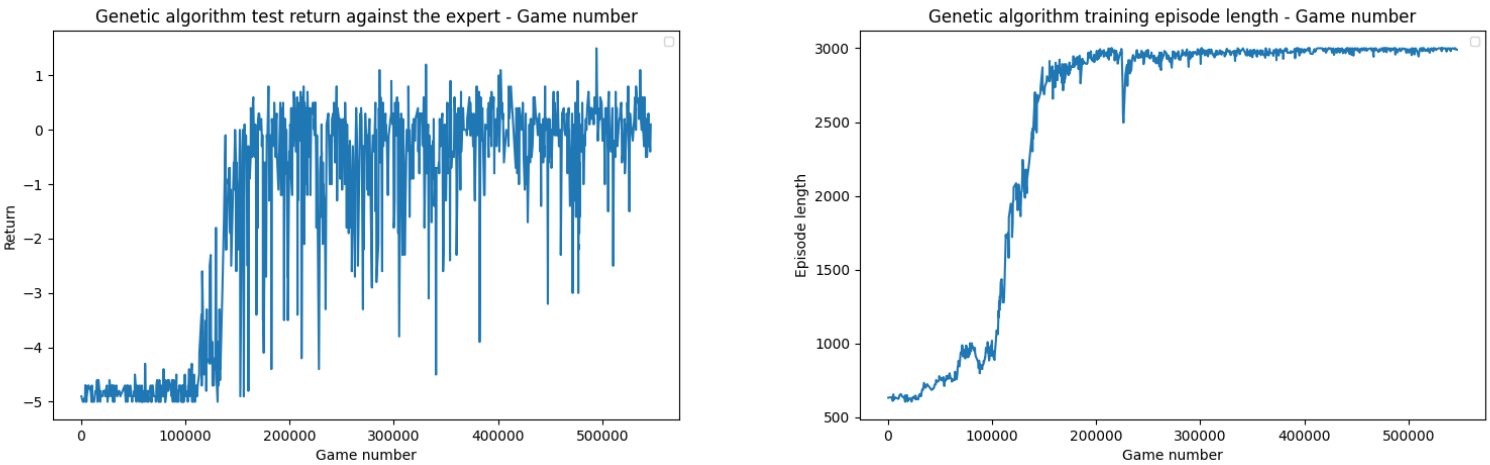

Figure 5: Test returns (Left) and training episode length (Right) as a function of the training step for the genetic algorithm

References

[1] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017. [Online]. Available: http://arxiv.org/abs/1707.06347

[2] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *CoRR*, vol. abs/1509.06461, 2015. [Online]. Available: http://arxiv.org/abs/1509.06461

[3] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," 2016.

[4] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," 2016.

[5] A. DiGiovanni and E. C. Zell, "Survey of self-play in reinforcement learning," 2021.

[6] S. Thede, "An introduction to genetic algorithms," *Journal of Computing Sciences in Colleges*, vol. 20, 10 2004.

[7] D. Ha, "Slime volleyball gym environment," https://github.com/hardmaru/slimevolleygym, 2020.

[8] E. Y. Yu, "Coding PPO from Scratch with PyTorch (Part 1/4) — medium.com," https://medium.com/analytics-vidhya/coding-ppo-from-scratch-with-pytorch-part-1-4-613dfc1b14c8, [Accessed 15-04-2024].

[9] P. Nakkiran, "Learning rate annealing can provably help generalization, even for convex problems," 2020.

[10] "Approximating KL Divergence — joschu.net," http://joschu.net/blog/kl-approx.html, [Accessed 15-04-2024].

[11] D. Ha, "Neural slime volleyball," *blog.otoro.net*, 2015. [Online]. Available: https://blog.otoro.net/2015/03/28/neural-slime-volleyball/

[12] Wikipedia contributors, "Elo rating system — Wikipedia, the free encyclopedia," 2024, [Online; accessed 15-April-2024]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Elo_rating_system&oldid=1218973913

[13] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.

[14] "Self-Play: a classic technique to train competitive agents in adversarial games - Hugging Face Deep RL Course — huggingface.co," https://huggingface.co/learn/deep-rl-course/en/unit7/self-play, [Accessed 17-04-2024].