Training an agent to play Breakout using Reinforcement Learning

Gabriel C. Ullmann

April 11, 2022

Abstract

In this project, I use three Reinforcement Learning algorithms (PPO, A2C, and DQN) to train an OpenAI Gym agent to play the game Breakout. Agents were trained using different combinations of algorithms, training steps, and reward approaches to determine which one reaches the maximum average score and number of lives in the game. On average, the best-performing agents were those trained with the PPO algorithm, in a range of 100K to 1M steps and a *break-and-follow* or *follow* reward approach. By observing the agents trained with this combination, I concluded that they not only learn the basic strategy to succeed on the game, but also how to improve it and score higher.

1 Introduction

Over the last few years, research has been done on applying Reinforcement Learning agents to play video games [2] [3] [6]. While experiments frequently use different sets of algorithms to train the agents over millions of steps, they do not usually compare the use of different reward functions in conjunction with these algorithms, neither they describe how this influences agent behavior. In this project I test different combinations of algorithms, training steps and reward approaches to answer the following questions:

- RQ1: Which RL algorithm makes the agent perform the best?
- RQ2: Which reward approach makes the agent perform the best?
- RQ3: Which combination of algorithm and reward makes the agent perform the best while requiring the least amount of training steps?

In the context of these research questions, the word "performance" means how skillfully the agent is able to play the game. Therefore, if the agent is able to complete gaming sessions with as high a score as possible while losing as few lives as possible, it is performing well.

2 Methodology

I took two projects available on GitHub as starting points for my project: Mustafa Abushark's implementation of an OpenAI Gym agent to play the game Snake [1], and a Breakout implementation made with the PyGame library by John Cheetham [2]. I also used GitHub to manage my own code for this project [3].

On Abushark's code, the game and agent logic are coupled, which makes the code harder to read and maintain. Therefore, after replacing the original Snake code with Breakout, I separated the concerns into two classes: BreakoutAgent (agent) and BreakoutGame (game). Both classes work in conjunction via the Observer pattern: at every step of execution, the game notifies changes to its current state (e.g position of the ball and bat, score, etc.) to the agent, which uses these observations to learn and choose its next action.

2.1 Algorithms and training steps

Then I proceeded to compare three algorithms: PPO, A2C, and DQN. These were chosen because they have

[1] https://github.com/laserany/snake-ai-model
[2] https://github.com/johncheetham/breakout
[3] https://github.com/nkinesis/comp6321-project

been applied successfully to Breakout by other researchers [1] [4] [5]. Agents were trained with these algorithms in five different amounts of steps: 10K, 50K, 100K, 500K, and 1M. Since Abushark's approach reached good results by training with 100K steps, I decided to test with 2 ranges of values above and below this value in order to observe how they affect agent performance. All agents were trained with the algorithms default hyperparameters as described at StableBaselines3 documentation [4]. This approach was chosen by two reasons: 1. There are several hyperparameters to test, and it would be hard to understand which are more worth exploring without first testing the defaults, plus the different combinations of rewards. 2. The three algorithms do not share the same hyperparameters, which would make it hard to make subsequent comparisons. The same idea was applied when choosing policies. While StableBaselines3 provides six policies, plus the possibility of implementing a custom one, I chose MlpPolicy, which implements an actor-critic architecture using a MLP with 2 layers and 64 neurons. This policy has also been applied successfully in the aforementioned research, including Abushark's.

2.2 Reward functions

In Breakout, the player hits the ball with a bat with two goals in mind: keeping the ball from touching the bottom of the screen, while also making it hit and break as many blocks as possible. Initially, however, it is not clear which of these actions has the largest influence on the agent's success.

My initial hypothesis is that rewarding the agent for following the ball is enough to reach reasonable performance, as hitting the blocks is only a consequence of this behavior. On the other hand, rewarding the agent for scoring (breaking blocks) has been the approach of choice in all aforementioned research, and it was successful. To understand which approach is the best and therefore answer RQ2, three combinations were tested: rewarding the agent by breaking blocks only, by following the ball only, or by doing both. I will thereby reference these approaches as *break*, *follow* and *break-and-follow* respectively.

[4] https://stable-baselines.readthedocs.io/en/master/modules/ppo2.html

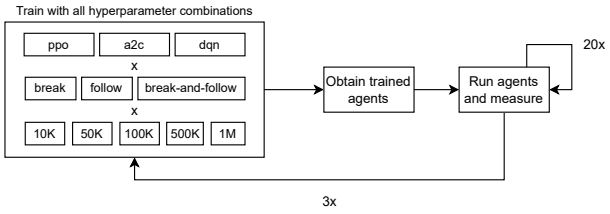

Figure 1: Agent training and testing flow

2.3 Measuring agent performance

I wrote a script to train the agents with all combinations of algorithms, steps and rewards. After trained, each agent was tested in 20 game sessions lasting 30 seconds and its score/lives were recorded. The session may end prematurely if the agent manages to lose five lives (therefore reaching a "game over" state). The ball spawns on a random position along the x-axis on every play through to assure the agent is not repeating the same actions at the start of every iteration, but effectively learning to adapt to a changing environment.

Since randomness plays an important role in training, the training/testing process was repeated 3 times as a way to assure no agent was performing well or badly merely by chance. Figure 1 describes the iterations of this process.

I set up StableBaselines3 to save logs during training that could later be visualized in Tensorboard. An analysis of this data is provided on Section 3.2.

As a baseline, the score and number of lives achieved by a dummy agent across 60 game sessions of 30 seconds were also recorded. This data is also helpful to assure the agents are learning to play, and not only taking actions by chance. This number of sessions was chosen because it is the same number of times each trained agent will be tested (3 rounds of training * 20 tests after each round).

3 Experimental Results

RQ1: best algorithm The agents trained with the PPO algorithm were the best performing on average, followed by A2C and DQN (Figures 3b and 3d). The same trend can be seen if we consider the absolute scores (Table 1). From the top 10 scores, five were obtained by agents

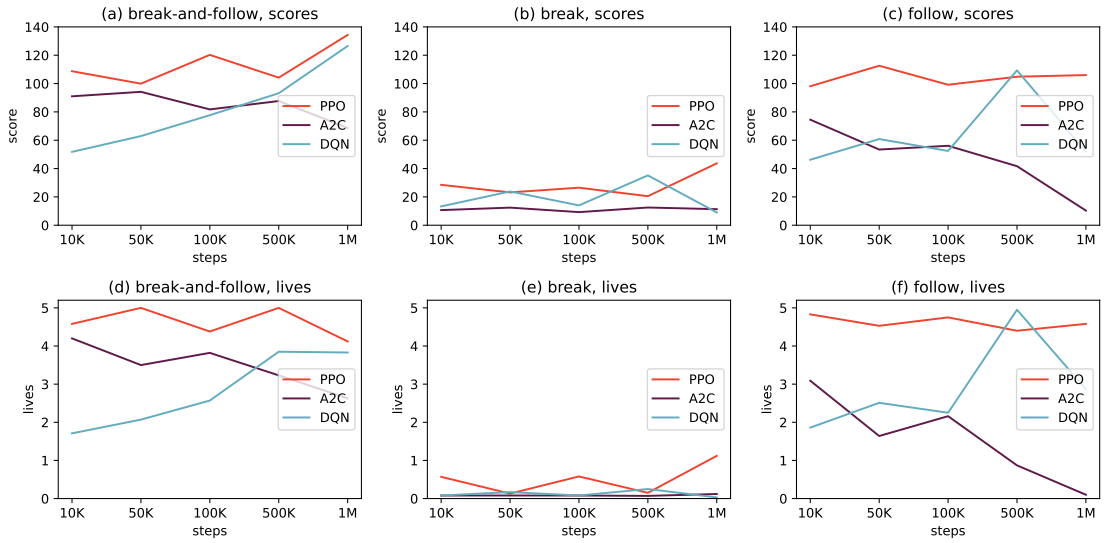

Figure 2: Average scores (top) and lives (bottom) obtained by training with each combination of reward approaches, algorithms and steps.

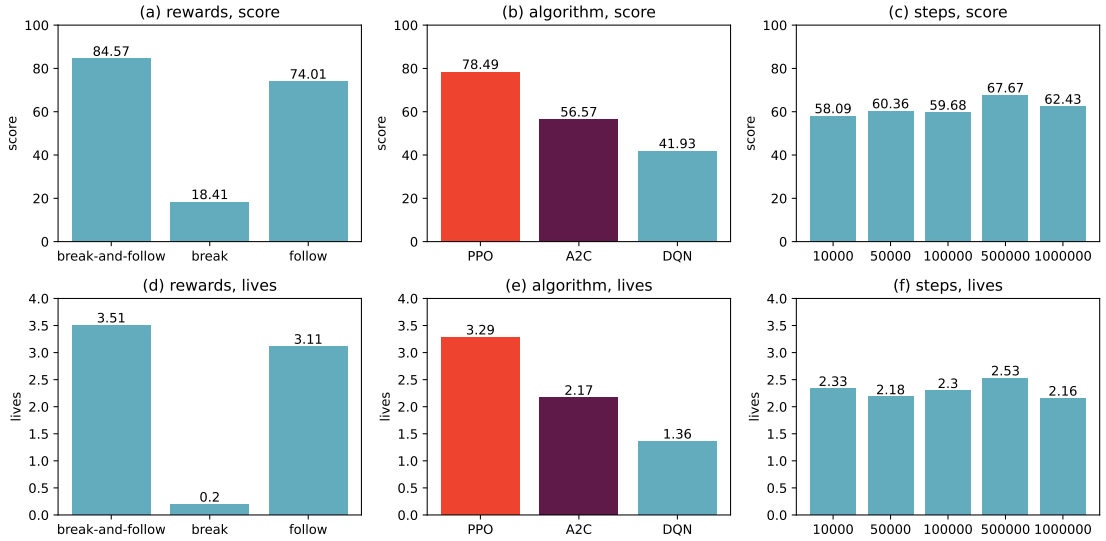

Figure 3: Average scores (top) and lives (bottom) grouped by reward approach (a, d), algorithm (b, e) and number of steps (c, f).

#	Algorithm	Steps	Reward	Score	Lives
1	AC2	10K	BF	390	5
2	A2C	50K	F	360	5
3	DQN	1M	BF	310	5
4	DQN	1M	BF	290	5
5	PPO	50K	F	280	5
6	PPO	100K	BF	280	5
7	PPO	500K	BF	280	5
8	PPO	1M	BF	280	5
9	A2C	10K	F	270	5
10	PPO	1M	BF	270	5

Table 1: Top 10 absolute scores and lives obtained by trained agents. On the reward column: B = Break, F = Follow, BF = Break-and-follow

trained with PPO, and they managed to reach this result without losing any lives. While both A2C and DQN also appear at the top of the table, the agents trained with these algorithms did not perform as well as PPO on average.

RQ2: best reward approach As shown in Figures 3a and 3d, the agents trained with *break-and-follow* and *follow* rewards obtained similar averages, but *break-and-follow* performed slightly better. The best absolute combination of score/lives obtained with a *break* reward was 250 points and 3 lives, which did not make to the top 10. By comparing Figure 3e with the others it can also be observed that the *break* approach performed much worse than its counterparts.

RQ3: best combination of algorithm, reward and steps Figure 2 shows the average score and number of lives obtained for each combination of algorithm, number of steps, and reward approach.

Agents trained with the PPO algorithm performed consistently well in all scenarios, especially when trained with a *break-and-follow* reward and with a number of steps between 100K and 1M. Figures 3c and 3f show this range of steps is indeed the one with the highest averages.

While A2C and DQN performed similarly, DQN agents reached equal or higher average scores and lives than their A2C counterparts when trained with more than 100K steps. DQN agents even surpassed PPO on average when trained with 500K steps with a *follow* reward. However, since the objective is to determine which agent performs the best with the least amount of training, the combination of PPO + 100K-1M steps + *break-and-follow* reward is still the optimal combination.

3.1 Comparison to baseline

The "dummy" agent obtained an average of 13.33 points and 0.05 lives across 60 game sessions. Regardless of the tested combination, all trained agents performed better than the baseline on average, even though many of them have hit zero lives, especially when they have been paired with a *break* reward approach (Figure 2e).

3.2 Influence of reward approaches on agent behavior

The agents trained with *break-and-follow* and *follow* rewards were the best performing, respectively. Therefore, it confirms the hypothesis I proposed in Section 2.2: the agent indeed learns well when rewarded by following the ball, even though it learns even better when a score reward is put to work together with this system.

Considering the Tensorboard logs obtained from training with a *break-and-follow* reward, we can visualize an increase on both episode duration (Figure 4a) and average reward (Figure 4b) as the number of steps increases. This indicates the agent is both surviving longer and also scoring more when trained with more steps. Further remarks about Tensorboard logs on training loss can be found in Appendix A.

As for the actions, I could observe that when trained either with a *break-and-follow* or *follow* reward, the agents learned how to follow the ball reasonably well with as little as 10K steps (with PPO), usually hitting the ball with the center of the bat. As they go through more training steps, eventually they learn that hitting the ball with the corners of the bat makes it go faster, and therefore score higher in less time. On the other hand, when trained with a *break* reward the agent moved without following any recognizable pattern. I comment further on possible reasons for this behavior in Section 4.

4 Related work

In this section, I will summarize academic work related to RL applied to Breakout and other Atari games, while also comparing it to my own.

Regarding the choice of algorithm, there seems to be no consensus. When comparing DQN and A3C (a variation of A2C), Jeerige et al. [4] claim that A3C is "much better than DQN in terms of training time, stability and higher score". Adamski et al. [1] used four variations of the A3C algorithm to train agents to play six Atari games, including Breakout. The most basic version of the A3C algorithm was the best performing algorithm, reaching the highest score with the least training time.

On the other hand, Mnih et al. [5], the creators of DQN, mention it was created with stability in mind and also that it was able to learn the "optimal strategy" for Breakout, "which is to first dig a tunnel around the side of the wall allowing the ball to be sent around the back to destroy a large number of blocks". However, reaching this level of skill on the game does not only depend on the algorithm itself but also reward approach, policy, batch normalization and hyperparameters such as clipping range.

The focus on a more broad hyperparameter search such as done by Mnih et al. and others may be the explanation on why all these works, differently from mine, obtained good results even though they used score as their only reward.

As for the YouTube sources, little information could be found. ForrestKinght [5] used NEAT, a genetic algorithm that is different from those applied in the aforementioned works. While he could train his Breakout agent successfully, they did not compare it with other algorithms in terms of performance or stability. NamePointer[6] only goes as far as showing the training process but does not elaborate on their methodology.

5 Conclusions

In this project, I applied Reinforcement Learning to train agents to play the game Breakout. Several agents were produced by trying different combinations of RL algorithms, number of training steps, and reward approach.

After training, testing, and analyzing the scores and lives obtained by each agent, my conclusion is that agents trained with a combination of the PPO algorithm, in a range of 100K to 1M steps and a *break-and-follow* reward approach were, in average, the best performing both in terms of score and lives. The *follow* reward approach also yielded good results.

By observing agent playthrough, I concluded it did not only learn how to reach the goals that were rewarded by the environment, but also found ways to be rewarded in even quicker and more consistent ways. However, such behavior was only observed for agents trained with the *break-and-follow* and *follow* rewards and not *break*. Taking these successful reward approaches into consideration, agent performance could be further improved by conducting a broader hyperparameter search.

By developing this project I had the opportunity to understand more deeply how Reinforcement Learning works, which hyperparameters are most important on the learning process of the agent, how an OpenAI Gym environment works and how different rewards affect the behavior of an agent.

[5] https://www.youtube.com/watch?v=9ZHnJCxq0LU

[6] https://www.youtube.com/watch?v=ejtJHnzJyGw

References

[1] Igor Adamski, Robert Adamski, Tomasz Grel, Adam Jedrych, Kamil Kaczmarek, and Henryk Michalewski. Distributed deep reinforcement learning: Learn how to play atari games in 21 minutes. In *Lecture Notes in Computer Science*, pages 370–388. Springer International Publishing, 2018. 2, 5

[2] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017. 1

[3] Mirco Giacobbe, Mohammadhosein Hasanbeig, Daniel Kroening, and Hjalmar Wijk. Shielding atari games with bounded prescience. In *Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '21, page 1507–1509, Richland, SC, 2021. International Foundation for Autonomous Agents and Multiagent Systems. 1

[4] Anoop Jeerige, Doina Bein, and Abhishek Verma. Comparison of deep reinforcement learning approaches for intelligent game playing. In *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0366–0371, 2019. 2, 5

[5] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb. 2015. 2, 5

[6] Matthew Stephenson and Jochen Renz. Deceptive angry birds: Towards smarter game-playing agents. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*, FDG '18, New York, NY, USA, 2018. Association for Computing Machinery. 1

A Appendix: Extra Results

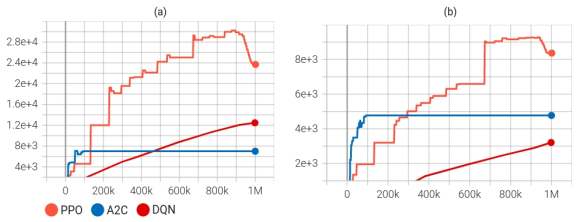

Figure 4: Evolution of average session length (a) and average reward (b) for PPO, A2C and DQN algorithms across 1M training steps

In this section, I will present training loss data for agents trained with PPO, A2C, and DQN algorithms under the parameters found to be the best performing: 1M training steps and a *break-and-follow* reward approach. While I could not find many detailed explanations on how to interpret this data, the StableBaselines3 documentation can be used as a starting point to understand the purpose of each metric.

Different algorithms log different loss function outputs during training. PPO and DQN record a metric called "loss", which corresponds to the current total loss. This metric drops significantly for PPO between 600K and 800K steps, which is also the best performing interval on average. However, the loss never reached zero for PPO, which means its policy is still changing due to the exploration of new possibilities of action inside the game.

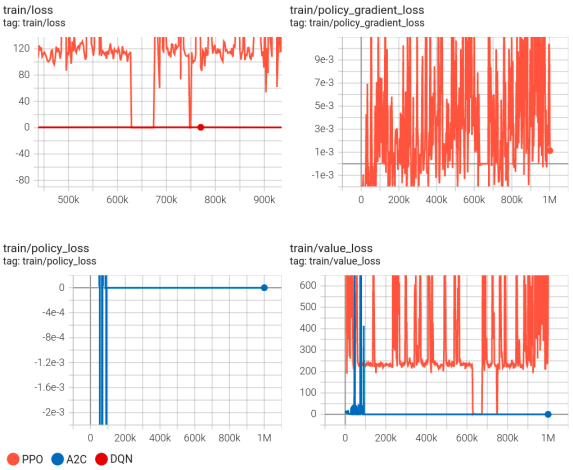

Figure 5: Loss measurements for PPO, A2C and DQN, as visualized on Tensorboard

The agent trained with DQN, on the other hand, is always logging "loss" equal to zero, which might indicate that it is not exploring its possibilities of action as broadly as it should and therefore considers it has reached "perfect" performance.

Since both PPO and A2C are on-policy algorithms, they log a metric called "value_loss", which according to StableBaselines3 documentation, represents the "the value function loss (...) usually error between value function output and Monte-Carle (sic) estimate" [7]. The "value_loss" drops to zero for A2C after 200K steps of training, which indicates it is no longer learning after this point (which is corroborated by the fact the mean reward also stops increasing, as since in Figure 4). The "policy_loss" is logged only by A2C and displays the same behavior in the same interval.

Finally, there is the "policy_gradient_loss" metric is logged only by PPO. According to StableBaselines3 documentation, "its value does not have much meaning" and is probably used only for debugging the algorithm itself. However, in this case, it presents a similar behavior as "loss": it is generally very unstable, but its value drops in the range of 600K-800K training steps.

[7] https://stable-baselines3.readthedocs.io/en/master/common/logger.html