ACTOR VERSUS CRITIC

Osmany L. Corteguera
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA 02139
`osmanyc@mit.edu`

May 16, 2019

ABSTRACT

This project compares the performance of value function approximation and policy optimization methods for solving reinforcement learning problems. The Deep Q-Network (DQN) [1] architecture was chosen as a representative of a value function approximation method, and Proximal Policy Optimization (PPO) [2] as a policy optimization method. I implemented the DQN architecture and used OpenAI's baseline implementations of DQN and PPO [3] to learn evaluate the algorithms on Atari 2600 Pong from OpenAI Gym [4]. The vanilla DQN architecture was not able to learn a successful policy, so an implementation of DQN with prioritized experience replay [5] was used for comparison with PPO. In the experiments, the DQN player learns a successful policy much more quickly than the PPO player and also with less variance, thus DQN seems to be a better algorithm for the game of Pong.

1 Introduction

Model-free deep reinforcement learning has two major paradigms: value function approximation and gradient optimization. Each of them tries to solve the problem of finding the policy $\pi^*(s)$ that maximizes the future returns of the agent. Value function approximation attempts to solve the problem by estimating the state-action value function, $Q(s, a)$ and greedily choosing the best action at every state. On the other hand, gradient optimization approaches the same problem by optimizing the policy $\pi(s)$ directly.

Each of these methods usually rely on deep neural nets to approximate their respective functions, so that they are parametrized by θ: $Q(s, a|\theta)$, $\pi(s|\theta)$. Both methods, then, rely on experience from the environment to optimize their respective functions with respect to θ. This project chooses DQN and PPO as representative algorithms for these two approaches since they have shown to be successful in solving deep RL problems.

In this project I explore the questions of whether there are significant differences in the performance of DQN and PPO on RL tasks, and in what situations it would be preferable to use one over the other. Metrics considered are the sample efficiency, the stability of training, and the effectiveness of the agent as we approach convergence.

I chose to test these algorithms on the Atari 2600 game Pong, since it is an environment with relatively dense rewards, and many different algorithms have been successful in solving it.

2 Implementing DQN

2.1 First Model

2.1.1 Architecture

First, I tried to implement the DQN architecture as described in the DQN 2013 preprint [6]. In this model, we first preprocess the 210x160x3 RGB screen frames we get from the simulator, and turn them into 84x84 grayscale frames. For each timestep, we then feed the previous 4 frames of the simulation (an 84x84x4 tensor) to the Q-network. The

Q-network is a convnet, with [16, 8, 4] ReLU first layer, [32, 4, 2] ReLU second layer, fully-connected 256 ReLU third layer, and a fully-connected 6 unit linear layer as the output, one unit per action in Pong. I built this Q-network using PyTorch [7].

The other part of the model is the replay memory. In it we store the last N transitions seen in the simulated play. We can also sample transitions from it for the purpose of training. I first implemented it as a Python object which held a list of transitions $(\phi_j, a_j, r_j, done_j, \phi_{j+1})$, where ϕ_j is the stack of 4 processed frames at timestep j.

2.1.2 Training

For training, I used the setup given in [6]. The optimizer used was the RMSProp implementation from PyTorch with their default parameters. The size of the minibatches was 32; the behavior policy was ϵ-greedy, annealed linearly from 1 to 0.1 over the first 1 million frames.

Unlike the setup in [6], I trained on every frame, and used a replay buffer smaller than 1 million frames.

2.1.3 Challenges

The first challenge I ran into with this architecture was in storing the transitions in the replay buffer. I had used a Python double-ended queue as the storage for the transitions, appending each new transition as it became available. However, after over 100,000 training steps training would grind to a halt due to the RAM being depleted by the increasingly large replay memory.

I worked around this by not storing ϕ_t at each transition, which is 4 frames, and instead storing a frame per transition, and rebuilding the 4 frames at the time of sampling. This increased the effective capacity of the replay memory, but still wasn't able to reach the 1 million transition buffer in [6]. The final version of the replay memory used pre-allocated PyTorch tensors with floating-point precision, which allowed me to reach a capacity of 400,000 without running out of RAM. Trying to allocate the full 1 million frames would take 26GB of RAM.

The biggest challenge I had with this architecture was that it failed to learn a useful policy. After training for more than 1 million frames, the average reward per game still hovered around -20, which is close to the minimum of -21. Playing the saved model revealed that the points were usually scored when random actions were chosen, rather than what the policy had learned.

2.2 Second Model

I also used the Q-network as described in [1], which has a different architecture and could possibly learn a good policy for Pong quickly. I also used the same hyperparameters as [1], with the only exception being that the replay memory's size was 400,000 instead of 1,000,000. The same training routine was used, running gradient descent every 4 steps on a random batch of 32 transitions, repeating the chosen action for the following 4 steps.

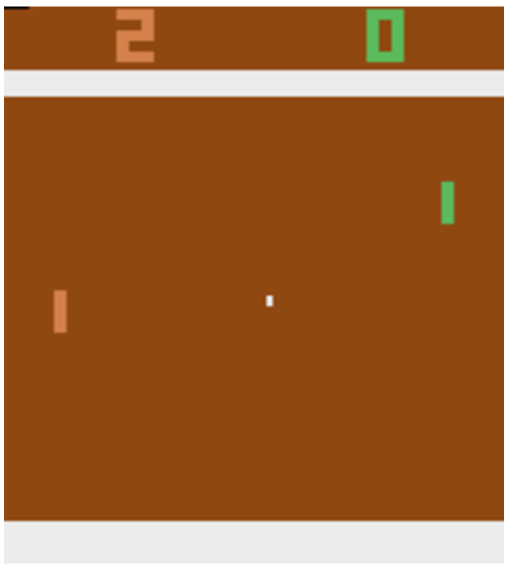
This setup also struggled to learn any useful policy. After being trained for 5 million frames, the Q-network didn't make any progress in increasing its rewards. A video of this agent can be found in the appendix. Figure 1 shows the Q-network's output for a sample game frame. Notice how all the actions have very negative values, and the values are all almost the same. [1]

One possible reason why the agent did not learn is that it wasn't trained for a sufficiently long number of frames. While I used 5 million frames of training experience, [1] use 50 million frames, which is significantly more experience. Another reason why it might have failed to learn is that the replay buffer was not as large as in [1], though this seems unlikely since the distributions of transitions in a replay buffer of size 400,000 should not be so different from one of size 1,000,000. Finally, there could be bugs in my implementation, and I don't have a test set to rule out this possibility.

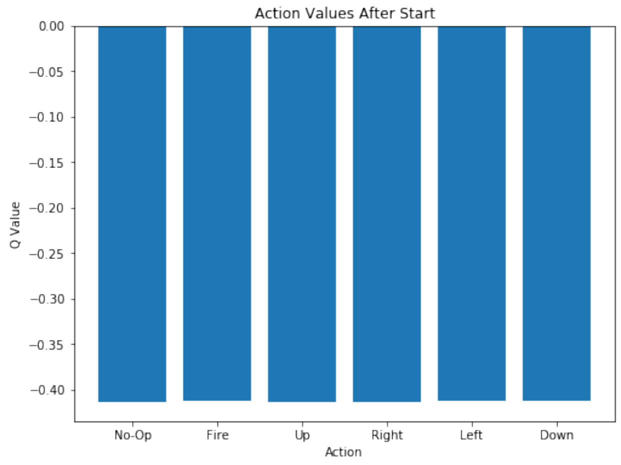
3 Algorithms to Compare

I order to carry out the experiments comparing DQN and PPO, I used the OpenAI baseline implementations of these algorithms [3] and trained them on OpenAI gym's Pong environment [4].

[1]Code of DQN implementation at: `https://github.com/osmanylc/dqn-player/blob/master/dqn.py`



(a) Frame of beginning of serve.



(b) State action values.

Figure 1: DQN state-action values, frame of serve.

3.1 DQN

This DQN model adds a few features on top of the model in [1]. First, it uses prioritized experience replay (PER) [5]. PER is a method for sampling from the DQN replay memory. Instead of sampling transitions uniformly at random, as in vanilla DQN, transitions are sampled proportionally to their TD error. This means that the transitions sampled will be those that the Q-network has trouble predicting well, which has the effect of increasing the efficiency of the sampling. Finally, this model uses the dueling architecture for DQN [8]. This architecture separates the estimation of the Q-function into estimation of the value function and an advantage function. This architecture has shown to learn the Q-function more efficiently [8].

3.2 PPO

The PPO implementation follows the one in [2], using the PPO-Clip version.

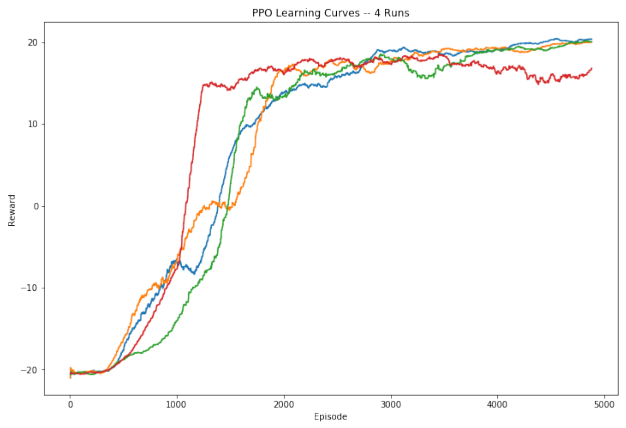
4 Experiments

The DQN and PPO players were trained on the Pong environment for more than 1 million frames each. Below we show a comparison of their performance. The PPO graph shows a plot of the moving average of rewards for the previous 5 episodes for 4 different PPO agents trained for around 5000 episodes. The DQN graph shows the moving average of the previous 100 episodes for 3 different agents, trained between 1000 and 3000 episodes.

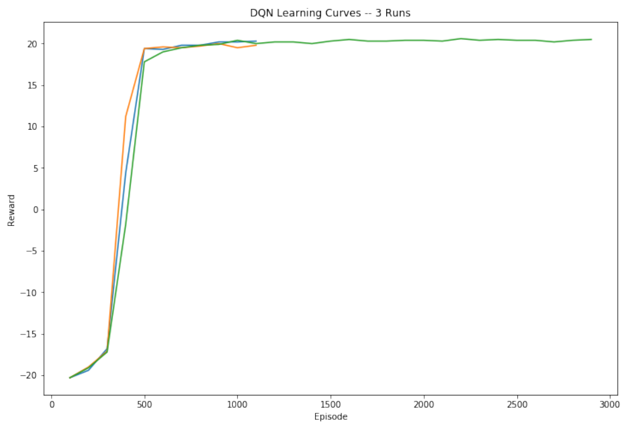
As we can see in Figure 2 the DQN solves the problem much more quickly than the PPO player. After around 400 episodes, the 3 DQN players learned already had very successful policies, while it took the PPO players around 3,000 episodes to get to the same level of performance. We can also see that the DQN agents trained have lower variance in their learning curves than the PPO agents.

5 Conclusion

This work explored the difference between value function approximation approaches and policy optimization approaches to deep reinforcement learning. I trained one algorithm from each class, DQN and PPO, respectively, on the Pong environment. The DQN algorithm showed better performance and lower variance for fewer samples than PPO, showing that it is a better approach to solving the game of Pong.



(a) PPO agents.



(b) DQN agents.

Figure 2: Comparison of learning curves for DQN and PPO agents on Pong.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[2] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[3] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines. `https://github.com/openai/baselines`, 2017.

[4] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[5] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[7] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[8] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1995–2003, 2016.

6 Appendix

6.1 Agent Videos

- Vanilla DQN agent: `https://drive.google.com/file/d/1Dal0r-YtIUnomaVvbZMLFmgSP1ds-Ygc/view?usp=sharing`
- PPO agent: `https://drive.google.com/file/d/1BGRr55npixgoE4PfSB0nnTFdYVFXnM57/view?usp=sharing`