Playing Atari Bowling with Reinforcement Learning and Proximal Policy Optimization

Alessandro Pomponio
Department of Computer Science and Engineering (DISI)
University of Bologna, Italy
alessandro.pomponio2@studio.unibo.it

Abstract

Reinforcement learning has been on the rise in the last few years, thanks to levels of performance that can even go beyond what humans are able to achieve in certain tasks. A great deal of work is being put into autonomous and adaptive systems that learn with experience, especially in robots and cars. In this work, we leverage Proximal Policy Optimization to create an autonomous agent that plays the Atari version of Bowling. We then compare the results we obtained to the ones available in the original paper, suggesting ways to improve the agent's behavior in this environment.

1 Introduction

Artificial intelligence and machine learning have reached incredible levels of popularity nowadays, rushing into our lives in more or less apparent ways. When it comes to the general public, most people will associate artificial intelligence with "smart assistants" such as Amazon Alexa, Google Assistant and Apple Siri; or with image classification algorithms categorizing pictures and videos on our smartphones. Reinforcement learning, a subset of machine learning enabling intelligent agents to learn from experience, has only recently started to gain traction with the masses, mainly due to the efforts that automotive manufacturers have been pouring into the development of autonomous vehicles. While fully autonomous driving is still far-fetched as of now, reinforcement learning has been able to achieve incredible results in board and video games. The most notable examples include Google DeepMind's AlphaGo defeating the European Go champion Fan Hui in October 2015 [1] and AlphaStar defeating 99.8% of human players in StarCraft 2 [2].

This work applies Proximal Policy Optimization, a family of algorithms presented by Schulman et al. in 2017 [3] to the Atari Bowling game, showing that an agent can learn to play a video game with good results by means of only the video feed and the game score. In the end we compare the results we obtained with the ones in the original paper and present a few additional considerations that may improve the results obtained by the agent.

2 Background

2.1 OpenAI Gym

As anticipated in the previous section, this work uses reinforcement learning techniques to develop an agent and train it to play the Atari Bowling video game by limiting its inputs to the game screen and the game score. We leverage OpenAI Gym, a toolkit for developing reinforcement learning algorithms that gives the user a plethora of environments with a standardized interface, greatly simplifying the development process. In fact, the developer can choose one of the available environments by means of a string, specifying whether they want pixels or RAM dumps as inputs, enabling or disabling frame

skipping and other characteristics. In addition to that, the reward function has already been set up, typically by mirroring the game score, easing once again the developer off on this task.

2.2 Proximal Policy Optimization

Proximal Policy Optimization, PPO for short, is a family of policy gradient methods which alternate between sampling data through interaction with the environment, and optimizing a "surrogate" objective function using stochastic gradient ascent [3]. They have been proposed due to the lack of simple, efficient, scalable and robust policy gradient algorithms for reinforcement learning. At the time in which the paper has been published, in fact, the most common algorithms were *Q*-learning (which struggled in problems with continuous action spaces, and was generally poorly understood) and Trust Region Policy Optimization (which is complicated and not compatible with architectures that include noise or parameter sharing).

PPO attempts to achieve the same data efficiency and reliable performance of TRPO, while using only first-order optimization. This was achieved by means of a novel objective with clipped probability ratios, which forms a pessimistic estimate (i.e., a lower bound) of the performance of the policy. Policy optimization was obtained by alternating between sampling data from the policy and performing several epochs of optimization on the sampled data [3].

A general skeleton of PPO was presented in [3] in pseudo-code. We decided to copy it here as well to help the reader understand the rest of the content.

Algorithm 1: PPO, Actor-Critic style

for $iteration = 1, 2, \ldots$ **do**
  for $actor = 1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{old}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}, \ldots, \hat{A_T}$
  Optimize surrogate loss with respect to θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{old} \leftarrow \theta$

Despite the original paper using an asynchronous implementation, for the sake of simplicity and due to the lack of hardware powerful enough to benefit from such parallelism, we decided to limit it to a single actor.

We now show the three most important parts of PPO: the advantages, the clipped surrogate objective, and the total loss function.

2.2.1 Advantages

Following TRPO's blueprints, the authors of PPO decided to leverage Generalized Advantage Estimation [4], initially introduced by Schulman et al. in 2015, modifying it to include two smoothing factors instead of one. Advantages are important because they measure "how good" a certain state proved to be compared to the original estimates, a key piece of information when performing policy improvement. The formula for computing the advantage is as follows:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1}$$
$$\text{where} \quad \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

2.2.2 Clipped surrogate objective

In order to limit the size of the policy update, TRPO applied a constraint β to a "surrogate" objective function based on the KL-divercence between the old and the new policy.

In theory TRPO should have used a penalty instead of a hard constraint, but choosing a value of β that performs well across different problems is very difficult. The authors of PPO, then took TRPO's surrogate objective:

$$L^{CPI}(\theta) = \hat{\mathbb{E}}_t\Big[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t\Big] = \hat{\mathbb{E}}_t\Big[r_t(\theta)\hat{A}_t\Big]$$

and modified it to penalize changes to the policy that move $r_t(\theta)$ away from 1 (as it would lead to an excessively large policy update without constraints). They proposed the following:

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

This new objective clips the probability ratio to remove the incentive for moving r_t outside the interval around 1 delimited by ϵ and returns the minimum between the unclipped and the clipped objective, effectively acting as a lower bound [3].

2.2.3 Total loss function

We will be using a neural network architecture that shares parameters between the policy and the value function; this requires us to use a loss function that combines the policy surrogate and a value function error term. This objective can further be augmented by adding an entropy bonus to ensure sufficient exploration.

Combining these terms, we obtain the following objective, which is (approximately) maximized each iteration:

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}\big[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)\big]$$

where c_1, c_2 are coefficients, and S denotes an entropy bonus, and L_t^{VF} is a squared-error loss $\big(V_\theta(s_t) - V_t^{targ}\big)^2$ [3].

3 Neural network architecture

To keep in line with what was proposed in the original paper, we will use the same architecture shown by Mnih et al. in [5].

The neural network is as follows, with a plot shown in figure 1:

1. An input layer, accepting inputs of dimensions 80×80.
2. A convolutional layer with 16 filters of size 8×8 with stride 4.
3. A convolutional layer with with 32 filters of size 4×4 with stride 2.
4. A fully connected layer with 256 hidden units.
5. The actor output, a fully connected layer with 6 nodes.
6. The critic output, a fully connected layer with only 1 node.

4 Preprocessing

In this section, we will introduce the preprocessing actions that were taken to try and make the training as efficient as possible.

We started by implementing the strategies suggested in [6], namely:

- Converting the game frames to greyscale.
- De-noising the image using Adaptive Gaussian Thresholding.
- Normalizing values to be between 0 and 1.
- Resizing the image to be 80×80.
- Stacking 4 frames to allow the network to understand movement.

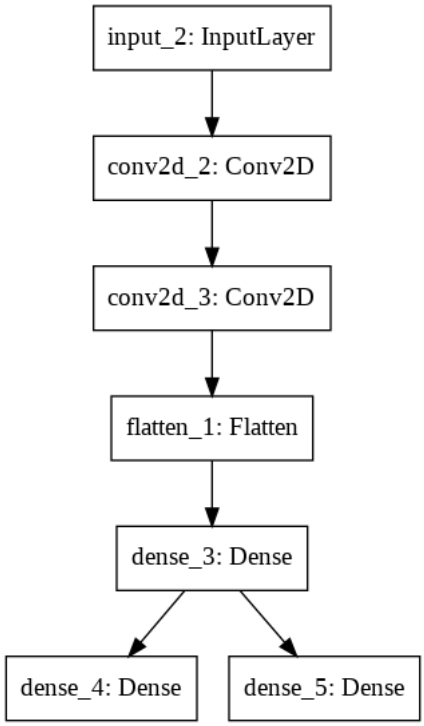
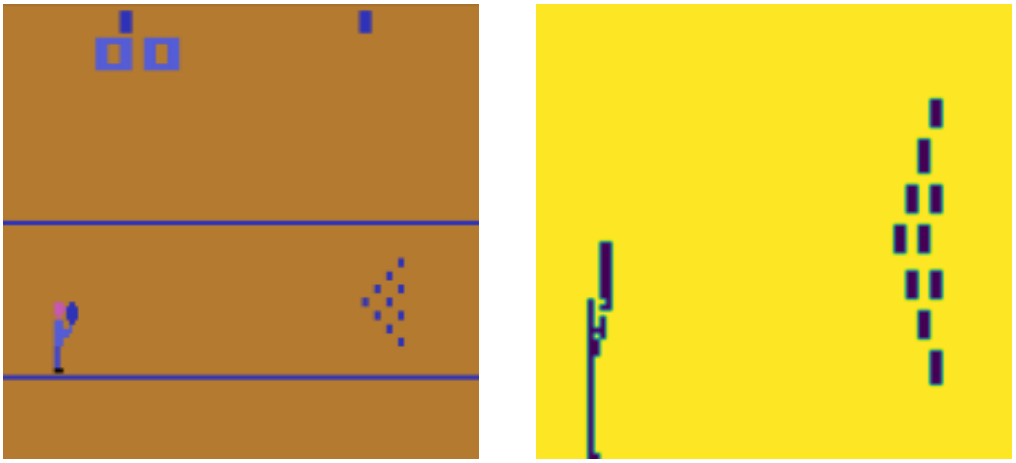


Figure 1: Plot of the neural network used

In addition to that, we cropped the observation to focus the network's attention on just the bowling lane. A before and after of this preprocessing can be seen in figure 2.



(a) The original Bowling environment

(b) The preprocessed Bowling environment

Figure 2: Before and after of the Bowling environment

5 Results and conclusions

After training the agent for around two days, we reached a peak score of 109 and a peak average score of 74.52, performing better than the original paper, which peaked at an average of around 57 points..

A plot of the rewards we obtained in each episode can be found in figure 3, while one for the average reward is found in figure 4.

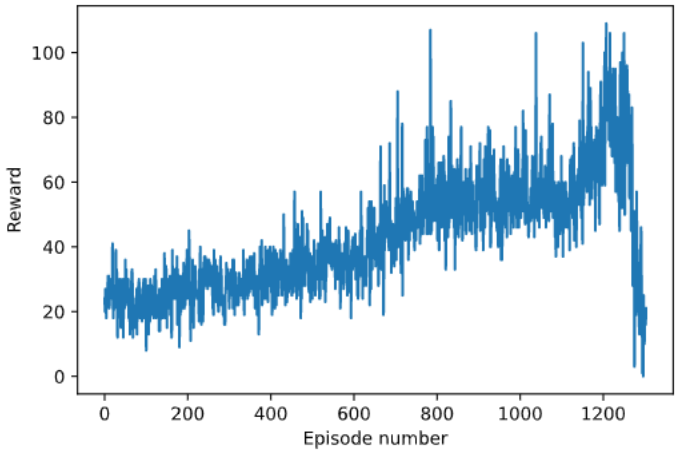


Figure 3: Rewards per episode

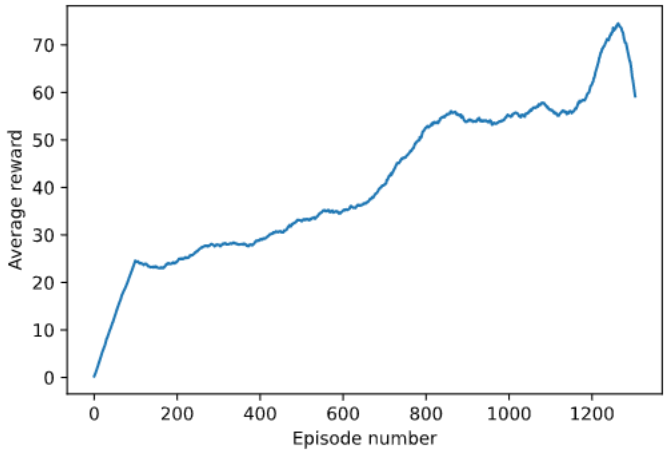


Figure 4: Average reward per episode

Some videos were then recorded, to see what techniques the agent was employing. By using the model saved roughly when the average score peaked, we see the agent playing reasonably well, apart from some instances where it throws the ball in a straight line, missing all the pins. Interestingly enough, the agent can (rarely) use curveballs to perform strikes.

In the future, the experiment may be repeated modifying the reward function to encourage the agent to hit as many pins as possible, possibly by giving a punishment for every pin standing after the throw. Further attempts may include limiting the episode duration to only two throws (or one in case of a strike), so as to make the agent focus on performing strikes or spares, which yield the most points.

References

[1] Silver, D. & Huang, A. & Maddison, C. et al. (2016) Mastering the game of Go with deep neural networks and tree search. *Nature* **529**, 484–489.

[2] Vinyals, O. & Babuschkin, I. & Czarnecki, W.M. et al. (2019) Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature* **575**, 350–354.

[3] Schulman, J. & Wolski, F. & Dhariwal, P & Radford, A. & Klimov, O. (2017) Proximal Policy Optimization Algorithms *Arxiv* 1707.06347.

[4] Schulman, J. & Moritz, P. & Levine, S. & Jordan, M. & Abbeel, P. (2015) High-Dimensional Continuous Control Using Generalized Advantage Estimation *Arxiv* 1506.02438.

[5] Mnih, V. et al. (2016)Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning* - Volume **48** (ICML'16). JMLR.org, 1928–1937.

[6] Chuchro, R. & Gupta, D (2017) Game Playing with Deep Q-Learning using OpenAI Gym.