# Deep Reinforcement Learning with Proximal Policy Optimization for a Custom Pygame Shooting Game

**Yuri Di Biasi**
Department of Computer Science and Engineering (DISI)
University of Bologna
yuri.dibiasi@studio.unibo.it

**Abstract**

Reinforcement learning (RL) algorithms have shown remarkable success in training agents to perform complex tasks by interacting with their environment and learning from feedback. In this paper, we explore the application of Proximal Policy Optimization (PPO) in training an agent to play a custom Pygame shooting game. The game involves two players competing in a static 2D map, where the objective is to survive and eliminate the opponent. By leveraging the power of deep neural networks and the flexibility of PPO, we develop an agent that learns to make strategic decisions, navigate the game environment, and outperform the opponent.

## 1 Introduction

Reinforcement learning has gained significant attention as a powerful approach for training agents to autonomously learn and make decisions in complex environments. One popular algorithms in the field, Proximal Policy Optimization (PPO)[3], has demonstrated remarkable success in various domains and it is known for its stability, sample efficiency, and compatibility with continuous action spaces. This paper aims to provide a comprehensive evaluation on the results obtained, focusing on its performance and efficiency when applied to a custom environment.

## 2 Background

### 2.1 Pygame

Pygame is a popular Python library that provides a simple and efficient framework for developing 2D games. It offers a range of functionality for handling graphics, sound, and user input, making it suitable for creating interactive and visually appealing game environments. The implementation at hand represents a shooting game, wherein two players engage in a head-to-head contest within a stationary 2D map. The objective is to eliminate the opponent, and the last surviving player emerges as the victor. Initially, both participants possess full health, which is indicated by a value of 100 out of 100. However, whenever a player sustains a bullet hit, its health decreases by 25 units. Within the realm of Reinforcement Learning (RL), Pygame can be effectively utilized to generate customized environments that amalgamate its extensive graphical capabilities with the standardized interface and evaluation tools provided by OpenAI Gym.

## 2.2 OpenAI Gym

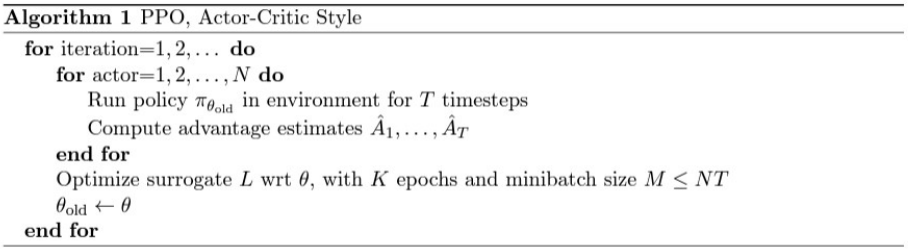
OpenAI Gym is a widely-used framework for developing and evaluating RL algorithms. It provides a standardized interface for interacting with the environments, making it easier to compare and benchmark different algorithms. Gym environments typically provide the agent with observations of the environment state and allow the agent to select actions based on the available actions in the environment. The environment then provides feedback in the form of rewards or penalties, indicating the quality of the agent's actions. After experimenting with various reward settings, I have identified the reward scheme that appears to facilitate optimal learning of the game dynamics. The chosen reward settings are as follows:

- At each step, a reward of -5 is provided to encourage the player to eliminate the opponent quickly, as seen in [1]
- A reward of +100 is given for hitting the opponent with a bullet.
- Being hit by a bullet results in a penalty of -50.
- A reward of +50 is awarded each time a bullet is dodged.
- Losing the game results in a substantial penalty of -1000.
- Winning the game is rewarded with a variable reward of +4000 multiplied by a ratio. The ratio is calculated based on the player's remaining health compared to their initial health. This encourages the player to aim for a win with as much health as possible.

Furthermore, following the approach proposed by Mnih et al. [2], the agent observes and selects actions every $k^{th}$ frames instead of every frame. The agent's last action is repeated for the skipped frames. In this implementation, $k$ was fixed to 4.

## 2.3 Proximal Policy Optimization

Proximal Policy Optimization (PPO) is a model-free, on-policy reinforcement learning (RL) algorithm designed to optimize the policy function in order to maximize the expected return. PPO employs a surrogate objective function that sets bounds on the policy update, ensuring stability during the training process. Unlike value-based methods such as Deep Q-Learning, PPO focuses on directly optimizing the policy without explicitly estimating the value function. The PPO algorithm operates in an iterative manner, collecting data by interacting with the environment and subsequently updating the policy using stochastic gradient ascent to maximize the expected return. PPO has gained considerable popularity due to its stability, sample efficiency, and its ability to effectively handle continuous action spaces. It has been successfully applied in a wide range of domains, including robotic control, autonomous driving, and game playing. Here is a pseudo-code representation of the algorithm:



**Algorithm 1** PPO, Actor-Critic Style

```
for iteration=$1, 2, \ldots$ do
    for actor=$1, 2, \ldots, N$ do
        Run policy $\pi_{\theta_{\text{old}}}$ in environment for $T$ timesteps
        Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
    end for
    Optimize surrogate $L$ wrt $\theta$, with $K$ epochs and minibatch size $M \leq NT$
    $\theta_{\text{old}} \leftarrow \theta$
end for
```

Advantage Estimation, Clipped Surrogate Objective, and Loss Function are three crucial components of the Proximal Policy Optimization (PPO) algorithm.

### 2.3.1 Advantage Estimation

The advantage estimate measures the advantage of taking a particular action compared to the expected value. It indicates how much better or worse the action is in terms of cumulative rewards. Accurate estimation of advantages is crucial for proper policy updates and learning.

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

### 2.3.2 Clipped Surrogate Objective

Clipped surrogate objective helps in controlling the policy update. The clipped surrogate objective is a modification to the traditional policy gradient objective function. It introduces a constraint on the policy update to prevent large policy changes that could destabilize the learning process. The objective is formulated as a surrogate objective that approximates the expected improvement in policy performance.

The clipped surrogate objective function is defined as follows:

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

where $r_t(\theta)$ denotes the probability ratio:

$$r_t(\theta) \;=\; \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}$$

### 2.3.3 Loss Function

If use a neural network architecture that shares parameters between the policy and value function, it is imperative to employ a loss function that encompasses not solely the policy surrogate and a value function error term, but also incorporates an entropy bonus to guarantee an adequate degree of exploration. Combining these terms, we obtain the following objective, which is maximized each iteration:

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

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

## 3 Preprocessing

As suggested by Mnih et al [2], directly processing raw frames as inputs can be computationally intensive. In the context of my implementation, the raw frames are 300 x 300 pixel images with a 255-color palette. To address this challenge and reduce the dimensionality of the input, a basic preprocessing step is applied. I have made the deliberate choice not to convert the observation into grayscale, despite the fact that it would have led to a slightly larger number of parameters. This preprocessing step aims to simplify the input representation, making it more computationally efficient and manageable for subsequent processing and analysis.

1. Cropping operation to remove useless information
2. Normalize values to be between 0 and 1
3. Resize image to be 84x84
4. Stack 4 frames to allow networks to detect movement
5. Additionally, reward normalization is employed to enhance the stability of the learning process.

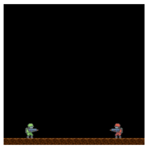


Figure 1: Original observation

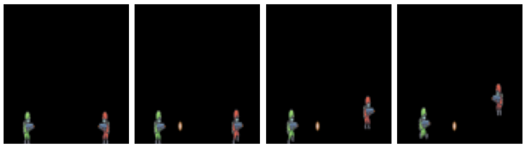


Figure 2: Final observation

## 4 Neural Networks Architecture

The neural network architecture plays a pivotal role in facilitating the agent's learning and decision-making capabilities. In this implementation, I employed an Actor-Critic network, which closely adheres to the structure proposed by Mnih et al. [2]. The architecture is as follows:

1. The input layer taking input of dimension 4x84x84x3.
2. A first convolutional layer convolves 16 8x8 filters with stride 4.
3. A second convolutional layer convolves 32 4x4 filters with stride 2.
4. A dense layer with 256 rectifier units.
5. **Actor** output, fully-connected layer with 12 nodes.
6. **Critic** output, fully-connected layer with 1 node.

By utilizing this Actor-Critic network architecture, we aim to enhance the agent's ability to effectively learn from its environment, capture policy information, and estimate value functions, thus enabling improved decision-making and performance in the given task or domain.

## 5 Results and Analysis

For an accurate representation of the obtained results, it is imperative to note that the opponent's actions were constrained to solely shooting. This constraint was imposed to expedite the training process and due to the observation that if the opponent played randomly, the player controlled by our neural network appeared to exhibit sub-optimal learning and exhibited pseudo-random behavior. The duration of the training process was set at 2 million frames. However, a predefined condition was established wherein if the agent achieved 5 consecutive victories in games played every 2,000 frames, repeated for a total of 30 consecutive occurrences, the training process would be terminated. The agent fulfilled this condition after 136,000 frames.

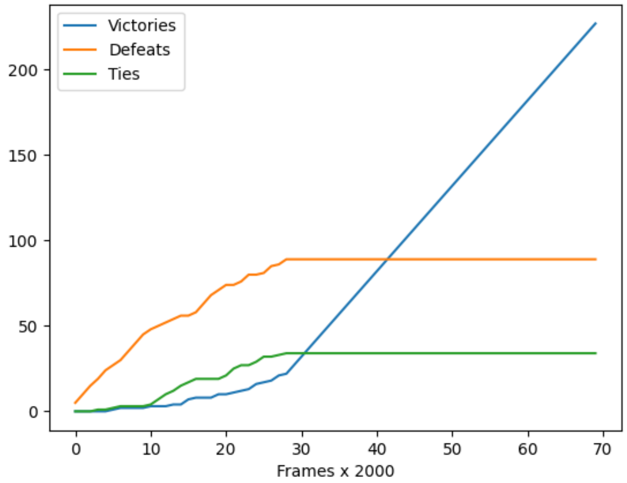


Figure 3: Training results

The graph presented above clearly indicates that the agent was able to discover an effective strategy that resulted in consistent victories in all subsequent matches after approximately 60,000 frames.

Implementing modifications to the reward functions, as well as introducing variable skipped frames during gameplay based on the positions of the two players, can potentially result in performance enhancements. Adjusting the number of skipped frames to a higher value when the players are far apart and reducing it when they are in close proximity may facilitate improved gameplay dynamics. Furthermore, these modifications have the potential to enable our agent to engage in self-play and potentially even compete against human players.

# References

[1] Dmitry Akimov and Ilya Makarov. "Deep reinforcement learning with vizdoomfirst-person shooter". In: (2019).

[2] Volodymyr Mnih et al. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[3] John Schulman et al. "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347* (2017).