Procgen benchmarks with A2C

Gianmiriano Porrazzo
University of Bologna
gianmiriano.porrazzo@studio.unibo.it

Abstract

The following is an implementation study on A2C applied to three games in the OpenAI Gym environment. This study focus is on the implementation of the previous mentioned algorithm to obtain an agent that learn to play the games in a better way than the random agent.

1 Introduction

1.1 Objective

The main objective of the project is to apply Reinforcement Learning techniques to create an agent that can solve video game by learning how to do it itself. The desired outcomes are:

- Implement an agent that learn how to play the games
- Create an agent that perform better than the random choice for all the tested games

1.2 Environment

The environment used to make the measurments is Procgen [1]. In particular are used three games:

- Heist: a maze like game
- Leaper : a game in wich the player must cross several lanes to reach the finish line
- Starpilot: a side scrolling shooter game

In the first one the agent has to collect the keys in the maze and unlock the final lock, passing from the previous locks.
In the second one the player must cross several lines avoiding obstacles before reaching the finish line, so the reward is given at the end. The lanes can contain: cars that needs to be avoided and logs on a river to hop on for crossing the river.
The last game is a scrolling shooter in which the enemies can change velocity so the player needs to be reactive and dodge the enemies, but also try to kill them to collect more points and avoid dying.
For all the games, at each timestamp, the enviroment provides an observation (or state) from the raw frame data, centered on the agent and as a 64x64x3 RGB image. The agent can perform 15 possible actions, such as: move right, left, go up and down, etc.
All this makes the environments not trivial to train an agent on, due to the fact that the observation space is large and the rewards are quite sparse

1.3 Algorithms

Neural Network
To process the state, represented by the image, it can be helpfull to make a neural network. To do that the Neural Network is composed by a small Convolutional Network that takes in input the image of the state and reduces it to a vector of features.

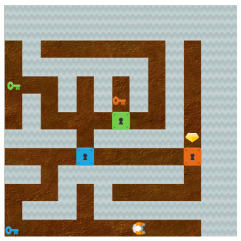
Figure 1: Heist

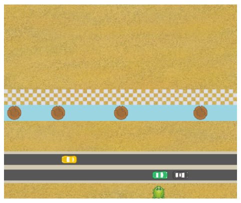
Figure 2: Leaper

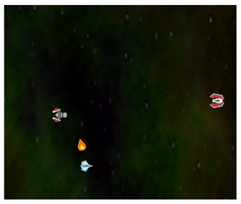
Figure 3: Starpilot

Advantage Actor Critic
Advantage Actor Critic (or A2C) is an Actor Critic method based on the [2] OpenAI algorithm. As an Actor Critic it has a policy network and a value network, in A2C the critic part is done using the Advantage function instead of an action value function.

1.4 Implementation tricks

To improve the performance and stability of Deep Reinforcement Learning algorithms some tricks are used. One of these tricks consist into modify the input of the network so that it is processed before entering in the neural network itself. This is called *Input Formatting*. Some of the techniques used in this study are:

- Frame Stacking
- Pixel Normalization
- Return calculation for terminated episodes
- Reward standardisation or normalization

A distinct possibility is to use *Optimisation stability* to improve training stability. Some tricks of that are:

- Orthogonal weight initialization
- Gradient clipping

2 Implementation

2.1 Neural Network

Since the environments return an image as an observation, it can be used as the input of a neural network. Also, thanks to the environment wrappers, it is possible to convert from the shape of the image returned by the environment to the one that it is used in the model of the neural network. This is done because the network used to learn from the image is the model explained in the Nature article [3], that is at the base of the DQN implementation for solving atari games.

2.2 A2C

The Actor Critic algorithm combines aspect from **policy-based methods (Actor)** and **value-based methods (Critic)**, this helps to address the limitations of each methdos when applied individually.
The A2C algorithm introduces the **Advantage function**, that it's used to measure how much better an action is compared to the average action in a given state. Using that information, A2C, focuses the learning process on actions that have an higher value than the typical action taken in that state.
The main difference between A2C and a simple AC algoithm is that the usage of the advantage function can help improving the learining process.

The A2C algorithm is the following [4]:

Algorithm 1 Advantage Actor Critic

```
// Assume $\theta$ and $\theta_v$ as parameter vectors
Initialize step counter $t \leftarrow 1$
Initialize episode counter $E \leftarrow 1$
repeat
    Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$
    $t_{start} = t$
    Get state $s_t$
    repeat
        Perform $a_t$ according to policy $\pi(a_t|s_t;\theta)$
        Receive reward $r_t$ and new state $s_{t+1}$
        $t \leftarrow t + 1$
    until terminal $s_t$ or $t - t_{start} == t_{max}$
    $R = \begin{cases} 0 & \text{for terminal} s_t \\ V(s_t, \theta_v) & \text{for non-terminal} s_t \text{//Bootstrap from last state} \end{cases}$
    for $i \in \{t-1, \dots, t_{start}\}$ do
        $R \leftarrow r_i + \gamma R$
        Accumulate gradients wrt $\theta$ : $d\theta \leftarrow d\theta + \nabla_\theta \log \pi(a_i|s_i;\theta)(R - V(s_i;\theta_v))+$
                                    $\beta_e \partial H(\pi(a_i|s_i;\theta))/\partial\theta$
        Accumulate gradients wrt $\theta_v$ : $d\theta_v \leftarrow d\theta_v + \beta_v(R - V(s_i;\theta_v))(\partial V(s_i;\theta_v)/\partial\theta_v)$
    Perform update of $\theta$ using $d\theta$ and $\theta_v$ using $d\theta_v$
    $E \leftarrow E + 1$
until $E > E_{max}$
```

In which $a_t, s_t$and$R$ are the action, the state and the discounted return at time t, γ is the discount factor.

π represent the policy, parametrized by the neural network parameters θ, V is the value function, parametrized by θ_v, that estimetes the expected return from the staste s_t. Also we can see:

- $H(\pi)$ that is the entropy of th policy π.
- β and β_v that are the hyperparameters controlling the contribution of the value estimate loss and the entropy regularization term.

From all the above informations one can observe that the advantage function is not mentioned. Thi is because it can be found in the loss function. So the loss function is:

$$\begin{aligned}\nabla\mathcal{L} &= \nabla\mathcal{L}_\pi + \nabla\mathcal{L}_v + \nabla\mathcal{L}_H \\ &= \frac{\partial \log \pi(a_t|s_t;\theta)}{\partial\theta}\delta_t(s_t;\theta_v) + \beta_v \delta_t(s_t;\theta_v)\frac{\partial V}{\partial\theta_v} \\ &+ \beta_e\left[\frac{\partial H(\pi(a_t|s_T;\theta))}{\partial\theta}\right]\end{aligned}$$

In which we can found $\delta_t(s_t;\theta)$ that is the *n-step* return TD error that provides an estimate of the **advantage fucntion** for the Actor Critic.
This delta is defined as:

$$\delta_t(s_t;\theta_v) = [R_t - V(s_t;\theta_v)]$$

and R_t is:

$$R_t = \sum_{i=0}^{k-1}[\gamma^i r_{t+1} + \gamma^k V(s_{t+k};\theta_v)]$$

2.3 Memory

This part of the algorithm is used to collect the trainsition during each iteration of the training phase, so the training performances can be enhanced. In fact there are stored some valuable informations for the Advantage Actor Critic algorithm, for example:

- The state, action and relative reward
- The value of the advantage function

All those are used for the gradient update at each training step. For that here are, also, stored all the information to calculate the *advantage function*,and store its value after, that is calculated as using the following formula:

$$A(s_t, a_t) = R_{t+1} + \gamma V(S_{t+1}) - V(s_t)$$

Where R_{t+1} is the future reward V are the result given by the value functions and γ is the discount factor. Another foundamental role performed by the memory is to store and manage the mini-batches for mini-batch backpropagation. Instead of performing the backpropagation on all the sampled steps, there a used only a subset to perform the backward pass. This action is repeated for n times over the entire batch.

2.4 Random agent

To have a comparison model to check if the A2C is doing well, a simple random agent is implemented with the objective of choosing the action to perform in a random way.

3 Tricks

After some first tests the result weren't so good: the agent get stuck in some points or never start to play the game. So, after some researches, some improving techniques were added to the learning process [5]. In particular the *Input Formatting* ones were added in the building of the environment through the wrappers [6], instead the *Optimisation Stability* ones were inserted directy were necessary.

3.1 Input Formatting

Input formatting is usefull to process the inputs before inserting them into the neural network.

Frame Stacking
In environments with image observations, an agent needs information about how the state changes over time to infer new informations like direction, that cannot be obtained by a single image. To do so frame stacking is used. It consist in not changing state at each action, but skipping some frames. Most commonly 4 image frames are stacked, as is done in the implementation. This can be helpfull in games such starpilot in wich there are a lot of movement performed by the agent and the enemies, such as *starpilot*.

Pixel Normalisation
Image pixel are in the range [0, 255] and neural networks that process these ranges are slower. So, to reduce the complexity of the computation, the values can be normalised to the range [0, 1].

Return calculation for terminated episodes
A2C uses the critic part to implement the value function, that require the returns to be calculated as the target during the training, but it's also needed to handle the time steps when an episode ends. In that case, the returns at the next time step shouldn't be used in the calculation. If the episod ends because the agent found the terminal state the returns can be masked when the episode ends. In procgen there is also a time limit to truncate each episode so that it deesn't continue for too long (that is inherited from gym [6]). For truncated episodes bostrapping using the predicted value of the state is a good option. This allows the agent to train using the estimate of what the future returns would be at that state if the episode had not been truncated.

Reward standardisation
This is the process in wich, starting from a collection of rewards, this is transformed into a standard range value of rewards. It can be helpfull because having rewards that vary a lot can create an unstable learning process, so limiting them could help to approximate the target better. This can be helpfull when the environment produces rewards within a big range of value.

3.2 Optimisation Stability

This category of techniqes aims to improve the training process, in particular its stability

Orthogonal Weight Initialization
Orthogonal Initialization (OI) is one of the possible initialization technique for the learnable parameters of a model. This is helpfull because the agent thend to converge and be more stable during the training process with respect to the agents without OI.

Gradient Clipping
The aim of gradient clipping is to rescale the model parameter's gradients in a way that make the norm of the vector containing the model gradients clamped under a limit. In other words, performing the clipping of the gradiente causes the upper bound for the norm of the gradient itself. This can be helpfull when the model have large gradients during training, maybe because it's a long train or the steps are wide. For example it can be caused by a log probability (used in the A2C implementation) of an action that has almost 0 probablity, that is sampled by the policy.

3.3 Vectorized Environments

Since the implementation of the algorithm uses the mini-batch backpropagation and wrappers were used, there was a small cost on using *Vectorized Environments* [6]. Those are envirnoments that run multiple and indipendent copies of the same environment. They take as an input a batch of actions and return a batch of states (or observations): this can be usefull for collecting data for mini-batch backpropagation.

4 Results

4.1 Setup

To evaluete the performance of the algorithm, each training step is alternated with a validation one. The two steps run on differrent environments with fixed seeds. Also the evaluation is done using Google Colab [7] to use better hardware.

4.2 Outcomes

In the following figures it is shown how the algorithm performed with respect to various games and training length.
First the graphs that shows how the agent with A2C learn with respect to the increase the number of total steps, both in the train and evaluation process (called test in the graphs).

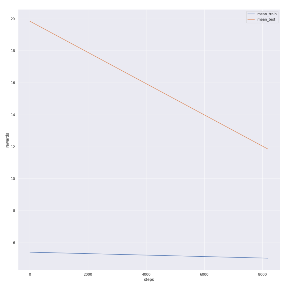

Figure 4: starpilot 10000 steps

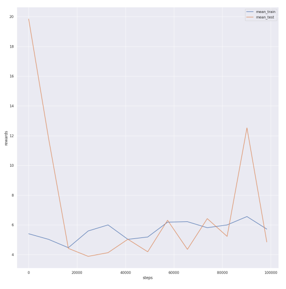

Figure 5: starpilot 100000 steps

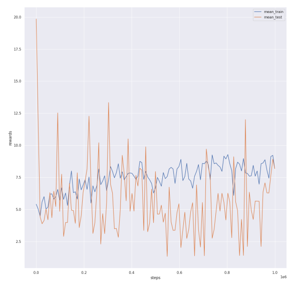

Figure 6: starpilot 1000000 steps

From the above graphs it is gathered that there's some learning during the training phase and, giving more the algorithm more time to train can benefit for the final result in all the cases taken in consideration. More can be found in the appendix, but in all game this is the trend.
Then a comparison with the random agent.

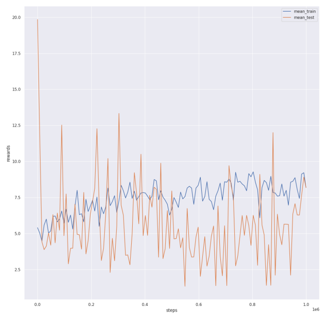

Figure 7: a2c starpilot

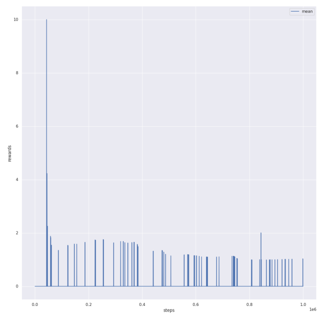

Figure 8: random starpilot

From all the graph above, it's evident that in all the choosen games the implemented actor critic learn better than the random agent, since the mean reward tend to increase instead of having huge jumps between steps.

5 Improvements

The implemented algorithm is not perfect and other tricks can be implemented to improve its performances. Another improvent could be based on more detailed tests, changing some hyperparameters that may give more insights.

6 Conclusion

The implementation achieves good result and has margin for improvment. The implementation of the algorithm wasn't easy, and some steps aren't well documented in the study for brevity. Also hyperparameter tuning is another crucial part of that determine the performances of the algorithms, for this a more detailed tuning phase should achieve better results, but it's not done in the current study. At the end more tests could be done to check and find more crucial insigths to understand better where to act.

References

[1] Cobbe K. & Hesse C. & Hilton J. & Schulman J. Leveraging procedural generation to benchmark reinforcement learning. 2020. doi: https://doi.org/10.48550/arXiv.1912.01588. URL `arXiv:1912.01588v2`.

[2] Mnih V. & Badia A.P. & Mirza M. & Graves A. & Harley T. & Lillicrap T.P. & Silver D. & Kavukcouglu K. Asynchronous methods for deep reinforcement learning. 2016. doi: https://doi.org/10.48550/arXiv.1602.01783. URL `arXiv:1602.01783v2`.

[3] Mnih V. & Kavukcouglu K. & Silver D. & Rusu A. A. & Veness J. & Bellemare M. G. & Graves A. & Riedmiller M. & Fidjeland A. K. & Ostrovski G. & Petersen S. & Beattie C. & Sadik A. & Antonoglou I. & King H. & Kumaran D. & Wierstra D. & Legg S. & Hassabis D. Human-level control through deep reinforcement learning. 2015. doi: http://www.nature.com/doifinder/10.1038/nature14236. URL `https://www.nature.com/articles/nature14236`.

[4] Wang J. X. & Kurth-Nelson Z. & Kumaran D. & Tirumala D. & Soyer H. & Leibo J. Z . & Hassabis D. & Botvinick M. Prefrontal cortex as a meta-reinforcement learning system. 2018. doi: https://doi.org/10.1038/s41593-018-0147-8. URL `https://www.nature.com/articles/s41593-018-0147-8`.

[5] Dunnion M. & Garcin S. & McInroe T. Reinforcement learning implementation tips and tricks. 2022. URL `https://agents.inf.ed.ac.uk/blog/reinforcement-learning-implementation-tricks`.

[6] Brockman G & Cheung V. & Pettersson L. &Schneider J. & Schulman J. & Tang J. & Zaremba W. Openai gym. 2016. doi: https://doi.org/10.48550/arXiv.1606.01540. URL `arXiv:1606.01540v1`.

[7] Inc Google. Google colab, 2017. URL `https://colab.google`.

A Extended results

Here are displayed other graphs relative to the training phase with different steps, for the other games.

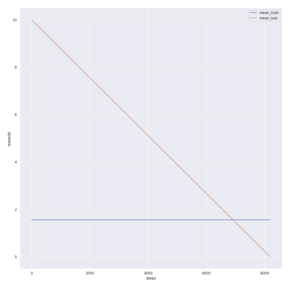

Figure 9: heist 10000 steps

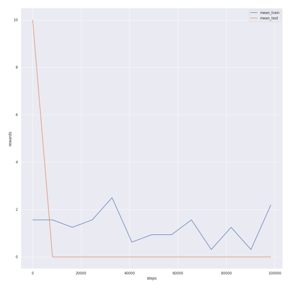

Figure 10: heist 100000 steps

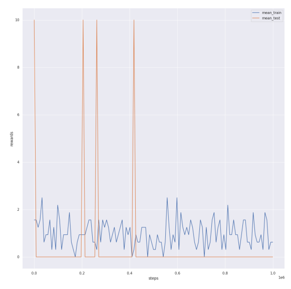

Figure 11: heist 1000000 steps

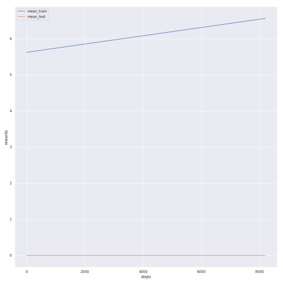

Figure 12: leaper 10000 steps

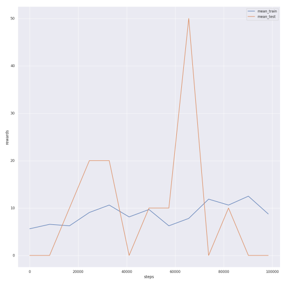

Figure 13: leaper 100000 steps

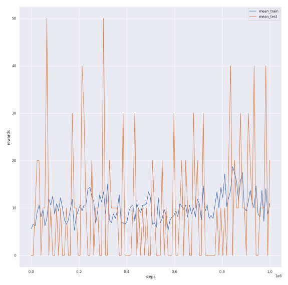

Figure 14: leaper 1000000 steps

The following are the graphs related to the comparison between the random agent and the a2c agent for *leaper* and *heist*.

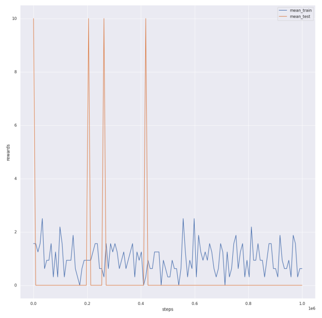

Figure 15: a2c heist

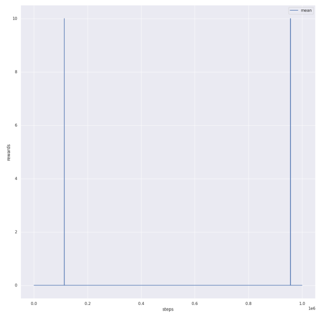

Figure 16: random heist

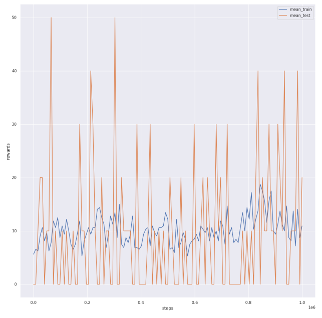

Figure 17: a2c leaper

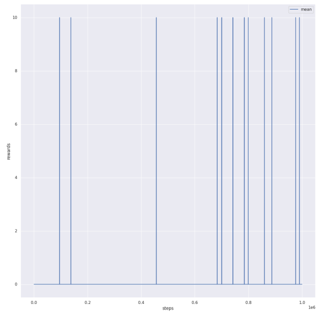

Figure 18: a2c heist

The last study done is related to how parallel envs perform in respect to a single env. What can be seen is that parallel envs make the training phase more stable and also a bit faster.

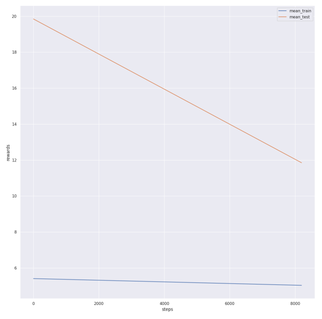

Figure 19: 32 envs 10000 steps starpilot

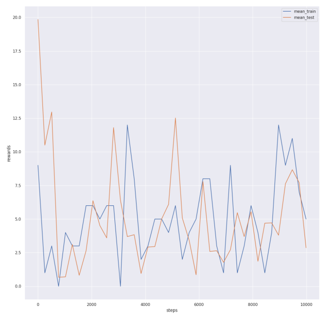

Figure 20: 1 env 10000 steps starpilot