

Reinforcement Learning Project

Jesús Copado Rodríguez (802648), Filippo Bedendo Bordon (771326)

08 December 2019

Contents

1	Introduction	2
2	Review of external sources	2
3	Approach and Method	2
3.1	First approaches	2
3.2	PPO approach	4
3.2.1	Policy	4
3.2.2	Preprocessing	4
3.2.3	Hyperparameters	5
3.3	Final refinements	5
4	Results and Performance analysis	6
4.1	Results	7
4.1.1	Results with two actions	7
4.1.2	Results with three actions	7
4.2	Performance analysis	8
5	Conclusions	9
6	Acknowledgement	9

1 Introduction

The project consists on creating an agent for the Pong environment 'wimble-pong', a modified version of Atari Pong. Our final agent has been trained using PPO Gradient Descent algorithm for reinforcement learning, but many algorithms has been tested from the beginning of the project. The neural network of the Policy has been developed using Pytorch Libraries and CUDA Libraries, whose required advanced computing capabilities. Hence the training has been done with a Nvidia Quadro P5000 GPU of the computers in the building Maari - Paniikki at Aalto University.

2 Review of external sources

For this project we have used many external resources of reinforcement learning algorithms for Pong.

All these resources work on a different Pong environment, but were good starting point to face the problem and to get good inspirations.

It will follow the list of the external resources used for the project:

1. <http://www.sagargv.com/blog/pong-ppo/> An agent implementing Proximal Policy Optimization in Atari Pong
2. <https://arxiv.org/abs/1707.06347> The paper presenting the theory of PPO Policy gradient for reinforcement learning and useful examples of hyperparameters.
3. <http://karpathy.github.io/2016/05/31/rl/> The blogpost by Andrej Karpathy that introduced us the basis of Pong from pixels.
4. <https://arxiv.org/abs/1611.01224> Sample Efficient Actor-Critic with Experience Replay

3 Approach and Method

3.1 First approaches

At first, we just wanted to train a decent agent with the "simpler" algorithms and approaches we have seen in the previous exercises and during the lecture before jumping into more complex algorithms as PPO or ACER.

Among all those "simple" algorithms we learnt during the course we decided to go first with the Actor Critic as during the exercises it proved to have the best performance. In addition, we opted for AC in the first place because policy gradients methods usually show faster convergence rate than Q-learning methods as DQN, although they tend to converge to a local optimal. Moreover, since PG

methods model probabilities of actions, it is capable of learning stochastic policies, and hence there's no need to create an exploration strategy like e-greedy for the case of Q-learning. Furthermore, learning parametric policies directly without learning value functions is usually easier.

For our first implementation, we used the policy architecture already implemented in "SomeAgent" and "SomeOtherAgent", with a CNN. After spending several hours trying different hyper-parameters configurations and different pre-processing techniques nothing resulted in a successful training. Unfortunately we didn't save any training plots at this point because the performance was very poor.

Then, we decided to lower down complexity by a bit and try to learn a Policy Gradient both with the implementation of Karpathy directly over numpy [3] and with an implementation of PG inspired by the previous exercises of the course using Pytorch. Here again we could not achieve any inspiring results, getting to just 3% win ratio after several thousands of episodes.

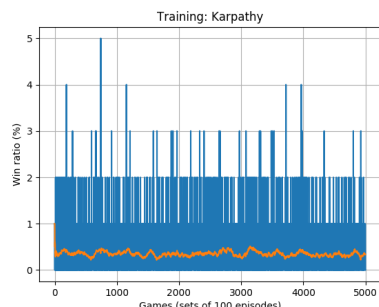


Figure 1: Karpathy Model

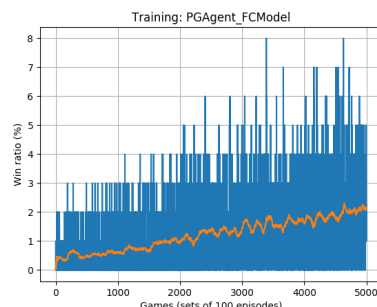


Figure 2: Policy Gradient

At this point, the frustration was quite extensive as we didn't understand (and still don't) why the same Karpathy's code the agent was learning in the original Pong environment but not in the wimblepong environment. We then sent the code to the Teaching Assistants of the course to help us find what we were doing wrong but, unfortunately, as they were also unable to find anything wrong with it we decided to start over again with more complex algorithms hoping to get better results once we use approaches with more sample efficiency.

After a short reviewing of the literature given on the project PDF and looking at benchmarks on performance in the Pong environment as [this one](#), we decided to implement either PPO [2] or ACER [4] as both showed to have very strong performance and we believed we had (or could get to have) a good understanding of them. Finally decided to go with PPO as we got to know that both "SomeAgent" and "SomeOtherAgent" were trained on PPO and got very decent results, and after all the work we have put so far we were desperately looking for a solution that actually works.

3.2 PPO approach

We have found Proximal Policy Optimization (PPO) as a good algorithm for our Pong environment that offers the possibility to update the loss with minibatches from a certain number of epochs [2] and handles a problem that a policy gradient methods might suffer: updating once the policy leads to a high change of action probabilities and this is something we want to avoid because usually there is also a certain amount of noise with the gradients [1]. The PPO algorithm instead, adds a penalty to the loss function penalizing large probability changes.

With this more sample efficient approach we finally started to see how our agent was able to learn. The results presented in Section 4 show our best results. To obtain such performance we used the following.

3.2.1 Policy

The policy network was constructed with two fully connected layers with 512 hidden neurons. We tried also with a bigger FC network with three layers, which didn't produce good results for us, and also with a CNN in order to speed up the training time, but sadly we didn't manage to make it learn as in the first few iterations it was always given 100% for one action which led to no exploration at all. We would have liked to keep on trying to improve it but we didn't have enough time for it.

3.2.2 Preprocessing

The preprocessing used consisted first on downsampling by a factor of two the input image and at the same time using just one color channel, which made our observation shrink from 200x200x3 to 100x100x1 still without losing valuable information. The next step was to transform the image into black and white, e.g. just using 0 and 1 values, 0 for the background and 1 for the paddle and ball, a technique used commonly on Reinforcement Learning as well as in Machine/Deep Learning to get rid of unwanted complexity and ease the training performance. And finally the latter step was to concatenate that current preprocessed observation with the previous one resulting on an input tensor of (2,100,100) for our policy network.

An other approach that we have used to feed the neural network is a difference between two images. As described above we crop the image to 200x200x3 into 100x100x1 taking one layer as setting the values of the image to 0 for the background and 1 for the paddles and for the ball, as explained in Karpathy implementation [3]. Then we do the difference between the current observation and the previous one scaled by 1/2. The picture below shows an example of subtraction of two consequential frames implemented in an early version of the project.

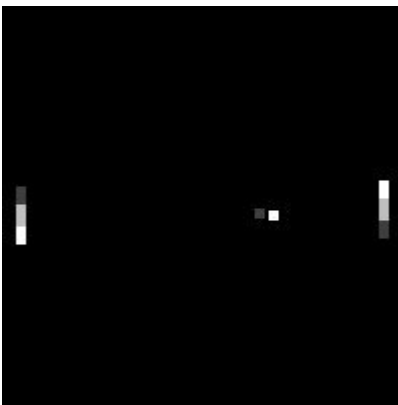


Figure 3: Trace of the ball from the previous obs.

The advantage of these two methods is that the neural network will be able to understand the direction of the ball and the paddles and to act consequently and thus have better performances.

3.2.3 Hyperparameters

The choice of hyperparameters is mainly due to both the publication on PPO [2] and Sagar Gubbi’s article [1]. As a summary we present here a table with all the hyperparameters used for our PPO algorithm:

Hyperparameter	Value
Episode Batch Size	200
Learning Rate	1×10^{-3}
Num. Epocs	5
Minibatch Size	70%
Discount factor	0.99
Clipping	0.1

Note that the minibatch size was then reduced to $\approx 20\%$ of the total batch but we are leaving in the table the value used for the results presented in the next section.

3.3 Final refinements

Finally, once we examined our agent already with decent performance (more than 50% win ratio on average) we noticed that it was mimicking the opponent, following it as a mirror, and just at the last moment tried to hit the ball with the corners in order to make the ball bounce a lot and win the game. Observing this we were afraid that this behaviour won’t be ideal against other kind of players.

To confirm this prognostic, we tested our agent competing against itself and here it was shown that in fact that behaviour was not good enough since the agents were not able to reflect the ball and as a result the games didn't last almost anything. The reason for this is that since the opponent was no longer following the ball (as the SimpleAI was) the player agent was not able to reflect the ball almost at any time. This fact made us believe that our performance in the competition was going to be very poor.

In order to improve this, we decided to start again another training in which we erased the opponent's paddle and concatenated three observations instead of two. With this approach we started to see slow but promising results but unfortunately our training got interrupted as we were using the shared machines of Panniki and at this point we were very short on time.

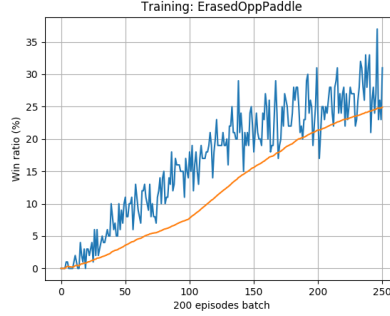


Figure 4: PPO Agent without opponent Paddle

4 Results and Performance analysis

The section below shows the result obtained with the Proximal Policy Optimization algorithm, that is the method that performs better with our implementation.

The graphics of two versions of the algorithm will be proposed.

- Results 1 - Action space of two (UP and DOWN)
- Results 2 - Action space of three (STILL, UP and DOWN)

Our submitted code and model are using the version with two movements because at delivery time was more mature and having better performances.

4.1 Results

4.1.1 Results with two actions

The plot below was expected to be one plot but we propose it split in three parts because during the training that took about two days, the training process had been killed and restored twice.

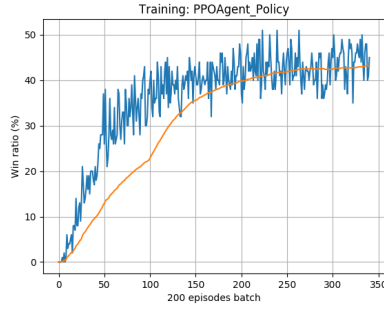


Figure 5: Two actions - Part 1

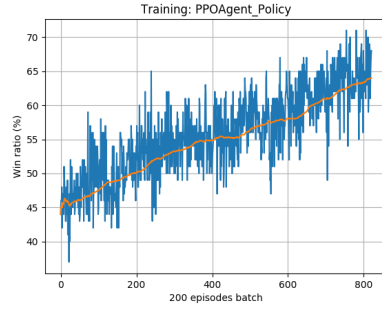


Figure 6: Two actions - Part 2

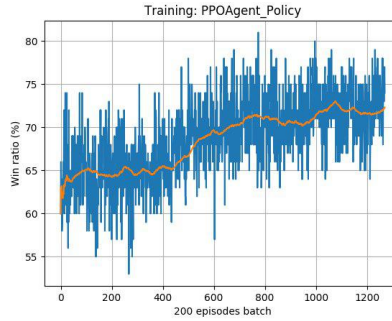


Figure 7: Two actions - Part 3

4.1.2 Results with three actions

In the plots above the win ratio is higher than the one achieved by the model submitted for the competition because the two agents had more time to train and scored better results. We believe that if we will leave our networks for other few days the performance will be even higher.

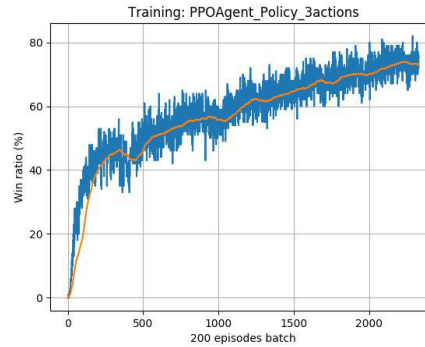


Figure 8: Three actions

4.2 Performance analysis

From the graphics we can see that both the versions of PPO reached a win ratio of 40% just after 200 iterations. At model submission time the 3-actions algorithm was performing well, but the 2-action PPO was more mature and with a higher win ratio, so we decided to upload that one.

So for the following section we will analyse the performance of 2-action algorithm against the SimpleAI, that was the same algorithm used for the training. Since the result of a game of 100 episodes suffers of variance we decided to play a game of 1000 episodes.

It will follow the results of the test:

```

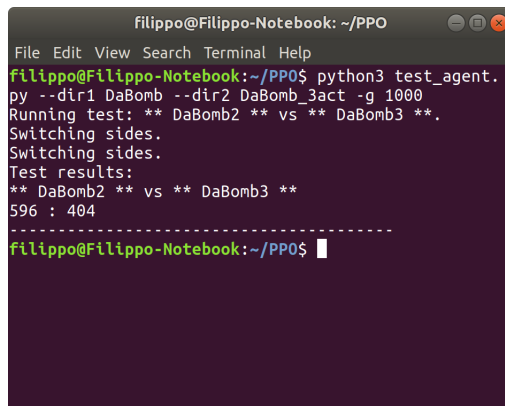
filippo@Filippo-Notebook: ~/PPO
File Edit View Search Terminal Help
filippo@Filippo-Notebook:~/PPO$ python3 test_agent.py
--dir1 DaBomb -g 1000
Running test: ** DaBomb2 ** vs SimpleAI.
Switching sides.
Switching sides.
Test results:
** DaBomb2 ** vs SimpleAI
683 : 317
-----
filippo@Filippo-Notebook:~/PPO$

```

Figure 9: Our 2-Action PPO Agent vs SimpleAI

As we can see we have obtained a win ratio of about 68% against the SimpleAI that is in line with the results of the plot above.

Then we tried to analyse the performance of our agent when it played against the 3-Action version of PPO. Also in this case we will propose a game of 1000 episodes.

A terminal window titled 'filippo@Filippo-Notebook: ~/PPO' with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows the execution of a Python script 'test_agent.py' with arguments '--dir1 DaBomb --dir2 DaBomb_3act -g 1000'. The output indicates a test between 'DaBomb2' and 'DaBomb3', with 'Switching sides.' printed twice. The final 'Test results:' show '596 : 404' for 'DaBomb2' vs 'DaBomb3'.

```
filippo@Filippo-Notebook: ~/PPO
File Edit View Search Terminal Help
filippo@Filippo-Notebook:~/PPO$ python3 test_agent.py --dir1 DaBomb --dir2 DaBomb_3act -g 1000
Running test: ** DaBomb2 ** vs ** DaBomb3 **.
Switching sides.
Switching sides.
Test results:
** DaBomb2 ** vs ** DaBomb3 **
596 : 404
-----
filippo@Filippo-Notebook:~/PPO$
```

Figure 10: 2-Action PPO vs 3-Action PPO

As we can see from the image the 2-Action Agent performs slightly better than the 3-Action one, probably because it has been trained for more time rather than the 3-Action PPO.

5 Conclusions

This project has allowed us to have a better understanding on more complex algorithms of Reinforcement Learning, such as PPO or ACER. We also learnt that RL approaches take more time to train than we initially expected and how sensitive they are to the configuration of hyperparameters. Also related to the time needed to train we also experienced how crucial the sample efficiency is.

As a general overview we think that working on this project has taught us the difference that there is between an exercise (homework) and a real task in which we have to implement a reinforcement learning algorithm from scratch. We think that working with the project have required us to learn a broader knowledge not only about reinforcement learning but also how neural networks work (i.e. CNN) and has shown us what are the real aspects and limitations in a reinforcement learning project (training time requirements, hardware capabilities, right algorithms and hyperparameters).

6 Acknowledgement

We would like to thank the teaching assistants for their support through all the development of this project and for their valuable feedback.