Reinforcement Learning

Rainbow model: analysis and evaluation

Author :
NASCIMENTO, Leandro
MULIUKOV, Artem

E-mail :
dearaujo@telecom-paris.fr
artem.muliukov@telecom-paris.fr

6 mars 2020

This project is a study of the Rainbow Reinforcement Learning Model, from the article : "Rainbow : Combining Improvements in Deep Reinforcement Learning", available at : `https://arxiv.org/abs/1710.02298` [1]

This work is an analysis of the model described on the paper and an evaluation of it in Atari games, using the OpenAI gym python framework.All the notebook files and complementary material are also available at our GitHub repository (`https://github.com/leandrones/Rainbow_RL`).

1 Paper analysis

1.1 The Model

The Rainbow model basically assembles 6 different DQN (Deep Q-Network) modifications in order to achieve an overall more robust RL agent, specially when tested with the Atari games environment. The 6 DQN modifications chosen by the authors were :

- DDQN (Double DQN) - PER (Prioritized Experience Replay) - Dueling Network - Multi-step learning - Distributional RL - Noisy Networks

We will do an overview of each of these variations as well as of the integrated model (Rainbow).

1.1.1 DDQN

As it is widely known that regular Q-learning has an over-optimism towards its estimation of the Q-values for the states (even with the ϵ-greedy strategy), the DDQN tries to solve this issue by partially decoupling the maximization step of the Q-function. This transforms of the loss function of DQN, that was :

$$\left(R_{t+1} + \gamma_{t+1} \max_{a'} Q_{\bar{\theta}}(S_{t+1}, a') - Q_\theta(S_t, A_t)\right)^2$$

into :

$$(R_{t+1} + \gamma_{t+1} Q_{\bar{\theta}}(S_{t+1}, \arg\max_{a'} Q_\theta(S_{t+1}, a; \theta_t)) - Q_\theta(S_t, A_t))^2$$

This separation of selection of action from the evaluation reduces the general overestimations of DQN.

1.1.2 Prioritized Replay (PER)

The paper shows that, in many games, this was the factor that most improved performance on the Rainbow model. This is because it changes the way data is sampled from the replay buffer, a major part of DQN. Originally, DQN samples uniformly at random from the buffer. PER changes this uniform distribution into a new one p_t, proportional to :

$$p_t \propto |R_{t+1} + \gamma_{t+1} \max_{a'} Q_{\bar{\theta}}(S_{t+1}, a') - Q_\theta(S_t, A_t)|^\omega$$

Where ω is an another hyper-parameter. This method creates a bias for choosing more recent transitions. (We should notice that on the code implementations, this ω is often called as β).

1.1.3 Dueling networks

This method is basically a decomposition of the Q-value function in a "value" and "advantage" stream, with is an encoder for the state variable. This results in the following function :

$$q_\theta(s,a) = v_\eta(f_\xi(s)) + a_\psi(f_\xi(s), a) - \frac{\sum_{a'} a_\psi(f_\xi(s), a')}{N_{\text{actions}}}$$

In this case, θ is the concatenation of ξ, η, ψ

1.1.4 Multi-step learning

The idea here is to make a forward view greater than one step for the reward, as it is normally done. So instead of using R_{t+1}, we define : $R_t^n \equiv \sum_{k=0}^{n-1} \gamma_t^k R_{t+k+1}$. Also, we replace γ_{t+1} in the loss function for : $\gamma_t^n = \prod_{i=1}^{k} \gamma_{t+i}$

1.1.5 Distributional RL

This variation probably is the one which changes DQN the most, in terms of implementation. DQN minimizes (the expectation) of the loss, since it is stochastic (or mini-batch). The idea is then to try to find the distribution of returns instead of the expected return. This is done with probability masses. We have a vector $\mathbf{z}$, where each $z^i = v_{\min} + (i-1)\frac{v_{\max} - v_{\min}}{N_{\text{atoms}} - 1}$ with $i \in \{1, ..., N_{\text{atoms}}\}$. Then, we have that the distribution of each atom i at time t is $d_t = (\mathbf{z}, \mathbf{p}_\theta(S_t, A_t))$.

Finally, the new loss to be minimized will be the Kullback-Leiber divergence (D_{KL}) between d_t and the target $d_t' \equiv R_{t+1} + \gamma_{t+1}\mathbf{z}, p_{\bar{\theta}}(S_{t+1}, \bar{a}_{t+1}^*)$, with $\bar{a}_{t+1}^* =$

$\arg\max_a q_{\bar{\theta}}(S_{t+1}, a)$ being the greedy action for the optimal policy. More precisely, the D_{KL} is calculated between the L2-projection of d'_t and d_t, that is, $D_{KL}(\mathbf{\Phi_z} d'_t || d_t)$.

1.1.6 Noisy Networks

The premise of Noisy Nets is that, for complex systems, the smallest difference on the weights of the network will make the agent too state dependent after multiple time-steps. So, the idea is to add a noise on the network in order to improve the agents exploration in the state space. This happens because the network will eventually ignore the noise, but at different rates and different parts of the state space.

Given a linear network as $\mathbf{y} = \mathbf{Wx} + \mathbf{b}$, we transform it into :

$$\mathbf{y} = \mathbf{Wx} + \mathbf{b} + \mathbf{b_{noisy}} \odot \epsilon^b + (\mathbf{W}_{noisy} \odot \epsilon^w)\mathbf{x}$$

where the ϵ are factorized Gaussian noise random variables.

1.1.7 The integrated agent

As the authors recommend, the first step is to change the distributional loss to a n-step version, such that $d_t^n = (R_t^n + \gamma_t^n z, p_{\bar{\theta}}(S_{t+n}, \bar{a}^*_{t+n}))$

Then, a greedy action in S_{t+n} gets us our DDQN. In the PER, our p_t will be proportional to $p_t \propto (D_{KL}(\Phi_{\mathbf{z}} d_t^n || d_t))^\omega$

As the Noisy Nets are trivial to adapt, we have left only the Dueling Networks. We create the shared representation $\phi = f_\xi(s)$ and then we have our probabilities of the atom z^i as :

$$p_\theta^i(s,a) = \frac{\exp(v_\eta^i(\phi) + a_\psi^i(\phi,a) - \bar{a}_\psi^i(s))}{\sum_j \exp(v_\eta^j(\phi) + a_\psi^j(\phi,a) - \bar{a}_\psi^j(s))}$$

,

with $\bar{a}_\psi^i = \frac{1}{N_{\text{actions}}} \sum_{a'} a_\psi^i(\phi, a')$, and we are finished with the integration.

2 Our work

So, as we can see, this Rainbow Model has, at the same time, a high complexity of implementation and a long time for learning. So we decided to analyze the different implementations of Rainbow that are available on GitHub. Sure enough, we encountered many different ones, using TensorFlow or PyTorch. However, we found one of the PyTorch implementations ("Rainbow is all you need" [2]) that did not use the CNN part of the Neural Network, because this model was tested only with the CartPole Model (that is much simpler than Atari games). Also his feature network was only a Fully Connected Layer of 128 neurons, so it was capable to learn with the RAM memory option that openAI Gym gives. It is important to say already that all implementations studied were using OpenAI Gym rather than the paper's choice, which was the Arcade Learning Environment (ALE).

So, we decided to take this implementation, study it in detail, and modify it in order to make it learn with the pixels of Atari Games, as it is done on the paper. This entails the use of the Gym environment Wrappers. Those are modules available on OpenAI [3]. Then, we basically chose and coded some of the wrappers that were important for us. For instance, we need the sub-sampling wrapper to reduce the dimensionality of the Atari frames. They are originally of size (210x160x3) and they are changed to (84x84x1). Then, another wrapper stacks 4 consecutive frames and a state tensor becomes of shape (84x84x4).

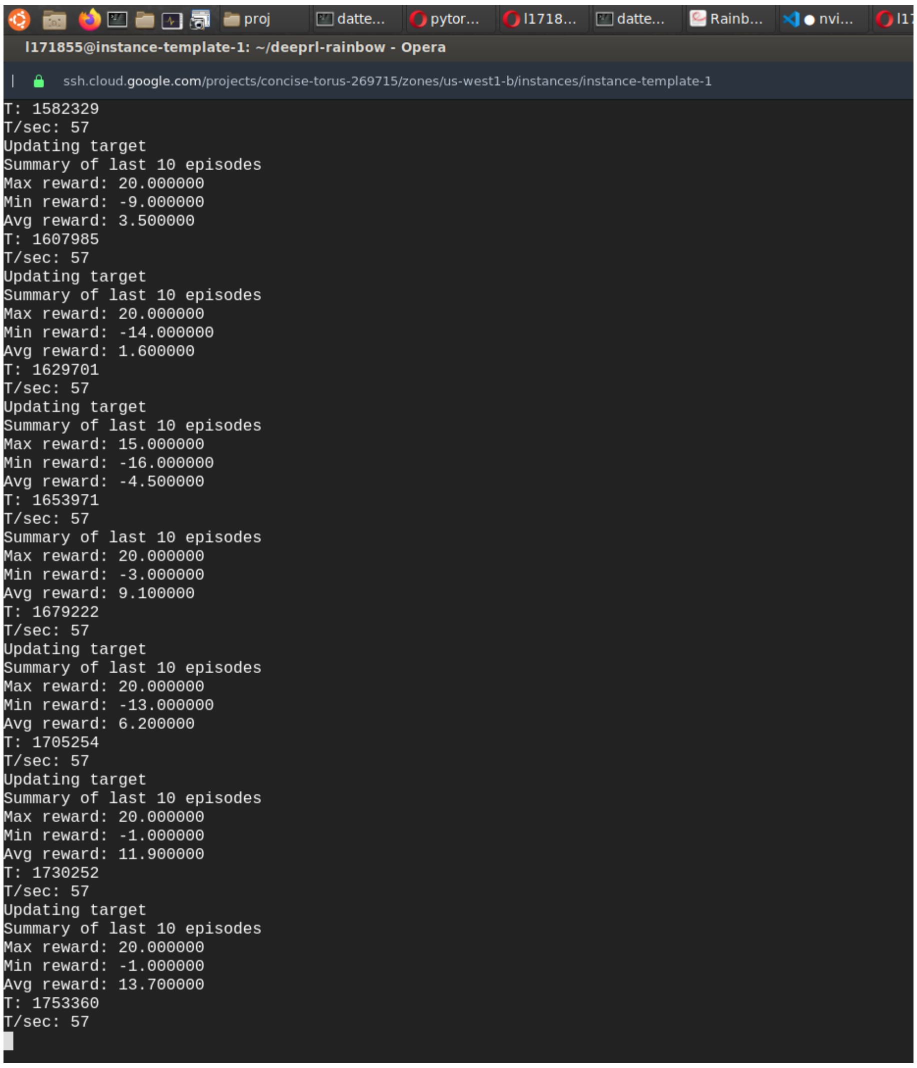
Moreover, this implementation also lacked some hyperparemeters, such as the 'min history to start learning', which can actually be crucial to the agent's progress. Hence, we modified the code to make sure we took these parameters into account. We discovered also from other implementations, that the best configuration of the openai Atari is "<gameName>NoFrameskip-v4", so we also chose this one.

Another important part of our work was how to tackle the time complexity that is intrinsic with deep reinforcement learning algorithms and even more so for Rainbow. For the lack of not having GPUs and for the fact that Google Colab limits free run times to 12 hours, we eventually turned to the GCP (Google Cloud Platform) solution. We benefited from the $ 300 credit from a new account, but now we had run into another issue of how to create a data science jupyter environment inside a cloud VM. So, following some references such as [4], we managed to instantiate a virtual machine inside GCP, with an 8 core CPU, 30GB of RAM and a Nvidia Tesla K80. However, soon we saw this VM did not contain not even the Nvidia drivers nor GCC or python. Therefore, amongst other sources, with followed this reference [5] to install (citing the main parts) the drivers, Cuda, CudNN, Anaconda, PyTorch, and then also OpenAI, OpenCV, and TensorFlow (as we ended up needing it).

2.1 Preliminary, first tests

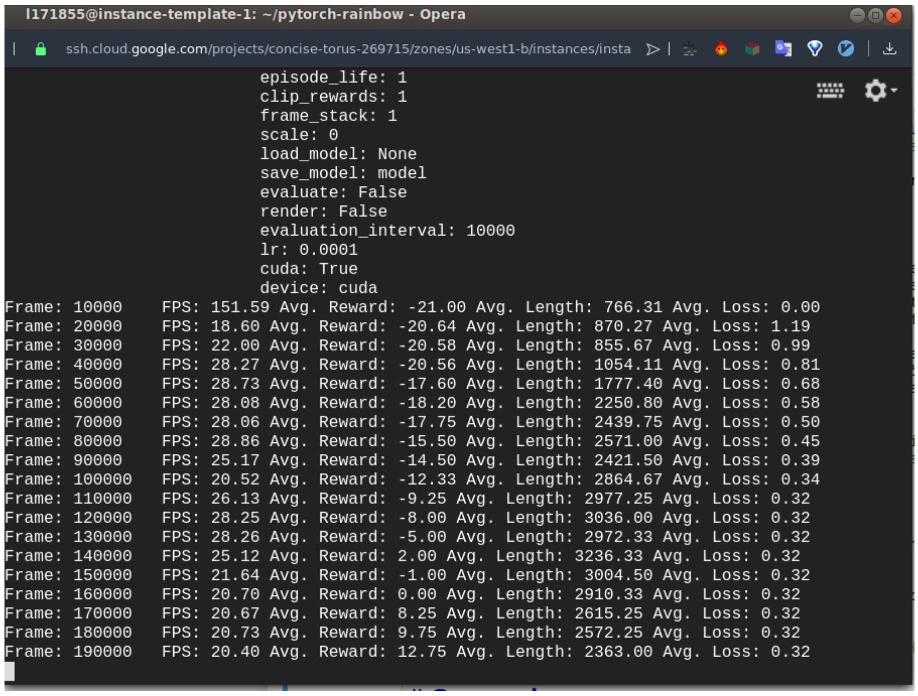
After not having too much success with Colab, we went to execute our modified code on the GCP. The file .ipynb with the modified version of Rainbow is all you need repository is the one called `rainbow_is_all_you_need_08_modified`. Our choice for the game was Pong, which even though other simpler algorithms can also learn it, we saw from the paper's appendix, that it had maybe the smallest number of frames necessary to learn how to play the game.

Meanwhile, we also decided to benefit from the cloud GPU to test other Rainbow implementations, that already used convolutional layers and the correct Wrappers. So we tested with the same Pong game, one implementation using TensorFlow [6] and another PyTorch [7]. We can see the results in fig. 2.1 and 2.2.


proj
datte...
pytor...
l1718...
datte...
Rainb...
nvi...
l171855@instance-template-1: ~/deeprl-rainbow - Opera
ssh.cloud.google.com/projects/concise-torus-269715/zones/us-west1-b/instances/instance-template-1

```
T: 1582329
T/sec: 57
Updating target
Summary of last 10 episodes
Max reward: 20.000000
Min reward: -9.000000
Avg reward: 3.500000
T: 1607985
T/sec: 57
Updating target
Summary of last 10 episodes
Max reward: 20.000000
Min reward: -14.000000
Avg reward: 1.600000
T: 1629701
T/sec: 57
Updating target
Summary of last 10 episodes
Max reward: 15.000000
Min reward: -16.000000
Avg reward: -4.500000
T: 1653971
T/sec: 57
Summary of last 10 episodes
Max reward: 20.000000
Min reward: -3.000000
Avg reward: 9.100000
T: 1679222
T/sec: 57
Updating target
Summary of last 10 episodes
Max reward: 20.000000
Min reward: -13.000000
Avg reward: 6.200000
T: 1705254
T/sec: 57
Updating target
Summary of last 10 episodes
Max reward: 20.000000
Min reward: -1.000000
Avg reward: 11.900000
T: 1730252
T/sec: 57
Updating target
Summary of last 10 episodes
Max reward: 20.000000
Min reward: -1.000000
Avg reward: 13.700000
T: 1753360
T/sec: 57
```

FIGURE 2.1 – Terminal screenshot of a Rainbow TensorFlow implementation [6].


l171855@instance-template-1: ~/pytorch-rainbow - Opera
ssh.cloud.google.com/projects/concise-torus-269715/zones/us-west1-b/instances/insta

```
                          episode_life: 1
                          clip_rewards: 1
                          frame_stack: 1
                          scale: 0
                          load_model: None
                          save_model: model
                          evaluate: False
                          render: False
                          evaluation_interval: 10000
                          lr: 0.0001
                          cuda: True
                          device: cuda
Frame: 10000    FPS: 151.59 Avg. Reward: -21.00 Avg. Length: 766.31 Avg. Loss: 0.00
Frame: 20000    FPS: 18.60 Avg. Reward: -20.64 Avg. Length: 870.27 Avg. Loss: 1.19
Frame: 30000    FPS: 22.00 Avg. Reward: -20.58 Avg. Length: 855.67 Avg. Loss: 0.99
Frame: 40000    FPS: 28.27 Avg. Reward: -20.56 Avg. Length: 1054.11 Avg. Loss: 0.81
Frame: 50000    FPS: 28.73 Avg. Reward: -17.60 Avg. Length: 1777.40 Avg. Loss: 0.68
Frame: 60000    FPS: 28.08 Avg. Reward: -18.20 Avg. Length: 2250.80 Avg. Loss: 0.58
Frame: 70000    FPS: 28.06 Avg. Reward: -17.75 Avg. Length: 2439.75 Avg. Loss: 0.50
Frame: 80000    FPS: 28.86 Avg. Reward: -15.50 Avg. Length: 2571.00 Avg. Loss: 0.45
Frame: 90000    FPS: 25.17 Avg. Reward: -14.50 Avg. Length: 2421.50 Avg. Loss: 0.39
Frame: 100000   FPS: 20.52 Avg. Reward: -12.33 Avg. Length: 2864.67 Avg. Loss: 0.34
Frame: 110000   FPS: 26.13 Avg. Reward: -9.25 Avg. Length: 2977.25 Avg. Loss: 0.32
Frame: 120000   FPS: 28.25 Avg. Reward: -8.00 Avg. Length: 3036.00 Avg. Loss: 0.32
Frame: 130000   FPS: 28.26 Avg. Reward: -5.00 Avg. Length: 2972.33 Avg. Loss: 0.32
Frame: 140000   FPS: 25.12 Avg. Reward: 2.00 Avg. Length: 3236.33 Avg. Loss: 0.32
Frame: 150000   FPS: 21.64 Avg. Reward: -1.00 Avg. Length: 3004.50 Avg. Loss: 0.32
Frame: 160000   FPS: 20.70 Avg. Reward: 0.00 Avg. Length: 2910.33 Avg. Loss: 0.32
Frame: 170000   FPS: 20.67 Avg. Reward: 8.25 Avg. Length: 2615.25 Avg. Loss: 0.32
Frame: 180000   FPS: 20.73 Avg. Reward: 9.75 Avg. Length: 2572.25 Avg. Loss: 0.32
Frame: 190000   FPS: 20.40 Avg. Reward: 12.75 Avg. Length: 2363.00 Avg. Loss: 0.32
```

FIGURE 2.2 – Terminal screenshot of a Rainbow model with PyTorch (CNN already implemented) [7].

As we can see, both these implementations are capable of learning Pong. It is important to notice that we did not show results of millions of iterations, as it is commonly done. It was not our purpose, as it would take too long to train them completely, even with the GPU. Nevertheless, we can see how fast Rainbow is able to start learning. This is in agreement with the appendix graph of the paper, that shows Pong results jumping very quickly from negative to positive scores. This was the same case with us in both versions, as we see in both terminals that in one print, the average score is for instance -4 and in the next it is 9.

Other interesting comparisons can be made regarding the speed of each version. Using the same VM on the cloud, the TensorFlow version showed us it has a higher frame rate processing that the PyTorch one : the first being 57 FPS and the second, around 25. This can be due maybe for the different frameworks, but not only. For instance, the buffer implementation can vary a lot and this may also influence on the time efficiency of the algorithm.

2.2 Modifications made

Note : all modifications made by us are marked with a commentary above written `our code`.

Coming back to our modified code. Firstly, we basically coded some important wrappers. For instance, we need the sub-sampling wrapper to reduce the dimensionality of the Atari frames. They are originally of size (210x160x3) and they are changed to (84x84x1). Then, another wrapper stacks 4 consecutive frames to tensor of shape (84x84x4). So we could see the evolution of the game fields over the time.

The initial implementation for the code was tested on the more common tasks, such as CartPole. The reward when working with this algorithm is much easier to calculate (we reward every second of survival of the inverted pendulum). In the case of the Pong game, learning is much tougher, since the reward is issued only at the end of the game. So, we tried to slightly change the issued reward. For example, encourage every second of the game (long game - the algorithm knows how to hit the ball). That may significantly influence to the speed of training and performance, as we could give any possible value, and to give 0.1 or 0.001 may be two completely different approximation tasks. But we haven't come to the perfect value.

Plus, we have tried to change various hyperparameters of the network, such as gamma alpha and beta (which is omega in the article). Although this helped, for instance, to get rid of the problem of sticking the strategy to one solution (the agent always press the same button), it did not improve the final game performance.

2.3 Learning Atari using RAM

As we did not have the best results with this modification, we decided to try to use the other PyTorch implementation that worked with CNN [7], and try to make it work with the RAM memory of the Atari The Atari memory is composed only by 128 bytes and it comes as a state array of size 128 with integer values from 0 t0 255. So, following this article [8], we changed the CNN and Flatten parts to 2 Fully Connected and not Noisy Layers with 128 neurons each, and we changed the 2 Noisy Layers that exists in Rainbow and have 512 neurons, we made them with 128. However, the Rainbow Model did not seem to even start learning within the 200 K iterations. Also, we tried only with the 2 Noisy layers, still being based on [8], which actually had slight better results than with the extra 2 layers. These lats training result values can be seen on the `pytorch_rainbow_RAM_Pong` notebook file. We selected also a new best learning rate $\approx 10^{-7}$ (the old one $\approx 10^{-3}$ leaded us to the over-weighted net).

2.4 Other games

We also took this PyTorch implementation [7] to be tested with games other than Pong. This was the only version with a pretrained model for Pong that could be verified that really worked, so we decide to see if it performed well for another game. This game was Qbert, that from the paper also showed as being learned quite quickly by rainbow. The result we obtained is present in detail on the file `pytorch_rainbow_Qbert`, but we can say that this implementation does learn how to play Qbert, however it seems quite slow. The paper showed us a graph in a scale of 10^4, but we, with 240 K iterations, obtained an average reward of 4.62. However, as we said, this reward was globally increasing, but not quite fast.

3 Conclusion

What we could learn from this project was all the theory details of the Rainbow Model and how complex it is to adapt 6 modifications in the base DQN. Not only in theory, we read and understood how complex it can be also to implement all of these features at once and how tricky it is to manipulate PyTorch to make your model learn the parameters and optimize the loss function.

We also saw the different performances of two different Rainbow versions, one using TensorFlow and one, PyTorch and we saw that the implementation we chose to modify is much slower and has some issues to address. Moreover, the RAM state model was found to be not as consistent as learning from the pixel frames, which makes sense since the literature uses in general much more the frame state than the RAM state. Finally, we were able to test a functioning implementation for Pong and saw that it also learns the game Qbert, but apparently not as quickly as the paper model did.

Références

[1] "Rainbow : Combining Improvements in Deep Reinforcement Learning". `https://arxiv.org/abs/1710.02298`

[2] Curt-Park/rainbow-is-all-you-need. `https://github.com/Curt-Park/rainbow-is-all-you-need`

[3] Openai baselines. `https://github.com/openai/baselines/blob/master/baselines/common/atari_wrappers.py`

[4] Running Jupyter Notebook on Google Cloud Platform in 15 min. `https://towardsdatascience.com/running-jupyter-notebook-in-google-cloud-platform-in-15-min-61e16da34d52`

[5] 0.2 - Install PyTorch on Ubuntu. https://www.youtube.com/watch?v=PSCORVcubnA

[6] coreystaten/deeprl-rainbow (TensorFlow implementation). https://github.com/coreystaten/deeprl-rainbow/

[7] belepi93/pytorch-rainbow. https://github.com/belepi93/pytorch-rainbow

[8] Learning from the memory of Atari 2600. Jakub Sygnowski and Henryk Michalewski. https://arxiv.org/pdf/1605.01335.pdf