Research Project Report : Rainbow Algorithm

Paul Carfantan
Institut Supérieur de l'Aéronautique et de l'Espace (ISAE-SUPAERO), 31055 Toulouse, FRANCE
paul.carfantan@student.isae-supaero.fr

Abstract

This research project focuses on implementing a state-of-the-art algorithm that consists of multiple major improvements on an elementary Reinforcement Learning algorithm. That algorithm will both learn tasks more efficiently and perform those tasks better after the training. Therefore we will need to base ourselves on already existing codes, and combine their functions in a way that enhances the global performance.

1 Introduction

The SuReLI (Supaero Reinforcement Learning Initiative) research team's global purpose is to find the best ways to solve problems through Reinforcement Learning. The main concern in this field of Artificial Intelligence is to make a machine learn how to chose actions for an agent in a given environment in order to maximize a reward.

One of the well-known building points of Reinforcement Learning is named DQN for Deep Q-Networks [1]. In this algorithm, we use a neural network, which is a complex structure composed of inter-connected elementary computational units called neurons. Those units are the parameters of the network, and may be modified to change its behavior.
Once the neural network is well trained on a task, and given a state as input, it has to be able to output which action we should take from this state to get the highest long-term reward.
But how to train a neural network ? For that we need the notion of loss : a function used to evaluate the current performances of the network, that decides how well the task is satisfied. By a process of gradient descent on this loss, it is then possible to optimize the parameters of the network to minimize the loss, which means fitting better the given task.

As several works based themselves on DQN [1], one of them stands out from the mass by its state-of-the-art performance. Known as Rainbow [2], it combines 6 major improvements of DQN from other papers [3, 4, 5, 6, 7, 8] into one single algorithm. The performance of Rainbow compared to DQN's and to the individual improvements' can be seen on Figure 1 below describing the results of [2].

2 Aim

As the results presented (see Figure 1) are really appealing, it would be interesting for SuReLI to possess an implementation of Rainbow and to use it inside the current research works, preferably under the PyTorch library for practical reasons. Hessel et al. wrote a paper in 2017 about their works on Rainbow [2], however the code itself has not been published. Some tries have been made to replicate it but unfortunately, none of those made available on the internet managed to implement all 6 elements of Rainbow with the desired python library.

That is why this project aims to implement Rainbow algorithm in PyTorch following the information given in the corresponding article. Once implemented, the code will have to be tested and validated on various videogame benchmarks.

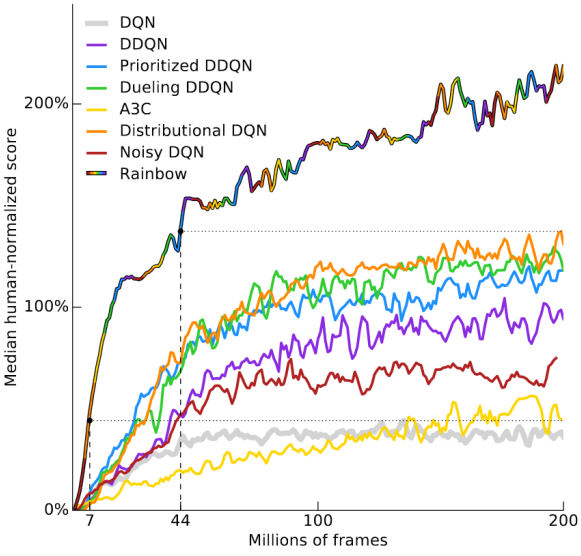

Figure 1: Performance of Rainbow compared to other algorithms [2]

3 Implementation

The first step of this project consisted of a bibliography work, each one of the 6 improvements being described in a specific paper : the good understanding of those was essential for a rigorous manipulation of the associated algorithms. For some of them, the code was fortunately provided either by the authors or by a third party. Hence the next step was to get started on the different codes, by reading and commenting them through many tests to truly understand the function of each part. Then, I based myself on a working DQN code, and added one by one the different improvements that compose Rainbow.

0. **DQN** : in a classical DQN [1], we commonly use one neural network to compute the following loss function and optimize the network thanks to this function :

$$\mathcal{L}_{DQN} = [R_{t+1} + \gamma \max_{a'} q_\theta(S_{t+1}, a') - q_\theta(S_t, A_t)]^2 \quad (1)$$

where R_{t+1} is the reward returned by the environment, γ the discount factor, q the state-action value function, representing the expected reward we should get by taking a given action in a given state. θ characterizes the parameters of the neural network through which q is calculated ; S_{t+1}, S_t and A_t represent respectively the next state returned by the environment, the current state and the action taken from this state.
An Experience Replay principle is used in DQN : that means that for each action taken, we store uniformly the transition in a buffer to make the network "inspire" from those samples. A transition is characterized by the current state, the action taken from it, the new state and the reward obtained, along with a variable traducing wether the game just finished with that action or not.
Finally the common process of DQN is based on ϵ-greedy exploration, meaning that when it chooses an action, a random action is taken with a probability ϵ instead of using the network.

1. **Double DQN** : the difference with DQN is that we dissociate the choice of the next action (corresponding to the maximization step) from the evaluation of the impact of this action on the global reward. For that, a second similar neural network is introduced and updated regularly to resemble to the main one. Those two networks are called current and target network. It leads to the use of a new loss function [3] :

$$\mathcal{L}_{DDQN} = [R_{t+1} + \gamma q_{\overline{\theta}}(S_{t+1}, \underset{a'}{\text{argmax}}(q_\theta(S_{t+1}, a'))) - q_\theta(S_t, A_t)]^2 \tag{2}$$

with θ and $\overline{\theta}$ meaning respectively that q is evaluated respectively through the current network and the target network.

The concrete modification to the DQN code here was to initialize a second network (target) with the same structure than the first one (current), and create a function to copy all the weights of current network into the target network. This function is called regularly so that the computations using target network remain valid. This target network is then included into the loss as described above.

2. **Dueling DQN [4] :** this improvement consists of decoupling a part of the network to make it compute separately a term called Advantage and an other called Value, and then combine them to form the output of the network, in the following way :
$Out = Val + Adv - \text{mean}(Adv)$. That implies that those two parts of the network will be optimized independently from each other, which brings a better efficiency.
Here, only the structure of the neural network has been modified by adding the two streams of computation at the end of the network, and then a "merger" channel to form the wanted output.

3. **Multistep return :** for the computation of the loss in classical DQN, we only use the current reward, the current state and the next one, and the action we take to get there.
In multistep [5], instead of the current state, we rather take into account the n-previous state, and instead of the current reward, a n-step reward $R^{(n)}$ as defined below :

$$R^{(n)} = \sum_{k=0}^{n-1} \gamma^k R_{k+1} \tag{3}$$

where R_1 is the last reward obtained and R_n the n-previous one.
This brings to different modifications to the code. First, the Experience Replay part : instead of stocking only one reward, it now takes a list of the n last rewards, and instead of the current state, it takes the n-previous one. Second, the loss computation is affected too as we replace in it the current reward by the n-step one. We also add a factor that cancels the using of n-step reward if the game is finished. In future works, this factor may have to be modified as its implementation coherence remains uncertain.
Multistep return enables a better propagation of the current rewards to the previous states and hence a better optimization of the network.

4. **Noisy Networks :** in order to get a better exploration in our process (meaning that the algorithm will less get stuck into local minima instead of searching the global one), we add to the network some noisy layers of computation [6]. This method replaces the ϵ-greedy exploration characterizing DQN.

For this one, I reused a code that had already been implemented by the SuReLI team.

5. **Prioritized Replay :** instead of DQN's Experience Replay where the transitions are stored uniformly in the buffer, Prioritized Experience Replay [7] assigns priorities to each transition, influencing the probability to be selected. These priorities are proportional to the loss : in this way, transitions with a bigger learning potential will more likely be sampled. Moreover, each transition gets associated with a weight translating that learning potential. These weights will then be used as factors of the loss function.
   Here, I used the Prioritized Replay code from OpenAI Baselines which uses a SegmentTree data structure, enabling more efficient calculations and queries through the buffer. This one had to be modified to fit with the multistep return, as explained above for Experience Replay. Finally, the loss function also had to be modified to take the weights into account, and to update the priorities.

6. **Distributional DQN :** Instead of giving as output of the neural network a mean of probabilities which forms the expected global reward ; Distributional DQN [8] proposes to output the whole probability distribution of returns as a histogram. This allows far more precise results in terms of neural network optimization.
   Unfortunately, this part of my Rainbow implementation remains incomplete. Indeed, I first tried to use an already existing SuReLI Distributional DQN, but the results were unsatisfying, and I got to the conclusion that I needed a Distributional DQN of my own, adapted to my Rainbow implementation. Regrettably I did not have the time to achieve this, thus it will have to be treated in future works.

4 Results

After the inclusion of each one of the improvements, many tests were performed, first to validate the implementation and correct the potential coding errors, and second to tune the hyperparameters as precisely as possible to get the best results. Those experiments were made on Gym benchmark, especially on CartPole-v0, LunarLander-v2 and Acrobot-v1. I will show below the most significant graphs obtained through the experimentation process.

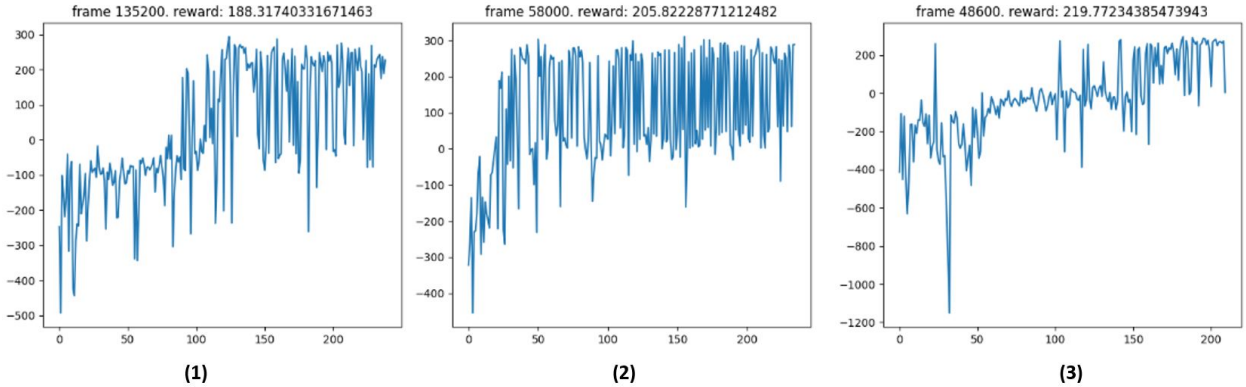

Figure 2: Experimentations on game LunarLander at three phases of implementation

On Figure 2 is shown the evolution of global reward through the training of our network on LunarLander at three different steps of my work. The game is considered solved when the reward is 200 or higher ; the ordinate axis corresponds to the reward, and the abscissa axis to the number of episodes (how many games have been played).

(1) corresponds to the performance of the basic DQN, (2) to the performance of the combination of Dueling DQN, Double DQN and multistep return, and (3) to the final implementation, i.e. all the improvements except Distributional DQN.
We can observe that (2) reaches positive rewards and even 200-reward far earlier than (1), meaning that the training is more efficient ; however there is still no real convergence and the reward remains very noisy between 50 and 300. This problem seems to be addressed in (3), where the convergence looks really better after the same number of episodes ; however the training is again longer.

In (3) we can observe three different phases : first, a noisy one until 50 episodes, then a stabilization between -100 and 0 for around 100 episodes, and a climb to finally reach a quasi-convergence around 200. Knowing that LunarLander is a game where we have to make a rover land in a given zone by controlling 3 motors, we can interpret these phases as following. The first one corresponds to quasi-pure exploration, then the agent *understands* how to control the rover, and makes it fly nearly in a stationary way (which explains the still poor reward), but continues a little exploration ; and finally exploration shows that a greater reward was obtained if the rover landed after having flown : thus the agent sticks to this strategy.

These graphs, along with some other results of experiments which resemble the previously described ones, tend to validate the implementation in terms of performance.

In future works, the inclusion of Distributional DQN should give more precise results and maybe a better convergence ; the algorithm will also have to be tested on more games, as the benchmark we used here is limited in terms of difficulty and resolution time.

5 Conclusion

This research project was initially supposed to go further than the implementation of Rainbow : indeed one final objective was to implement by myself an improvement of Rainbow (*Rainbow++*) by including Implicit Quantile Networks (IQN) [9]. The latter algorithm being a major improvement of Distributional DQN [8], it would have replaced it in our *Rainbow++* to improve the performance.

There has unfortunately been an underestimation of the time I would need to complete this project, which may come from a lack in research methods and in knowledge in general given my experience. Hence, the implementation of Rainbow was too long, and I could not finish it : Distributional DQN has been left out for eventual future works, as well as Implicit Quantile Networks.
Future works could also include a more extensive analysis and tuning of the hyperparameters, along with more experiments on various environments to check that this implementation is flexible enough and still valid on longer experiments.

As I started from zero in Reinforcement Learning, I had to follow a *Ramp-up* including video courses, bibliography and codes, which brought me understanding and some skills, that in addition to the project, could be useful for me later. This project was very constructive, because of the knowledge I gained through it, but also because of the methods of work I acquired all along.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *NIPS Deep Learning Workshop*, 2013.

[2] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. G. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," *CoRR*, 2017.

[3] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," *CoRR*, 2015.

[4] Z. Wang, N. de Freitas, and M. Lanctot, "Dueling network architectures for deep reinforcement learning," *CoRR*, 2015.

[5] R. S. Sutton and A. G. Barto, "Reinforcement learning: An introduction," *IEEE Transactions on Neural Networks*, 1998.

[6] M. Fortunato, M. G. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, C. Blundell, and S. Legg, "Noisy networks for exploration," *CoRR*, 2017.

[7] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *CoRR*, 2016.

[8] M. G. Bellemare, W. Dabney, and R. Munos, "A distributional perspective on reinforcement learning," *ICML*, 2017.

[9] W. Dabney, G. Ostrovski, D. Silver, and R. Munos, "Implicit quantile networks for distributional reinforcement learning," *ICML*, 2018.