Progetto Computazionale di Reti Neurali

Emanuele Marconato
Eugenio Mazzone

July 2020

Abstract

In this project we implement the DQN algorithm to explore the efficiency on the gym environment for Atari Arcade videogames. We apply some techniques of *Reinforcement Learning* to evaluate the optimal policy for the Atari *"Space Invaders"* game. The outline of this report is the following:

- **Introduction -** We briefly review the off-policy algorithms in the *Temporal Difference* approximation (TD) and we explain why it's necessary to estimate the Q-value function with a neural network $Q_\theta(a, s)$.
- **Pseudocode -** We explain the structure of the algorithm and the actual difficulties encountered to train the network.
- **Results -** We present the results obtained with the same DQN architecture in the seminal paper.
- **Conclusions -** We present some conclusions (theoretical and practical) for the following case study and stress the importance of hybrid RL algorithms.

1 Introduction

Our purpose has been to develop an algorithm to deal with model-free RL environments, following chapter 5 in [1]. We chose the Atari 2046 games collection to have a pre-processed framework to test the DQN architecture. In our case study every episode is too long to sample a complete trajectory and it turns unecessary, therefore we use the TD technique to train the agent with two consecutive states. Precisely, we update the action-value function with $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$, where $\gamma = 0.99$ is the discount factor and α is the learning rate. This type of update is off-policy, meaning that we use the previous experience to update the action-value function disregarding the current policy.
In these model-free RL algorithms, the environment is not tabular, due to finite memory, indeed it turns necessary to use a function approximation to evaluate the Q-function on all possible states. We use two CNN to learn the optimal Q-function, one online $Q_\theta(s, a)$ that gets updated for every learning step and one offline (target) $\hat{Q}_{\theta'}(s, a)$ that gets updated every T steps. The online network is used to compute the optimal greedy moves of the agent, while the target network has been used to compute $y_i = r_i + \gamma \max_{a'} \hat{Q}_{\theta'}(s_i, a')$, since it does not affected the gradient on θ. We reduce the mse-LOSS given by:

$$L(\theta) = E_{s,a,r,s'}[(y_i - Q_\theta(s_i, a_i))^2]$$

We apply a mini-batch GD, with Adam Optimizer, on the online network to update the network's parameters:

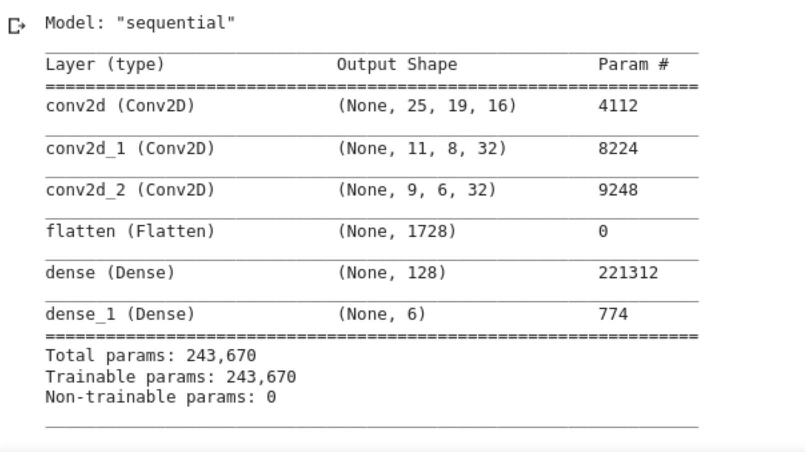
$$\theta \leftarrow \theta - \alpha \nabla_\theta L(\theta)$$

2 Pseudocode

Since the data are strongly related to each other in any episode, we sample a mini-batch from a memory buffer to restore the approximation of IID data. Then every mini-batch sampled is used to train the online network. The memory-buffer is a class containing a storage memory of every agent movement: (s, a, r, s'). The logic of storage is FIFO. We apply ϵ-greedy decisions to make the agent explorative in the first learning steps. The pseudocode of algorithm, applied to *SpaceInvaders-v0* is:

```
for M episodes:
    #initialize the environment
    env.reset()
    while not episode_ends:
        a = eps-greedy(s)
        s',r,finish = env_observation(a)
        #memory storage
        D + (s,a,r,s')
        #update the model
        sample a mini-bath (s,a,r,s') from D
        y_j = r_j    if finish'=True
        y_j = r_j + \gamma max_a' Q_\theta' (s',a')  else
        Perform GD with the mse-LOSS on \theta
        #update the target network
        Every T steps: \theta' = \theta
        s = s'
        if finish: break
        #eps-decay
        eps -= eps_decay_rate   if steps > control
```

For the online and target networks architecture, see figure1.



```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d (Conv2D)              (None, 25, 19, 16)        4112      
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 11, 8, 32)         8224      
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 9, 6, 32)          9248      
_________________________________________________________________
flatten (Flatten)            (None, 1728)              0         
_________________________________________________________________
dense (Dense)                (None, 128)               221312    
_________________________________________________________________
dense_1 (Dense)              (None, 6)                 774       
=================================================================
Total params: 243,670
Trainable params: 243,670
Non-trainable params: 0
_________________________________________________________________
```

Figure 1: The input layer is of size (105,80,4), where every input is a stack of 4 consecutive frames. The network is composed of 3 layers of convolution, respectively with kernerl sizes (8,4,3) and strides (4,2,1), with activation "ReLU", a flatten layer followed by a dense layer of 128 units, with activation "ReLU" and an output dense layer with as many units as the possible actions of the agent, with linear activation.

3 Results

While the code architecture succeed to outperform the task for many Atari games [2], see fig 2 and fig 3, the computational cost is very high and the training process requires at least 5 days for a standard hardware device (using colab with integrated GPU), thus we are opting for cloud services to train the networks. As a qualitative result we refer to the environment *"Breakout-v0"* as the agent successfully moves after few learning steps.We report the result obtained by DeepMind group [2].

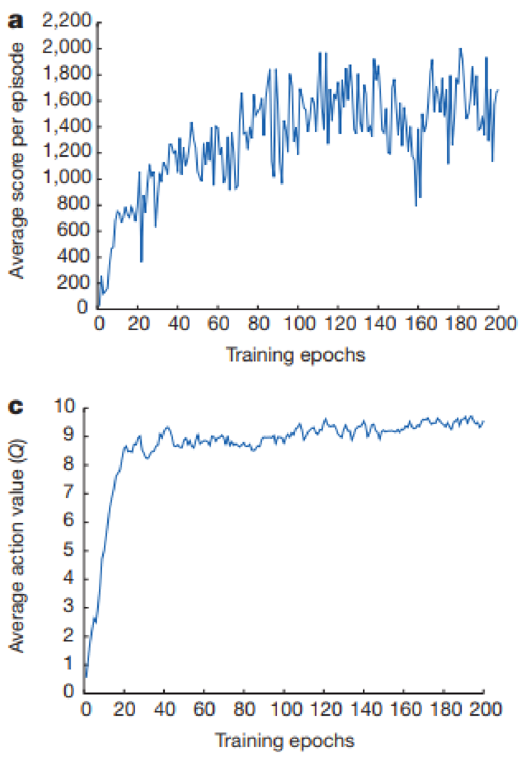


Figure 2: **a.** Each point is the average score per episode after running ϵ-greedy for 520k frames, on Space Invaders. **c.** Each point is the average predicted action-value on a held-out set of states, referred to training epochs above. [2]

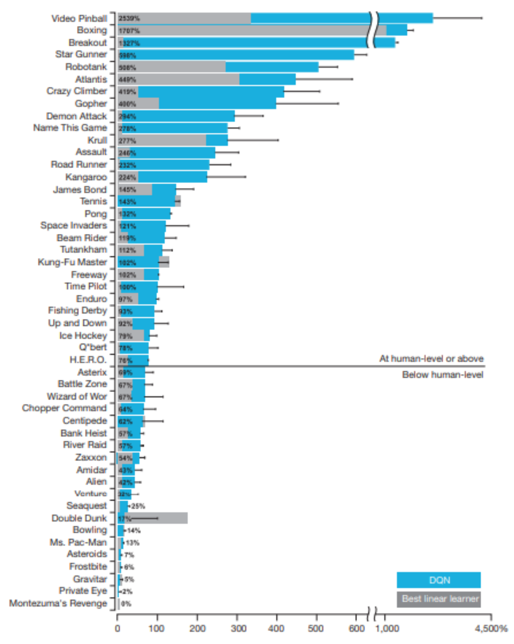


Figure 3: A comparison of DQN performances in different Atari games normalized over the best human professional player's performance: $score = 100 \times (DQNscore - RANDOMscore)/(HUMANscore - RANDOMscore)$.[2]

4 Conclusions and Future Developments

On the theoretical side, while the algorithm performs very well in many Atari environments, it fails in others where a long-short memory is required (e.g. *"Montezuma Revenge"*) or other approaches. The solution to these instances is found in different typologies of RL, where updates on the Q-function and Policy Gradients are combined together to perform better in those environments.
On the practical side, we are implementing an extension to transfer the trained weights of *"Space Invaders "* environment to another one. While the computational cost is high to train just the mother target network, it's not clear if a pre-trained network could perform well in different environments with a fine tuning of its training variables. We leave this question open.
The code can be found at link-to-master-directory. We don't own all the parts of the code, since we took many functions and implementations from Space-Invaders Dueling QN, actually we changed the network architectures and the update of the target network.

References

[1] A. Lonza. *Reinforcement Learning Algorithms with Python: Learn, understand, and develop smart algorithms for addressing AI challenges*. Packt Publishing, 2019.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc Bellemare, Alex Graves, Martin Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–33, 02 2015.