DQN vs REINFORCE for Atari Boxing

Federico Battistella[*]
0000926542
Artifcial Intelligence Master Degree
Unibo
Bologna
federico.battistella@studio.unibo.it

July 7, 2021

Abstract

In this paper we will compare two methods for reinforcement learning applied to the famous Boxing Atari game. We will compare a value approximation method as the Q-Learning, using a deep Q network for training, and a policy gradient method as Reinforce, which will be trained using two architectures, a deep neural network for the state values and another for the policy improvement.

1 Value Approximation methods vs Policy Gradient methods

In deep reinforcement learning we can divide the learning algorithms in two main classes: value approximation methods and policy gradient methods, we can so solve the reinforcement learning problem in two different ways that we are going to briefly describe in the following paragraphs, assuming we are describing the deep learning versions of these methods.

1.1 Value Approximation Methods

Value approximation methods focus on the estimation of the state-value function of a certain environment. The state-value function is learned in two phases: the first phase where we let the agent act in the environment following a stochastic policy that needs to be improved through time, or a heuristic which at the beginning will encourage the agent to explore the action space to get a better knowledge of the values of the non-greedy action and then will progressively change towards the exploitation of the best actions, or just . Then we have

[*]Use footnote for providing further information about author (webpage, alternative address)—*not* for acknowledging funding agencies.

a deep neural network used to predict the state (or state-action) values that will enable us to compute and determine which action will bring us the highest reward in each time step. The algorithm will, in the end, get real values about the environment by self-play and will then learn to predict them, at the end of the training, the deep neural network should be able to find the optimal action in each step, predicting the value for each state.

1.2 Policy Gradient Methods

Policy gradient methods focus on the estimation/improveement of the policy, so the distribution that represents the probability to select each action available in a given state.The aim is learning a parametrised policy that can select actions without consulting a value function. Like for value approximation, we need first to let the agent explore the environment trying different actions to start learning the values of each state, and then exploitation to refine values of the best paths found. Then we can proceed in a similar way as we did before: we feed our deep neural network with the states of the environment and we aim to predict values for each state-action pair. However this time we need to obtain a probability distribution as output of the network not a vector of values. We can switch from numerical preferences over actions to a probability distribution applying a softmax function in the last fully connected layer of our model.

$$\pi(a|s,\theta) = \frac{e^{h}(s|a,\theta)}{\sum_b e^{h}(s|b,\theta)}$$

Where h(s|a,θ) is the vector of numerical preferences over each state-action pair. This way we will obtain a probability distribution with the same number of elements as the number of state-action pairs, and the overall sum of the new values will be 1. Using softmax an action preferences allows to approach a deterministic policy, but it will never be reached, we can then find a stochastic optimal policy.

2 On Policy exploration vs Off Policy exploration

There are two different methods to explore the action space: On Policy and Off Policy. With On Policy exploration we follow the policy we have available, while we can try to improve it at the same time. Off Policy exploration involves the adoption of another method to visit all the possible states of the environment, for example we can use an ϵ-greedy policy, that enable us to choose the actions to take in a stochastic way.

3 Monte Carlo Methods vs Temporal Difference Methods

Monte Carlo and Temporal Difference methods can be used for 3 different problems: - Prediction problem - Policy improvement problem - Control problem

3.1 Monte Carlo Methods

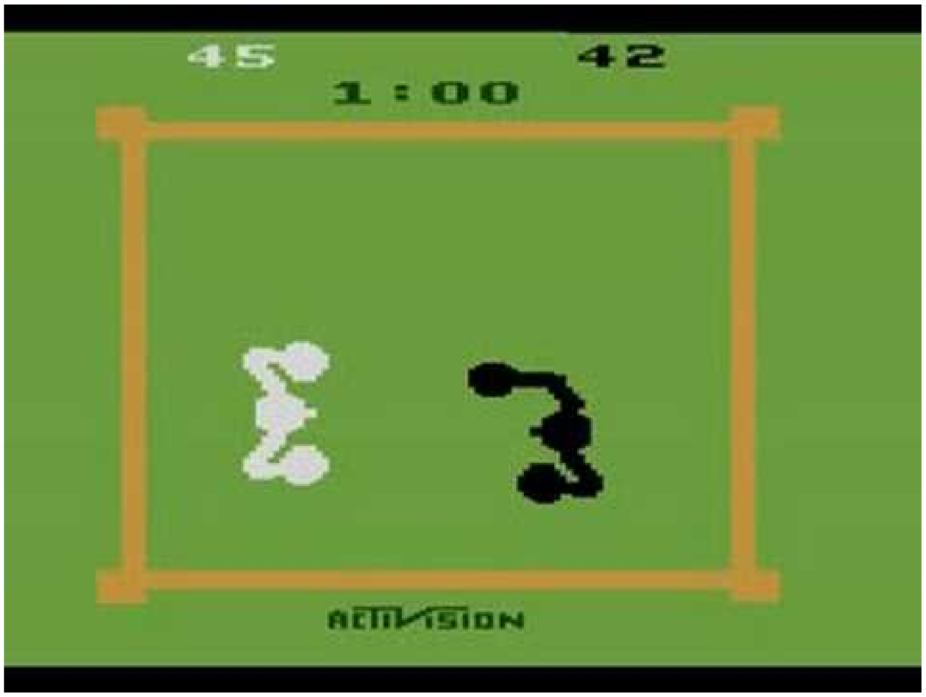
Monte Carlo methods are ways of solving the reinforcement learning problems based on averaging the sample returns of each episode, they learn from experience. The values estimates and the policy change can only be done at the end of each episode, as the reward of each action depends on the following actions taken during the same episode.

3.2 Temporal Difference Methods

Temporal Difference methods, like Monte Carlos, can learn directly from experience. But, on the other hand, Temporal Differences methods update estimates based in part on other learned estimates, without waiting for the final outcome.(Bootstrap)

4 Proposed project

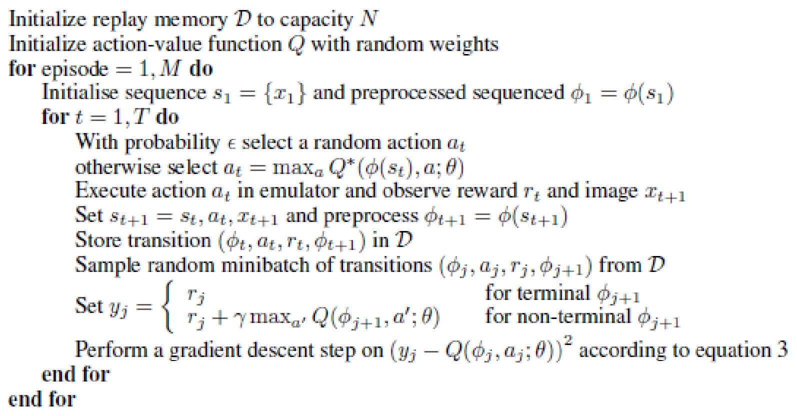
The proposed project focuses on a comparison between two very different reinforcement learning methods: Reinforce and DQN. The two methods will be applied to the gym environment for Atari games simulation, in particular we will consider the game of Boxing analyzing the differences between the two methods.



4.1 DQN Learning

The first method used to let the agent learn how to play Atari Boxing is DQN; it's a value approximation, temporal-difference, off-policy method. In this project the DQN has been implemented in a simple way: The agent starts playing with an epsilon greedy policy, with the epsilon that decreases gradually during training, every step performed by the agent is stored in a replay buffer in form of a SARS tuple, so we are filling the replay buffer step by step adding new tuples. For the training we adopted a convolutional network that during each training step samples SARS tuples from the replay buffer. The model takes in input a batch of sequences of 4 observations which are preprocessed images of the state of the game at a given time step and the previous three steps. Each training step is divided in different points:

- Sample a batch from the replay buffer
- Predict the Q values for the states from the batch
- Compute the maximum target Q values
- Apply MSE between predicted and target maximum Q values
- Adam optimizer apply gradients to the MSE loss function to update the training weights of the model



```
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode $= 1, M$ do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
    end for
end for
```

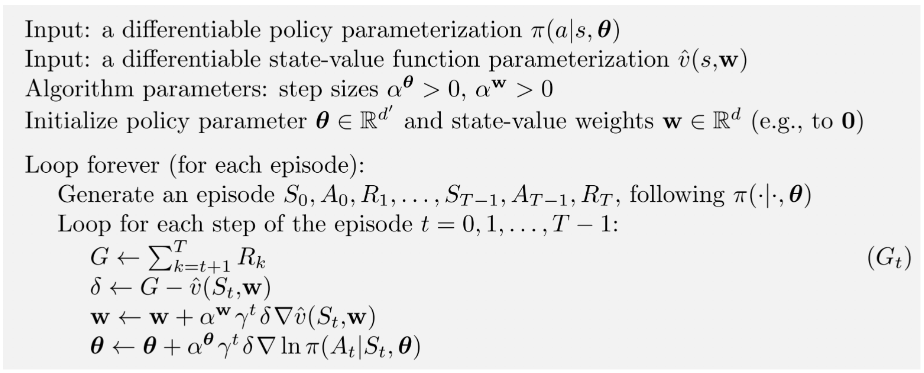
Figure 1: Picture from:`https://www.programmersought.com/article/45434715638/`

The model is then evaluated every 25 episodes, the total average reward over 10 episodes is compared with the best value recorded until that time and if the new value is higher the best average score is updated.

4.2 Reinforce

The second method used to let the agent learn how to play Atari Boxing is Reinforce; it's a policy gradient, Monte Carlo, on-policy method. In this case, applying deep learning to reinforce, we used a variation of the algorithm, we adopted, in fact, the Reinforce with Baseline, which substracts to the episode total reward a value. We adopted convolutional neural networks taking the same input as the DQN process, but instead of using a sequence of 4 frames we just used one, as the input of the network are temporally sequential.In fact, Reinforce train after completing a whole episode, Monte Carlo methods need to complete the full episode to give a reward value to each state. In this case we use two models, the first is necessary to predict the value *(v)* of the given state, which represents the baseline value that need to be substracted to the total episode reward *(G)* during each training step; the other network, instead, try to improve the policy, getting the first network loss value as part of the policy loss function. The training in each episode is developed in this way:

- Let the agent play a full episode to compute the total reward at the end of it
- After completing a full episode we can start the training process, we will first compute the loss function for the value network, then we will use this value to compute the loss function for the policy network.
- Compute the gradients for the to loss functions
- Apply the optimizer (Adam) to update the weights of the two models



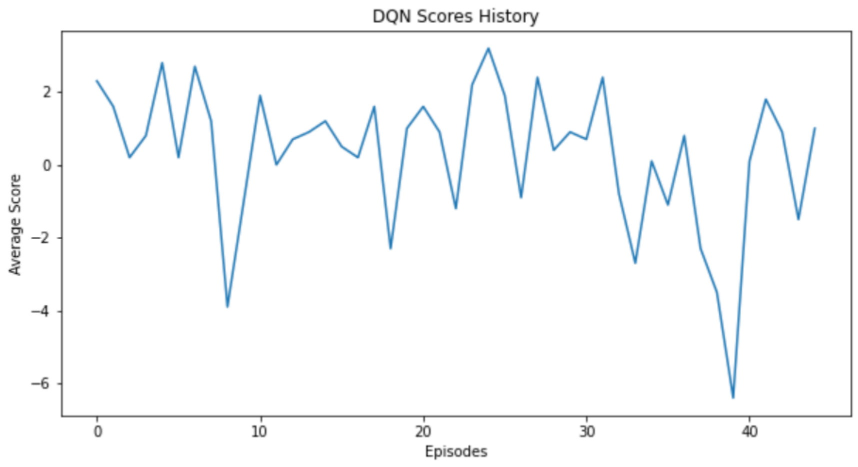
```
Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Algorithm parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
    Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
    Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
        $G \leftarrow \sum_{k=t+1}^{T} R_k$                                                  $(G_t)$
        $\delta \leftarrow G - \hat{v}(S_t,\mathbf{w})$
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \gamma^t \delta \nabla \hat{v}(S_t,\mathbf{w})$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \gamma^t \delta \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$
```

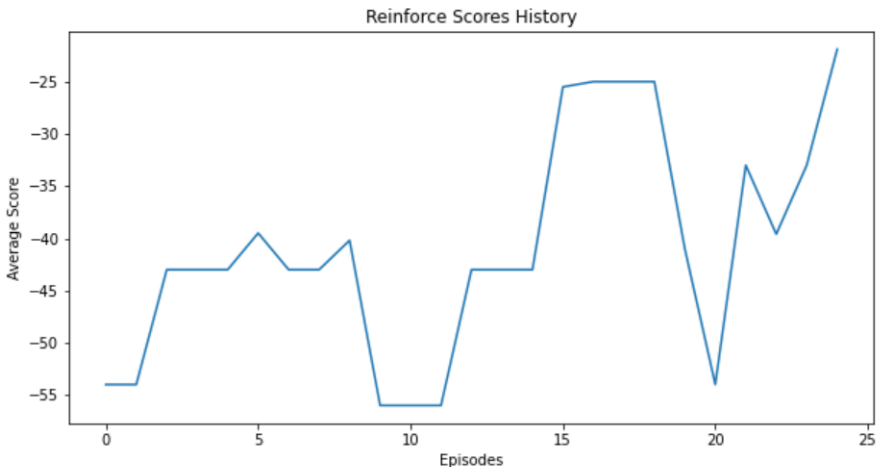
Figure 2: `https://stats.stackexchange.com/questions/340987/how-can-i-understand-reinforce-with-baseline-is-not-a-actor-critic-algorithm`

5 Implementation and results

The project has been developed in Python, using the Gym AI environment to manage the Atari framework, TensorFlow has been used for the deep learning module to train the models. The DQN module has been developed on Google Colab, due to the intensive and harder training, while the Reinforce run in local on Visual Studio, this has been done to allocate more resources to the DQN project and to train the two models in parallel. The DQN requires a much more intensive training, as we used a batch of dimension 32 for each training step and we concatenated 4 frames to get a single input for the neural network, moreover we need to a larger use of memory to store the full replay buffer. On the other hand the Reinforce algorithm has a simpler and faster training because we just have a single frame as input for the ntworks. although the agent playing part is slower beacuse it requires to complete a full episode before strarting the training process for that given episode. The results obtained show, as expected, that the DQN method is able to learn much faster than the Reinforce, it converges quicker because it suffers from high variance and this can slow down the learning process.

We now show the plots of the average reward for episode


DQN Scores History
Average Score
Episodes


Reinforce Scores History
Average Score
Episodes