University of Rome "La Sapienza"
Department of Ingegneria Informatica, Automatica e Gestionale

Reinforcement Learning

Assignment 3

1-step Actor-Critic & CarRacing-v2

Gianmarco Scarano

Matricola:
2047315

December 7, 2023

Contents

Chapter 1

Theory

1.1 1-step Actor-Critic

From theory, one knows that the Actor-Critic algorithm is explained as follows:

Algorithm 1 1-step Actor-Critic

```
1: Initialize $S$ (first state of the episode)
2: $I \leftarrow 1$
3: Loop while S is not terminal (for each time step):
4:     $A \sim \pi(\cdot|S, \theta)$
5:     Take action $A$, observe $S', R$:
6:     $\delta \leftarrow R + \gamma \cdot \widehat{v}(S', w) - \widehat{v}(S, w)$
7:     $w \leftarrow w + \alpha^w \cdot \delta \cdot \nabla_w \widehat{v}(S, w)$
8:     $\theta \leftarrow \theta + \alpha^\theta \cdot I\delta \cdot \nabla_\theta \ln \pi(A|S, \theta)$
9:     $I \leftarrow \gamma \cdot I$
10:    $S \leftarrow S'$
```

Along the parameters given by the Assignment PDF, we have to compute the policy for action 0 and 1.

1. $\pi_0(a = 1|s) = \frac{1}{1+e^{-(\theta^T x(s))}}$
2. $\pi_0(a = 0|s) = 1 - \frac{1}{1+e^{-(\theta^T x(s))}} = \frac{e^{-(\theta^T x(s))}}{1+e^{-(\theta^T x(s))}}$

Let's simply apply this definition on our problem.

- First of all we must compute δ, which is:

$$\begin{aligned}\delta &= 0 + 0.9 \cdot (w_1^T \cdot x(s_1)) - (w_0^T \cdot x(s_0)) \\ &= 0 + 0.9 \cdot 0 - 0.8 = -0.8\end{aligned} \tag{1.1}$$

Computations for w_0:
- Then, we must compute w_0 as follows:

$$\begin{aligned} w_0 &= (0.8, 1)^T + 0.1 \cdot (-0.8) \cdot \nabla_w Q_w(S, a) \\ &= (0.8, 1)^T - 0.08 \cdot (1, 0)^T \\ &= (0.8, 1)^T - (0.08, 0) \\ &= (0.72, 1)^T \end{aligned} \tag{1.2}$$

Computations for w_1:
- Then, we must compute w_1 as follows:

$$\begin{aligned} w_1 &= (0.4, 0)^T + 0.1 \cdot (-0.8) \cdot \nabla_w Q_w(S, a) \\ &= (0.4, 0)^T + -0.08 \cdot (1, 0)^T \\ &= (0.4, 0)^T + (-0.08, 0) \\ &= (0.32, 0)^T \end{aligned} \tag{1.3}$$

Computations for θ:

$$\theta = (1, 0.5)^T + 0.1 \cdot (-0.8) \cdot \nabla_\theta \ln \pi(A|S, \theta) \tag{1.4}$$

We now know the term $\pi(A|S, \theta)$ when $s = x(s_0)$ and $a = a_0$, since we calculated it at the very beginning of this exercise.

$$\begin{aligned} \nabla_\theta \ln \pi_\theta(s, a_0) &= \nabla_\theta \ln\left(\frac{e^{-(\theta^T x(s))}}{1 + e^{-(\theta^T x(s))}}\right) \\ &= \frac{-(x(s) \cdot e^{\theta^T x(s)})}{e^{\theta^T x(s)} + 1} = (-0.7311, 0)^T \end{aligned} \tag{1.5}$$

$$\begin{aligned} \theta &= (1, 0.5)^T + 0.1 \cdot (-0.8) \cdot (-0.7311, 0)^T = \\ &= (1, 0.5)^T - 0.08 \cdot (-0.7311, 0)^T = \\ &= (1, 0.5)^T + (0.0584, 0)^T = \\ &= (1.0585, 0.5)^T \end{aligned} \tag{1.6}$$

Chapter 2

Practice

2.1 CarRacing-v2

For this practical, I decided to implement the PPO algorithm in a continuous environment, starting from the pseudocode that can be found in the paper about A2C:

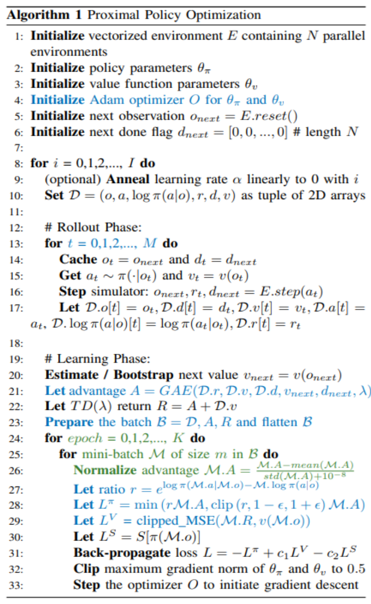
2.1.1 Implementation

With this being said, let's analyze the functions:

- ***Model architecture***: I designed a Proximal Policy Optimization (PPO) model with Convolutional layers for feature extraction. The model included separate branches for the Actor and Critic networks, incorporating a Gaussian distribution for policy sampling, using the SoftPlus activation function. Orthogonal initialization of weights have been added in order to improve robustness to the learning process.

- ***Training***: Adam optimizer was employed for training the PPO agent. The rollout mechanism was implemented to collect experiences, and policy loss, value loss, and entropy loss have been calculated during training and backpropagated through the network. The application of gradient clipping prevented exploding gradients. During training (and testing as well) I adjusted the values of the actions in order to make them more suitable (and stable) for the environment, through a simple clipping mechanism. This helped a lot in preventing the agent from crashing the car or slipping on the road/grass very often.

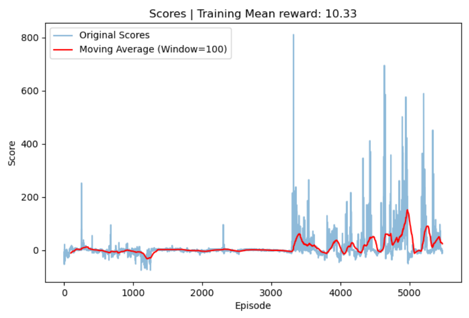
Figure 1: Proximal Policy Optimization pseudo-code.



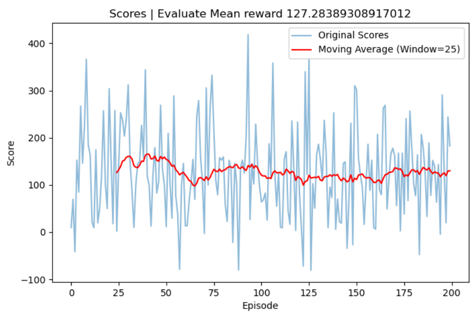
```
Algorithm 1 Proximal Policy Optimization
Initialize vectorized environment $E$ containing $N$ parallel environments
Initialize policy parameters $\theta_\pi$
Initialize value function parameters $\theta_v$
Initialize Adam optimizer $O$ for $\theta_\pi$ and $\theta_v$
Initialize next observation $o_{next} = E.reset()$
Initialize next done flag $d_{next} = [0, 0, ..., 0]$ # length $N$

for $i$ = 0,1,2,..., $I$ do
    (optional) Anneal learning rate $\alpha$ linearly to 0 with $i$
    Set $\mathcal{D} = (o, a, \log \pi(a|o), r, d, v)$ as tuple of 2D arrays

    # Rollout Phase:
    for $t$ = 0,1,2,..., $M$ do
        Cache $o_t = o_{next}$ and $d_t = d_{next}$
        Get $a_t \sim \pi(\cdot|o_t)$ and $v_t = v(o_t)$
        Step simulator: $o_{next}, r_t, d_{next} = E.step(a_t)$
        Let $\mathcal{D}.o[t] = o_t, \mathcal{D}.d[t] = d_t, \mathcal{D}.v[t] = v_t, \mathcal{D}.a[t] = a_t$, $\mathcal{D}.\log \pi(a|o)[t] = \log \pi(a_t|o_t), \mathcal{D}.r[t] = r_t$

    # Learning Phase:
    Estimate / Bootstrap next value $v_{next} = v(o_{next})$
    Let advantage $A = GAE(\mathcal{D}.r, \mathcal{D}.v, \mathcal{D}.d, v_{next}, d_{next}, \lambda)$
    Let $TD(\lambda)$ return $R = A + \mathcal{D}.v$
    Prepare the batch $\mathcal{B} = \mathcal{D}, A, R$ and flatten $\mathcal{B}$
    for $epoch$ = 0,1,2,..., $K$ do
        for mini-batch $\mathcal{M}$ of size $m$ in $\mathcal{B}$ do
            Normalize advantage $\mathcal{M}.A = \frac{\mathcal{M}.A - mean(\mathcal{M}.A)}{std(\mathcal{M}.A) + 10^{-8}}$
            Let ratio $r = e^{\log \pi(\mathcal{M}.a|\mathcal{M}.o) - \mathcal{M}.\log \pi(a|o)}$
            Let $L^\pi = \min(r\mathcal{M}.A, \text{clip}(r, 1-\epsilon, 1+\epsilon)\mathcal{M}.A)$
            Let $L^V$ = clipped_MSE$(\mathcal{M}.R, v(\mathcal{M}.o))$
            Let $L^S = S[\pi(\mathcal{M}.o)]$
            Back-propagate loss $L = -L^\pi + c_1 L^V - c_2 L^S$
            Clip maximum gradient norm of $\theta_\pi$ and $\theta_v$ to 0.5
            Step the optimizer $O$ to initiate gradient descent
```

- ***Results***: The training process lasted several days no-stop. I monitored the mean rewards throughout the whole training phase, also by saving the scores in a JSON file, which were really helpful in providing insights into the agent's performance. While the final results showcased commendable performances, a noteworthy observation was the agent's difficulty in recognizing the correct direction to take. The agent sometimes successfully recovers the car when deviating slightly off-road. However, in some instances, it struggled to regain control, going very sharply to the left. This behavior suggests a potential limitation in the agent's ability to learn an optimal policy for navigating right turns, especially when encountering obstacles like kerbs. Also, as discussed in class, the reward for going off-road is the same as the one when it's completely still. This makes it even harder for the model to know that going off-road is a very negative behaviour.

  Despite these challenges, the overall mean rewards attained by the agent were fine, as we can see in the following plots:



(a) Training - Mean reward: 10.33



(b) Evaluation - Mean reward: 127.28

Figure 2: Rewards for training (left) over 5870 episodes and testing phase (right) for the best model over 200 testing episodes.

Also, a discrete version has been implemented, but it didn't perform as well as the continuous one, so I decided to keep the latter. Parallel environments have been implemented as well, but they didn't improve the performances, so I decided to keep the single environment.

I used the json Python library for saving results and used tqdm v4.66.1 for the progress bar.

Chapter 3

Collaborations

I discussed this assignment with the following students:

- Giancarlo Tedesco