Actor-Critic Algorithm

Compute the transition of w_0, w_1 and θ.

- $\pi_\theta(a=1|s) = \sigma(\theta^T x(s)) = \frac{1}{1+e^{-(\theta^T x(s))}}$
- $Q_w(s, a=0) = w_0^T x(s)$
- $Q_w(s, a=1) = w_1^T x(s)$
- $w_0 = (0.8, 1)^T$
- $w_1 = (0.4, 0)^T$
- $\theta_0 = (1, 0.5)^T$
- $\alpha_w = \alpha_\theta = \alpha = 0.1$
- $\gamma = 0.9$

Algorithm. The actor-critic algorithm consists in:

(1) Given: policy π, parametrization Q_w and learning rates α_w and α_θ. Initialize policy parameters and weights $\theta, \mathbf{w}$

(2) Iterate:
- Initialize the state S
- While S is not a terminal state, at iteration i
  (a) Sample the action from the policy $A \sim \pi(\cdot|S, \theta)$
  (b) Observe reward r and next state S' and sample next action $A' \sim \pi(\cdot|S, \theta)$
  (c) Compute $\delta = r + \gamma Q_w(S', A') - Q_w(S, A)$
  (d) Update $\mathbf{w} = \mathbf{w} + \gamma^{i-1}\alpha_w \delta \ \nabla_w Q_w$
  (e) Update $\theta = \theta + \gamma^{i-1}\alpha_\theta \delta \nabla_\theta \log \pi(A|S, \theta)$

In this specific case we have, other than the variables defined at the beginning, $x(s_0) = (1, 0)^T$, $a_0 = 0$, $r_1 = 0$, $x(s_1) = (0, 1)^T$, $a_1 = 1$.
Before updating the parameters we need to compute the gradients:

- $\nabla_w Q_w$: Let's say that we have either action i or j, with $i, j \ in 0, 1 \ i \neq j$ then:

$$\nabla_w Q(s, a=i) = \begin{cases} \nabla_{w_i} w_i^T x(s) \\ \nabla_{w_j} w_i^T x(s) \end{cases} = \begin{cases} x(s) & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$

  In this example we have $a_0 = 0$ so this means that

$$\nabla_w Q_w = \begin{cases} x(s) & \text{for } w_0 \\ 0 & \text{for } w_1 \end{cases} = \begin{cases} (1, 0)^T & \text{for } w_0 \\ (0, 0)^T & \text{for } w_1 \end{cases}$$

- $\nabla_\theta \log \pi(A|S,\theta)$: We know that $\pi(a=1|s)$ is the sigmoid and that $a_0 = 0$, this means that $\pi(a=0|s) = 1 - \sigma(\theta^T x(s))$. Now let's compute the gradient:

$$\begin{aligned}\nabla_\theta \log \pi(a=0|S,\theta) &= \nabla_\theta \log(1-\sigma(\theta^T x(s))) = \\ &= \frac{\nabla_\theta(1-\sigma(\theta^T x(s)))}{1-\sigma(\theta^T x(s))} = \frac{-\nabla_\theta(\sigma(\theta^T x(s)))}{1-\sigma(\theta^T x(s))} = \\ &[\textit{Given that } \sigma' = \sigma(1-\sigma)] \\ &= \frac{-\sigma(\theta^T x(s))\cancel{(1-\sigma(\theta^T x(s)))}\nabla_\theta \theta^T x(s)}{\cancel{(1-\sigma(\theta^T x(s)))}} = \\ &= -\sigma(\theta^T x(s)) x(s)\end{aligned}$$

In this example we have $\theta_0 = (1, 0.5)^T, x(s_0) = (1,0)^T$, then $\theta_0^T \sigma(x(s_0)) = \sigma(1) = \frac{1}{1+e^{-1}} \approx 0.73$. So $\nabla_\theta \log \pi(a=0|S,\theta) = -0.73 * (1,0)^T$

Together with the gradients, also δ has to be computed in order to update the parameters, and:

$\delta = r_1 + \gamma Q_w(s_1,a_1) - Q_w(s_0,a_0) = 0 + 0.9 * (0.4,0)(0,1)^T - (0.8,1)(1,0)^T = 0.9 * 0 - 0.8 = -0.8$

So now for the update itself:

- $\theta_1 = \theta_0 + \gamma^0 \alpha \delta \nabla_\theta \log \pi(a=0|S,\theta_0) = (1,0.5)^T + -0.8 * 0.1 * (-0.73,0)^T = (1.0584,0.5)^T$
- $w_0 = w_0 + \gamma^0 \alpha \delta \nabla_{w_0} Q(s_0, a=0) = (0.8,1)^T + -0.8 * 0.1 * (1,0)^T = (0.72,1)^T$
- $w_1 = w_1 + \gamma^0 \alpha \delta \nabla_{w_1} Q(s_0, a=0) = (0.4,0)^T + -0.8 * 0.1 * (0,0)^T = w_1$

Code Explanation

A2C. For the pratical part of the homework I decided to implement the **A2C** *Advantage Actor Critic* algorithm. My implementation of the algorithm is contained inside the given **Policy** class.
The class is composed of 5 functions (other then the 3 given functions **load, save** and **to**):

- **Initialization function**: That initilizes the class. Inside this function I defined all the class variables and most important I defined **model** and **optimizers**.

- **Forward**: That takes as input x (which could be either a batch of states or one single one) and return 3 objects, the predicted value computed using the critic network, the probability of each of the actions given by the actor network and an action sampled from the probability.

- **Act**: That given a single state returns, after obtaining the probability of the actions via the actor network, the action to be performed.

- **Train**: The main function of the class, that is used to train the network. To implement this function I followed the given paper, so divided the training in 2 phases and iterated over them:

  (1) **Rollout phase** in which I just make decisions using the forward function and store in some arrayes various informations, such as: predicted values, taken actions (and logarithm of the probability of those actions), rewards, states (and whether or not those where terminal states) and the entropy.
  (2) **Learning phase** in which I compute the loss using the stored informations and optimize the parameters of the model using the **RMSprop**. The loss to use for this method is $loss = -loss_{actor} + c_1 \cdot loss_{critic} - c_2 \cdot entropy$, where

$$\begin{aligned} loss_{actor} &= -\log \pi(a|s) \cdot advantages \\ loss_{critic} &= (returns - predicted\_values)^2 = advantages^2 \\ advantages &= returns - predicted\_values \end{aligned}$$

- **Compute Advantages** function use to compute the advantages after the Rollout phase. Given as input the rewards, values, dones (to check the terminal states) and the next predicted value, in this fuction I compute the advantages in the following way:
  (1) Define the array $returns = [0, \dots, 0, next\_value]$ of length equal to the number of steps in the rollout phase (k) plus 1.
  (2) Iterate for i that decreases form k to 1 and compute the return at time i: $returns_i = reward_i + \gamma returns_{i+1}(1 - dones_i)$
  (3) At this point it's easy to compute the advantages as $advantages_i = returns_i - values_i$ for $i \in 0, \dots, k$.

- **Preprocess states** that given a state, it transform it in the form I want for the network. See the first of the notes below for specifics.

Notes

(1) **States**. For my implementation of the algorithm I did some changes to the states. Originally the states where some RGB frames –so shape (3, 96, 96)– to try and make the learning process simpler I decided to make 3 changes.
First I used the *torchvision* library to transform the frames form RGB to the gray scale.
Second I wanted to give the model some termporal informations, so my input to the model was not just the current frame but the latest 4 stacked together.
Third I cropped the image in a way that the black line at the bottom is not included.
So at the end my states were (4, 84, 84) tensors.

(2) **Model structure**. Since I'm dealing with frames and images, for the models I chose the following structure:
- 3x(Conv2d + ReLU + MaxPool)
- Flatten
- 1x(Linear+ReLU)
- The last layer is different for the actor and critic network. For the actor: (Linear$(256, 5 = n_actions)$ + Softmax), while the critic is just a (Linear$(256, 1)$).

Double DQN. I also implemented the Double DQN with proportional prioritization. I divided the code in this algorithm in two separate files: *student.py* and *auxiliary.py*.

auxiliary.py, I used this file to define the classes for both the **Replay Buffer** and the **Network Architecture**.

- *Replay Buffer class* composed of 3 main functions:
  - **push**: That takes as input the experience observed during the evironment step, so *state, action, done, reward* and *next state*, and assign it a priority that will be:
    $$p_t = \begin{cases} 1 & \text{if the buffer is empty} \\ max_{i<t}(p_i) & \text{otherwise} \end{cases}$$
    In this way I'll give a sample just experienced the maximum priority, that will later be modified. Note that if the buffer is full the new sample will take the place of the one with the least priority.
  - **get_samples**: That, given the preset batch size, will select a batch of sample from the one stored in the buffer. The selection of the elements will be done by assigning a probability to each of the samples –$\mathbb{P}(sample_i) = \frac{p_i^\alpha}{\sum_j p_j^\alpha}$, where the p_i's are the priorities of the samples– and drawing the wanted amount of samples according to the probabilities.
    Together with the samples, this function return the *importance sampling weights* that are used to try and balance out the fact that the $\mathbb{P}(sample_i)$ could be highly inbalanced, the weights are computed as follows:
    $$w_i = \Big(\frac{1}{size\_buffer} \cdot \frac{1}{\mathbb{P}(sample_i)}\Big)^\beta$$
    And then the weights are all normalized between $[0, 1]$ by dividing by $max_j w_j$. Note that I will let the parameter β gradually go from the initial value of 0.5 to 1.
  - **update_priorities**: That take as input the index of the sampled elements and a list of the new priorities. The old priorities in fact are replaced by the TD-errors: $\delta_t = R_t + (1 - done) \cdot Q\_target(state_{t+1}, \text{argmax}_a\, Q(state_{t+1}, a)) - Q(state_t, a_t)$

- *Network class* that is mainly used to define the sturcture of the network and the **Adam** optimizer. The **initialization function** of the class take as input *output_size, input_channels* and *learning rate* and define a nework with:
  - 3x(Conv2d + ReLU + Batch normalization + MaxPool)
  - Flatten
  - 1x(Linear+ReLU)
  - 1x(Linear)

  I used this class to define both the target and the local networks.

student.py the given file, in which is just present the *Policy class*. In the class there are various functions:

- **Initialization**: In which everything is defined and initialized.

- **Forward**: That given the input x gives back the output of the local network.

- **Act**: That uses an ϵ greedy policy, if a random element is under epsilon then i return the action that maximizes $Q(state, a)$, otherwise just a random action. Note that during the evaluation phase this function will just return the action that maximize the output of the network.

- **Preprocess States**: Function that implement what I already did for the **A2C** algorithm: *Gray scale, stack frames* and *crop the image*.

- **Learn**: The function that compute the loss and perform the optimizer step. Takes the samples for the buffer and use them to compute the TD-errors: $\delta_t = R_t + (1 - done) \cdot Q\_target(state_{t+1}, \text{argmax}_a\, Q(state_{t+1}, a)) - Q(state_t, a_t)$. Once the δ_t's are compute the loss is defined as $\sum_{sample_t \in samples} (\delta_t w_t)^2$, where w_t are the importance sampling weights computed in the buffer, and once the backprobagation is completed the new model parameters of the local network are copied to the target one. Note that once the δ_t's are computed they are given to the **update_priorities** function of the buffer to be used as new priorities for the samples.

- **Train**: In this function I iteratively select an action –with the **act** function– given the state and store the observation in the buffer. The rest of the training loop depends on which phase I'm in:
  (1) *Exploration phase*, that currespond to the first 25k iteration in which $\epsilon = 1$ fixes. During this phase ϵ remain the same and every 128 iteration I call the **Learn** function to use the random experience to train the network.
  (2) ϵ *decay*, all the iterations in which $\epsilon \neq 1, 0.1$. Every 128 iteration I, other then updating the parameter using **Learn**, update $epsilon = epsilon * epsilon\_decay$, where $epsilon\_decay = 0.995$.
  (3) *Exploitation phase*, $\epsilon = 0.1$. Just update the model weights every 128 iteration using the experience taken by applying the network.

- **Reset Env**: Once an episode is completed I have to reset the environment and I use this function to do so. I implemented a function for this step because in this way I can, other than reset the env, skip the first 60 frame, that are just a zoom in of the track, and store the last 4 to create my custom state just as explained before.