Theory It's asked to expand the updates of the monte-carlo reinforce algorithm, which is a policy gradient based algorithm, which takes as target value the expected return of an episode. Since in this case T = 2, we'll have to "analyze" two iterations, which happen at t=0 and t=1. In general, as stated in Sutton Barto, the G per timestep can be written as

$$G = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$$

After that we need to upgrade the weights according to gradient of the logarithm of the probabilities (given by the parametrized policy) associated to each action. The update is the following

$$w = w + \alpha \gamma^t \frac{\nabla \pi(A_t|S_t, w_t)}{\pi(A_t|S_t, w_t)}$$

The calculations were made in Matlab and are reported afterwards.

Matlab Code

```
w = sym("w",[2,1]);
x = sym("x",[2,1]);

w0 = [0.8;1]; x0 = [1;0]; x1 = [1;0]; x2 = [0;1];
alpha = 0.1; gamma = 0.9;

%probability of executing action 1 in state s
sigma = @(w,x)(1/(1+exp(-dot(w,x))));
sigma(w0,x0) %action 1 in state 1
```

```
ans = 0.6900
```

So, we see that the first action is more likely to be selected by the agent.

```
policy(2) = sigma(w,x);     %probability of action 1
policy(1) = 1 - policy(2);  %probability of action 0
g1 = gradient(policy(2),w)/policy(2); %gradient of logprob of action 1
g0 = gradient(policy(1),w)/policy(1); %gradient of logprob of action 0
```

Action selected at first step is 0 with reward 0 (agent goes in exploration)

```
t = 0;
G = 0*gamma^0 + 1*gamma^1;
gradient = eval(subs(g0,[w,x],[w0,x0]));
w1 = w0 + alpha*(gamma^t)*G*gradient
```

```
w1 = 2x1
    0.7379
    1.0000
```

```
sigma(w1,x1)
```

```
ans = 0.6765
```

Action selected at second step is 1 with reward 1 (agent selects the most probable action)

```
t = 1;
G = 1*gamma^0;
gradient = eval(subs(g1,[w,x],[w1,x1]));
w2 = w1 + alpha*(gamma^t)*G*gradient
```

```
w2 = 2x1
    0.7670
    1.0000
```

```
sigma(w2,x1)
```

```
ans = 0.6829
```

Practice The practical exercise asked to implement an algorithm for the RacingCar environment. I tried to implement Double DQN with proportional prioritization. The structure of the network is that of "Human-level control through deep reinforcement learning", so with 3 convolutional layers and some linear layers with ReLU activation functions. The input dimension of the neural network is (4,84,84), i.e. four frames stacked upon each other after being preprocessed (greyscaled and resized), while the output dimension consists of the dimension of the action space, which in the case of racing car is 5. Double DQN proceeds in this way: it first selects an action (with epsilon greedy strategy) and then stores the observed values in the buffer. After that, obviously the crucial part is how the weights of the neural network are updated according to the temporal difference error. At every updating step, the algorithm samples a bunch of experiences to whose altogether to update the weights. So, not only the last observed state is used, but also past experiences.

Proportional Prioritization This process has the goal to output a batch of transitions from the buffer in which many past experiences are stored. This is achieved with a particular data structure called sum tree, in which very node is the sum of its two children, the root contains the sum of all the leaves and the priorities are stored in the leaves. I implemented a slightly different version than the ones found online, I tested it and it works. The sampling happens simply by selecting k (the dimension of the batch) values between 0 and the value of the root, and by that we go down the tree until we find a matching priority. I used proportional prioritization. Higher priorities mean higher probability to be matched by the sample, which means higher probability to be reconsidered in future samplings, in order to influence the update of the nn weights. After having sampled some transitions, which means adjusting the loss function to lower the bias. This consists in adjusting the weights according to the probabilities (the normalized priorities). The weights gave me problems because they always became nan for a reason i didn't manage to understand.
It was a bit challenging, and in fact I didn't manage (for time issues) to implement a fully working solution. THe versione without importance sampling worked, also on cartpole. Nonetheless, I tried to implement the algorithm more or less on my own.

Resources

- Sutton Barto
- Foundations of Deep Reinforcement Learning
- https://github.com/jcborges/dqn-per