Alexandre Allani, CID : 1797836

November 21, 2019

Reinforcement Learning Part 2: Coursework Part 2
The agent that I have developed, reaches the goal during the training, however, it gets stuck while testing with the greedy policy. I tried to save the state of the network when it is supposed to work greedily. However, it works from time to time. I'm explaining in this document what I tried to do.

What is in *agent.py* file

The *agent.py* file contains 4 different classes:

- Agent: Contains everything about the agent. From how it is rewarded to how it trains the deepQ network behind it
- Network, DQN & ReplayBuffer: Contains specificities of a deepQ network

Specificities of my implementation from the lectures

Line : 478 From the advanced technique that we saw in class, I tried to implement two of them. The first one is doubleQ learning. After implementing this, I saw indeed some small improvement in the evaluation of the Qvalues. However, I can't say if this is a plus for the problem that was given to us. It only seems to work better, but I don't have any proof of it.
I tried also to implement a weighted ReplayBuffer. However, with my computer, without the weighted ReplayBuffer, I could train for around 100K steps. With the replay buffer, I was able to train only for around 30K steps. I think my implementation was too "unoptimized", resulting in that loss. I was using a deque object and a numpy array to store weights and transition. There must be a better way to do it, however, I did not find it, so I did not let the weighted buffer in my final agent.

Specificities of my implementation

Epsilon Greedy and exploration : The function in the code is *_update_epsilon*() **(Line : 88)**. I chose to decrease the value of ϵ with a negative exponential function. The e_decay variable measures how much steps my agent should take before having an ϵ that allows more exploration. I choose to do that so that my agent has enough time to explore most of the stage while having a greedy component in its policy. I also choose to let the agent explore for 5000 steps at the beginning**(Line : 56)**. Again, this was done to find a good balance between exploration and intensification. By exploring too much, my agent was not able to reach the goal with a full greedy policy. And with too much intensification, my agent did not explore enough. So it was not reaching the goal. In the end, values chosen for the ϵ-greedy policy were the best trade-off between intensification and exploration that I could make. As you can see in the $get_next_action()$**(Line : 167)** function, if the agent got stuck hitting the wall on with a "Right" action, it should go up and down until he gets unblocked. Although this was done so that the agent would not get stuck, it is not very useful because the agent rarely gets blocked multiple with a "Right" action. Usually, it goes left. Finally, after a certain amount of steps, I consider that the network has explored enough the starting area, so I'm applying a full greedy policy for the first 20 steps at each run.

Reward Function: The function in the code is $_reward()$ **(Line : 96)**. This function is in the end quite basic. It is $(1-d)**2$. As expressed in part 1, the difference of reward should be non-linear so that the agent gets more reward for going toward the goal. I added also a greater reward when going to the final state. Therefore, when the networks arrive at the goal state, Qvalues are updated toward the goal state. Moreover, we know that the goal state is always left, so the agent gets a bit more reward when going to the right, and a bit less reward when going to the left or hitting the wall. At first, I tried with a heavy negative reward for going to the left, but my Qvalues were not converging. I guess gradients were exploding, so I kept small values. The more complicated the reward function I made, the less it was working, so I kept it simple for the final run.

Saving the best network: The first if statement in $set_next_state_and_distance()$ is here to save the best network that reaches the goal. To do so, after a certain amount of time, I test a full greedy policy with the deepQ network (without training it). If it reaches the goal within a number of steps that were never achieved before, the state of the network is saved.

Network architecture: I added a layer to see if there is any improvement. I changed the activation function to leaky relu, because we don't want to have dead neurons in the neural network.

Other hyper-parameters **Gamma** was a hard parameter to work around. With a very high gamma (0.99), my agent started looping around the same states, because it was indeed giving better result than just going to the state within 200 steps (length of my episode). With a low gamma, my agent preferred to follow the best instant reward and was getting easily stuck in a local maximum. Finally, I chose an episode of length 200, so that the agent could explore more the "maze" during one episode.