Navigation with DeepRL in a Unity environment

I/ Implementation

At first, I took the ***model.py*** and ***agent.py*** files from the lesson on the Lunar Lander in Udacity Nano-degree called Deep Reinforcement Learning.

Then I Modified the DeepQNetwork, from ***model.py***. It has now two fully connected layers, each containing 18 nodes. Those layers are using the ReLU activation function, but I guess that another would have made the job as fine. I choose to have a quit little network in order to make my calculation easier (my internet connection is bugging and most of the simulations that I made on Udacity workspace where interrupt and so unusable).

In the ***agent class***, I tried to change some variable values, but due to the time it takes to correctly test the benefits of one change, I decided to keep the one already in place which seemed functional.

Finally, on the ***navigation_2 file***, I implemented the DQN function as we already made in previous lessons.

II/ How does that algorithm works?

To begin, we create an agent with a Deep Q-Network. This DQN can be any shape and size wanted (here, I use two hidden layers). Then we'll train this agent with experience tuples which is a set of state, action, reward, next State and done. (done let us know if the experience is final in the episode).

The class Agent contains two Q-Networks. One is the local, and the other is the target network. The local network is updated on every step once there are enough samples in the replay buffer to make a batch. The target network is updated every UPDATE_EVERY step so that it is slightly closer to the local version by the following equation:

*target_parameters = TAU * local_parameters + (1 - TAU) * target_parameters*

Then the Q-value of the next state is calculated based on the target network parameters and its chosen action.

Finally, thanks to the ReplayBuffer class, our agent can learn every UPDATE_EVERY which mean it takes a batch of experience randomly selected experiences (with "simple" DQN at least) and update its weights upon them.

During all the simulations, I have been using the following ***hyperparameters***:

```
BUFFER_SIZE = int(1e5)   # replay buffer size
BATCH_SIZE = 64          # minibatch size
GAMMA = 0.99             # discount factor
TAU = 1e-3               # for soft update of target parameters
LR = 5e-4                # learning rate
UPDATE_EVERY = 4         # how often to update the network for Fixed Q-targets
```

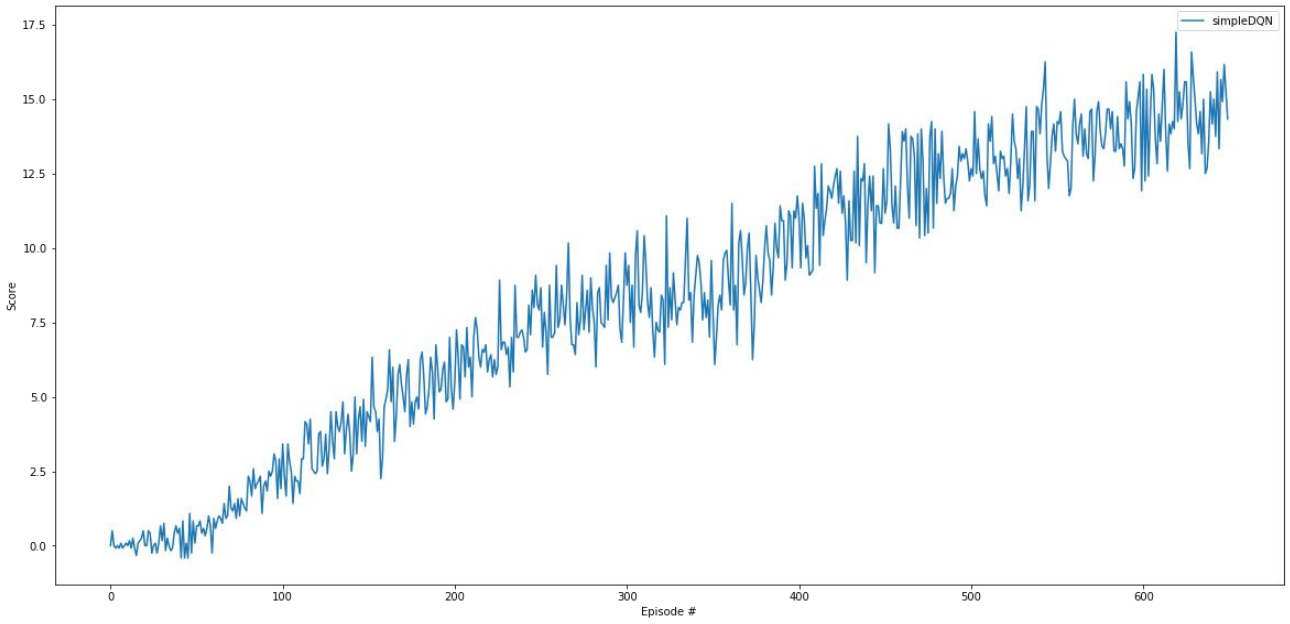
III/ Training and testing

When I was sure my program would work, I trained an agent for 2000 episodes in order to see how it would improve with time. This is the one saved on ***MyModel.pth***. I saw that after a few time (about 1000 episodes) the agent would not get better continually and seemed to be kind of unstable in its learning instead. However, it still allowed me to have a well-trained agent for later tests.

Then I decided to see how well the agent was improving until it gets to a score of about 13 in the last 100 episodes. It was near 600 to 700 episodes so I planned to train a dozen of agent for 650 episodes to have an average of their results along time.

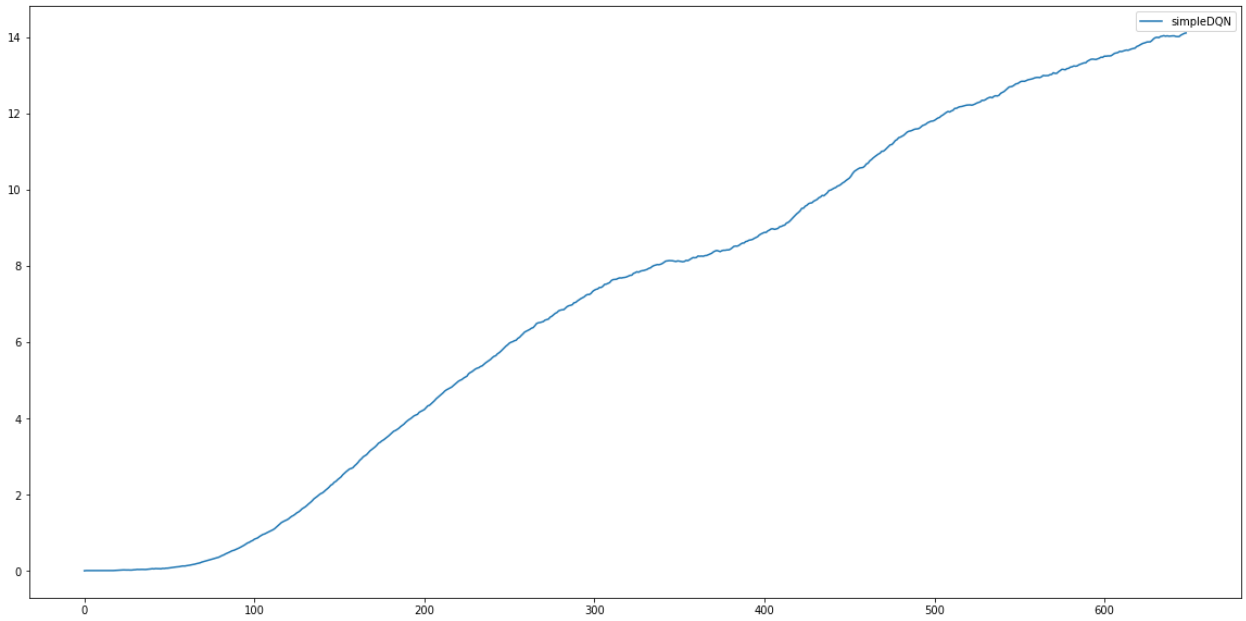
This are the rough results:

Fig I: Testing of 12 Agents for 650 episodes each (rough curve)



Then in order to have a readable graphic, I have made an average of the 100 values before each episode. For exemple, for the 120th episode, we will average episode 20 to 120.

Fig II: Testing of 12 Agents for 650 episodes each (smoothed curve)



This curve is like a Low pass and allows us to only see the slowest guiding improvement. The 99 first value are falsified due to the average but the overall shows us what we can expect if we launch a simulation of an agent for less than 650 episodes. The results are quit satisfying except for the part around 350 episodes that seems to be flattened, maybe du to the low number of tested agents.

IV/ What to do next

It would be interesting to implement double DQN, dueling DQN and prioritized replay to see whether it has major influence on the Agent or not. Maybe should we launch a big enough simulation to see how each variable influence the learning time. Testing changes in the deep Network should be very productive too. We could try other activation function, other kinds of layers or even a different number of layer and nodes.

Finally, I tried the double DQN that was proposed by *Sachinruk* on GitHub (https://github.com/sachinruk/dqn_bananas) and found **not much difference** with "simple" DQN.

I made an experiment on 12 different agents for both algorithms, at first, for 300 episodes, then for 650.

Here the results are smoothed with an average of the 100 preceding values for each episode. It acts like a Lowpass Filter and make it easier to see the results. Though, it falsifies the 99 firsts values.

Fig III: Testing of 12 Agents from "simple" and double DQN for 300 episodes each

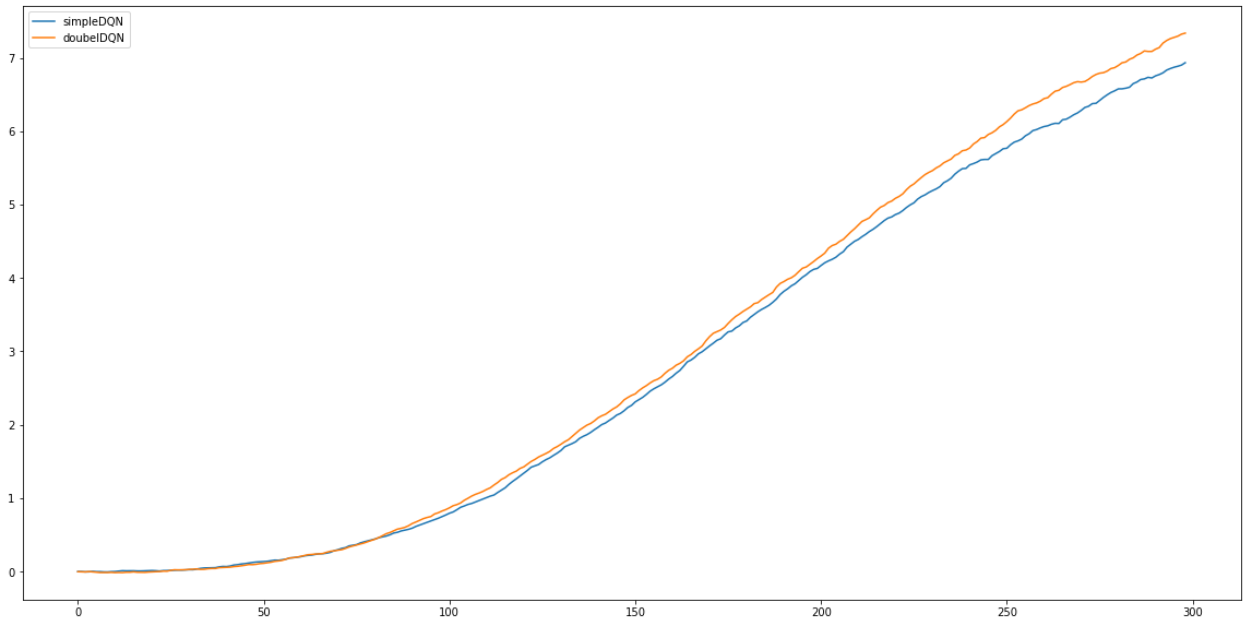
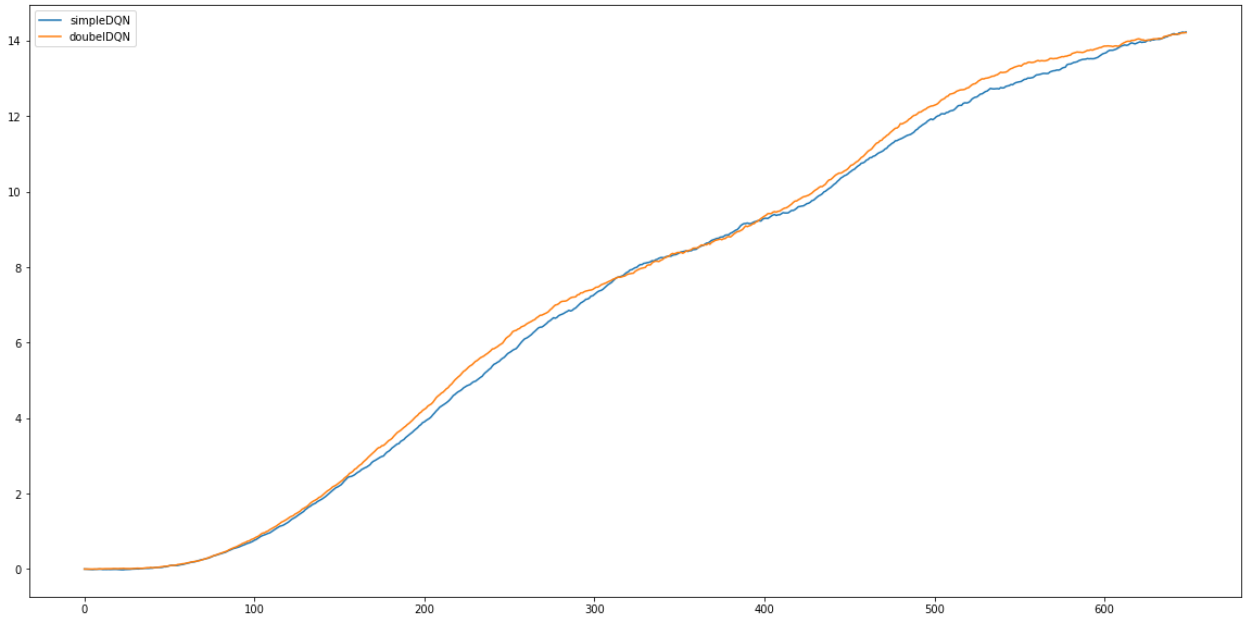


Fig IV: Testing of 12 Agents from "simple" and double DQN for 650 episodes each



It's interesting to see that here too, the curve is flattening around episode 300 to 400. It would be interesting to test changes in the variables and see what is responsible for that.