**Deep RL project report**

**Plan**

Description of our environment

Learning algorithms

Implementation

Detailed result

1. ***Description of our environment***

**MountainCar v0**: A car is on a one-dimensional track, positioned between two "mountains". The goal is to drive up the mountain on the right; however, the car's engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum

**Observation:**

Num Observation Min Max

0 position -1.2 0.6

1 velocity -0.07 0.07

**Actions:**

Type: Discrete (3)

Num Action

0 push left

1 no push

2 push right

**Reward:**

-1 for each time step, until the goal position of 0.5 is reached. As with MountainCarContinuous v0, there is no penalty for climbing the left hill, which upon reached acts as a wall.

**Starting state:**

Random position from -0.6 to -0.4 with no velocity.

**Episode Termination:**

The episode ends when you reach 0.5 position, or if 200 iterations are reached.

1. **Learning algorithms**

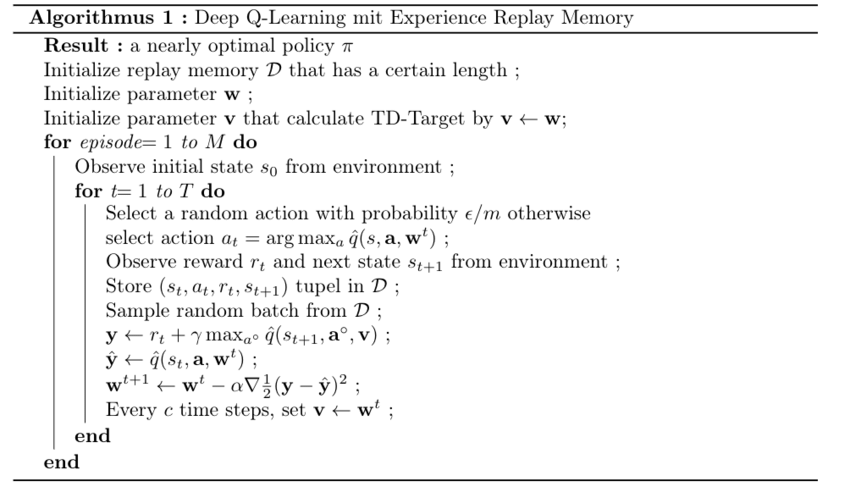
* DQN

We’ll be using experience replay memory for training our DQN. It stores the transitions that the agent observes, allowing us to reuse this data later. By sampling from it randomly, the transitions that build up a batch are decorrelated. It has been shown that this greatly stabilizes and improves the DQN training procedure.

For this, we’re going to need two classes:

Transition: a named tuple representing a single transition in our environment. It essentially maps (state, action) pairs to their (next\_state, reward) result, with the state being the screen difference image as described later on.

ReplayMemory: a cyclic buffer of bounded size that holds the transitions observed recently.

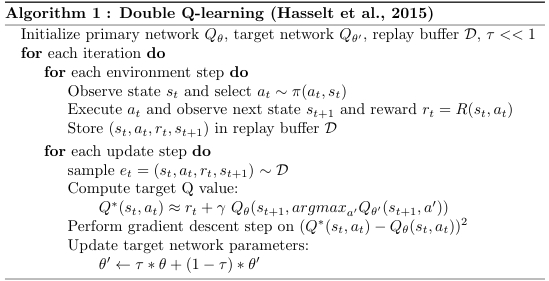


* Double DQN

Taking the maximum overestimated values as such is implicitly taking the estimate of the maximum value. This systematic overestimation introduces a maximization bias in learning. And since Q-learning involves bootstrapping learning estimates from estimates such overestimation can be problematic.

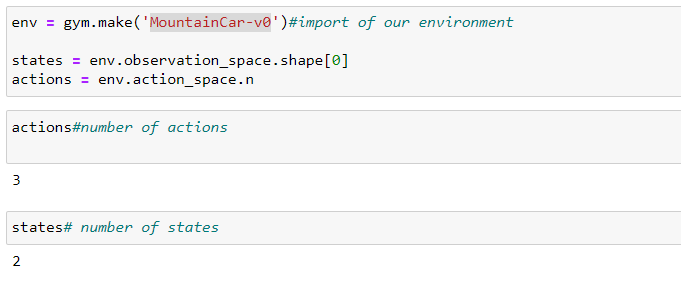
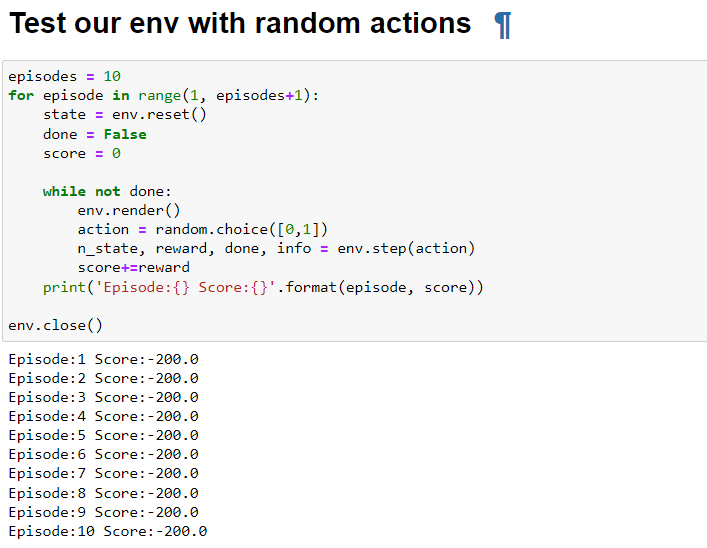
The solution involves using two separate Q-value estimators, each of which is used to update the other. Using these independent estimators, we can unbiased Q-value estimates of the

actions selected using the opposite estimator. We can thus avoid maximization bias by disentangling our updates from biased estimates.

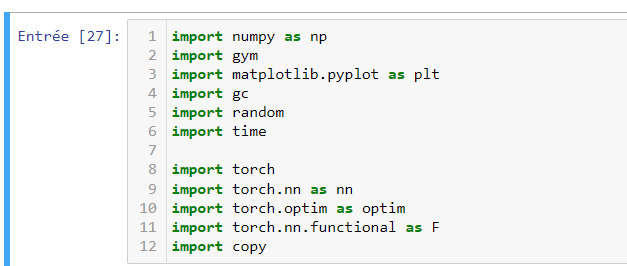


1. **Implementation**

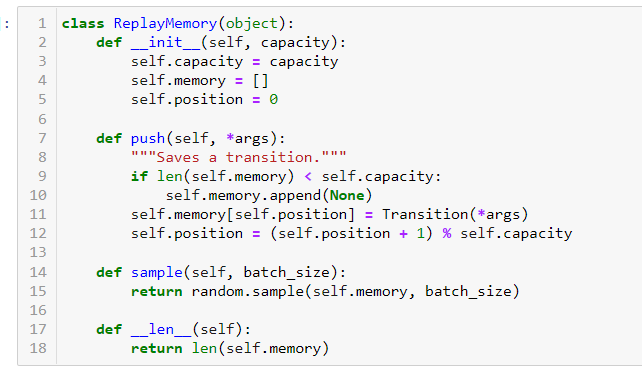
We imported our environment MountainCar-v0 using gym library, and as we see below we have 3 actions and 2 states

After, we tried to visualize how the environment evolve using random action, just to have a little perspective about it.

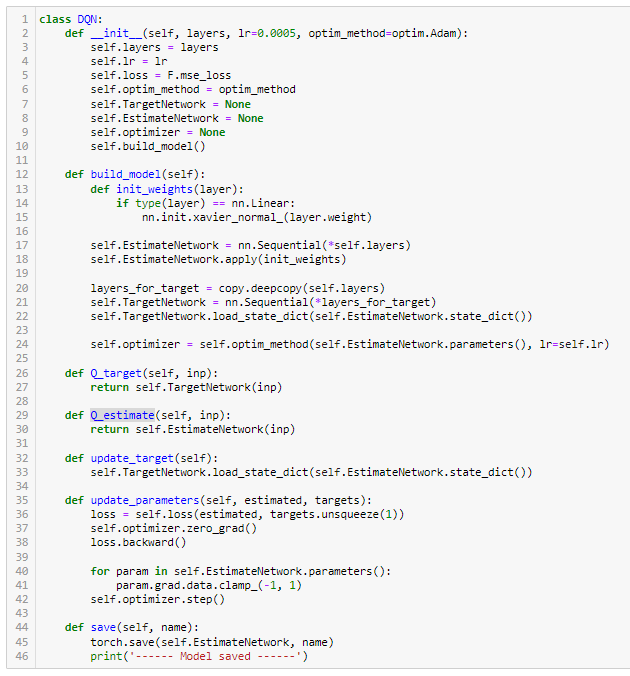
* **DQN**

Import of libraries

Here we created a replaymemmory class so we can save the previous transitions. The attribute memory will contain the transitions taking into account the maximum capacity. Sample will return us a random sample from the memory.

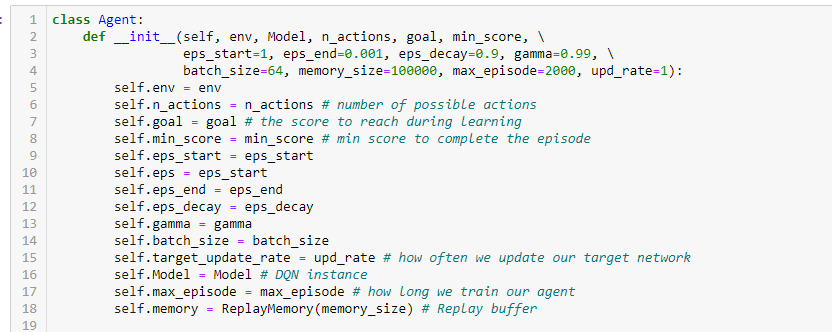


Now we are creating DQN class which will help us create the 2 networks, estimated network and target network. Update\_parameter will be used later to update the loss so we can do the training of the model.

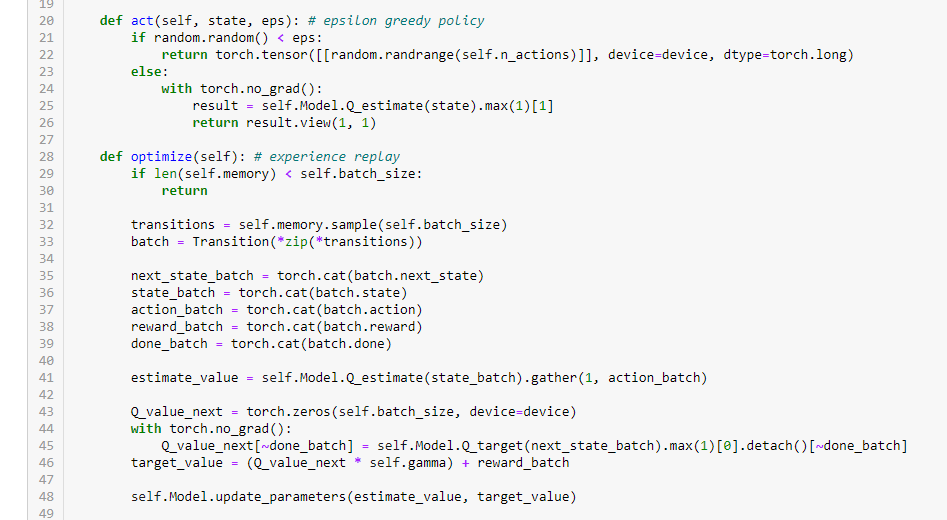


Now, we need to create our agent so we can do the training on our environment, so we created a class agent with attributes as mention in this picture below

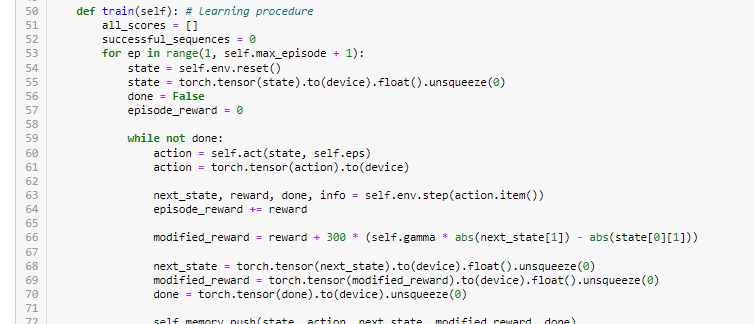
Memory\_size is the capacity of the replay memory, gamma is fixed on 0.99, and batch size is the length of the sample taken from the memory of the replaymemory and env is our environment.



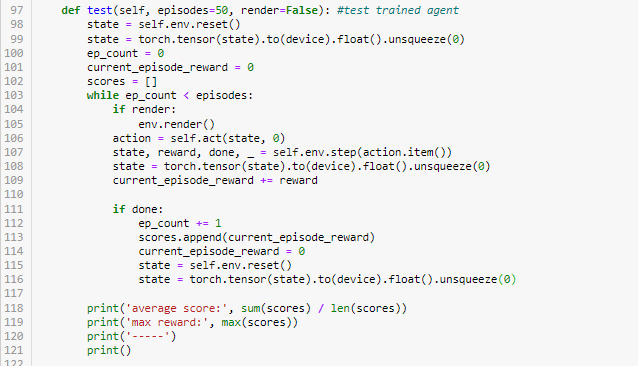
Act function to perform the greedy policy and select actions to fill the replay buffer. Optimize is to test the length of replay buffer and if it is superior of the batch size we update the parameters of the model using a specific Q\_value formula.



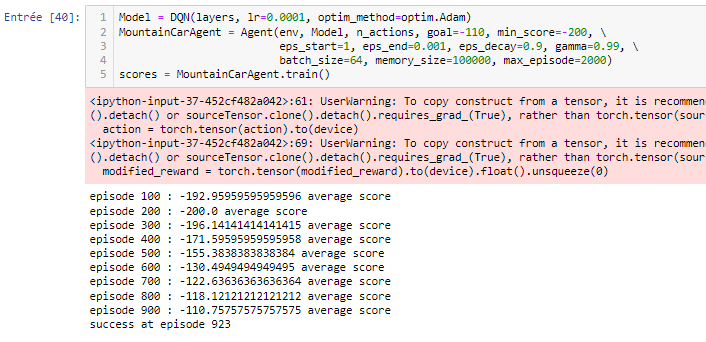
The training function is to train our target network on a certain number of episodes(max\_episode) while calculating the score of each one.



Test function to test the trained model (agent) on 50 episode while printing in the end the average score and the max reward.

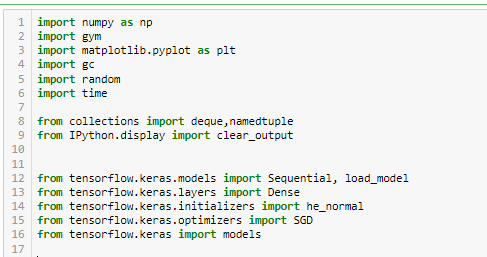


After initializing our model we created the agent with a goal to have at least -110 as a score and succeeded to reach his goal in 923 episode.

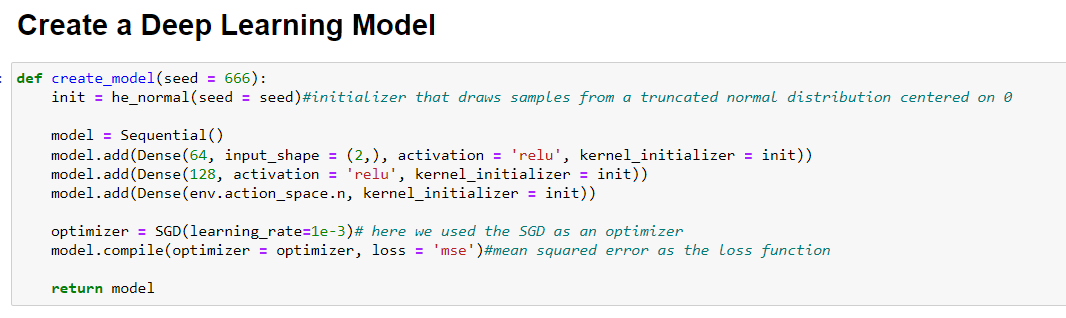


* **Double DQN**

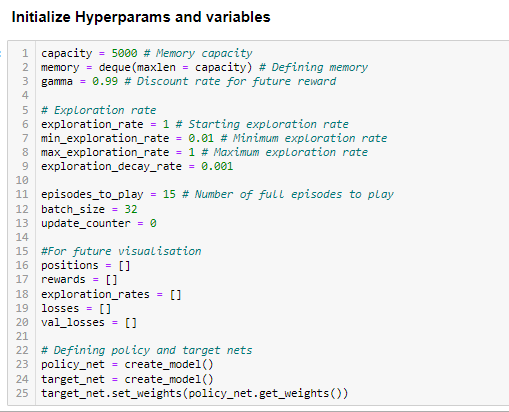
Import of libraries



Create\_model is used to create two identical neural network models. One learns during the experience replay, just like DQN does, and the other one is a copy of the last episode of the first model. These two networks are composed from three layers. The number of neurons in the input layer is the environmental state dimension. The number of neurons in the output layer is set to the environmental action dimension for all the models. Relu (Rectified Linear Unit) is used as the activation function, and the SGD algorithm is selected as an optimizer.



in this part we initialized the different hyperparameters. The capacity of the memory is set to 5000, gamma the same as dqn algorithm, 15 episodes to play,batch size set to 32and the exploration rate for the greedy strategy is fixed on 0.001.

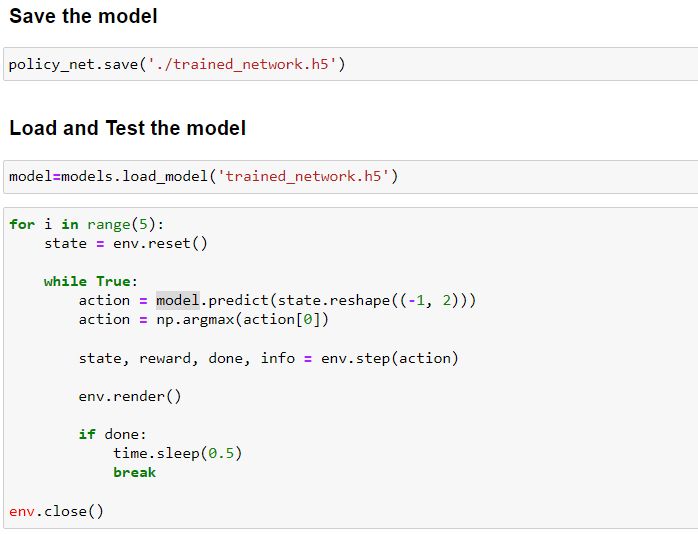


Here we started to implement the algorithm described in the learning algorithm section,so as a beginning we created a loop on 15 episodes and in each episode we did a while loop.

This while contain:

* Action selection according to the greedy strategy, a random action in selected if the exploration\_rate\_tresh is inferiour to exploration rate else it is selected using policy\_net network
* Calculate the reward and append [state, action,reward,new\_state] to the replay buffer
* If the size of the replay buffer is superiour then batch\_size the start the learning process
* Using the target\_net network we calculate the q\_optimal which will help us fit the policy\_net
* “target\_net.set\_weights(policy\_net.get\_weights())” this line of code will copy the weights of the policy\_net to be equal to th target\_net. This Strategy wil help us avoid the over-estimation of the return.



In the end, we saved our model “trained\_network.h5” and then we load it to test it on our environment and see how preferment it is.

1. **Conclution**

Deep Q Networks take as input the state of the environment and output a Q value for each possible action. The maximum Q value determines, which action the agent will perform. The training of the agents uses as loss the TD Error, which is the difference between the maximum possible value for the next state and the current prediction of the Q-value (as the Bellman equation suggests). As a result, we manage to approximate the Q-tables using a Neural Network.

So far so good. But of course, there are a few problems that arise. It’s just the way scientific research is moving forward. And of course, we have come up with some great solutions.

The trick is that the target value is not automatically produced by the maximum Q-value, but by the Target network. In other words, we call forth the Target network to calculate the target Q value of taking that action at the next state. And as a side effect, we also solve the moving target problem. Neat right? Two birds with one stone. By decoupling the action selection from the target Q-value generation, we are able to substantially reduce the overestimation, and train faster and more reliably.

We can relate here with our code performance for the two algorithme,the DQN take over 932 episode to converge but double DQN take only 50 episodes.