Project 1. Navigation

1 DQN Algorithm

The DQN (or Deep Q-Learning) algorithm is used in this project. DQN is an off-policy value-based method for discrete action environments. It uses an experience replay buffer to minimize correlations between samples, by randomly sampling a minibatch of samples at every learning step. It also uses a separate target $\hat{Q}$ network to calculate the target expected value and to have more stability of the targets along time. Both Q and $\hat{Q}$ networks have the same architecture, so that their weights can be copied between them. The algorithm was first introduced in [1] to learn to play Atari games. Its main steps are the following:

Algorithm 1 Deep Q-Learning with Experience Replay

Initialize replay memory $\mathcal{D}$ with capacity N
Initialize action-value network Q with random weights θ
Initialize target network $\hat{Q}$ with weights $\theta^- = \theta$
for episode $= 1 \dots M$ **do**
  Observe initial state s_1
  for $t = 1 \dots T$ **do**
    With probability ϵ select a random action a_t. Otherwise select $a_t = \text{argmax}_a Q\left(s_t, a; \theta\right)$.
    Execute action a_t and observe reward r_t and next state s_{t+1}
    Store tuple (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
    Sample random minibatch of tuples (s_j, a_j, r_j, s_{j+1}) from $\mathcal{D}$
    Set target $y_j = r_j + \gamma \max_a \hat{Q}\big(s_{j+1}, a; \theta^-\big)$
    Perform a gradient descent step on $\big(y_j - Q\left(s_j, a_j; \theta\right)\big)^2$ with respect to the network parameters θ
    Every C time steps reset $\theta^- = \theta$
  end for
end for

The double DQN improvement [2] has also been implemented. The main motivation of the double DQN is to solve the overestimation of the action values performed by the Deep Q-Learning algorithm [3]. Double Q-Learning has been shown to work well in practice to help with this. It consists of substituting the calculation of y_j in Algorithm 1 by:

$$y_j = r_j + \gamma\, \hat{Q}\left(s_{j+1}, \text{argmax}_a\, \hat{Q}\big(s_{j+1}, a; \theta\big), \theta^-\right)$$

That is, we first find the action that maximizes the output of the $\hat{Q}$ network with weights θ, and then we take the Q-value of the output of $\hat{Q}$ corresponding to this action, evaluated with weights θ^-. This technique prevents the algorithm from propagating incidental high rewards that may have been obtained by chance. It prevents Q-values from exploding in early stages of learning or fluctuating later on.

2 Python Code

The code is divided in three files: `model.py`, `dqn_agent.py` and `p1.py`. We have taken the DQN code provided in the course as a basis (in the directory `dqn/solution` of the Udacity repository).

The file `model.py` contains the *QNetwork* class. It defines the neural networks for Q and $\hat{Q}$. Both networks have identical architectures: two hidden layers of 64 units and a ReLU activation function in each one. The input of the networks is a state and the output is the action-value for the four possible actions.

The file `dqn_agent.py` contains the hyperparameters, the *Agent* class and the *ReplayBuffer* class. In both classes, a variable *self.device* has been added to specify the architecture in which the calculations are to be carried out (CPU or GPU). The double DQN technique has been added at the beginning of the *learn* method of the *Agent* class. Specifically, in line 92, the local network Q is used to get the positions of the actions corresponding to the maximum Q-values for the next states. Then, in line 94, we get the Q-values associated with these actions by performing a forward step of the target $\hat{Q}$ network with the next states.

The hyperparameters and their values are defined at the top of `dqn_agent.py`:

```
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 64         # minibatch size
GAMMA = 0.99            # discount factor
TAU = 1.0               # for soft update of target parameters
LR = 5e-4               # learning rate
UPDATE_EVERY = 2        # how often to apply the learning algorithm
UPDATE_TARGET = 4       # how often to update the target network weights
```

The learning algorithm is executed every 2 time steps, and the weights of the target $\hat{Q}$ network are updated every 4 time steps. Note that, since $\tau = 1$, a direct copy of the weights of the local Q network is performed. The Adam algorithm is used in the optimization process.

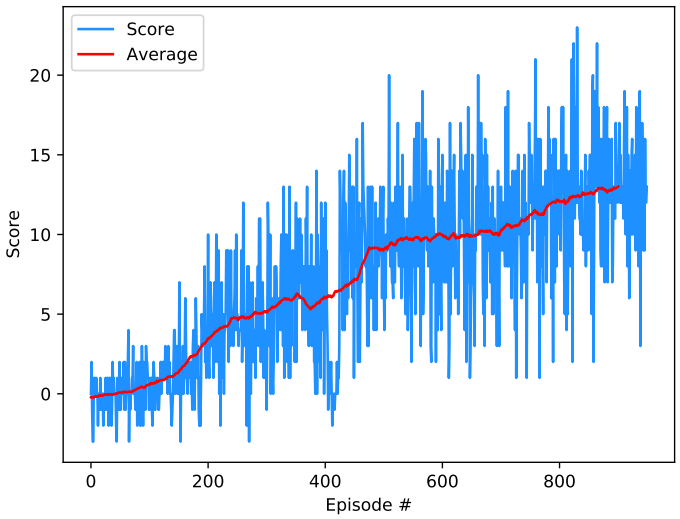
Finally, the file `p1.py` is the main file which contains the *main* function. It also has two more functions: *train_dqn*, which is called in the learning process, and *test*, called in the testing process.

3 Results

The code has been executed using Ubuntu 18.04 and Python 3.6.9. The environment is solved (i.e. an average score of +13 is reached over 100 consecutive episodes) in 850 episodes, and the text output is the following:

```
Episode 100 Average Score: 0.081
Episode 200 Average Score: 1.38
Episode 300 Average Score: 4.73
Episode 400 Average Score: 6.14
Episode 500 Average Score: 7.22
Episode 600 Average Score: 9.64
Episode 700 Average Score: 9.91
Episode 800 Average Score: 11.26
Episode 900 Average Score: 12.55
Episode 950 Average Score: 13.00
Environment solved in 850 episodes! Average Score: 13.00
```

The next graph shows the score reached at each episode. The red line is the moving average of 100 episodes:



By testing the obtained agent with the first 100 episodes of the environment we get an average score of 15.64.

4 Future Work

In order to get a better agent, we can try several things:

- Perform a longer training and reach a higher average score.
- Test other improvements that have been proposed for the DQN algorithm, such as the prioritized experience replay [4] or the dueling DQN [5].
- Test the Rainbow algorithm [6], which incorporates six extensions to the DQN algorithm, outperforming all of these extensions individually and reaching state-of-the art performance on Atari 2600 games.

References

[1] V. Mnih et al. Human-level control through deep reinforcement learning. *Nature*, 518, 529–533, 2015.

[2] H. van Hasselt, A. Guez, D. Silver. Deep reinforcement learning with double Q-learning. `arXiv:1509.06461v3 [cs.LG]`, 2015.

[3] S. Thrun, A. Schwartz. Issues in using function approximation for reinforcement learning. Proceedings of the Fourth Connectionist Models Summer School, 1993.

[4] T. Schaul et al. Prioritized experience replay. `arXiv:1511.05952v4 [cs.LG]`, 2015.

[5] Z. Wang et al. Dueling network architectures for deep reinforcement learning. `arXiv:1511.06581v3 [cs.LG]`, 2015.

[6] M. Hessel et al. Rainbow: combining improvements in deep reinforcement learning. `arXiv:1710.02298 [cs.AI]`, 2017.