Deep Reinforcement Learning : Navigation

In this project repository, I detail my results for the Project 1: Navigation.

Learning Algorithm

In this project, two *value-based* method, deep Q-learning with experience repaly and Doulbe Q-learning are implemented to achieve the goal.

Deep Q-learning

Deep Q-learning combine the Q-learning method (SARSA Max) and the deep neural network to represent the Q-table for approximation. However, the naive Deep-Q-Learning has two main problems:

- It will only learn from a single example, and this learning is effectively discarded each time. However, out function approximator is deep neural network, which is a high dimensional space. This make the training tends to be caught in a local minimum finally.
- It using the same network to evaluate the value and the maximum action as well when choosing the maximum action, which will change the correlations between them.

These problems were solved by the work of DeepMind publication: ["Human-level control through deep reinforcement learning" (https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf)](https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf). The two key ideas of DQN are:

- Experience replay: removing correlations in the observation sequences and smoothing over changes in the data distribution
- Q targets that are only periodically updates: reducing correlations with the target

The pseudo code of DQN algorithm from ["Human-level control through deep reinforcement learning" (https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf)](https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf) is shown as below.

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode $= 1, M$ **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1$,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^{-}\right) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

The architecure and the chosen parameters for the deep Q network are shown as follows:

```
QNetwork(
  (fc1): Linear(in_features=37, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=128, bias=True)
  (fc3): Linear(in_features=128, out_features=4, bias=True)
)
```

The implementation of the DQN is shown in `dqn_agent.py` of the source code.

Double Q-learning

Although DQN show promising performance, [study (https://arxiv.org/pdf/1509.06461.pdf)](https://arxiv.org/pdf/1509.06461.pdf) show DQN more likely to select overestimated values, resulting in overoptimistic value estimates. To tackle this, in Double Q-learning (DDQN), two value functions are learned by assigning each experience randomly to update one of the two value functions, such that there are two sets of weights, θ and θ'. For each update, one set of weights is used to determine the greedy policy and the other to determine its value. For comparison, the learning erros of DQN and DDQN are compared as below:

$$Y_t^{\mathrm{Q}} = R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\operatorname{argmax}}\, Q(S_{t+1}, a; \boldsymbol{\theta}_t); \boldsymbol{\theta}_t)$$

$$Y_t^{\mathrm{DoubleQ}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\operatorname{argmax}}\, Q(S_{t+1}, a; \boldsymbol{\theta}_t); \boldsymbol{\theta}'_t)$$

The implementation of the DDQN is shown in `ddqn_agent.py` of the source code.

Hyperparameters

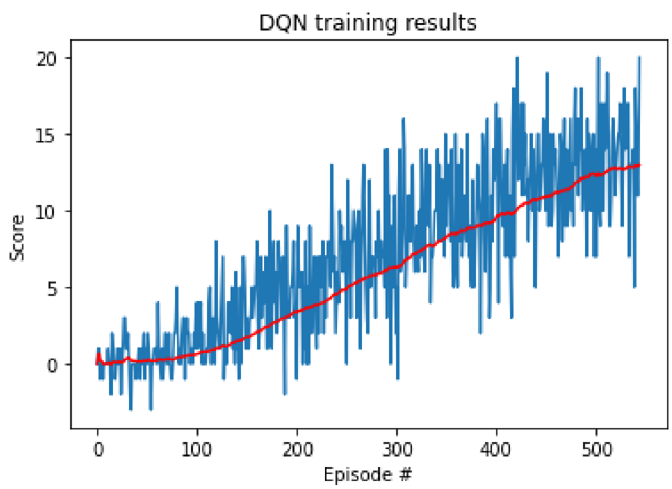
The chosen hyperparameters for both agents are shown as below:

```
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
```

Plot of Rewards

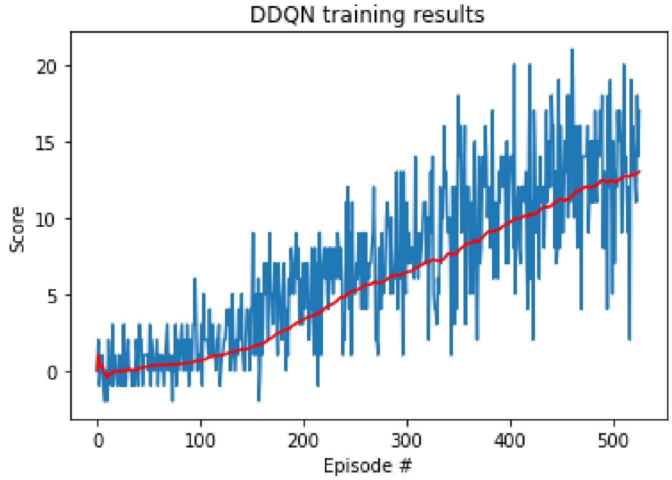
The results of DQN are shown as below:

```
Episode 100	Average Score: 0.61
Episode 200	Average Score: 3.39
Episode 300	Average Score: 6.31
Episode 400	Average Score: 9.47
Episode 500	Average Score: 12.37
Episode 544	Average Score: 13.00
Environment solved in 444 episodes!	Average Score: 13.00
```



The results of DDQN are shown as below:

```
Episode 100	Average Score: 0.63
Episode 200	Average Score: 3.27
Episode 300	Average Score: 6.39
Episode 400	Average Score: 9.66
Episode 500	Average Score: 12.39
Episode 526	Average Score: 13.00
Environment solved in 426 episodes!	Average Score: 13.00
```



Ideas for Future Work

For the future work, the performance of the agent can be improved by introducing:

- Dueling DQN
- Prioritized experience replay