DRL-Banana-Navigation - DQN Learning Algorithm - Description of the Implementation.

Deep Q-Learning needs several components to work:

1. Two Neural Networks
2. Exploration-Exploitation (e-greedy) Algorithm
3. Experience Replay Memory
4. Rules to update the target neural networks' parameters

1. Two Neural Networks: 'local' and 'target'

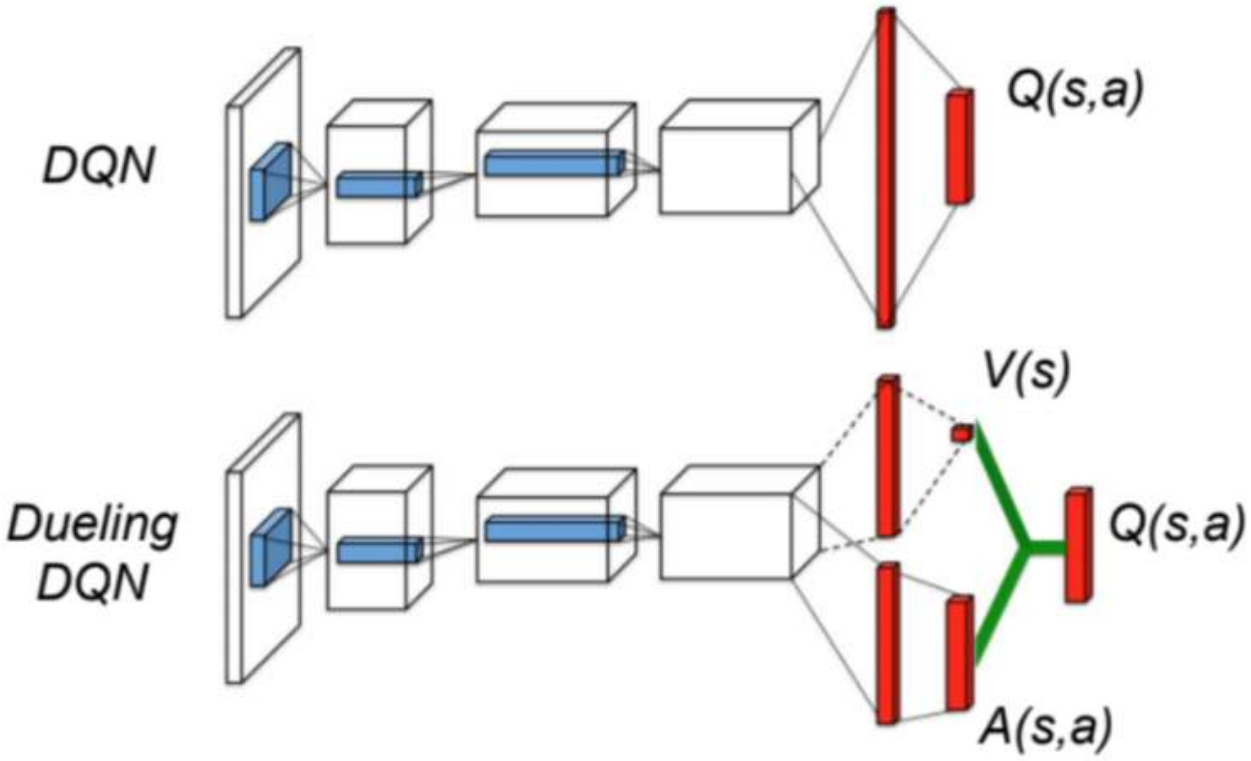
DQN uses a neural network to learn Q values. In this implementation, to stabilize the training, we are using two Duel Deep Q Networks, a main network, called 'local', and a second one, called 'target'. The target' weights are updated exclusively and periodically. These weights are fixed for the largest part of the training, this way, the Temporal Difference error, or TD-error, is calculated using a stable target updated only occasionally.

Both 'local' and 'target' networks are implemented using the **DuelQNetwork** class in **model.py**. The definition of a **DuelQNetwork** contains three neural networks with fully connected layers: feature, value approximator and advantage approximator models. They are all combined into one network, in the **forward**() function, to take 37 inputs, from state size, they contain 296 and 148 nodes on the hidden layers and provide 4 outputs, equal to the action size (296 and 148 come from calculating '2 * state size * action size' and 'state size * action size' respectively).

Duelling network: Split Q-network into two channels

- Action-independent value function $V(s, v)$
- Action-dependent advantage function $A(s, a, \mathbf{w})$

$$Q(s, a) = V(s, v) + A(s, a, \mathbf{w})$$



2. Exploration-Exploitation (e-greedy) Algorithm

The DQN agent explores and exploits the environment many times to learn the best execution policy. Exploration is about learning something new, while exploitation is using existing knowledge to choose the best action to take in the current environment. Switching between exploration and exploitation is controlled by the epsilon variable, hence the Epsilon-Greedy Policy algorithm name. During training epsilon is supposed to start in full exploration mode, therefore 'epsilon_start' is 1. Because we want the algorithm to converge with time, we lower the probability to take a random action less frequently. Starting from 1, the initial exploration probability 'epsilon_start', with each episode, epsilon decays to 0.995 of its current value, iteratively, until it reaches the value of 0.01, denoted as 'epsilon_end'. This means that occasionally, the agent will still perform a random move, without considering the optimal policy.

3. Experience Replay Memory

When a Q network is trained on successive states, the data from timestep to timestep will be strongly correlated and the network would tend to overfit. This means that the network has a significant risk that it forgot what it hasn't seen in a while. Using data stored in a replay buffer, the agent can be trained on old data and reduce the correlation between chronological experiences through uniform sampling and replay. This also increases the learning speed especially when learning from high-TD error transitions and reusing past transitions to avoid catastrophic forgetting. An improvement of experience replay is called prioritized experience replay where transitions are not sampled randomly with equal probability, but with one proportional to the TD error, because there is more to learn from transitions with higher errors than from others.

Prioritised replay: Weight experience according to surprise

- Store experience in priority queue according to DQN error

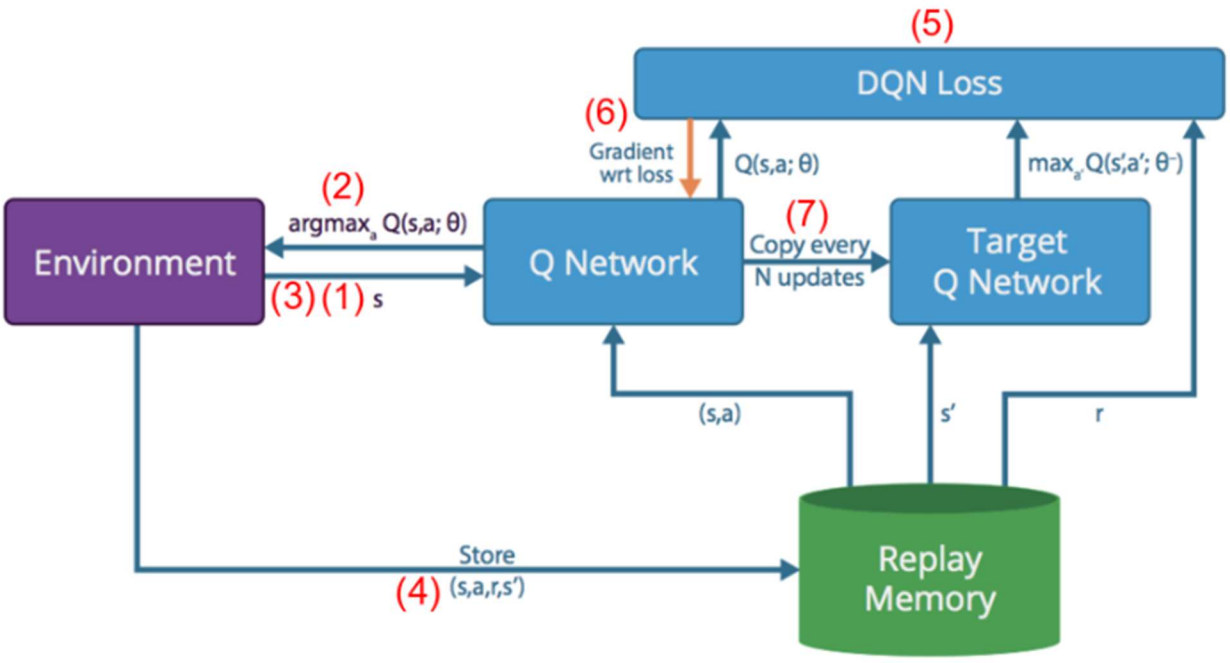
$$\left| r + \gamma \max_{a'} Q(s', a', \mathbf{w}^-) - Q(s, a, w) \right|$$

4. **Rules to update the target neural networks' parameters**

The target network weights can be updated gradually to track the 'local' learning network, via 'soft updates', or suddenly, via 'hard updates'. Sudden updates usually require a lot of steps. Therefore, they happen once every N > 10^4, before they can occur again, and training is slow. Some simple problems converge with small, 'soft updates' at smaller intervals, where N ≤ 10. In this implementation we are using the 'soft' approach, where N is 10.

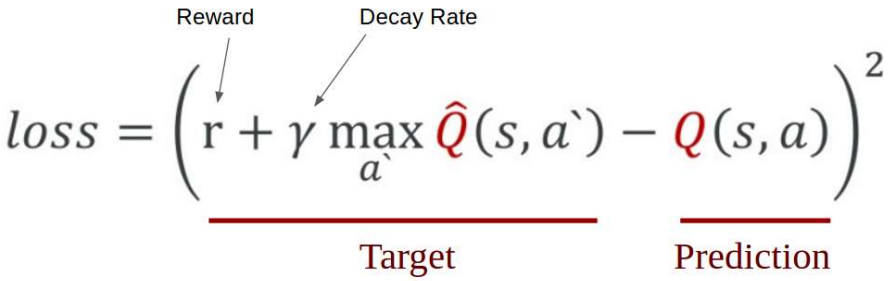
Implementation

In this project, the implementation uses **Python** 3.7 and **PyTorch** with two networks, as mentioned above, a 'local' and a 'target'. The input size is equal to the state size of 37, and the output is equal to the actions size of 4. The Adam optimizer is used by the model as it is shown to perform well in the typical case. The agent can be trained with a GPU or a CPU. The DQN agent updates the 'target' **Duel DQNetwork** every N = 10 time steps during training.

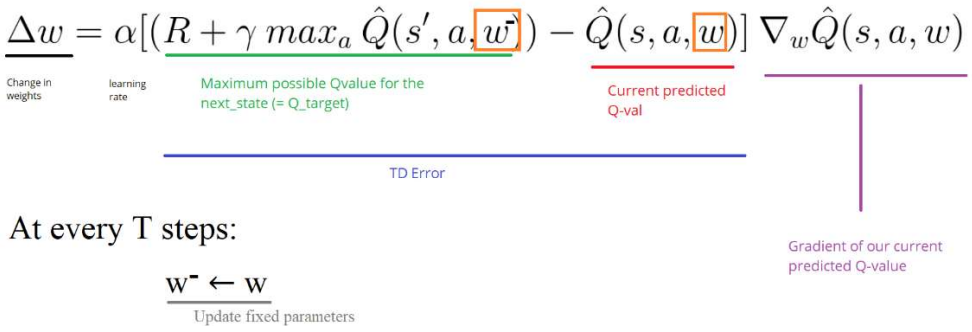


Description of the steps involved in the implementation of duel deep Q-network (DQN) follow below:

1. Take environment information and get the state from it, the current state.
2. The agent selects an action for current state and applies it on the environment. Action selection is using the epsilon-greedy policy. With the probability epsilon, a random action is selected (exploration). With probability 1 - epsilon, an action that has a maximum Q-value, such as a = argmax(Q(s,a,Θ)), is selected (exploitation).
3. In turn the environment sends back feedback containing a new state, reward (and terminal info, whether the game has ended or not)
4. The agent takes the feedback from the environment, this transition is stored in the replay buffer and the agent samples some random batches of historical data
5. The value parameters are updated every '**UPDATE_EVERY**' number of iterations and a second, 'target', network is used to generate the target-Q values that will be used to compute the loss.
6. Gradient descent is performed with respect to our 'local' network parameters in order to minimize this loss between the 'target' and estimated 'local' Q-values (prediction).



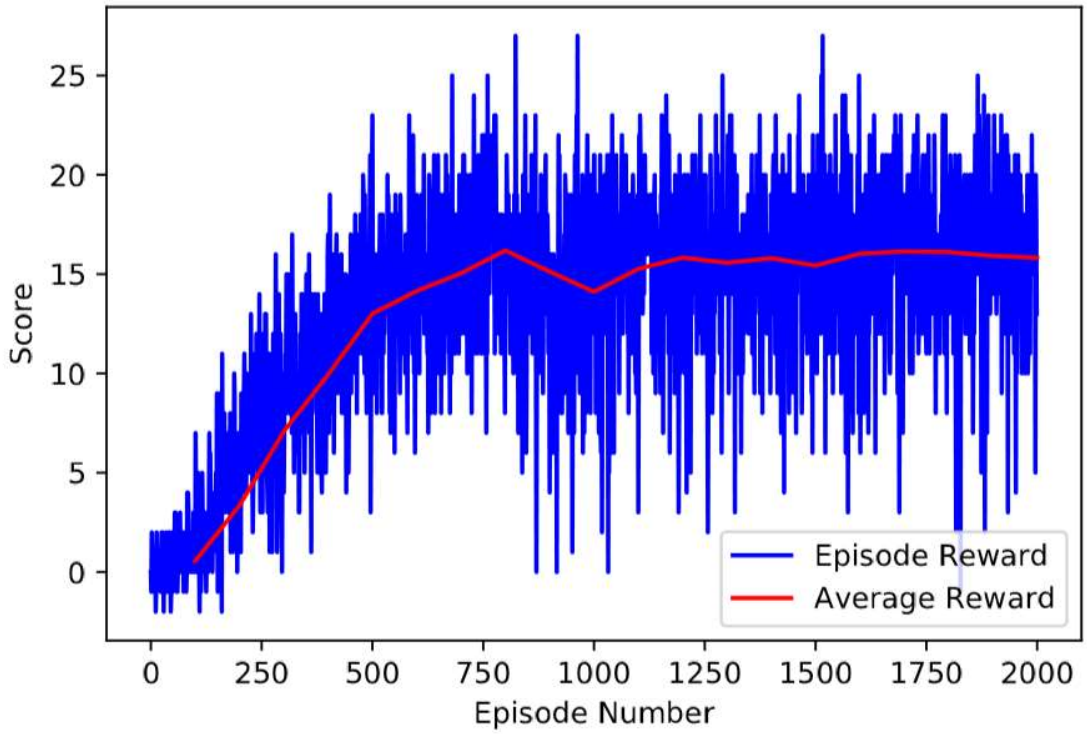
7. A soft update of the 'target' network weights is done by copying from the 'local' network, using an interpolation parameter Tau
8. Repeat these steps for 2000 episodes



Training Result

The agent takes 500 episodes to achieves an average reward of 13:

```
Episode 100     Average Score: 0.56
Episode 200     Average Score: 3.35
Episode 300     Average Score: 7.08
Episode 400     Average Score: 9.94
Episode 500     Average Score: 13.01
Episode 600     Average Score: 14.15
Episode 700     Average Score: 15.04
Episode 800     Average Score: 16.19
Episode 900     Average Score: 15.12
Episode 1000    Average Score: 14.11
Episode 1100    Average Score: 15.26
Episode 1200    Average Score: 15.83
Episode 1300    Average Score: 15.57
Episode 1400    Average Score: 15.80
Episode 1500    Average Score: 15.43
Episode 1600    Average Score: 16.03
Episode 1700    Average Score: 16.14
Episode 1800    Average Score: 16.11
Episode 1900    Average Score: 15.92
Episode 2000    Average Score: 15.82
```



Even though the training performance was oscillating between 15 and 16 in '**Navigation**.ipynb', executing '**Navigation-Viewer**.ipynb', the trained model can achieve a score of 18.

Future Improvements

Using a **Duel DQN** agent achieves a pretty good result, a score above 13, within 500 episodes, but there is number of areas we can improve.

In this implementation we are using Uniform Experience Replay. However, as mentioned in the introduction, a better candidate would be **Prioritized Experience Replay** because it gives higher priority to experiences with higher error, and therefore more to learn from, while discarding the other ones.

Also, the next improvement that would follow naturally would be a **Dueling Double Q-Learning** implementation, with **Prioritized Experience Replay.**