Report of Prioritized DDQN for Navigation Problem (Udacity DRLND)

1. Implementation and model architectures

We mainly follow the algorithm of prioritized double deep Q network with experience replay:

Initialize replay memory with capacity **BUFFER_SIZE**

Set up action-value function neural network Q with random weights θ

Set up target action-value function neural network $\hat{Q}$ with weights $\theta^- = \theta$

For episode = 1, M **do**

Initialize environment and acquire initial state s_1.

For t = 1, T **do**

Select action using epsilon-greedy, with varying eps = max(**eps_end**, eps***eps_decay**) and $Q(\theta)$

Execute action in environment and observe reward r_t, state s_{t+1} and done information.

Use $Q(\theta)$ to choose a* which gives the max action value in state s_{t+1}, and produce the TD-difference delta δ_t by $\hat{Q}(\theta^-)$

$$\delta_t = |y_t - Q(s_t, a_t; \theta)|$$

where

$$y_t = \begin{cases} r_t & \text{if done} \\ r_t + \gamma \hat{Q}(s_{t+1}, a^*; \theta^-) & \text{otherwise} \end{cases}$$

Store tuple $(s_t, a_t, r_t, s_{t+1}, \delta_t, \text{done})$ in replay memory.

If the size of replay memory is larger than **BATCH_SIZE**, and $(t + 1)\%$**UNPATE_EVERY** == 0,

select mini-batches of tuples = $\{tu_1, tu_2, \ldots, tu_{\mathbf{BATCH\_SIZE}}\}$ as training set obeying the prioritized sampling method, where $tu_i = (s_i, a_i, r_i, s_i', \delta_i, \text{done})$. The probability for picking the i-th tuple is given by

$$P_i = \frac{\delta_i^{\mathbf{Exp\_A}}}{\sum_{i=1}^{\mathbf{BUFFER\_SIZE}} \delta_i^{\mathbf{Exp\_A}}}$$

Train network $Q(\theta)$ with loss function

$$L = \frac{1}{\mathbf{BATCH\_SIZE}} \sum_{i=1}^{\mathbf{BATCH\_SIZE}} (\mathbf{BATCH\_SIZE} \times P_i)^{-\mathbf{Exp\_B}} \delta_i^2$$

and update $\theta^- = (1 - \mathbf{TAU}) \times \theta^- + \mathbf{TAU} \times \theta$, where $\mathbf{Exp\_B} = \mathbf{Exp\_B}^{0.995}$.

End for

End for

Architecture of the action value Q network

We use four-layer fully connected networks as a surrogate of the Q-table. Each layer (except for the last layer) is applied a relu activation function. This net is sketched as follows:

```python
        super(QNetwork, self).__init__()
        self.seed = torch.manual_seed(seed)
        "*** YOUR CODE HERE ***"

        # linear layers
        self.fc1 = nn.Linear(state_size,64)
        self.fc2 = nn.Linear(64,256)
        self.fc3 = nn.Linear(256,32)
        self.fc4 = nn.Linear(32,action_size)

        # activation functions
        self.relu = nn.ReLU(inplace=True)
        self.softmax = nn.Softmax()

    def forward(self, state):
        """Build a network that maps state -> action values."""

        x = self.fc1(state)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.fc3(x)
        x = self.relu(x)
        x = self.fc4(x)

        return x
```

Figure 1. Neural network

2. Parameter set up and preliminary results

Table 1 shows the values of the hyper-parameters above.

Table 1. Hyper-Parameters

Para.	Value	Para.	Value
BUFFER_SIZE	10^5	BATCH_SIZE	128
LR	5×10^4	UPDATE_EVERY	10
GAMMA	0.99	TAU	5×10^{-3}
eps_start	1.0	eps_end	0.01
eps_decay	0.995	Exp_A	0.5
Exp_B	0.001		

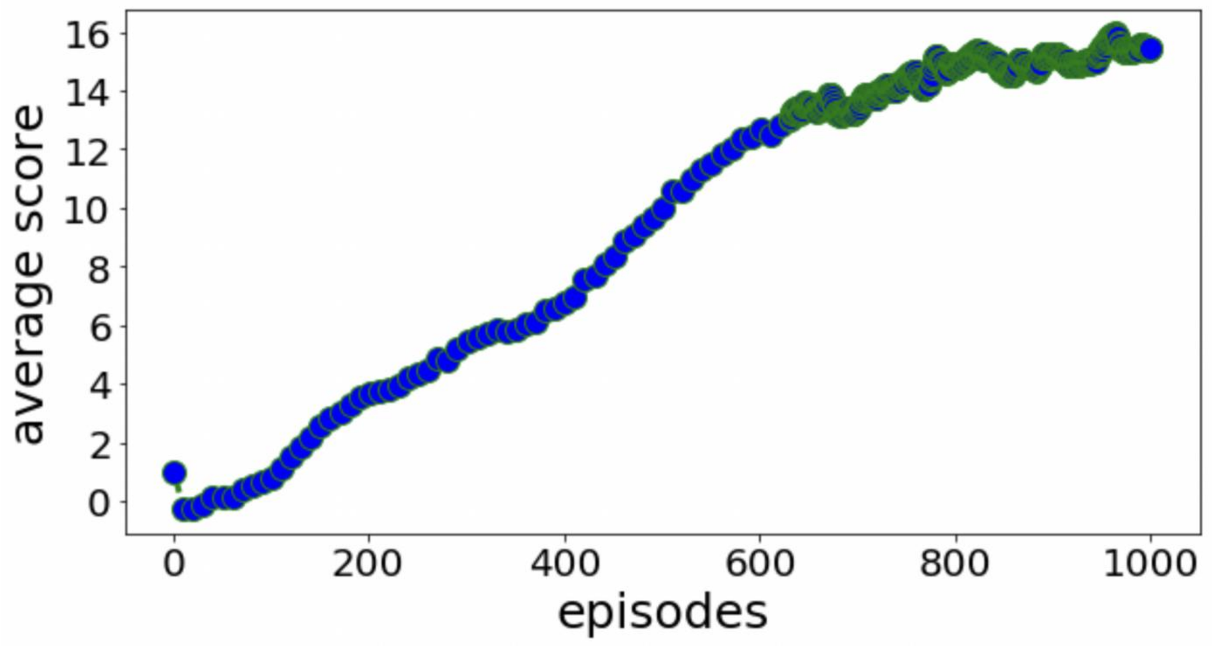

Figure 2. The average rewards per episode.

The agent is able to receive an average reward (over 100 episodes) of 13.02 in 630 episodes.

3. Tendencies that might improve the performance

3.1 Increase the number of the DQN layers. We used four full-connected layers (state_size*64, 64*256, 256*32, 32*action_size) in the DQN networks resulting in satisfied results. However, as the decrease of the number of the layers, the training results becomes worse. Besides, decreasing the nodes in the hidden layers, for instance, changing 256 nodes to 128 nodes will greatly make DQN perform worse.

3.2 Adopt the proper hyper parameters. How to choose hyper parameters is actually complicated, which needs lots of experiences and experiments. For this project, the suitably larger **BATCH_SIZE**, smaller **LR** and **TAU** will give stable and nice training. However, it is not correct to use extremely small **LR** or **TAU** which decrease the training efficiency.

3.3 Employ better active functions. We only use Relu function in our project, and we think other proper active functions can increase the training performance to some extent.