Udacity Deep Reinforcement Learning Nanodegree

Project 1 solution: Banana Navigation

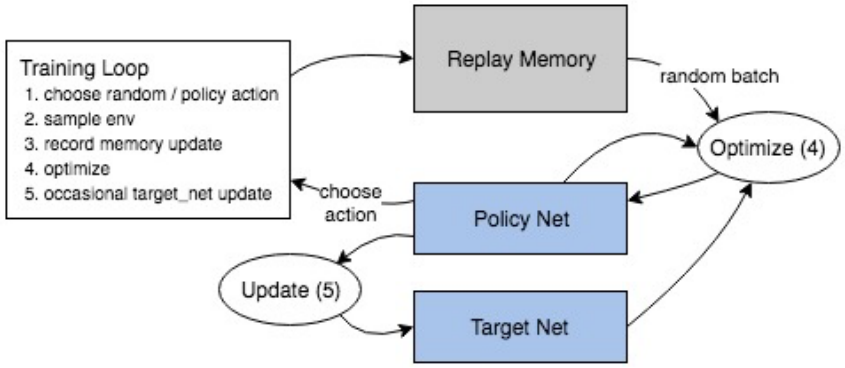
Summary of DQN

The idea of double Q-learning is to reduce overestimations by decomposing the max operation in the target into action selection and action evaluation.

In the vanilla implementation, the action selection and action evaluation are coupled. We use the target-network to select the action and estimate the quality of the action at the same time.

The target-network calculates $Q(s, a_i)$ for each possible action a_i in state s. The greedy policy decides upon the highest values $Q(s, a_i)$ which selects action a_i.

This means the target-network selects the action a_i and simultaneously evaluates its quality by calculating $Q(s, a_i)$. Double Q-learning tries to decouple these procedures from one another.



Another improvement is to use the replay buffer, this means instead of running Q-learning on state/action pairs as they occur during simulation or actual experience, the system stores the data discovered for $[state, action, reward, next\_state]$ - typically in a large table. Note this does not store associated values - this is the raw data to feed into action-value calculations later.

The advantages of experience replay buffer include:

- Use previous experience more effectively by learning multiple times. This is important when obtaining real-world experience is expensive, and you can make the most of all experiences through the environment. The traditional Deep Q learning updates are incremental and do not converge efficiently, so using multiple passes of the same data is beneficial, especially when the immediate outcomes (reward, next state) of the same state, action pair are less different.
- Better convergence behavior when training function approximators. Part of the reason is that the data is more i.i.d..

The difference between basic DQN and double DQN

The main difference between basic DQN and double DQN are shown:

Basic Q-Learning

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(R_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$$

Double Q-Learning

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(R_{t+1} + \gamma \boxed{Q'(s_{t+1}, \boxed{a})} - Q(s_t, a_t))$$

estimated/expected Q-value

$$\boxed{a} = \max_a Q(s_{t+1}, a)$$

$$q_{estimated} = \boxed{Q'(s_{t+1}, \boxed{a})}$$

And the implementation of DQN is:

Algorithm 1 Double Q-learning

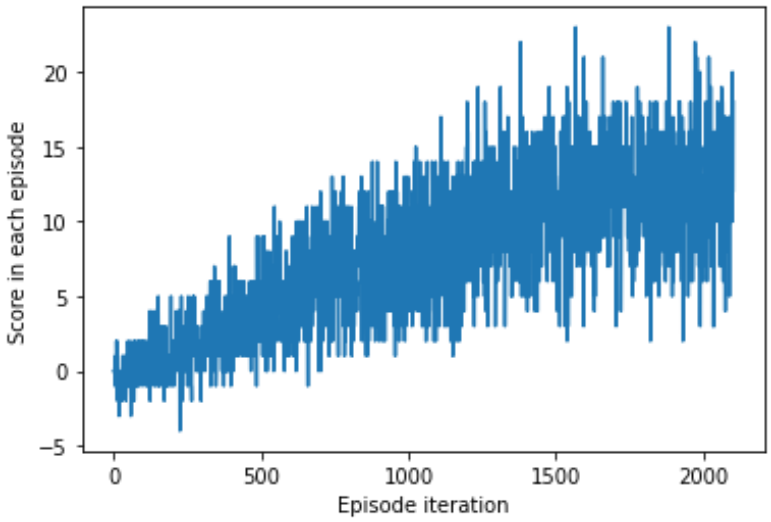
Initialize Q^A, Q^B, s
repeat
  Choose a, based on $Q^A(s, \cdot)$ and $Q^B(s, \cdot)$, observe r, s'
  Choose (e.g. random) either UPDATE(A) or UPDATE(B)
  if UPDATE(A) **then**
    Define $a^* = \arg\max_a Q^A(s', a)$
    $Q^A(s, a) \leftarrow Q^A(s, a) + \alpha(s, a)\left(r + \gamma Q^B(s', a^*) - Q^A(s, a)\right)$
  else if UPDATE(B) **then**
    Define $b^* = \arg\max_a Q^B(s', a)$
    $Q^B(s, a) \leftarrow Q^B(s, a) + \alpha(s, a)(r + \gamma Q^A(s', b^*) - Q^B(s, a))$
  end if
  $s \leftarrow s'$
until end

The code structure:

- Navigation.ipynb: the jupyter notebook file runs the training and demonstration session.
- dqnAgent.py: this python file defines the agent using double DQN to learn the Q function, as well as the replay buffer.
- model.py: this python file defines neural network structure used for estimating the DQN value estimation function.

Results

The average result for every 100 episodes is shown in this plot.



Next step

The next step for this project is to try the implementation of DQN based on pixels. Also, try different NN structures to see if the converge speed would be improved.