Tsinghua University

Deep Reinforcement Learning

Spring 2024

Assignment 2

Subject: Deep Q-Network

Author: Tonghe Zhang(张同和)
Email: zhang-th21@mails.tsinghua.edu.cn
Affiliation: Department of Electronic Engineering, Tsinghua University

April 12, 2024

Contents

1 About this Report **1**

2 Training Results and the Analysis **2**

2.1 Deep Q-Network 2
2.2 Double DQN 2
2.3 Dueling DQN 3
2.4 Prioritized Experience Replay 4
2.5 N-Step Replay 5

1 About this Report

Contents of the project We implemented the network architecture of vanilla Deep Q-net with target network, Double Deep Q-net, and the Dueling DQN. We also hand-crafted N-step replay buffer and Prioritized replay buffer to stablize training. We tested the network using Cartpole-v1 envirnomet.

Analysis in the report In this short report we will highlight several technical details that are easily overlooked but can significantly influence the training results. We also try to distinguish several confusing concepts emerged in the various algorithms.

2 Training Results and the Analysis

2.1 Deep Q-Network

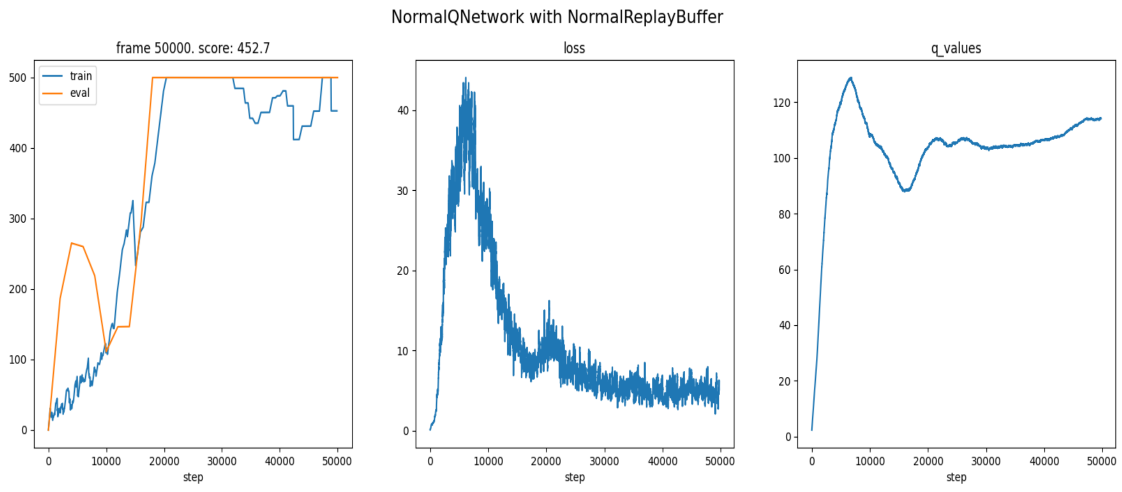

Fig 1: Deep Q Network Training Result

The implementation of the Deep Q-Network is rather straightforward. And the evaluation values converges eventually.

2.2 Double DQN

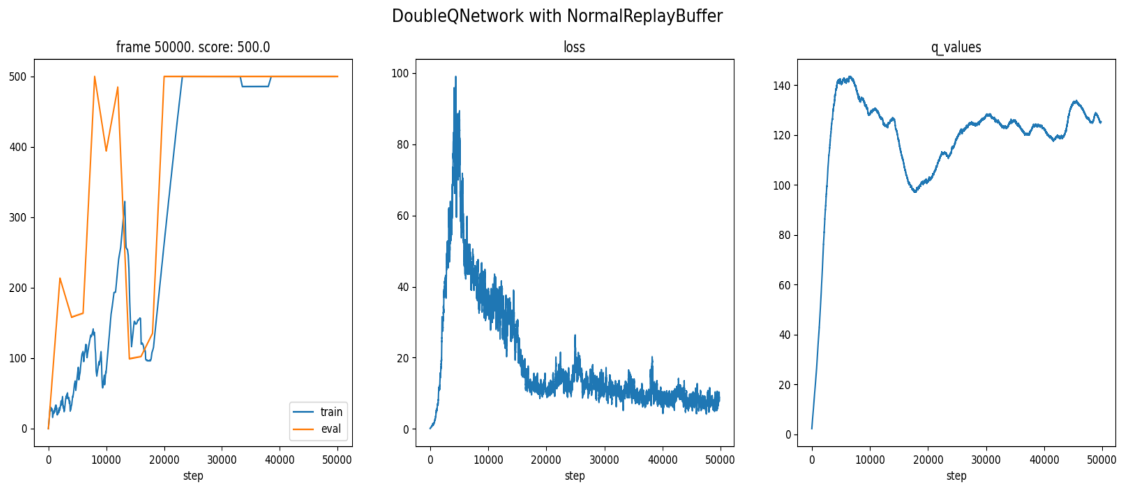

Fig 2: Double Deep Q Network Training Result

Distinguish between the target network and Doublde/Dueling DQN:

- Since Vanilla DQN can easily cause over-estimation problem due to trainig errors, we introduce the target network θ_t as an obsolete copy of the original Q-network. We periodically update its parameters from the original Q-network θ by the following formula $\theta^- = \tau * \theta + (1 - \tau)\theta^-$, where $\tau \in (0, 1)$ is the mixing ratio.We call this process "soft-update" of the target network.

- Double DQN and Dueling DQN are just other two types of methods to stablize DQN. In particular, Double DQN is used to update the target network $Q(\cdot,\cdot;\theta^-)$, while Dueling DQN modifies the update of the original Q-network θ. In both implementation, the soft-update is still used periodically, after we compute the Q-functions and its target nets.
- In Double DQN, the target update is changed to: $Q(s,a;\theta^-) = r(s,a) + \gamma Q(s', \arg\max_{a'} Q(s',a';\theta);\theta^-)$ where $Q(s,a;\theta^-)$ is the target net and $Q(s',a';\theta)$ is the original Q-net. We choose action based on the most recent Q-values, while we evaluate the policy using the older target values.
- In Dueling DQN, the original Q-net is not the output of a single series of neuron networks, but the output of two suites of nets sharinng the saem input feature map. $Q(s,a;\theta) = V(s;\theta) + \underset{a}{\text{Nomalize}} A(s,a;\theta)$

2.3 Dueling DQN

Very importanct notes on the implementation of Dueling Q-Network:

- The value net and advantage net should share the same input feature. If we adopt separate features, the training and evaluation curves will be highly unstable. This is possibly because more parameters may cause the model to overfit. Using shared features is also a common practice.
- We should manually normalize the advantage function. Even though it is normalized by definition, it is not necessary be when we use neuron nets to approximate its value. For simplicity, we do not average the advantage function using the policy as the weight. Instead we directly subtract its value with the mean.

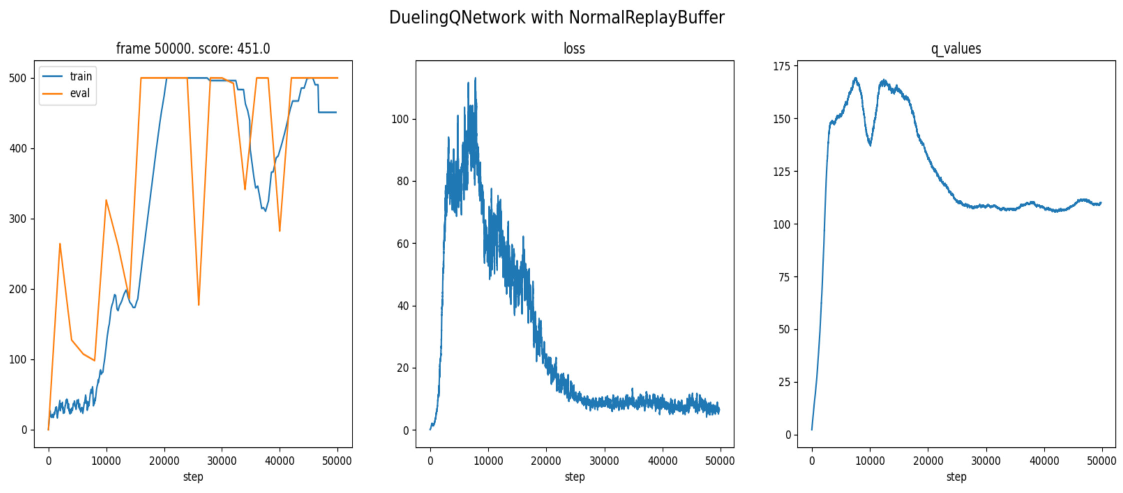

Fig 3: Dueling Deep Q Network Training Result

2.4 Prioritized Experience Replay

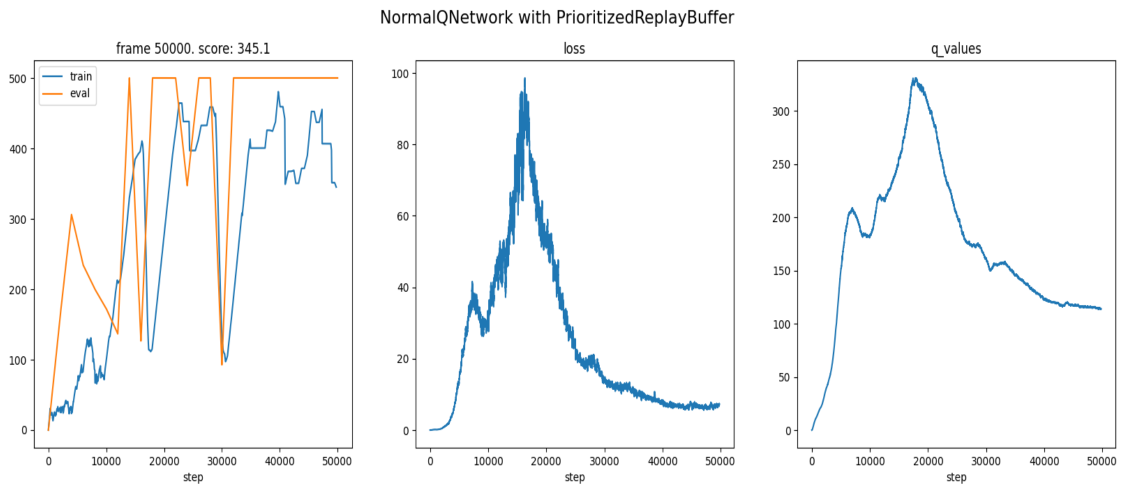

Fig 4: Prioritized Experience Replay Training Result

Several remarks:

- The algorithm and mathematical descriptions of PER are listed on page 4-5 of (Schaul et al., 2016).
- The $P(i)$ in PER is completely different from $p(i)$. We only update the $P(i)$ for the sampled batch, while we update all the entries of $p(i)$ in the entire buffer:
  p=self.priorities/self.priorities.sum()
  P=torch.as_tensor(self.priorities[sample_idxs]/self.priorities[sample_idxs].sum())
  Also be aware that we use P instead of p to update the weights w.
- The N used in the importance sampling step should be the actual numeber of samples 'self.size', but not the capacity of the buffer 'self.capacity' If it is wrongly specified, the reward curves will oscillates to the quite dramatically. self.size is the effective size of the data loader, or the actual number of samples that can be used. self.size==min(self.capacity, self.size + 1)
- I discover that if we normalize priorities as 1, instead of 0, the algorithm converges better asymptotically. This coincides with the design of (Schaul et al., 2016)(see line 2, Algorithm 1).

2.5 N-Step Replay

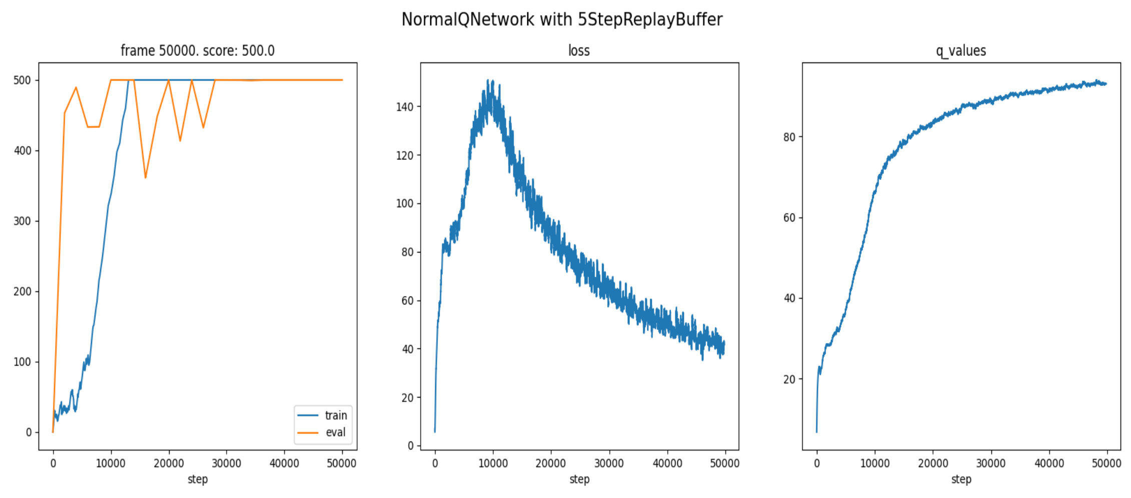

Fig 5: N-Step Return Training Result

N-step Replay buffer is very useful to accelrate convergence. It forces the agent to learn with a sense of farsightedness.

Technically speaking, we create an "N-step replay buffer" in addition to the larger "main buffer" to store sampled trajectories. We use the N-step replay buffer to rectify the instand rewards with accumulated N-step reward-to-go, and add the modified trajectories $(s_t, a_t, \sum_{k=0}^{N-1} \gamma^k R_{t+k-1}, s_{t+N})$ to the main buffer. The agent will train on these modified trajectories to fit the Q value using gradient descent. The procedure is as follows:

- The main buffer is a stack of arrays, each array represents "states", "actions", "rewards", "next states" and "done flag of the current state".
- The N-step replay buffer does not store the next state, but the rest are the same with the main buffer.
- During the sampling process, we first store newly sampled trajectories in the N-step replay buffer(instead of the main buffer), continuously filling it up until the N-step replay buffer is full.
- We then retrieve the first (s,a) pair from the N-step replay buffer, which is our s_t, a_t.
- Then we calculate the N-step reward-to-go: $R_t^n := \sum_{k=0}^{N-1} R_{t+k-1}$.
- We will record whether there is a done flag from s_t (included) to s_{t+N-1} (included). If there is one, then we set variable Done as True.
- The next state s_{t+N} is the last state in the N-step replay buffer.
- We put $(s_t, a_t, R_t^n, s_{t+n}, Done)$ to the main buffer. We will then sample a batch from main buffer, compute Bellman update and td error to update the parameters using gradient descent.

References

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2016.