Deep Q-learning for Solving the Cart-Pole problem
Project Report

Tao Li

May 19, 2020

Abstract

In this report, we discuss a python implementation of deep Q-learning algorithm based on pytorch and its application to the cart-pole problem. The report consists of the following parts. We first give a quick review about deep Q-learning and some techniques for stabilizing the training process. We then move to the python implementation and provide details about this project. We start with an introduction of the cart-pole problem and the related OpenAI gym environment, which we use as a simulator. Based on this environment, we illustrate the design of the python program. Finally, we present experiment results, showing that the problem is successfully solved. For the python implementation, simply run `DQN_experiment` and gifs and plots will pop out.

1 Deep Q-Learning

For simplicity, we skip the introduction of Markov Decision Process (MDP) and reinforcement learning [Sutton and Barto, 2018], as we assume readers are familiar with the background and we follow the standard MDP notation: MDPNv1 specified in [Thomas and Okal, 2015]. Based on Bellman principle, for learning the optimal action-value function $Q^*(s,a) := \max_\pi \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots | s_t = s, a_t = a, \pi\right]$, one can start with an initialization of $Q(s,a)$ and perform the following Q-learning update [Watkins and Dayan, 1992] with any transition sample (s_t, a_t, r_t, s_{t+1}):

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

where α is the learning rate and this update rule is based on tabular estimation. In order to deal with complicated tasks such as Atari games [Bellemare *et al.*, 2013], where large state and action spaces make eq. (1) impractical, Mnih *et al.* [2015] propose to use nonlinear function approximators based on neural networks in Q-learning, which allows for greater powers of generalization as well as a richer representation class in challenging MDP environments. More specifically, we can parameterize an approximate value action-value function $Q(s,a;\theta)$ using a neural network, where θ is the weights of the network and it is updated by minimizing the following mean square loss function

$$L(\theta) := \mathbb{E}_{(s,a,r,s')}\left[\left(r + \gamma \max_{a'} Q\left(s',a';\theta\right) - Q\left(s,a;\theta\right)\right)^2\right]. \tag{2}$$

As we compare eq. (2) with eq. (1), we can see that instead of using temporal-difference learning [Sutton and Barto, 2018] , eq. (2) tries to reduce difference by doing least-square fitting.

However, when using nonlinear function approximators, it is known that learning process is unstable or even unable to converge. This instability arises from the following facts as pointed out in [Mnih *et al.*, 2015]: (1) the correlations present in the sequence of observations violate the

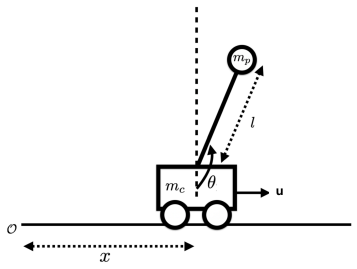


Figure 1: The cart-pole system: the difficulty of the cart-pole system is that we can only move the cart back and forth to balance the pole.

i.i.d assumption in least square fitting, (2) small updates to the neural network may result in significant changes to the greedy policy and thus lead to the change of data distribution and the correlations between $Q(s, a; \theta)$ and $r + \gamma \max_{a'} Q(s', a')$. This makes it difficult for the network to learn because the target is always moving and from optimization perspective, this means that the descent direction at the same point (same transition sample) is always changing. To address these issues, two key ideas have been proposed in the same paper. The first is experience replay that randomly selects samples at each iteration and hence removes the correlations between observations and the second one is target network, a copy of the Q network that is updated periodically. This means that the error is going to be stable for some time allowing the Q network to learn.

In addition to experience replay and target networks, another method for improving deep Q-learning is to use double Q-learning[Hasselt, 2010; Hasselt *et al.*, 2016], which is proposed to address overestimation issue in deep Q-learning. As observed in [Thrun and Schwartz, 1993; Hasselt *et al.*, 2016], in noised environment, Q learning tends to overestimate the performance of an action and the estimation error asymptotically leads to sub-optimal policies. Double deep Q-learning (DDQN) resolves this issue by separating action selection and action evaluation. More concretely, during the training process, we first use the Q-network to find the optimal action, i.e., $a' = \arg\max_a Q(s, a; \theta)$ and the evaluate its Q value using the target network $Q^t(s, a'; \theta^t)$, where Q^t is the target network parameterized by θ^t. Therefore, in DDQN, eq. (2) is replaced by

$$L(\theta) := \mathbb{E}_{(s,a,r,s')} \left[\left(r + \gamma Q^t \left(s', \arg\max_a Q(s, a; \theta); \theta^t \right) - Q\left(s, a; \theta\right) \right)^2 \right]. \tag{3}$$

The details and implementations of all these techniques are discussed in the following section, where we use what we have introduced to solve the cart-pole problem. We conclude this section with the following pseudocode (see algorithm 1) that summarize deep Q-learning and for double Q learning, we just replace the loss with eq. (3).

2 Python Implementation

2.1 The Cart-Pole Problem

The cart-pole problem is a classical control task [Barto *et al.*, 1983], where a pole is attached by an un-actuated joint to a cart, which moves along a frictionless track, see fig. 1. The system is controlled by applying a force of +1 or -1 to the cart and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright and the episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.

In our project, we use the environment 'CartPole-v0' developed by OpenAI [Brockman *et al.*, 2016] and the environment specification from the documentation is given below.

Algorithm 1 Deep Q-Learning with Experience Replay using $\epsilon-$greedy polciy

Input: exploration parameter ϵ, target network update frequency N and minibatch size
Initialize the Replay Memory D
Initialize the Q-network Q and the target networkQ^t with parameters θ and $\theta^t = \theta$ respectively
while not solved **do**
    for each episode **do**
        Reset the initial state s_0
        for each step t **do**
            Generate a random number p
            if $p < \epsilon$ **then**
                Choose a random action a_t
            else
                $a_t = \arg\max_a Q(s_t; \theta)$
            Take action a_t, receive a reward r_t and arrive at a new state s_{t+1}
            Store the transition sample (s_t, a_t, r_t, s_{t+1}) into the replay buffer D
            Sample a mini-batch of transition samples from D and get the loss

$$L(\theta) = \begin{cases} \left(r_t - Q\left(s_t, a_t; \theta\right)\right)^2, & \text{if } s_{t+1} \text{ is a terminal state} \\ \left(r_t + \gamma \max\limits_{a'} \left(Q^t\left(s_{t+1}, a; \theta^t\right)\right) - Q\left(s_t, a_t; \theta\right)\right)^2, & \text{otherwise} \end{cases}$$

            Update θ with respect to $L(\theta)$
            Set $\theta^t = \theta$, for every N steps

```
Observation:
        Type: Box(4)
        Num Observation               Min             Max
        0 Cart Position             -4.8            4.8
        1 Cart Velocity             -Inf            Inf
        2 Pole Angle                -24 deg         24 deg
        3 Pole Velocity At Tip      -Inf            Inf
    Actions:
        Type: Discrete(2)
        Num Action
        0 Push cart to the left
        1 Push cart to the right
    Reward:
    Reward is 1 for every step taken, including the termination step
    Starting State:
        All observations are assigned a uniform random value in [-0.05..0.05]
    Episode Termination:
        Pole Angle is more than 12 degrees.
        Cart Position is more than 2.4 (center of the cart reaches the edge of
        the display).
        Episode length is greater than 200.
        A Flag done indicts whether the episode terminates (done = True) or not (done = False).
        Solved Requirements:
        Considered solved when the average reward is greater than or equal to
        195.0 over 100 consecutive trials.
```

Table 1: The information provided by the environment

values	data type	details
observation	object	state information of the system
reward	float	reward signal received from the environment
done	boolean	whether the episode terminates or not

To sum up, when using this simulator, at each time step, a tuple comprising an observation, a reward and a flag is returned, see table 1 for details.

2.2 Program Design

As we have shown the generic framework of the algorithm in algorithm 1, in this section, we present some details about implementations of Q-networks and experience replay.

The Q-Network Since the problem is relatively a simple control task, we use a feed-forward neural network with 2 hidden layers and rectified linear function [Nair and Hinton, 2010] is chosen as the activation. Besides, we use Adam [Kingma and Ba, 2014] as the optimizer for training the neural network. Based on this Q network, we further define the following functions:

- `get_qvalue(self, state)`: for a state s, get $Q(s, a)$ for all actions.
- `greedy_action(self, state)`: get the greedy action based on Q values.
- `get_action(self, state, epsilon=0.05)`: get an action from the $\epsilon-$greedy policy.

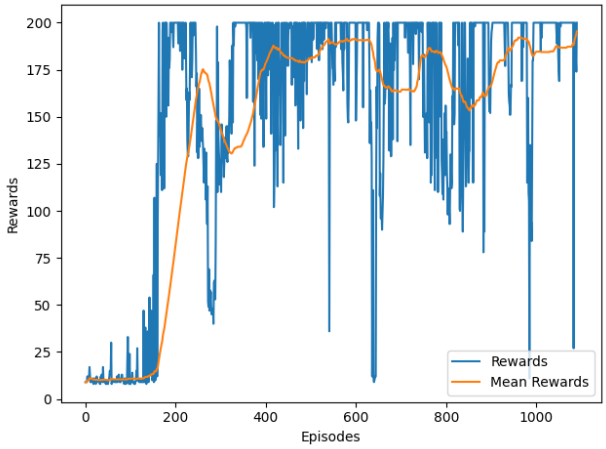
Replay Buffer For replay buffer with a certain size, we use the python object `deque` for dealing with incoming samples. That is, when the memory bank is full, we throw away earlier samples so that new samples can be stored. To perform experience replay, we define `append` for adding new samples and `sample_batch` for randomly selecting a batch of samples to update the network. Moreover, in practice, we require that the replay begins only after certain amount of samples are stored so that we have sufficient data for training. For this purpose, we define a function `ready` which returns `True`, once the number of stored samples is above the threshold.

Three Modes in Training We have three different modes where the agent adopts different action selection schemes. First, we define `explore_mode` for doing exploration with randomly selected actions. This mode is used to generate enough samples to fill the replay buffer and is implemented in the pre-training stage. Once the buffer is ready, we then begin deep Q-learning, see algorithm 1, and in this stage, we use `train_mode`, where actions are derived from the $\epsilon-$greedy policy. When the problem is solved, i.e., the mean reward over 100 consecutive episodes is greater than 195[1], we turn to `display_mode`, showing that the policy based on the learned Q function can successfully balance the pole. In this mode, greedy policy is adopted and there is no ϵ randomness.

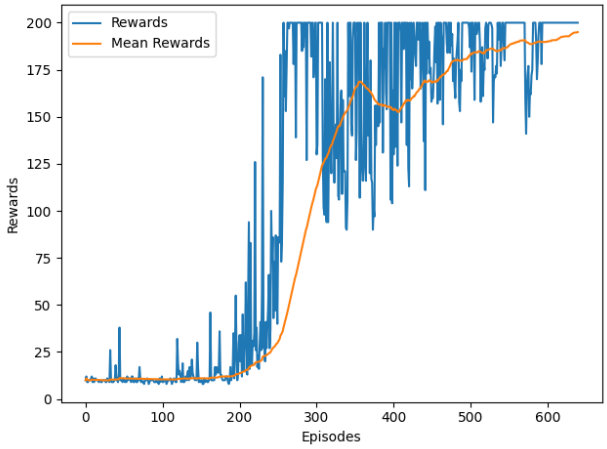
3 Experiment Results

As the algorithm involves randomness, we launch the experiment several times and we present two remarkable experiments for illustrating the difference between vanilla deep Q-learning (DQN) and double Q-learning (DDQN). As we can see from fig. 2, both can solve the cart-pole problem, with final mean rewards greater than 195. However, we note that DDQN is more stable than DQN in the following aspects.

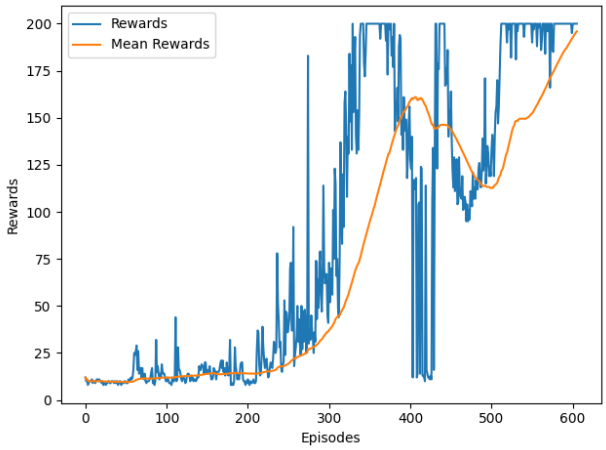
[1]Another criterion is to require that for 90 out of 100 consecutive episodes, the episodic rewards is above 195 and in practice, we find that this is more demanding than simply looking at the mean.



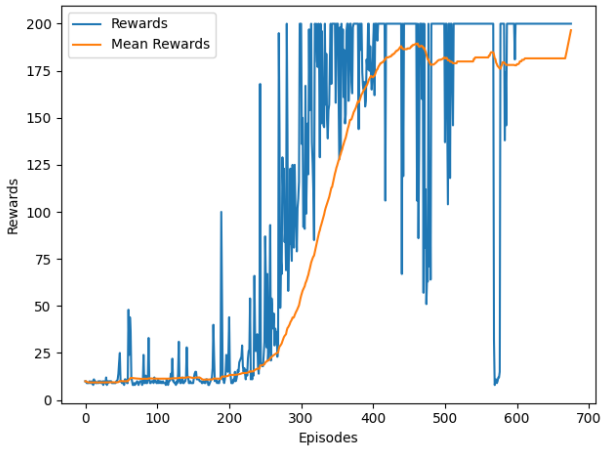
(a) Experiment 1: DQN



(b) Experiment 1: DDQN



(c) Experiment 2: DQN



(d) Experiment 2: DDQN

Figure 2: Two experiments about DQN and DDQN:

- The number of episodes: It generally take around 700 episodes for DDQN to finish the task, apart from the exploration episodes in pre-train stage, whereas DQN may take more than 1000 episodes, see fig. 2(a) and the number of required episodes varies from case to case.
- Mean rewards: As we can tell from figs. 2(b) and 2(d), on average, the mean rewards for DDQN increase, as more episodes are completed, whereas for DQN, chances are that scores may drop (see fig. 2(a)) and may drop significantly (see fig. 2(c)) during the learning process.

Our numerical results show that leveraging target networks can resolve the overestimation of Q-learning, leading to a more stale learning process.

References

Andrew G Barto, Richard S Sutton, and Charles W Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and*

Cybernetics, SMC-13(5):834–846, 1983.

Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

Hado van Hasselt, Arthur Guez, and David Silver. Deep Reinforcement Learning with Double Q-Learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, pages 2094–2100, Phoenix, Arizona, 2016. AAAI Press.

Hado V. Hasselt. Double Q-learning. In *Advances in Neural Information Processing Systems 23*, Advances in Neural Information Processing Systems, pages 2613—2621. Curran Associates, Inc., 2010.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

Vinod Nair and Geoffrey E. Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proceedings of the 27th International Conference on Machine Learning*, ICML10, page 807814, Madison, WI, USA, 2010. Omnipress.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press. MIT press, Cambridge, MA, 2018.

Philip S Thomas and Billy Okal. A Notation for Markov Decision Processes. *arXiv*, 2015.

Sebastian Thrun and Anton Schwartz. Issues in using function approximation for reinforcement learning. In *Proceedings of the 1993 Connectionist Models Summer School Hillsdale, NJ. Lawrence Erlbaum*, 1993.

Christopher J.C.H Watkins and Peter Dayan. Q-Learning. *Machine Learning*, 8(3-4):279 292, 1992.