Report II for MVIG Training

Yujing Shen
Shanghai Jiao Tong University
Shanghai, China
shenyujing@sjtu.edu.cn

Abstract

The recent DQN alogrithm is known for its brilliant performance in playing the Atari 2600 games. This report introduces DQN from the very basic concepts of reinforcement learning to the cutting-edge researches corresponded to it. The last part provides the result of a simple expriment on training a robot to play fappy bird and has several discussion on the result.

1. Basic Concept

Many contents in this section comes from Reinforcement Learning: An Introduction (Sutton and Barto[3]) and open course by David Silver of UCL.

1.1. MDP

Markov Decision Process or MDP is a memoryless stochastic process that the next state only depends on the present state and does not depend on the history.

$$p(s', r|s, a) = P\{S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a\} \tag{1}$$

Figure 1 shows the interaction between the agent and the environment.

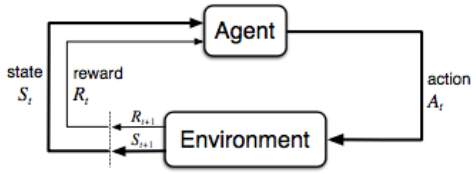

Figure 1. The agentenvironment interaction in reinforcement learning.

State-value function $v_\pi(s)$is the function of states (or state-action pairs) that estimate it is how good for an agent to be in a given state under policy π. Similarly, action-value function $v_\pi(s, a)$ evaluate taking action a in state s under policy π

State value function:

$$v_\pi(s) = E_\pi[G_t|S_t = s] \tag{2}$$

$$= E_\pi[\sum_{k=0}^{\infty} \gamma^k G_t|S_t = s] \tag{3}$$

Policy value function:

$$q_\pi(s, a) = E_\pi[G_t|S_t = s, A_t = a] \tag{4}$$

$$= E_\pi[\sum_{k=0}^{\infty} \gamma^k G_t|S_t = s, A_t = a] \tag{5}$$

1.2. Bellman Equation

Bellman functino is widly used in reiforcement leaning to get the greedy result of state-function and action function. For state-value function, it expresses a relationship between the value of a state and the values of its successor states. For policy value function, it almost the same.

Bellman equation for state-function:

$$v_*(s) = \max_a R_s^a + \gamma \sum_{ss'}^{a} v_*(s') \tag{6}$$

Bellman equation for action-function:

$$q_*(s) = R_s^a + \gamma \sum_{ss'}^{a} v_*(s') \tag{7}$$

1.3. DP in solving MDP

The Bellman equation can be expressed in the dynamic programming form. The basic reuirement for dynamic programming method or DP is that the environment has only finite states. DP is used to expolre the future states and evaluates the state or action.

Given below is one of the DP algorithms in solving MDP called policy iteration.

Algorithm 1 Policy iteration (using iterative policy evaluation)

Input: Random initial value function $V(s) \in R$; Random initial policy function $\pi(s) \in A(s)$; All states $s \in S$

Output: Stable value function $V \approx v_*$; Stable policy function $\pi \approx \pi_*$

Policy Evaluation

repeat
  $\Delta \leftarrow 0$
  for each $s \in S$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))[r + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
  end for
until $\Delta < \epsilon$

Policy Improvement

$policy - stable \leftarrow true$
for $s \in S$ **do**
  $old - action \leftarrow \pi(s)$
  $\pi(s) \leftarrow argmax_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
  if $old - action \neq \pi(s)$ **then**
    $policy - stable \leftarrow false$
  end if
end for
if policy-stable **then**
  Return $V \approx v_*$ and $\pi \approx \pi_*$
else
  Go to Policy Evaluation
end if

1.4. General Policy Iteration

Generalized policy iteration (GPI) refers to the general idea of letting policy evaluation and policy improvement processes interact, independent of the granularity and other details of the two processes. The evaluation and improvement processes in GPI can be viewed as both competing and cooperating.

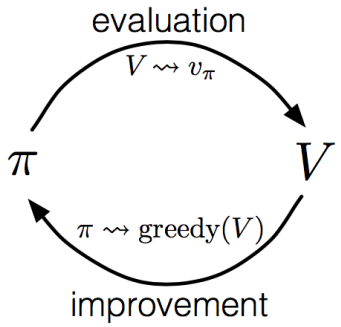

Figure 2. General policy iteration.

1.5. Model-free methods

Model-free methods require no prior knowledge of the environments dynamics and are used to train the system only based on the experience or simulation. Given below are some of the methods.

Monto Carlo (MC) is kind of like DFS. It estimate the value or policy function by doing massive experiments and getting experiences. In each experiment, MC always visits the terminal states to get G_t, one of the experiences. One good method to update the estimated value follows the equation below.

$$V(S_t) \leftarrow V(S_t) + \frac{1}{N(S_t)}(G_t - V(S-t)) \tag{8}$$

where $N(S_t)$ is the number of experiments. Moreover, this equation can be write as

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S-t)) \tag{9}$$

where α can determine how forgetful the system is.

Temporal Difference(TD) is kind of like BFS. It usually cost less time than MC. TD only explores some of S_{t+1} states usually updates estimation by following

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \tag{10}$$

where

$$R_{t+1} + \gamma V(S_{t+1}) \tag{11}$$

is called TD target and

$$\sigma_t = R_{t+1} + \gamma V(S_{t+1} - V(S_t)) \tag{12}$$

is called TD error.

N-step TD extends TD. TD can be seen as 1-step TD and n-step TD follows

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} \tag{13}$$

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t^{(n)} - V(S_t)) \tag{14}$$

When n approches infinity ($n \rightarrow \infty$), n-step TD turns to MC.

TD(λ) combines all n-step TD and can be written as

$$G_t^\lambda = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^(n) \tag{15}$$

$$V(S_t) \leftarrow (G_t^\lambda - V(S_t)) \tag{16}$$

Sarsa is short for S-A-R-S'-A' is a popular Q-learning method. It can be expressed as

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)) \tag{17}$$

1.6. Function Approximation Methods

Sometimes, the problem has an arbitrary large states space, but optimal policy or the optimal value function are in the limit of infinite time and data. Therefore, finding a good approximate solution using limited computational resources is a good idea. There are linear and unlinear methods to approximate value function.

Linear mothods usually follows the equation

$$\hat{v}(s,\theta) = \theta^{\intercal}\phi(s) = \sum_{i=1} n\theta_i\phi_i(s) \tag{18}$$

where θ are the weights to be trained and ϕ are the features of current state. The training methods are plentiful and semi-gradiant TD(0) can be one of them.

$$\theta_{t+1} = \theta_t + \alpha(R_{t+1}\phi_t - \phi_t(\phi_t - \gamma\phi_{t+1})^{\intercal}\theta_t) \tag{19}$$

The typical unlinear method is to use neuron network and one popular method is called DQN which will be illustrated in the later section.

1.7. Eligibility Traces

Eligibility traces, one of the basic mechanisms of reinforcement learning, unify and generalize TD and MC methods. For example, in the popular TD(λ) algorithm, the λ refers to the use of an eligibility trace. The eligibility trace vector is defined as

$$e_0 = 0 \tag{20}$$
$$e_t = \nabla\hat{v}(S_t,\theta_t) + \gamma\lambda e_{t-1} \tag{21}$$

and TD error for state-value prediction is

$$\delta_t = R_{t+1} + \gamma\hat{v}(S_{t+1},\theta_t) - \hat{v}(S_t,\theta_t) \tag{22}$$

Then the weight vecter is updated on each step propotional to scalar TD error and the vector eligibility trace:

$$\theta_{t+1} = \theta_t + \alpha\delta_t e_t \tag{23}$$

1.8. Categories on RL Methods

The table given below illustrates the features of value-based, policy-based and actor-cretic methods.

As Table 1 shows, value-based RL methods only need to learn the value function; policy-based methods need to learn the policy; actor-cretic methods need to learn both.

Requirement	value-based	policy-based	actor-cretic
learn value	T	F	T
learn policy	F	T	T

Table 1. Features of the three categories. T or F refer to this method has or does not have the requirement respectively

2. DQN

Deep Q-learing network or DQN (Mnih *et al.* [1]) is an end-to-end methods to approximate the Q function. It usually obeys the following equations.

$$a_t = argmax_a Q^*(\phi(s_t), a; \theta) \tag{24}$$

where a_t is the action the agent would like take, $\phi(s_t)$ is the features that feed in the network and θ are the parameters in the network.

Training DQN firstly requires memory D to store the experience. In every training episode, the first step is to sample random minibatch of transitions $(\phi_j, a_j, r)j, \phi_{j+1}$ from D. Then calculating the policy function of ϕ_{j+1}

$$y_j = \begin{cases} r_j & \text{for terminal}\phi_{j+1} \\ r_j + \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal}\phi_{j+1} \end{cases} \tag{25}$$

At last, getting loss by using $Q(\phi_j, a_j; \theta)$ as predictions and y_j as labels.

The training process ensures DQN has the ablility to see further than TD and taking last 4 scenes as the state provide DQN the ability to sense the agent locomotion. However, there are still some room for improvement. Take the process of sampling minibatch as an example. The probabilites for choosing these experiences may not be the same. Some good or terrible experience should be used more often.

The biggest puzzle for me is that how to distinguish whether the network is overfiting or not and could the result be better if the network is overfitting. It is quite difficult for me to get the answers.

3. DQN Improvements

Recently, there are plenty of improvements on DQN. I have read some of them and find double q-learning, duel network and prioritized memory replay are easier to understand.

Double Q-learning or Double DQN (Hasselt *et al.* [4]) use 2 same network to approximate the policy function. The idea, as the paper says, is to adjust the overestimated policy value in DQN. The algorithm trains network by following

$$y_j = r_j + \max_a(Q_1(\phi_{j+1}, argmax_{a'} Q_2(\phi_{j+1}, a'; \theta_2); \theta_1)) \tag{26}$$

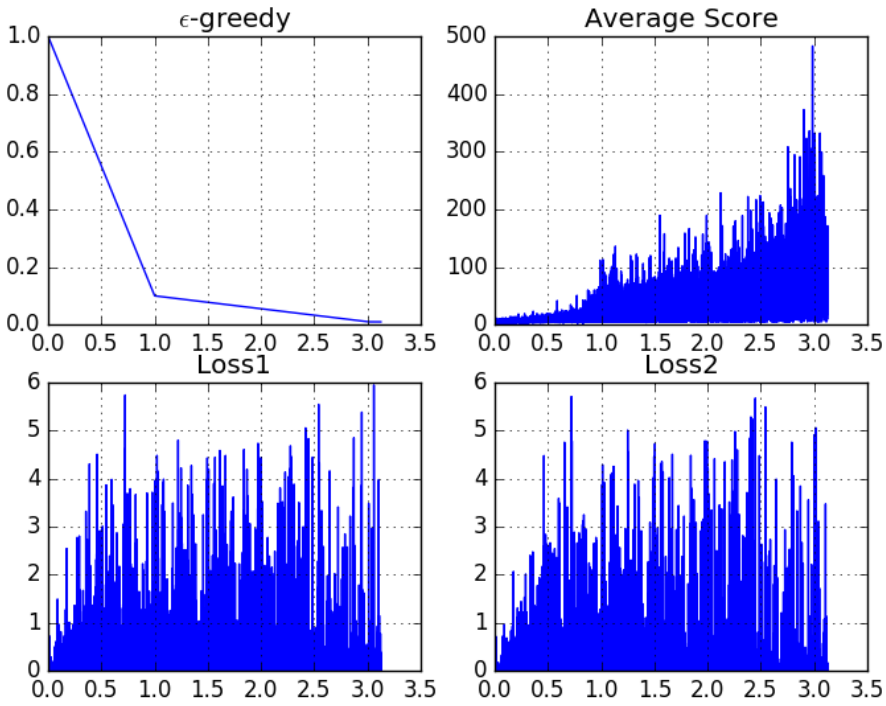

Figure 3. Empirical result

to get y_j when s_j is not terminal.

Prioritized memory replay (Schaul *et al*. [2]) is built on the DDQN. Its key idea is to increase the replay probability of experiences that have a high expected learning progress. The paper use TD error to measure it.

Duel network (Wang *et al*. [5])changes the network architecture. It split DQN's last 2 fully connected layers into 2 stream and add a new layer to combine the 2 seperate stream. This paper uses end-to-end method to train the network.

4. Experiment

In the experiment, I apply Double DQN to obtain better performance in playing flappy bird. Changes are also made in the network arthitecture to obtain nonlinearity. Besides, for the sake of better exploitation, I set the probability distribution of random actions as 0.9 and 0.1 or the model may easily fall into the local extremum.

4.1. Empirical Result

Figure 3 shows the empirical results of the training process. To explore the performance of the robot, please download the project on github and run code on demo mode.

4.2. Discussion

I have made several mistakes during the experiment. In the first place, I applied $y_j = r_j + \max_{a'} Q(\phi_j, a; \theta)$ in the training process which should have been $y_j = r_j + \max_{a'} Q(\phi_{j+1}, a; \theta)$. The first formula does not have ability to approximate the policy function and the future effects cannot propagate back correctly. Secondly, I forgot reset S_t when the game ends. That makes the game start with different ϕ_t with correlations to the last final state. Thirdly, the network includes relu layers, but I initialized convolution mask weights with gaussian distribution. In this way, some parts of the network may die at the begining of the training even with bias terms.

Based on observation, I find the Double DQN does have better performance than DQN and the nonlinearity of the network plays a key role in improving the score. Next time, I would like to try duel network, prioritized experience replay and other latest techniques on Atari games.

There are several difficult parts in training a DQN. First, if the conditions are strict, the model may have a cold start, i.e. the model may have little chance to get its first score. Besides, good experiences may be difficult to get and have low probability to be fed in the training process. Second, due to the discreteness of the rewards gaining, the agents

prefer random actions in no-reward areas. Good policy and bad policy are difficult to tell and the model may easily fall into local extremum. I am thinking about using some techniques like adversarial network to better the agents performance in these no-reward area.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013. 3

[2] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015. 4

[3] R. S. Sutton and A. G. Barto. Reinforcement learning: An introduction, 2011. 1

[4] H. Van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *AAAI*, pages 2094–2100, 2016. 3

[5] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015. 4