Playing Atari Games with Double DQN and Rainbow DQN

Linsong Guo
ACM Honors Class, Shanghai Jiao Tong University
gls1196@sjtu.edu.cn

Abstract

This report summarizes some extensions to DQN [Mnih *et al.*, 2013] and introduces my process of playing Atari games with two agents: Double DQN agent and Rainbow DQN agent. I evaluate the two agents with six Atari games and compare the performances of Double DQN [van Hasselt *et al.*, 2016] and Rainbow DQN [Hessel *et al.*, 2018].

1 Introduction

Deep Q-Network [Mnih *et al.*, 2013] is a deep reinforcement learning method that combines Q-learning with convolutional neural networks, which makes it possible to play Atari games at human level performance. However, the training of DQN is so slow that it may takes over ten days or even more to play in hard games. In recent years, the deep reinforcement learning community has proposed some independent extensions to DQN to improve its training speed and stability. [Hessel *et al.*, 2018] show that these extensions are indeed largely complementary and combine them into a integrated agent. In the course project, I implemented the integrated agent and compared its performance with Double DQN agent.

2 Background

2.1 Environments and Agents

In the project, agent interacts with the Atari game environment $\mathcal{E}$. At each time step $t = 0, 1, 2, \cdots$, the environment provides the agent with an image x_t representing the current screen, the agent selects an action a_t from the set of actions $\mathcal{A} = \{0, 1, \cdots, N_{\text{actions}} - 1\}$ and then receives a reward r_t representing the change of game score. Since it's impossible to fully understand the current situation from only the current screen x_t, we consider sequences of observations and actions $s_t = (x_1, a_1, \cdots, x_t, a_t)$, and we will learn game strategies that depend upon the sequences.

We define the future rewards $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$, where T is the terminal time step and γ is the discount factor. Under a policy π, the *state-action value function* is $Q^\pi(s, a) = \mathbb{E}[R_t | s_t = s, a_t = t, \pi]$, which is also called Q-function. Then the *value function* is $V^\pi = \mathbb{E}_{a\sim\pi(s)}[Q^\pi(s, a)]$ and optimal state-action function is $Q^*(s, a) = \max_\pi Q^\pi(s, a)$. Note that the optimal action-value function obeys *Bellman equation*, so we have

$$Q^*(s, a) = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s', a') \middle| s, a\right]$$

Using Bellman equation as an iterative update, we have

$$Q_{i+1}(s, a) = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q_i(s', a') \middle| s, a\right]$$

When $i \to \infty$, Q_i converges to Q^* [Sutton and Barto, 1998].

2.2 DQN

DQN [Mnih *et al.*, 2013] is a multi-layered neural network that for a given state s outputs a vector of action values $(Q(s, 0; \theta), Q(s, 0; \theta), \cdots, Q(s, K-1; \theta))^T$, where θ is the parameter of the network. DQN utilizes a technique known as *experience replay*: we store the agent's transition (s_t, a_t, r_t, s_{t+1}) at each time step t in a data set D, which is called *replay memory* [Lin, 1993]. Since $s_t = (x_1, a_1, \cdots, x_t, a_t)$ can be arbitrary length, it's difficult to store it into the replay memory. Thus we store $\phi(s_t)$ into the replay memory, where ϕ is a preprocessing mapping from s_t to the fixed length representation of s_t. At each update iteration i, the network is trained by sampling a mini-batch of transitions from D uniformly at random. Note the mini-batch is the form of $\{(\phi(s), a, r, \phi(s'))\}$, but for convenience we still describe it as the form of $\{(s, a, r, s')\}$ from here on. Then the loss function at update iteration i takes the from

$$L_i^{\text{DQN}}(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{U}(D)}\left[(y_i^{\text{DQN}} - Q(s, a; \theta_i))^2\right] \quad (1)$$

where $y_i^{\text{DQN}} = r + \gamma \max_{a'} Q(s', a'; \theta_{i-1})$ is the target value and θ_i is the parameter of the network at iteration i [Mnih *et al.*, 2013].

3 Ingredients of Rainbow DQN

There are some extensions to DQN, such as Double DQN [van Hasselt *et al.*, 2016], Dueling DQN [Wang *et al.*, 2015], Distributional DQN [Bellemare *et al.*, 2017], Prioritized Replay [Schaul *et al.*, 2016] and Noisy Nets [Fortunato *et al.*, 2018]. Based on the original DQN, each of above ingredients enables performance improvements in isolation. The key idea of Rainbow DQN [Hessel *et al.*, 2018] is to combine all the above ingredients.

3.1 Double DQN

In the equation 1, we use the same Q functions to calculate the target value $r + \gamma \max_{a'} Q(s^{'}, a^{'}; \theta_{i-1})$ and the current value $Q(s, a; \theta_i)$, which leads to the unstability of the update. Double DQN [van Hasselt *et al.*, 2016] use a separate network Q^- named *target network* to improve the stability. Every C updates we clone the network Q to obtain the target network Q^- and use Q^- to calculate target values in the following C updates to Q. Hence the loss function at update iteration i becomes

$$L_i^{\text{DDQN}}(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{U}(D)}\left[(y_i^{\text{DDQN}} - Q(s, a; \theta_i))^2\right]$$

where $y_i^{\text{DDQN}} = r+\gamma \max_{a'} Q^-(s^{'}, a^{'}; \theta_i^-)$ is the target value and θ_i^- is the parameter of the target network Q^- at iteration i [van Hasselt *et al.*, 2016].

3.2 Dueling DQN

Dueling DQN [Wang *et al.*, 2015] introduces *advantage function* which is related to Q function and value function:

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$$

Note that the definition $V^\pi = \mathbb{E}_{a\sim\pi(s)}[Q^\pi(s, a)]$, so we have

$$\mathbb{E}_{a\sim\pi(s)}[A^\pi(s, a)] = 0$$

In fact, the advantage function has the meaning of a relative measure of the importance of each action by subtracting the value function from Q function.

Dueling DQN consists two streams that represent value and advantage functions sharing a common convolutional layers. The value and advantage functions are combined into Q-function in the following form

$$Q(s, a; \theta) = V(s; \eta, \beta) + A(s, a; \eta, \alpha)$$

where η,β and α are the parameters of common layers, value layers and advantage layers, respectively and $\theta = \{\eta, \alpha, \beta\}$ is the parameter of the whole network. However, the real Q function we used in Deuling DQN takes the form

$$\begin{aligned} Q(s, a; \theta) &= V(s; \eta, \beta) \\ &+ \left(A(s, a; \eta, \alpha) - \frac{1}{N_{\text{actions}}} \sum_{a'} A(s, a^{'}; \eta, \alpha)\right) \end{aligned}$$

Note that subtracting the mean value doesn't change the relative rank of advantage values but narrows the range of Q function, so increases the stability of the optimization.

To my understanding, the efficiency of Dueling DQN is related to the following reasons. In DQN or double DQN, if we update $Q(s, a)$, the optimization is mainly for action a. However, in Dueling DQN, $Q(s, a)$ involves not only $A(s, a)$ but also $V(s)$. When updating $V(s)$, we also update the Q function for other actions instead of only the action a, which improves the efficiency of training.

3.3 Distributional Reinforcement Learning

In Distributional RL [Bellemare *et al.*, 2017], *return* $Z^\pi(s, a)$ is defined as a random variable of the total rewards obtained by starting from state s, taking action a and then following the policy π. In fact, the Q function $Q^\pi(s, a)$ is the expectation of return:

$$Q^\pi(s, a) = \mathbb{E}[Z^\pi(s, a)]$$

Similar to Bellman equation, the *distributional Bellman equation* takes the following form:

$$Z^*(s, a) = r + \gamma Z^*(s^{'}, a^*)$$

where r is the reward of taking action a, $s^{'}$ is the next state and $a^* = \arg\max_{a^{'}} Q^*(s^{'}, a^{'})$.

The main idea of Distributional RL is just to work directly with the distribution of return instead of the expectation of return. We model distribution of return as $d = (\boldsymbol{z}, p)$ with a discrete distribution function p whose support $\boldsymbol{z}$ is the set of atoms $\{z_i = V_{\text{MIN}} + i\triangle z : 0 \le i < N_{\text{atoms}}\}$, where $N_{\text{atoms}} \in \mathbb{N}$, V_{MIN},$V_{\text{MAX}} \in \mathbb{R}$ and $\triangle z = \frac{V_{\text{MAX}} - V_{\text{MIN}}}{N_{\text{atoms}} - 1}$. Since $z_0, z_1, \cdots, z_{N_{\text{atoms}}-1}$ are fixed, the network we designed just need to provide a probability vector $(p_0, p_1, \cdots, p_{N_{\text{atoms}}-1})$ for each (s, a). Thus the network takes s as input and outputs a probability matrix of size $N_{\text{actions}} \times N_{\text{atoms}}$. Thus we have

$$Q^\pi(s, a) = \mathbb{E}[Z^\pi(s, a)] = \sum_{i=0}^{N_{\text{atoms}}-1} p_i z_i$$

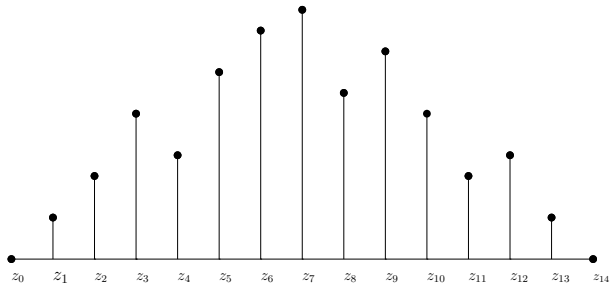

Figure 1: Distribution of return when $N_{\text{atoms}} = 15$
The height of z_i means p_i.

When updating the network, we also simple a mini-batch $\{(s, a, r, s^{'})\}$ from replay memory as we do in DQN or double DQN. The loss function at update iteration i is the distance between the two distributions: $d = (\boldsymbol{z}, p_{(s,a;\theta_i)})$ and $d^{'} = (r + \gamma\boldsymbol{z}, p_{(s',a^*;\theta_i')})$, where θ_i and $\theta_i^{'}$ are the parameter of network and target network respectively. The 'distance' here means the discrete version of *KL divergence*:

$$\text{KL}(p||q) = \sum_{i=0}^{N-1} p(z_i) \log \frac{p(z_i)}{q(z_i)}$$

where p and q are two probability distribution functions.

However, using a discrete distribution poses a problem: considering $r + \gamma\boldsymbol{z}$, the transformed atoms $r + \gamma z_0, \cdots, r + \gamma z_{N_{\text{atoms}}-1}$ aren't in the standard positions $z_0, \cdots, z_{N_{\text{atoms}}-1}$.

The solution is to construct a projection $\Phi_{\boldsymbol{z}}$, which splits each atom $r + \gamma z_j$ into the two neighbouring standard atoms and distribute its probability $p_{(s',a^*;\theta'_i)}(z_j)$ to the two neighbors. For example, in figure 2, the neighbors of $r + \gamma z_j$ are z_1 and z_2. Suppose the distance to z_1 is three times that to z_2, it makes sense that z_1 receives $\frac{1}{4}p_{(s',a^*;\theta'_i)}(z_j)$ and z_2 receives $\frac{3}{4}p_{(s',a^*;\theta'_i)}(z_j)$.

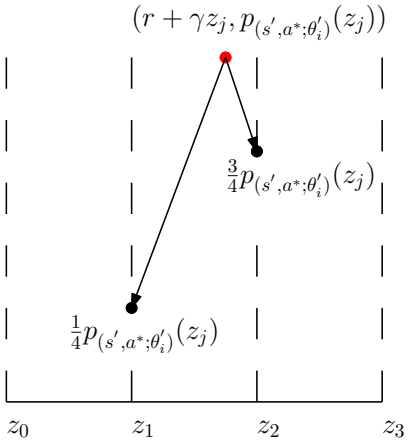

Figure 2: An example of what $\Phi_{\boldsymbol{z}}$ do

Suppose $\Phi_{\boldsymbol{z}}d^{'} = (\boldsymbol{z}, m)$, note that the supports of $\Phi_{\boldsymbol{z}}d^{'}$ and d are both $\boldsymbol{z}$ now. The loss function at update iteration i takes the form

$$\begin{aligned} & L_i(\theta_i) \\ = \; & \mathbb{E}_{(s,a,r,s')\sim\mathcal{U}(D)}\left[\mathrm{KL}(\Phi_{\boldsymbol{z}}d^{'}||d)\right] \\ = \; & \mathbb{E}_{(s,a,r,s')\sim\mathcal{U}(D)}\left[\sum_{j=0}^{N-1} m(z_j)\log\frac{m(z_j)}{p_{(s,a;\theta_i)}(z_j)}\right] \end{aligned}$$

3.4 Prioritized Replay

DQN samples uniformly from the replay memory. However, transitions may be more or less surprising, redundant or task-relevant [Schaul *et al.*, 2016]. Some transitions may not be immediately useful to the agent, but might become so when agent competence increases. Thus we want to sample more frequently those transitions which there are much to learn. The key idea of *prioritized replay* is to use a TD-error δ to measure the importance of a transition $(s, a, r, s^{'})$:

$$\delta = r + \gamma \max_{a^{'}} Q(s^{'}, a^{'}; \theta^{-}) - Q(s, a; \theta)$$

There are two choices needed to design: which transitions to store and which transitions to replay. For the first choice, new transitions are inserted into the replay memory with maximum priority, providing a bias towards recent transitions. For the second choice, the probability of sampling transition i is

$$P(i) = \frac{p_i^{\alpha}}{\sum_k p_k^{\alpha}}$$

where $p_i = |\delta_i| + \epsilon$ is the priority of transition i, ϵ is a small hyper-parameter that prevents the transition not being revisited once δ_i is zero and α determines how much prioritization is used [Schaul *et al.*, 2016].

3.5 Noisy Nets

In Reinforcement Learning, most exploration methods rely on random perturbations of the agent's policy, such as ϵ-greedy we used in DQN. However, the random perturbation may not lead to efficient exploration. Noisy Nets are neural networks whose weights and biases are perturbed by a parametric function [Fortunato *et al.*, 2018]. More precisely, consider a linear layer of a neural network with n inputs and m outputs:

$$\boldsymbol{y} = \boldsymbol{w}\boldsymbol{x} + \boldsymbol{b}$$

where $\boldsymbol{x} \in \mathbb{R}^n$ is the input layer, $\boldsymbol{w} \in \mathbb{R}^{m\times n}$ is the weight and $\boldsymbol{b} \in \mathbb{R}^m$ is the bias. Then the corresponding noisy linear layer is defined as:

$$\boldsymbol{y} = (\boldsymbol{\mu}^{\boldsymbol{w}} + \boldsymbol{\sigma}^{\boldsymbol{w}} \odot \boldsymbol{\epsilon}^{\boldsymbol{w}})\,\boldsymbol{x} + \boldsymbol{\mu}^{\boldsymbol{b}} + \boldsymbol{\sigma}^{\boldsymbol{b}} \odot \boldsymbol{\epsilon}^{\boldsymbol{b}}$$

where $\boldsymbol{\mu}^{\boldsymbol{w}} + \boldsymbol{\sigma}^{\boldsymbol{w}} \odot \boldsymbol{\epsilon}^{\boldsymbol{w}}$ and $\boldsymbol{\mu}^{\boldsymbol{b}} + \boldsymbol{\sigma}^{\boldsymbol{b}} \odot \boldsymbol{\epsilon}^{\boldsymbol{b}}$ replace $\boldsymbol{w}$ and $\boldsymbol{b}$ respectively in the original linear layer. Note that $\boldsymbol{\mu}^{\boldsymbol{w}}$, $\boldsymbol{\mu}^{\boldsymbol{b}}$, $\boldsymbol{\sigma}^{\boldsymbol{w}}$ and $\boldsymbol{\sigma}^{\boldsymbol{b}}$ are learnable but $\boldsymbol{\epsilon}^{\boldsymbol{w}}$ and $\boldsymbol{\epsilon}^{\boldsymbol{b}}$ are noise random variables [Fortunato *et al.*, 2018].

4 The Integrated Rainbow Agent

4.1 Network Architecture

We use the same architecture for Atari games we have trained. The input to the neural network is an $4 \times 84 \times 84$ image matrix produced by ϕ. In other words, the preprocessing map ϕ produces the last four frames of the history. The first hidden layer convolves 32 filters of 8×8 with stride 4 [Mnih *et al.*, 2015]. The second hidden layer convolves 64 filters of 4×4 with stride 2. The third hidden layer convolves 64 filters of 3×3 with stride 1. The three hidden layers are all followed by a rectifier nonlinearity. Then the stream is divided into 2 parts: value stream and advantage stream. The value stream is then fed into two connected *Factorised Gaussian noise* layers [Fortunato *et al.*, 2018] with N_{atoms} outputs in the form of $v_i(s)(0 \le i < N_{\text{atoms}})$. Similarly, the advantage stream is then fed into two connected Factorised Gaussian noise layers with $N_{\text{atoms}} \times N_{\text{actions}}$ output in the form of $a_i(s, a)(0 \le i < N_{\text{atoms}})$. For each atom z_i, the value and advantage outputs are aggregated and then passed through a *softmax* layer:

$$p_i(s, a) = \frac{\exp\left(v_i(s) + a_i(s, a) - \bar{a}_i(s)\right)}{\sum_{j=0}^{N_{\text{atoms}}-1}\exp\left(v_j(s) + a_j(s, a) - \bar{a}_j(s)\right)}$$

where $\bar{a}_i(s) = \frac{1}{N_{\text{actions}}}\sum_{a'\in\mathcal{A}} a_i(s, a^{'})$ is the mean advantage [Hessel *et al.*, 2018].

4.2 Changes to Prioritized Replay

In standard proportional prioritized replay [Schaul *et al.*, 2016], the priority of a transition $(s, a, r, s^{'})$ is

$$\begin{aligned} p = \; & |\delta| + \epsilon \\ = \; & \left|r + \gamma \max_{a'} Q(s^{'}, a^{'}; \theta^{-}) - Q(s, a; \theta)\right| + \epsilon \end{aligned}$$

However, Rainbow agent uses KL divergence as loss function [Hessel *et al.*, 2018], so the priority is

$$p = \left|\mathrm{KL}(\Phi_{\boldsymbol{z}}d^{'}||d)\right| + \epsilon$$

where d,$d^{'}$ and $\Phi_{\boldsymbol{z}}$ are symbols we used in section 3.3.

5 Experiments

5.1 Hyper-parameters

In the Rainbow DQN agent I implemented, I use the Adam optimizer [Kingma and Ba, 2017] of a learning rate $\eta = 6.25 \times 10^{-5}$ [Hessel *et al.*, 2018]. We sample a mini-batch with size 32 from the replay memory to update the network Q at each update iteration and clone Q to obtain the target network Q^- for every 1000 update iterations. For prioritized replay, we use the proportional prioritization as described in section 3.4, with priority exponent $\alpha = 0.5$, and linearly increased the importance sampling exponent β from 0.4 to 1 [Schaul *et al.*, 2016]. For explorations, we replace the linear layers with Factorised Gaussian noise layers rather than using ϵ-greedy, with $\sigma_0 = 0.5$ used to initialize the weights of the layer [Fortunato *et al.*, 2018]. The hyper-parameters are the same across all games we trained.

Hyper-parameter	value
Learning rate η	6.25×10^{-5}
Discount factor γ	0.99
Memory size	30K
Mini-batch size	32
Target network update frequency	1000
N_{atoms}	51
V_{MIN}	-10
V_{MAX}	10
Noisy Nets σ_0	0.5
Priority exponent α	0.6
Prioritization Importance sampling β	$0.4 \rightarrow 1$

Table 1: Hyper-parameters table

5.2 Performance

In the course, agents are evaluated in six Atari games: Pong, Freeway, Krull, Tutankham, Atlantis and Beam Rider. Firstly I implemented a Double DQN agent but found the training was very slow. So I implemented a Rainbow DQN agent later and found it work very well in Pong and Freeway. To compare the performance in the game Pong, I tried to found the optimal performances for both agents by tuning hyper-parameters, though the performance almost unchanged with the changing hyper-parameters in the limited range. In the following curve generated by Tensorboard, Rainbow DQN agent convergences much faster than double DQN agent.

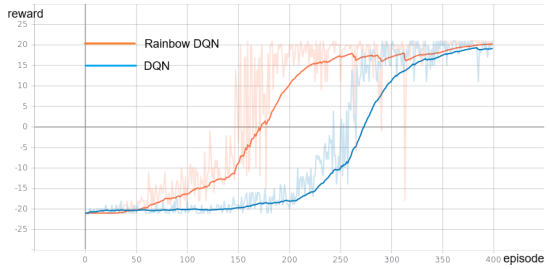

Figure 3: Rainbow DQN convergences fatser than DQN in Pong.

However, although my Rainbow DQN agent worked very well in easier games like Pong and Freeway, it had poor performances in harder games like Krull and Tutankham. For example, when I trained my Rainbow DQN agent in Tutankham, the reward increased very fast in the beginning but almost stopped increasing when the average reward reached to about 70. I struggled about several weeks and tried to find a good hyper-parameter setting but failed. So I had to give it up before the deadline and trained my Double DQN agent again. Although the training of Double DQN agent was slow, at least it worked in most evaluated games. But it was so late that I didn't have enough time to train Beam Rider and Atlantis, which may take about one week to train. The following table shows the evaluation results of Double DQN agent.

Game	result
Pong	20.8
Freeway	31.6
Tutankham	236.7
Krull	9532.8

Table 2: The Evaluation results of Double DQN agent

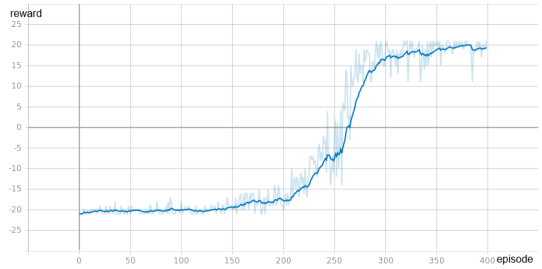

Figure 4: Reward curve of Double DQN in Pong.

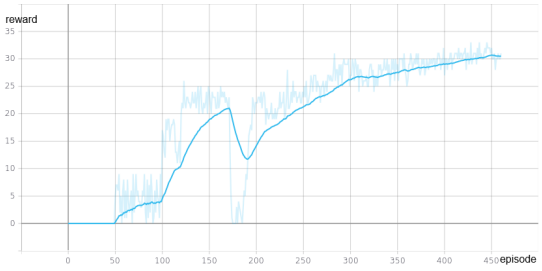

Figure 5: Reward curve of Double DQN in Freeway.

6 Conclusion

We summarized some extensions to DQN and evaluated the performance of Double DQN agent and Rainbow DQN agent I implemented. Although Rainbow DQN is more efficient than DQN, it has more complex structures and hyper-parameters, which may be a little difficult for the beginners to tune. The following are my personal suggestions about the course project.

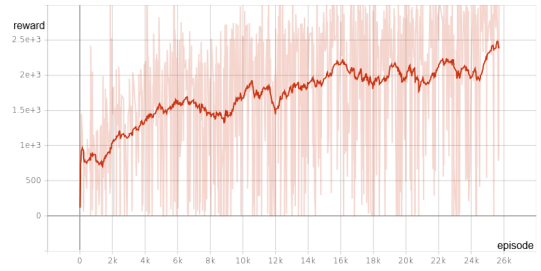

Figure 6: Reward curve of Double DQN in Krull.

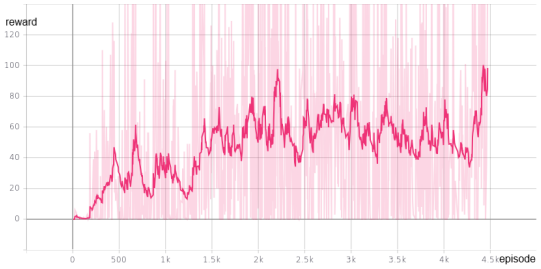

Figure 7: Reward curve of Double DQN in Tutankham.

- It's normal to have non-increasing rewards in the first several hundreds or even thousands of episodes in harder games. Don't mistakenly believe that training has no effect. The only thing to do is to calm down and wait for the flowers to bloom.
- Better model doesn't equal better performance. Tuning parameters of a better model is always more difficult.
- Sometimes poor performance is not because of bad parameters but bugs in your code. However, finding bugs in DQN is difficult because the agent is like a black box. A good way is to observe changes of network and loss function by using some tools like TensorBoard.

References

[Bellemare *et al.*, 2017] Marc G. Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. *ICML*, 2017.

[Fortunato *et al.*, 2018] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, and Ian Osband. Noisy networks for exploration. *ICLR*, 2018.

[Hessel *et al.*, 2018] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. *AAAI*, 2018.

[Kingma and Ba, 2017] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *ICLR*, 2017.

[Lin, 1993] Long-Ji Lin. Reinforcement learning for robots using neural networks. 1993.

[Mnih *et al.*, 2013] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Daan Wierstra Ioannis Antonoglou, and Martin Riedmiller. Playing atari with deep reinforcement learning. *NIPS Deep Learning Workshop*, 2013.

[Mnih *et al.*, 2015] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, and Joel Veness. Human-level control through deep reinforcement learning. *Nature*, 2015.

[Schaul *et al.*, 2016] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *ICLR*, 2016.

[Sutton and Barto, 1998] Richard S. Sutton and Andrew G. Barto. Reinforcement learning:an introduction. *MIT Press*, 1998.

[van Hasselt *et al.*, 2016] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double Q-learning. *AAAI*, 2016.

[Wang *et al.*, 2015] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning. *arXiv:1511.06581*, 2015.