Chapter 4
Deep Q-Networks

Yanhua Huang

Abstract This chapter aims to introduce one of the most important deep reinforcement learning algorithms, called deep Q-networks. We will start with the Q-learning algorithm via temporal difference learning, and introduce the deep Q-networks algorithm and its variants. We will end this chapter with code examples and experimental comparison of deep Q-networks and its variants in practice.

Keywords Temporal difference learning · DQN · Double DQN · Dueling DQN · Prioritized experience replay · Distributional reinforcement learning

4.1 Introduction

One of the most significant breakthroughs in reinforcement learning was the development of an off-policy temporal difference (TD) control algorithm, known as Q-learning, which is introduced in Chap. 2. Q-Learning has been proven to converge towards the optimal solution in a tabular case or using linear function approximation. However, it is known that Q-learning is unstable or even to diverge when using a non-linear function approximator such as a neural network to represent the Q-value function (Tsitsiklis and Van Roy 1996). With the advances in training deep neural networks, deep Q-networks (**DQN**) (Mnih et al. 2015) addressed this issue and ignited the research of deep reinforcement learning. In this chapter, we first review the background of Q-learning. Then we introduce DQN and its variants with detailed theories and explanations. Finally, in Sect. 4.10, we demonstrate their implementation details and empirical performance on the Atari games with code examples, for providing the readers a quick hands-on learning process. The

Y. Huang (✉)
Xiaohongshu Technology Co., Ltd., Shanghai, China

H. Dong et al. (eds.), *Deep Reinforcement Learning*,
https://doi.org/10.1007/978-981-15-4095-0_4

complete implementation of each algorithm is available in the repository provided together with the book.[1]

4.2 Background

Model-free methods provide a general way to tackle MDP-based decision-making problems, where "model" means an explicit model for the transition probability distribution and the reward function associated with the MDP. TD learning is a class of model-free methods. Recall that in Sect. 2.4 we discuss that if a perfect model of the MDP is available, one can get the optimal plan with dynamic programming by reusing the optimal solution of sub-problems recursively. TD learning follows such an idea that we can estimate the value of sub-problems with bootstrapping even though the estimation is not optimal all the time.

Sub-problems are represented by states in MDP. The value $v_\pi(s)$ of a state s with a policy π is defined by the expected return starting from s and acting with π:

$$v_\pi(s) = \mathbb{E}_\pi[R_t + \gamma v_\pi(S_{t+1})|S_t = s], \tag{4.1}$$

where $\gamma \in [0, 1]$ is the discount rate. TD learning decomposes the estimation above with bootstrapping. Given a value function $V : \mathcal{S} \to \mathbb{R}$, the simplest version, TD(0), is the following one-step bootstrapping:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_t + \gamma V(S_{t+1}) - V(S_t)], \tag{4.2}$$

where $R_t + \gamma V(S_{t+1})$ and $R_t + \gamma V(S_{t+1}) - V(S_t)$ are known as the TD target and TD error, respectively.

The value of a policy provides a way to estimate the acting performance. To further know how to select the action in a particular state, we would like to calculate the quality of the state-action combinations. Q-value allows such estimation:

$$q_\pi(s, a) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s, A_t = a]. \tag{4.3}$$

The simplest way to perform a better policy is acting greedily $\pi'(s, a) = \arg\max_{a'} q^\pi(s, a')$, where the improvement can be ensured with $q_{\pi'}(s, a) = \max_{a'} q_\pi(s, a') \geq q_\pi(s, a)$. An alternative for considering exploration is to act greedily most of the time, but with a small probability ϵ instead to select randomly from all actions with equal probability regardless of their Q-values. This method is

[1] Codes are available at: https://github.com/deep-reinforcement-learning-book/Chapter4-DQN.

called ϵ-greedy. We can calculate the Q-value of ϵ-greedy policy π' by

$$q_\pi(s, \pi'(s)) = (1-\epsilon)\max_{a\in\mathcal{A}} q_\pi(s,a) + \frac{\epsilon}{|\mathcal{A}|}\sum_{a\in\mathcal{A}} q_\pi(s,a). \tag{4.4}$$

Note that the sum of $\frac{\pi(s,a)-\epsilon/|\mathcal{A}|}{1-\epsilon}$ over $a \in \mathcal{A}$ is equal to 1. With the truth that the maximum is not less than the weighted average, we can get

$$\begin{aligned} q_\pi(s, \pi'(s)) &= (1-\epsilon)\max_{a\in\mathcal{A}} q_\pi(s,a)\sum_{a\in\mathcal{A}}\frac{\pi(s,a)-\epsilon/|\mathcal{A}|}{1-\epsilon} + \frac{\epsilon}{|\mathcal{A}|}\sum_{a\in\mathcal{A}} q_\pi(s,a) \\ &\geq (1-\epsilon)\sum_{a\in\mathcal{A}}\frac{\pi(s,a)-\epsilon/|\mathcal{A}|}{1-\epsilon} q_\pi(s,a) \\ &\quad + \frac{\epsilon}{|\mathcal{A}|}\sum_{a\in\mathcal{A}} q_\pi(s,a) = q_\pi(s,\pi(s)), \end{aligned} \tag{4.5}$$

which tells us that the Q-value of acting with the ϵ-greedy policy π' is not less than the origin policy π, i.e., ϵ-greedy method ensures policy improvement. We will discuss policy improvement with Q-value function in the next section.

4.3 Sarsa and Q-Learning

Similar to the update rule for the value function in TD(0), it is straightforward to update the Q-value function after every transition from a non-terminal state S_t:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_t + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)], \tag{4.6}$$

where both A_t and A_{t+1} are selected ϵ-greedily with respect to Q. If S_{t+1} is a terminal state, then $Q(S_{t+1}, A_{t+1})$ is defined as zero. We can continually estimate Q for the behavior policy π, and change π toward greediness with respect to Q at the same time. This algorithm is known as Sarsa. Note that π plays two roles in Sarsa—for experience generation and policy improvement. Basically, the policy used to generate behavior is called the behavior policy, and the policy that is evaluated and improved is called the target policy. The algorithm where the behavior policy and the target policy are the same, such as Sarsa, is known as the on-policy method.

On-policy methods are kinds of trial-and-error processes but only the experiences generated by the current policy are used for improvement. Off-policy methods address this issue with introspection, where the experience generated by the behavior policy is "off" (not following) the target policy. The off-policy technique allows reusing past experience. Q-learning is an off-policy method. Its simplest

form, one-step Q-learning, follows the update rule below:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_t + \gamma \max_{A_{t+1}} Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)], \tag{4.7}$$

where A_t is sampled by ϵ-greedy with respect to Q. Note that A_{t+1} is selected greedily, i.e., contrast to Sarsa, in Q-learning the behavior policy is also ϵ-greedy but the target policy is greedy. One-step Q-learning only takes in current transition. An alternative way to get more accurate Q-values in approximation case is to use multi-steps rewards, i.e., multi-steps Q-learning. Note that multi-steps Q-learning needs to consider the mismatches in subsequent rewards to keep the Q-value function approximating the expected return under the target policy as in Eq. (4.3). We will discuss multi-steps Q-learning in Sect. 4.9.

4.4 Why Deep Learning: Value Function Approximation

In tabular settings, the action-value functions can be represented by a big two-dimensional table, i.e., one entry for each discrete state and action. However, it is inefficient to deal with large-scale space such as raw pixels input, and let alone continuous control tasks. Fortunately, generalization from different inputs by function approximation has been widely studied, and we can utilize this technique in value-based reinforcement learning.

Let us consider the function approximation in Q-learning with some parameter θ. The approximator can be linear models, decision trees, or neural networks. Then the update rule in Eq. (4.7) is rewritten as

$$\theta_t \leftarrow \arg\min_{\theta} \mathcal{L}(Q(S_t, A_t; \theta), R_t + \gamma Q(S_{t+1}, A_{t+1}; \theta)), \tag{4.8}$$

where $\mathcal{L}$ represents the loss function, e.g., mean squared error. While one can solve the optimization problem above by collecting batch samples, which constructs the fitted Q iteration (Riedmiller 2005) shown as Algorithm 1, where S_i' is the successor state of S_i. An online stochastic variant with gradient is the online Q iteration algorithm presented in Algorithm 2.

Algorithm 1 Fitted Q iteration

for iteration $i = 1, T$ **do**
  Collect D samples $\{(S_i, A_i, R_i, S_i')\}_{i=1}^{D}$
  for t $= 1, K$ **do**
    Set $Y_i \leftarrow R_i + \gamma \max_a Q(S_i', a; \theta)$
    Set $\theta \leftarrow \arg\min_{\theta'} \frac{1}{2} \sum_{i=1}^{D} (Q(S_i, A_i; \theta') - Y_i)^2$
  end for
end for

Algorithm 2 Online Q iteration

for iteration $= 1, T$ **do**
  Select action a and observe (s, a, r, s')
  Set $y \leftarrow r + \gamma \max_{a'} Q(s', a'; \theta)$
  Set $\theta \leftarrow \theta - \alpha(Q(s, a; \theta) - y)\frac{\mathrm{d}Q(s,a;\theta)}{\mathrm{d}\theta}$
end for

Note that both the fitted Q iteration and online Q iteration are off-policy algorithms so that the past experience can be reused many times. We will discuss this topic further in the next section.

Recall that we introduce the convergence of value iteration in Sect. 2.4.2 with the Bellman optimality backup operator $\mathcal{T}^*$. We define a new operator $\mathcal{B}$ with function approximation by $\mathcal{B}V = \arg\min_{V' \in \Omega} \mathcal{L}(V', V)$, where Ω is the set of all possible value functions that can be approximated. Note that the arg min in $\mathcal{B}$ can be viewed as a projection from $\mathcal{T}^*V$ to Ω. So the backup operator with function approximation can be represented by $\mathcal{B}\mathcal{T}^*$. While $\mathcal{T}^*$ is contracted with ∞-norm and $\mathcal{B}$ is contracted with L2-norm for MSE loss. However, $\mathcal{B}\mathcal{T}^*$ is not contracted of any kind. So value iteration is unstable and might even diverge when a non-linear function approximator such as a neural network is used to represent the value function (Tsitsiklis and Van Roy 1997). We leave the discussion about the stability of training with deep neural networks in the next section.

4.5 DQN

In the last section, we introduce the method to learn action-value functions with approximation and its instability of convergence. To achieve the end-to-end decision-making in complex problems with raw pixel input, DQN combines Q-learning with deep learning with two key ideas to address the instability issue and achieves significant progress on Atari games.

The first one is known as **replay buffer**, which is a biologically inspired mechanism termed experience replay (McClelland et al. 1995; O'Neill et al. 2010; Lin 1993). At each time step t, DQN stores the experience of the agent (S_t, A_t, R_t, S_{t+1}) into replay buffer, and then draws a mini-batch of samples from this buffer uniformly to apply the Q-learning update. Replay buffer has several advantages over the fitted Q iteration. First, the experience in each step can be reused to learn the Q-function, which allows for greater data efficiency. Second, if there is no replay buffer, as in the fitted Q iteration, mini-batch samples are collected consecutively, i.e., they are highly correlated, which increases the variance of the updates. Third, experience replay avoids the situation that the samples used to train are determined by the previous parameters, which smooths out learning and reduces oscillations or divergence in the parameters. In practice, only the last N experience tuples are stored in the replay buffer to save the memory.

Table 4.1 The effects of replay and separating the target Q-network

Game	With replay and target Q	With replay but without target Q	With target Q but without replay	Without replay and target Q
Breakout	316.8	240.7	10.2	3.2
Enduro	1006.3	831.4	141.9	29.1
River raid	7446.6	4102.8	2867.7	1453.0
Seaquest	2894.4	822.6	1003.0	275.8
Space invaders	1088.9	826.3	373.2	302.0

Data comes from Mnih et al. (2015)

The second idea aims to further improve the stability with neural networks. Instead of the desired Q-network, a separate network, known as **target network**, is used to generate the Q-learning targets. Furthermore, at every C steps, the target network will be synchronized with the primary Q-network by copying directly (hard update) or exponentially decaying average (soft update). The target network makes the generation of the Q-learning target delay with old parameters, which reduces the divergence and oscillations much more. For example, the update making Q-value increase on action (S_t, A_t) may increase $Q(S_{t+1}, a)$ for all action a because of the similarity between S_t and S_{t+1}, where the training target constructed by Q-network will be overestimated.

The effect of two enhancements above on five Atari games is shown in Table 4.1. Agents were trained for 1e7 frames with the hyperparameters search. Each agent was evaluated every 250,000 training frames for 135,000 validation frames, and the highest average episode score is reported.

Since it is challenging to feed histories of arbitrary length as inputs to a neural network, DQN instead works on the fixed-length representation of histories produced by a function ϕ. More precisely, ϕ concentrates on the current and the previous three frame, which is useful for tracking temporal information, e.g., object moving. The full algorithm is presented in Algorithm 3. The raw frames are resized to 84×84 and converted to gray-scale. The function ϕ stacks the 4 most recent frames as the input to the neural network. In addition, the architecture of the neural network consists of three convolutional layers and two fully connected layers with a single output for each valid action. We will discuss more training details in Sect. 4.10.2.

4.6 Double DQN

Double DQN is an enhancement of DQN for reducing overestimation (Van Hasselt et al. 2016). Before taking a closer look, let us first illustrate the overestimation problem in classic DQN. The Q-learning target $R_t + \gamma \max_a Q(S_{t+1}, a)$ contains a max operator. Q is noisy, which may be caused by environment, non-stationarity, function approximation or any other reasons. Note that the expectation of maximum

Algorithm 3 DQN

Hyperparameters: replay buffer capacity N, reward discount factor γ, delayed steps C for target action-value function update, ϵ-greedy factor ϵ
Input: empty replay buffer $\mathcal{D}$, initial parameters θ of action-value function Q
Initialize target action-value function $\hat{Q}$ with parameter $\hat{\theta} \leftarrow \theta$
for episode $= 0, 1, 2, \ldots$ **do**
  Initialize environment and get observation O_0
  Initialize sequence $S_0 = \{O_0\}$ and preprocess sequence $\phi_0 = \phi(S_0)$
  for t $= 0, 1, 2, \ldots$ **do**
    With probability ϵ select a random action A_t, otherwise select $A_t = \arg\max_a Q(\phi(S_t), a; \theta)$
    Execute action A_t and observe O_{t+1} and reward R_t
    If the episode has ended, set $D_t = 1$. Otherwise, set $D_t = 0$
    Set $S_{t+1} = \{S_t, A_t, O_{t+1}\}$ and preprocess $\phi_{t+1} = \phi(S_{t+1})$
    Store transition $(\phi_t, A_t, R_t, D_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_i, A_i, R_i, D_i, \phi_i')$ from $\mathcal{D}$
    If $D_i = 0$, set $Y_i = R_i + \gamma \max_{a'} \hat{Q}(\phi_i', a'; \hat{\theta})$. Otherwise, set $Y_i = R_i$
    Perform a gradient descent step on $(Y_i - Q(\phi_i, A_i; \theta))^2$ with respect to θ
    Synchronize the target $\hat{Q}$ every C steps
    If the episode has ended, break the loop
  end for
end for

noise is not less than the maximum expectation of noises, i.e., $\mathbb{E}[\max(\epsilon_1, \ldots, \epsilon_n)] \geq (\max(\mathbb{E}[\epsilon_1], \ldots, \mathbb{E}[\epsilon_n]))$. So the next Q-values are always overestimated. Thrun and Schwartz (1993) provides further theoretical analysis and experimental results.

Note that the training target in standard DQN can be rewritten by

$$R_t + \gamma \hat{Q}(S_{t+1}, \arg\max_a \hat{Q}(S_{t+1}, a; \hat{\theta}); \hat{\theta}), \tag{4.9}$$

where $\hat{\theta}$ is used in both action selection and value evaluation. The central idea of double DQN is to decorrelate the noises in selection and evaluation by using two different networks in these two stages. The Q-network in the DQN architecture provides a natural candidate for the extra network. Recall that it is the evaluation role of the target network that improves the stability more. As a consequence, the Q-learning target used in double DQN is

$$R_t + \gamma \hat{Q}(S_{t+1}, \arg\max_a Q(S_{t+1}, a; \theta); \hat{\theta}). \tag{4.10}$$

Following Wang et al. (2016), we measure improvement in percentage (positive or negative) in score over the better of human and baseline agent scores:

$$\frac{\text{Score}_{\text{Agent}} - \text{Score}_{\text{Baseline}}}{\max(\text{Score}_{\text{Baseline}}, \text{Score}_{\text{Human}}) - \text{Score}_{\text{Random}}}. \tag{4.11}$$

The improvement over DQN are available in Fig. 4.1.

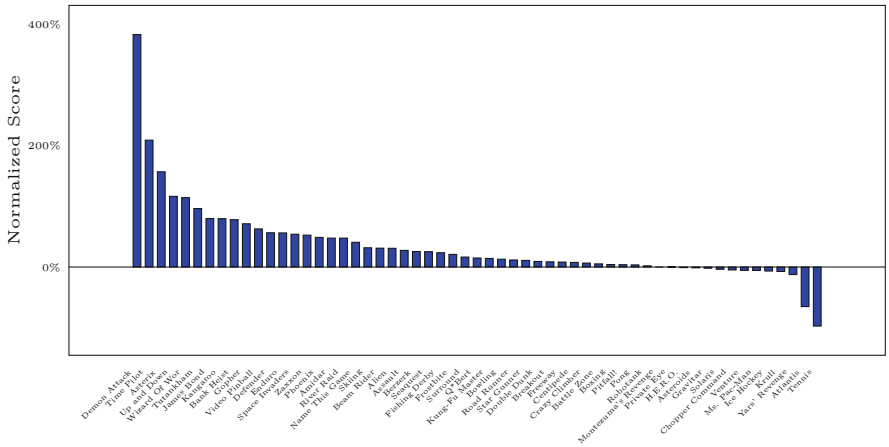

Fig. 4.1 Improvements of double DQN (Van Hasselt et al. 2016) over DQN (Mnih et al. 2015) in Atari benchmark, using the metric described in Eq. (4.11). All scores come from Wang et al. (2016) (Table 2)

4.7 Dueling DQN

For some states, different actions are not relevant to the expected value, and we do not need to learn the effect of each action for such states. For example, imagine standing on the mountain and watching the sunrise. The pleasant view comforts you a lot, which provides a high reward. You can stay here, and the Q-values of different actions do not matter. So decoupling the action-independent value of state and Q-value may lead to more robust learning.

Dueling DQN proposes a new network architecture to achieve this idea (Wang et al. 2016). More precisely, the Q-value can be split into state value part and action advantage part as following:

$$Q^{\pi}(s,a) = V^{\pi}(s) + A^{\pi}(s,a) \tag{4.12}$$

and dueling DQN separates the representations of these two parts by

$$Q(s,a;\theta,\theta_v,\theta_a) = V(s;\theta,\theta_v) + (A(s,a;\theta,\theta_a) - \max_{a'} A(s,a';\theta,\theta_a)), \tag{4.13}$$

where θ_v and θ_a are parameters of the two streams of fully connected layers, θ represent the parameters in convolutional layers. Note that the max operator in Eq. (4.13) ensures identifiability that the Q-value recovers state value and action advantage uniquely. Otherwise, the training may ignore the state value term and make the advantage function converge to Q-value only. Moreover, Wang et al. (2016) also proposed to replace max with average as the following for better stability:

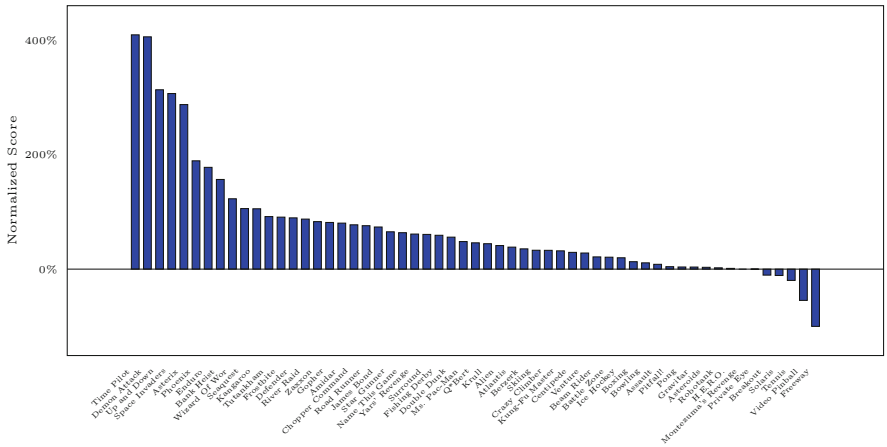

Fig. 4.2 Improvements of dueling DQN (Wang et al. 2016) over DQN (Mnih et al. 2015) in Atari benchmark, using the metric described in Eq. (4.11). All scores come from Wang et al. (2016) (Table 2)

$$Q(s,a;\theta,\theta_v,\theta_a) = V(s;\theta,\theta_v) + (A(s,a;\theta,\theta_a) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a';\theta,\theta_a)) \tag{4.14}$$

by which the advantage function only need to adapt to the direct of mean advantage instead of pursuing the optimal advantage.

Training of the dueling architectures, as with standard DQN, requires only more layers. The experiments show that dueling architectures lead to better policy evaluation in the presence of many similar-valued actions. The improvement over DQN is available in Fig. 4.2.

4.8 Prioritized Experience Replay

One remaining area of improvement in standard DQN is a better sampling strategy for experience replay. **Prioritized experience replay (PER)** is a technique for prioritizing experience, so as to replay important transitions more frequently (Schaul et al. 2015). The central idea of PER is to consider the importance of transitions with

TD error δ, which can be viewed as a surprising measure. Why this can be of help is that some of the experience might contain more information to learn as compared to the others. Giving those more information-rich experience a greater chance of being replayed will help make the whole learning process faster and more efficient.

The most direct idea is using TD error for prioritization directly. However, it has several issues. First, sweeping over whole memory is inefficient. In addition, it is sensitive to noises such as approximation error and stochastic rewards. Finally, greedy makes error shrink slowly, which may cause the beginning transitions with high error replayed frequently. To overcome these issues, Schaul et al. (2015) proposed to use the following sampling probability for transition i:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}, \tag{4.15}$$

where $p_i > 0$, known as the priority of transition i, α is an exponent hyper-parameter with $\alpha = 0$ corresponding to the uniform case, and k is enumerated on sampled transitions. There are two variants of p_i. The first one is proportional prioritization $p_i = |\delta_i| + \epsilon$, where δ_i is the TD error of transition i and ϵ is a small positive value for numerical stability. The second one is rank-based prioritization $p_i = \frac{1}{\text{rank}(i)}$, where $\text{rank}(i)$ is the rank of transition i according to $|\delta_i|$.

Remind that it is the random sampling from a large replay buffer that helps to decorrelate the samples. But the purely random sampling is abandoned when adding priority sampling. Decreasing the training weight for high priority transitions may make sense. PER uses the importance-sampling weights to correct this bias for transition i:

$$w_i = (NP(i))^{-\beta}, \tag{4.16}$$

where N is the size of replay buffer, P is the probability defined in Eq. (4.15), and β is a hyper-parameter annealed up to 1 during training because the unbiased nature of the updates will nearly converge at the end of the training. This weight is usually folded into the loss function to construct weighted learning.

For efficient implementation, the cumulative density function of sampling probability is approximated by a piece-wise linear function with k segments. More precisely, the priorities are stored in a query-efficient data structure called the segment tree. During run-time, PER first samples a segment, and then sample uniformly among the transitions within it. The improvement over DQN is available in Fig. 4.3.

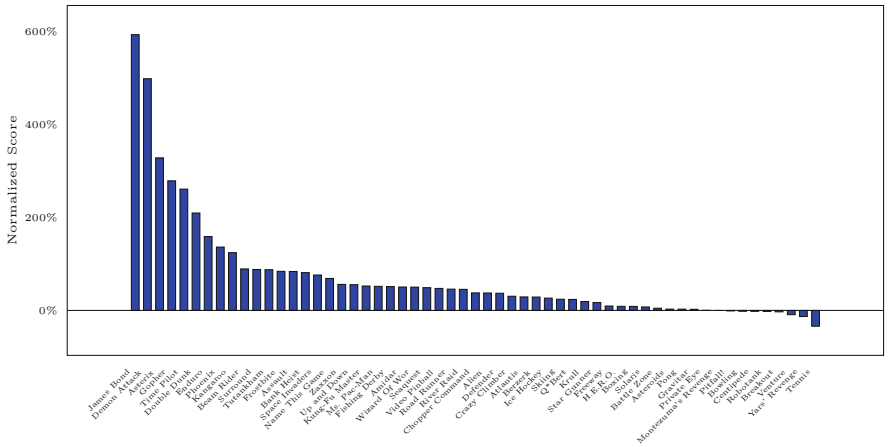

Fig. 4.3 Improvements of prioritized experience replay (Schaul et al. 2015) with rank-based prioritization over DQN (Mnih et al. 2015) in Atari benchmark, using the metric described in Eq. (4.11). All scores come from Wang et al. (2016) (Table 2)

4.9 Other Improvements: Multi-Step Learning, Noisy Nets, and Distributional Reinforcement Learning

Including double Q-learning, dueling architecture, and PER, **Rainbow** combines three more extensions to DQN and achieves significant results on the Atari domain (Hessel et al. 2018). We discuss them and their expansions in this section. The first one is multi-step learning. n-step return allows for accurate estimation and was proven to lead faster learning with suitably tuned n (Sutton and Barto 2018). However, there may exist a mismatch in the action selection between the target and behavior policy within the multi-steps during off-policy learning. One can find a systematic study about correcting this mismatch in Hernandez-Garcia and Sutton (2019). Rainbow uses the truncated n-step return $R_t^{(k)}$ from a given state S_t directly (Hessel et al. 2018; Castro et al. 2018), where $R_t^{(k)}$ is defined by

$$R_t^{(k)} = \sum_{k=0}^{n-1} \gamma^k R_{t+k}. \tag{4.17}$$

The Q-learning target in multi-step variant of Q-learning is then defined by

$$R_t^{(k)} + \gamma^k \max_a Q(S_{t+k}, a). \tag{4.18}$$

The second one is noisy nets (Fortunato et al. 2017). It is an alternative exploration algorithm for ϵ-greedy, especially for games requiring huge exploration such as Montezuma's Revenge. The noise is added into linear layer $\boldsymbol{y} = (\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$

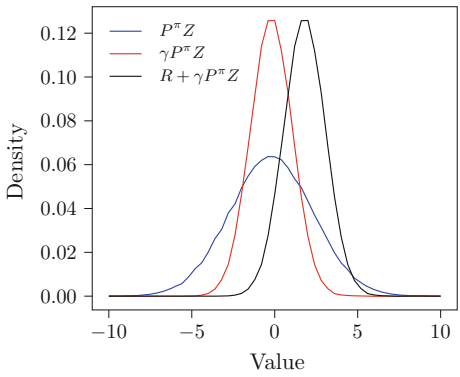

Fig. 4.4 A distributional Bellman operator in continuous case. Given the return distribution of the next state under policy π (blue curve), it is first discounted by the reward discounter γ (red curve), and then be shifted by the reward in current time step (black curve)

by an extra noisy stream

$$\boldsymbol{y} = (\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) + ((\boldsymbol{W_{noisy}} \odot \epsilon_w)\boldsymbol{x} + \boldsymbol{b_{noisy}} \odot \epsilon_b), \tag{4.19}$$

where $\odot$ refers to the element-wise product, both $\boldsymbol{W_{noisy}}$ and $\boldsymbol{b_{noisy}}$ are trainable parameters whereas ϵ_w and ϵ_b are random scales annealed down to zero. The experiment shows that noisy nets yield substantially higher scores for a wide range of Atari games over several baselines.

The last one is distributional reinforcement learning (Bellemare et al. 2017), which gives a new perspective on value estimation. Instead of considering the expectation of returns represented by random variable Z^π, Bellemare et al. (2017) proposed to estimate the distribution of Z^π with the distributional Bellman operator $\mathcal{T}^\pi$:

$$\mathcal{T}^\pi Z = R + \gamma P^\pi Z. \tag{4.20}$$

Figure 4.4 shows a continuous case of $\mathcal{T}^\pi$.

The distributional variant of DQN used in Rainbow, known as categorical DQN (Bellemare et al. 2017), models the action-value distribution by a discrete distribution parameterized by a vector $\boldsymbol{z}$ with N elements (also known as atoms) $z_i = V_{min} + (i-1)\Delta z$, where $[V_{min}, V_{max}]$ is the action-value range and $\Delta z = \frac{V_{max} - V_{min}}{N-1}$. In practice, N is usually specified to 51 so sometimes this algorithm is also called **C51**. The parametric model θ of C51 outputs the probabilities $p_i(s,a) = e^{\theta_i(s,a)} / \sum_j e^{\theta_j(s,a)}$ on each atom as distribution Z_θ. Note that discrete approximation causes disjoint supports of the Bellman update $\mathcal{T}^\pi Z$ and the parametrization Z_θ. C51 addresses this issue by projecting the target distribution $\mathcal{T}^\pi Z_{\hat{\theta}}$ onto the support Z_θ. More precisely, given a transition (S_t, A_t, R_t, S_{t+1}), the i-th component of projected target $\Phi \mathcal{T}^\pi Z_{\hat{\theta}}(S_t, A_t)$ with double Q-learning is

calculated by

$$\sum_{j=1}^{N} p_j \left(S_{t+1}, \arg\max_a \boldsymbol{z}^\intercal p(S_{t+1}, a; \theta); \hat{\theta} \right) [1 - \frac{|[R_t + \gamma z_j]_{V_{min}}^{V_{max}} - z_i|}{\Delta z}]_0^1, \tag{4.21}$$

where $[\cdot]_a^b$ bounds its argument in the range $[a, b]$. TD error cannot measure the difference between value distributions. As a result, C51 proposes to use the following Kullback–Leibler divergence as training loss:

$$D_{\mathrm{KL}}(\Phi \mathcal{T}^\pi Z_{\hat{\theta}}(S_t, A_t) || Z_\theta(S_t, A_t)). \tag{4.22}$$

In addition, the priority is also replaced by KL-Divergence for prioritized experience replay. For dueling architecture, the output distribution is also split into value stream and advantage stream, and the aggregated distribution is estimated by

$$p_i(s, a) = \frac{exp(V_i(s) + A_i(s, a) - \bar{A}_i(s, a))}{\sum_j exp(V_j(s) + A_j(s, a) - \bar{A}_j(s, a))}, \tag{4.23}$$

where $\bar{A}_j(s, a)$ is defined by $\frac{1}{|\mathcal{A}|} \sum_{a'} A_j(s, a')$.

The main drawback of C51 to achieve distributional reinforcement learning is that it can only estimate values on a fixed discrete set. Dabney et al. (2018b) proposed **quantile regression DQN (QR-DQN)** to address this issue by estimating the quantiles of the full distribution with quantile regression. Before introducing QR-DQN, we first review the quantile regression. Recall that empirical risk minimization with absolute loss function makes the prediction fit the medium value (50% quantile). More precisely, given random variable x and its label y, for estimation function f, the empirical mean absolute error is $\mathcal{L}_{\mathrm{mae}} = \mathbb{E}[|f(x) - y|]$. Then with the following partial difference:

$$\begin{aligned} \frac{\partial \mathcal{L}_{\mathrm{mae}}}{\partial f(x)} &= \frac{\partial}{\partial f(x)} (P(f(x) > y)(f(x) - y) + P(f(x) \le y)(y - f(x))) \\ &= P(f(x) > y) - P(f(x) \le y) = 0, \end{aligned} \tag{4.24}$$

we can get $F(x) = 0.5$, where F is the primitive function of f. Generally, for quantile τ, the quantile loss is defined as $\mathcal{L}_{\mathrm{quantile}}(\tau) = \mathbb{E}[\rho_\tau(f(x) - y)]$ with

$$\rho_\tau(\alpha) = \begin{cases} \tau\alpha, & \text{if } \alpha > 0 \\ (\tau - 1)\alpha, & \text{otherwise.} \end{cases} \tag{4.25}$$

Similarly, by $\frac{\partial \mathcal{L}_{\mathrm{quantile}}}{\partial f(x)}$ we can get $F(x) = 1 - \tau$, i.e., $f(x)$ is the τ quantile value of random variable y.

Specifically, QR-DQN considers N uniform quantiles $q_i = \frac{1}{N}$ for the value distribution. For a QR-DQN model $\theta : \mathcal{S} \to \mathbb{R}^{N \times |\mathcal{A}|}$, during sampling, the Q-value

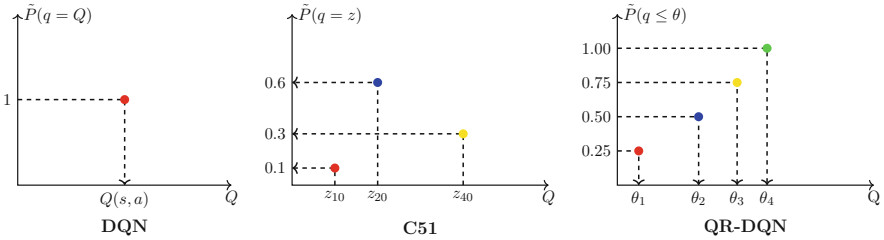

Fig. 4.5 Comparison of DQN, C51, and QR-DQN for state s and action a, where arrows point out the estimation and the number of quantiles in QR-DQN is specified to 4. The architecture of DQN only outputs the approximation of the actual Q-value. For distributional reinforcement learning, C51 estimates probabilities on several Q-value supports while QR-DQN provides quantiles of Q-value

of the state s and action a is the mean of N estimations: $Q(s, a) = \sum_{i=1}^{N} q_i \theta_i(s, a)$. During training, the greedy policy with respect to the Q-values in the next state provides $a^* = \arg\max_{a'} Q(s', a')$, and the distributional Bellman target is $\mathcal{T}\theta_j = r + \gamma\theta_j(s', a^*)$ according to Eq. (4.20). The Lemma 2 in Dabney et al. (2018b) points out that the following sum minimizes the 1-Wasserstein distance between the approximate value distribution and the ground truth:

$$\sum_{i=1}^{N} \mathbb{E}_j \left[\rho_{\hat{\tau}_i} (\mathcal{T}\theta_j - \theta_i(s, a)) \right], \tag{4.26}$$

where $\hat{\tau}_i = \frac{i}{N} - \frac{1}{2N}$.

Figure 4.5 shows a comparison of DQN, C51, and QR-DQN. There are further works in the flexibility or robustness of parameterized distribution for distributional reinforcement learning. Readers with more interest in this topic can find related resources from Dabney et al. (2018a), Mavrin et al (2019), Yang et al. (2019).

4.10 DQN Examples

In this section, we discuss more training details in DQN and its variants. Before that, we first demonstrate the process of setting up Atari environments and how to implement some useful wrappers that make training easy and stable.

4.10.1 Related Gym Environment

OpenAI Gym is an open-source toolkit for developing and comparing reinforcement learning algorithms. It contains a collection of environments, as shown in Fig. 4.6. One can install it with Atari extension directly from PyPI

```
pip install gym[atari]
```

or from source

```
git clone https://github.com/openai/gym.git
cd gym
pip install -e .
```

An environment object `env` can be created by

```
import gym
env = gym.make(env_id)
```

where `env_id` is a string that represents an environment. All possible `env_ids` are available at https://github.com/openai/gym/wiki/Table-of-environments.

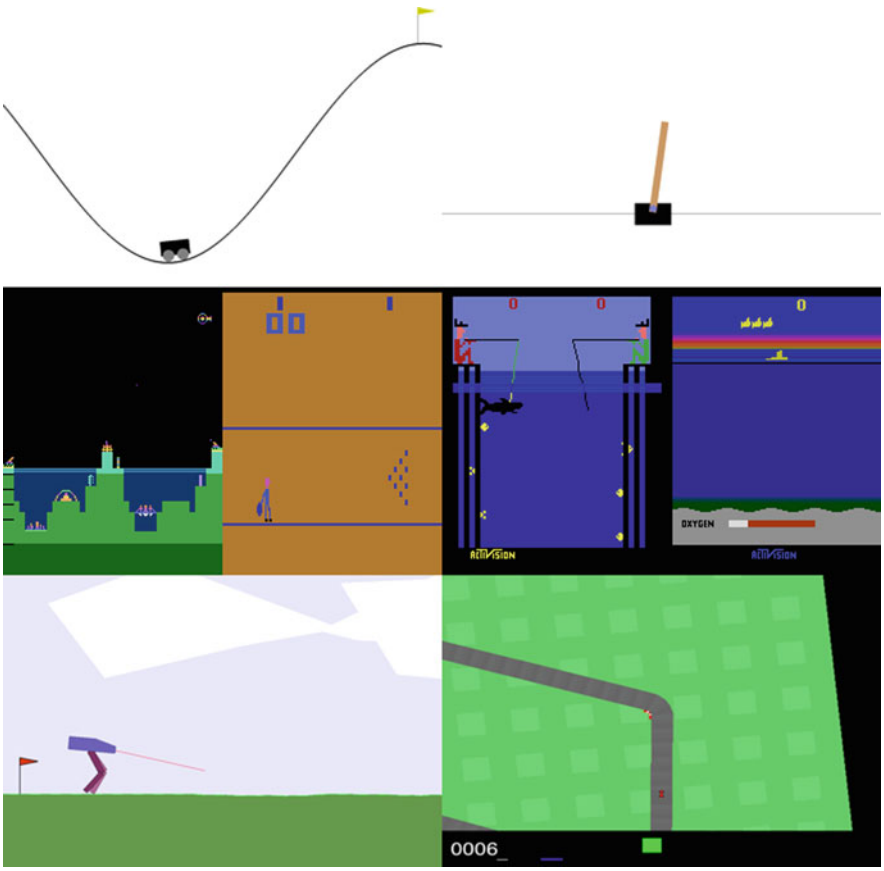

Fig. 4.6 Sample frames of some environments in OpenAI Gym

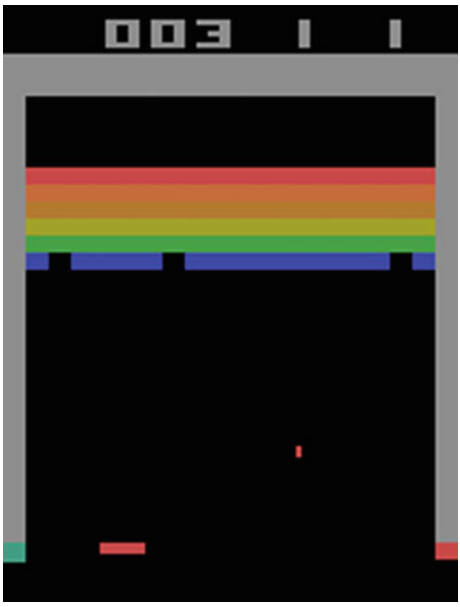

Fig. 4.7 An example frame of Breakout. There are several rows to destroy at the top of the screen. The agent can control the bar at the bottom of the screen to angle your shots at the images you want to smash with the ball. The observations are an RGB images of the screen with shape (210, 160, 3)

There are some important methods of `env`:

1. `env.reset()` resets the state of the environment and returns the initial observation.
2. `env.render(mode)` renders the environment with the given `mode`. The default mode is `human` which renders to the current display or terminal and returns nothing. You can specify `rgb_array` mode to make `env.render` return `numpy.ndarray` objects, which is suitable for generating videos.
3. `env.step(action)` runs one time step of the environment's dynamics with the given `action`, and then returns a tuple `(observation, reward, done, info)` where `observation` is the observation of the current environment, `reward` is the transition reward, `done` points out whether the episode has ended, and `info` contains some auxiliary information.
4. `env.seed(seed)` sets the seed manually, which is useful for reproduction.

Here is an example of classic game Breakout. It will run an instance of the `BreakoutNoFrameskip-v4` environment until the episode has ended. A sample frame shows in Fig. 4.7.

```
import gym
env = gym.make('BreakoutNoFrameskip-v4')
o = env.reset()
while True:
    env.render()
    # take a random action
    a = env.action_space.sample()
```

```
    o, r, done, _ = env.step(a)
    if done:
        break
env.close() # close and clean up
```

Note that `NoFrameskip` means no frame-skip and no action repeat, and `v4` means the 4th version which is the newest now. We will use this environment in following experiments.

Another useful feature in OpenAI Gym is the environment wrapper. It can wrap the environment object and make the training code more concise. Here is a time limit wrapper which limits the maximum length of each episode and is a default wrapper for Atari games.

```
class TimeLimit(gym.Wrapper):
    def __init__(self, env, max_episode_steps=None):
        super(TimeLimit, self).__init__(env)
        self._max_episode_steps = max_episode_steps
        self._elapsed_steps = 0

    def step(self, ac):
        o, r, done, info = self.env.step(ac)
        self._elapsed_steps += 1
        if self._elapsed_steps >= self._max_episode_steps:
            done = True
            info['TimeLimit.truncated'] = True
        return o, r, done, info

    def reset(self, **kwargs):
        self._elapsed_steps = 0
        return self.env.reset(**kwargs)
```

For efficient training, `gym.vector.AsyncVectorEnv` provides an implementation of vectorized wrapper that runs n environments in parallel. All interfaces receive and return n variables together. Furthermore, it is also possible to implement a vectorized wrapper with buffer whose interfaces also receive and return n variables but maintains $m > n$ workers in the background. It is efficient for environments where some transitions spend more time.

Included some classic control problems, Gym also provides standard interfaces of a collection of Atari 2600 games with RAM or screen images as input, using the Arcade Learning Environment (Bellemare et al. 2013). There are at most 18 different buttons in Atari 2600 games as follows:

1. Moving buttons: NOOP, UP, RIGHT, LEFT, DOWN, UPRIGHT, UPLEFT, DOWNRIGHT, DOWNLEFT
2. Fire buttons: FIRE, UPFIRE, RIGHTFIRE, LEFTFIRE, DOWNFIRE, UPRIGHTFIRE, UPLEFTFIRE, DOWNRIGHTFIRE, DOWNLEFTFIRE

where NOOP means do-nothing, and FIRE may also be used to start the game. For convenience, we will refer to buttons' names as actions later.

4.10.2 DQN

There are three more training tricks in DQN. First, the following wrappers are used in order for stable and efficient training:

1. `NoopResetEnv` takes random number of NOOPs in reset stage to ensure random initial states. The default maximum no-ops number is 30. This wrapper helps agent collect more beginning situations, which provides robust learning.
2. `MaxAndSkipEnv` repeats each action 4 times for efficient training. To further denoising observation, the returned frame is the max pooling result over pixels across the last two frames.
3. `Monitor` records the raw reward. We can also implement some useful functions such as speed tracer in this wrapper.
4. `EpisodicLifeEnv` makes the end of life equal to the end of episode, which helps value estimation (Roderick et al. 2017).
5. `FireResetEnv` takes action `FIRE` on reset for environments that need action `FIRE` to start the game. This is a prior knowledge for quick start of games.
6. `WarpFrame` converts the observations to 84×84 gray-scale images.
7. `ClipRewardEnv` wraps the rewards by their sign, which further improves the stability by not allowing any single mini-batch update to change the parameters drastically.
8. `FrameStack` stacks the last 4 frames. Recall that for capturing moving information, DQN preprocesses observations by concentrating the current frame and the previous three, represented by function ϕ. `FrameStack` and `WarpFrame` implement the ϕ. Note that we can optimize memory usage by storing common frames between the observations only once, which is also called lazy-frame trick.

Second, to avoid gradient explosion, DQN (Mnih et al. 2015; DeepMind 2015) uses gradient clipping of the squared error, which is equivalent to replace MSE by the Huber loss (Huber 1992) with $\delta = 1$. The Huber loss is given by

$$L_\delta(x) = \begin{cases} \frac{1}{2}x^2 & |x| \le \delta \\ \delta(|x| - \frac{1}{2}\delta) & \text{otherwise.} \end{cases} \tag{4.27}$$

Finally, replay buffer samples batch of experiences with replacement, and there are some warm start steps before updating for a stable beginning.

Note that all three tricks above are applied to all experiments in this section. Now we show how to build an agent to play Breakout. First of all, for reproducibility, we set random seeds in related libraries manually:

```
random.seed(seed)
np.random.seed(seed)
tf.random.set_seed(seed)
```

Then we build a Q-network with `tf.keras.Model`:

```
class QFunc(tf.keras.Model):
    def __init__(self, name):
        super(QFunc, self).__init__(name=name)
        self.conv1 = tf.keras.layers.Conv2D(
            32, kernel_size=(8, 8), strides=(4, 4),
            padding='valid', activation='relu')
        self.conv2 = tf.keras.layers.Conv2D(
            64, kernel_size=(4, 4), strides=(2, 2),
            padding='valid', activation='relu')
        self.conv3 = tf.keras.layers.Conv2D(
            64, kernel_size=(3, 3), strides=(1, 1),
            padding='valid', activation='relu')
        self.flat = tf.keras.layers.Flatten()
        self.fc1 = tf.keras.layers.Dense(512, activation='relu')
        self.fc2 = tf.keras.layers.Dense(action_dim,
            activation='linear')

    def call(self, pixels, **kwargs):
        # scale observation
        pixels = tf.divide(tf.cast(pixels, tf.float32),
            tf.constant(255.0))
        # extract features by convolutional layers
        feature =
            self.flat(self.conv3(self.conv2(self.conv1(pixels))))
        # calculate q-value
        qvalue = self.fc2(self.fc1(feature))

        return qvalue
```

The definition of a DQN object consists of attributes Q-network, target Q-network, number of time steps and optimizer, and synchronize Q-network and target Q-network as following:

```
class DQN(object):
    def __init__(self):
        self.qnet = QFunc('q')
        self.targetqnet = QFunc('targetq')
        sync(self.qnet, self.targetqnet)
        self.niter = 0
        self.optimizer = tf.optimizers.Adam(lr, epsilon=1e-5,
            clipnorm=clipnorm)
```

Declare an internal method to wrap the Q-network and then add a `get_action` method to DQN object for ϵ-greedy behavior:

```
@tf.function
def _qvalues_func(self, obv):
    return self.qnet(obv)

def get_action(self, obv):
```

```
    eps = epsilon(self.niter)
    if random.random() < eps:
        return int(random.random() * action_dim)
    else:
        obv = np.expand_dims(obv, 0).astype('float32')
        return self._qvalues_func(obv).numpy().argmax(1)[0]
```

where `epsilon` is a function that anneals ϵ linearly from 1.0 to 0.01 over the first 10% training time steps. For training, we use three common interfaces `train`, `_train_func`, `_tderror_func` for DQN and its variants in the following sections:

```
def train(self, b_o, b_a, b_r, b_o_, b_d):
    self._train_func(b_o, b_a, b_r, b_o_, b_d)

    self.niter += 1
    if self.niter % target_q_update_freq == 0:
        sync(self.qnet, self.targetqnet)

@tf.function
def _train_func(self, b_o, b_a, b_r, b_o_, b_d):
    with tf.GradientTape() as tape:
        td_errors = self._tderror_func(b_o, b_a, b_r, b_o_, b_d)
        loss = tf.reduce_mean(huber_loss(td_errors))

    grad = tape.gradient(loss, self.qnet.trainable_weights)
    self.optimizer.apply_gradients(zip(grad,
        self.qnet.trainable_weights))

    return td_errors

@tf.function
def _tderror_func(self, b_o, b_a, b_r, b_o_, b_d):
    b_q_ = (1 - b_d) * tf.reduce_max(self.targetqnet(b_o_), 1)
    b_q = tf.reduce_sum(self.qnet(b_o) * tf.one_hot(b_a,
        action_dim), 1)

    return b_q - (b_r + reward_gamma * b_q_)
```

where `train` calls `_train_func` and synchronizes the Q-network and target Q-network every `target_q_update_freq` time steps.

Finally, we build the main training procedure:

```
dqn = DQN()
buffer = ReplayBuffer(buffer_size)

o = env.reset()
nepisode = 0
t = time.time()
for i in range(1, number_time steps + 1):
    a = dqn.get_action(o)
```

```
    # execute action and feed to replay buffer
    # note that '_' tail in var name means next
    o_, r, done, info = env.step(a)
    buffer.add(o, a, r, o_, done)

    if i >= warm_start and i % train_freq == 0:
        transitions = buffer.sample(batch_size)
        dqn.train(*transitions)

    if done:
        o = env.reset()
    else:
        o = o_

    # episode in info is real (unwrapped) message
    if info.get('episode'):
        nepisode += 1
        reward, length = info['episode']['r'], info['episode']['l']
        print(
            'Time steps so far: {}, episode so far: {}, '
            'episode reward: {:.4f}, episode length: {}'
                .format(i, nepisode, reward, length)
        )
```

We run 10^7 time steps (4×10^7 frames) over three random seeds on Breakout. For better visualization, we smooth the episode rewards during training. Then we plot the mean and the standard deviation by following codes:

```
from matplotlib import pyplot as plt
plt.plot(xs, mean, color=color)
plt.fill_between(xs, mean - std, mean + std, color=color,
    alpha=.4)
```

The performance is shown in Fig. 4.8 with red area.

4.10.3 Double DQN

Double DQN can be implemented easily by using the following double Q estimation in `_tderror_func` of the agent:

```
# double Q estimation
b_a_ = tf.one_hot(tf.argmax(qnet(b_o_), 1), out_dim)
b_q_ = (1 - b_d) * tf.reduce_sum(targetqnet(b_o_) * b_a_, 1)
```

We also run 10^7 time steps over three random seeds on Breakout. The performance is shown in Fig. 4.8 with green area.

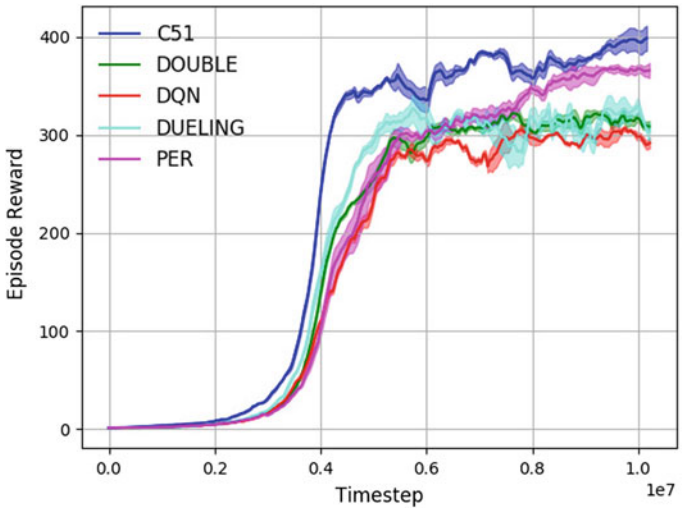

Fig. 4.8 Performances of DQN and its variants on breakout

4.10.4 Dueling DQN

The dueling architecture only changes the Q-network, which can be implemented by

```
class QFunc(tf.keras.Model):
    def __init__(self, name):
        super(QFunc, self).__init__(name=name)
        self.conv1 = tf.keras.layers.Conv2D(
            32, kernel_size=(8, 8), strides=(4, 4),
            padding='valid', activation='relu')
        self.conv2 = tf.keras.layers.Conv2D(
            64, kernel_size=(4, 4), strides=(2, 2),
            padding='valid', activation='relu')
        self.conv3 = tf.keras.layers.Conv2D(
            64, kernel_size=(3, 3), strides=(1, 1),
            padding='valid', activation='relu')
        self.flat = tf.keras.layers.Flatten()
        self.fc1q = tf.keras.layers.Dense(512, activation='relu')
        self.fc2q = tf.keras.layers.Dense(action_dim,
            activation='linear')
        self.fc1v = tf.keras.layers.Dense(512, activation='relu')
        self.fc2v = tf.keras.layers.Dense(1, activation='linear')

    def call(self, pixels, **kwargs):
        # scale observation
        pixels = tf.divide(tf.cast(pixels, tf.float32),
            tf.constant(255.0))
        # extract features by convolutional layers
```

```
        feature =
            self.flat(self.conv3(self.conv2(self.conv1(pixels))))
        # calculate q-value
        qvalue = self.fc2q(self.fc1q(feature))
        svalue = self.fc2v(self.fc1v(feature))

        return svalue + qvalue - tf.reduce_mean(qvalue, 1,
            keepdims=True)
```

We also run 10^7 time steps over three random seeds on Breakout. The performance is shown in Fig. 4.8 with cyan area.

4.10.5 Prioritized Experience Replay

There are three main changes in PER from standard DQN. First, the replay buffer maintains two segment trees with min operator and add operator to calculate the minimum priority and sum of priorities efficiently. More precisely, attribute `_it_sum` is the segment tree object with operation add with two interfaces, `sum` for getting the sum of elements in the given range and `find_prefixsum_idx` for finding the highest index i such that the sum of the smallest i elements is less than the input value.

Second, instead of uniform sampling, the sampling strategy of the proportional information is shown as follows:

```
res = []
p_total = self._it_sum.sum(0, len(self._storage) - 1)
every_range_len = p_total / batch_size
for i in range(batch_size):
   mass = random.random() * every_range_len + i * every_range_len
   idx = self._it_sum.find_prefixsum_idx(mass)
   res.append(idx)
return res
```

Third, apart from standard replay buffer, PER must return indexes and normalized weights of sampled experiences. Weights are used for weighting Huber loss, and indexes are used to update priorities. The sampling step is modified to

```
*transitions, idxs = buffer.sample(batch_size)
priorities = dqn.train(*transitions)
priorities = np.clip(np.abs(priorities), 1e-6, None)
buffer.update_priorities(idxs, priorities)
```

and the `_train_func` is modified to

```
@tf.function
def _train_func(self, b_o, b_a, b_r, b_o_, b_d, b_w):
   with tf.GradientTape() as tape:
```

```
        td_errors = self._tderror_func(b_o, b_a, b_r, b_o_, b_d)
        loss = tf.reduce_mean(huber_loss(td_errors) * b_w)

    grad = tape.gradient(loss, self.qnet.trainable_weights)
    self.optimizer.apply_gradients(zip(grad,
        self.qnet.trainable_weights))

    return td_errors
```

We also run 10^7 time steps over three random seeds on Breakout. The performance is shown in Fig. 4.8 with magenta area.

4.10.6 Distributed DQN

Distributional reinforcement learning estimates the distribution of the Q-value. In this section, we show how to implement one of these techniques, C51, to achieve distributed DQN. In game Breakout, the rewards are all positive. So we replace the value range $[-10, 10]$ used in Bellemare et al. (2017) by $[-1, 19]$, where -1 allows some approximation error. To implement C51, first of all, the Q-Network outputs 51 estimations for each action, which can be implemented by adding more output units in the last fully connection layer. Then instead of the TD error, KL-divergence between target Q distribution and the estimated distribution is used as loss:

```
@tf.function
def _kl_divergence_func(self, b_o, b_a, b_r, b_o_, b_d):
    b_r = tf.tile(
        tf.reshape(b_r, [-1, 1]),
        tf.constant([1, atom_num])
    ) # batch_size * atom_num
    b_d = tf.tile(
        tf.reshape(b_d, [-1, 1]),
        tf.constant([1, atom_num])
    )

    z = b_r + (1 - b_d) * reward_gamma * vrange # shift value
        distribution
    z = tf.clip_by_value(z, min_value, max_value) # clip the
        shifted distribution
    b = (z - min_value) / deltaz
    index_help = tf.expand_dims(tf.tile(
        tf.reshape(tf.range(batch_size), [batch_size, 1]),
        tf.constant([1, atom_num])
    ), -1)

    b_u = tf.cast(tf.math.ceil(b), tf.int32) # upper
    b_uid = tf.concat([index_help, tf.expand_dims(b_u, -1)], 2) #
        indexes
    b_l = tf.cast(tf.math.floor(b), tf.int32)
```

```
    b_lid = tf.concat([index_help, tf.expand_dims(b_l, -1)], 2) #
        indexes

    b_dist_ = self.targetqnet(b_o_) # whole distribution
    b_q_ = tf.reduce_sum(b_dist_ * vrange_broadcast, axis=2)
    b_a_ = tf.cast(tf.argmax(b_q_, 1), tf.int32)
    b_adist_ = tf.gather_nd( # distribution of b_a_
       b_dist_,
       tf.concat([tf.reshape(tf.range(batch_size), [-1, 1]),
                  tf.reshape(b_a_, [-1, 1])], axis=1)
    )
    b_adist = tf.gather_nd( # distribution of b_a
       self.qnet(b_o),
       tf.concat([tf.reshape(tf.range(batch_size), [-1, 1]),
                  tf.reshape(b_a, [-1, 1])], axis=1)
    ) + 1e-8

    b_l = tf.cast(b_l, tf.float32)
    mu = b_adist_ * (b - b_l) * tf.math.log(tf.gather_nd(b_adist,
        b_uid))
    b_u = tf.cast(b_u, tf.float32)
    ml = b_adist_ * (b_u - b) * tf.math.log(tf.gather_nd(b_adist,
        b_lid))
    kl_divergence = tf.negative(tf.reduce_sum(mu + ml, axis=1))

    return kl_divergence
```

We also run 10^7 time steps over three random seeds on Breakout. The performance is shown in Fig. 4.8 with blue area.

References

Bellemare MG, Naddaf Y, Veness J, Bowling M (2013) The arcade learning environment: an evaluation platform for general agents. J Artif Intell Res 47:253–279

Bellemare MG, Dabney W, Munos R (2017) A distributional perspective on reinforcement learning. In: Proceedings of the 34th international conference on machine learning, vol 70, pp 449–458. JMLR.org

Castro PS, Moitra S, Gelada C, Kumar S, Bellemare MG (2018) Dopamine: a research framework for deep reinforcement learning. http://arxiv.org/abs/1812.06110

Dabney W, Ostrovski G, Silver D, Munos R (2018a) Implicit quantile networks for distributional reinforcement learning. In: International conference on machine learning, pp 1104–1113

Dabney W, Rowland M, Bellemare MG, Munos R (2018b) Distributional reinforcement learning with quantile regression. In: Thirty-second AAAI conference on artificial intelligence

DeepMind (2015) Lua/Torch implementation of DQN. https://github.com/deepmind/dqn

Fortunato M, Azar MG, Piot B, Menick J, Osband I, Graves A, Mnih V, Munos R, Hassabis D, Pietquin O, et al (2017) Noisy networks for exploration. arXiv:170610295

Hernandez-Garcia JF, Sutton RS (2019) Understanding multi-step deep reinforcement learning: a systematic study of the DQN target. In: Proceedings of the neural information processing systems (advances in neural information processing systems) workshop

Hessel M, Modayil J, Van Hasselt H, Schaul T, Ostrovski G, Dabney W, Horgan D, Piot B, Azar M, Silver D (2018) Rainbow: combining improvements in deep reinforcement learning. In: Thirty-second AAAI conference on artificial intelligence

Huber PJ (1992) Robust estimation of a location parameter. In: Breakthroughs in statistics, Springer, Berlin, pp 492–518

Lin LJ (1993) Reinforcement learning for robots using neural networks. Tech. Rep., Carnegie-Mellon Univ Pittsburgh PA School of Computer Science

Mavrin B, Yao H, Kong L, Wu K, Yu Y (2019) Distributional reinforcement learning for efficient exploration. In: International conference on machine learning, pp 4424–4434

McClelland JL, McNaughton BL, O'Reilly RC (1995) Why there are complementary learning systems in the hippocampus and neocortex: insights from the successes and failures of connectionist models of learning and memory. Psychol Rev 102(3):419

Mnih V, Kavukcuoglu K, Silver D, Rusu AA, Veness J, Bellemare MG, Graves A, Riedmiller M, Fidjeland AK, Ostrovski G, Petersen S, Beattie C, Sadik A, Antonoglou I, King H, Kumaran D, Wierstra D, Legg S, Hassabis D (2015) Human-level control through deep reinforcement learning. Nature 518(7540):529–533

O'Neill J, Pleydell-Bouverie B, Dupret D, Csicsvari J (2010) Play it again: reactivation of waking experience and memory. Trends Neurosci 33(5):220–229

Riedmiller M (2005) Neural fitted Q iteration–first experiences with a data efficient neural reinforcement learning method. In: European conference on machine learning. Springer, Berlin, pp 317–328

Roderick M, MacGlashan J, Tellex S (2017) Implementing the deep Q-network. arXiv:171107478

Schaul T, Quan J, Antonoglou I, Silver D (2015) Prioritized experience replay. In: International conference on learning representations

Sutton RS, Barto AG (2018) Reinforcement learning: an introduction. MIT Press, Cambridge

Thrun S, Schwartz A (1993) Issues in using function approximation for reinforcement learning. In: Proceedings of the 1993 Connectionist Models Summer School Hillsdale. Lawrence Erlbaum, New Jersey

Tsitsiklis J, Van Roy B (1996) An analysis of temporal-difference learning with function approximation technical. Report LIDS-P-2322) Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Tech Rep

Tsitsiklis JN, Van Roy B (1997) Analysis of temporal-difference learning with function approximation. In: Advances in Neural Information Processing Systems, pp 1075–1081

Van Hasselt H, Guez A, Silver D (2016) Deep reinforcement learning with double Q-learning. In: Thirtieth AAAI conference on artificial intelligence

Wang Z, Schaul T, Hessel M, Hasselt H, Lanctot M, Freitas N (2016) Dueling network architectures for deep reinforcement learning. In: International conference on machine learning, pp 1995–2003

Yang D, Zhao L, Lin Z, Qin T, Bian J, Liu TY (2019) Fully parameterized quantile function for distributional reinforcement learning. In: Advances in neural information processing systems, pp 6190–6199