Deep Q-Networks

Sunmook Choi felixchoi@korea.ac.kr

September 11, 2023

1 Introduction

Deep reinforcement learning is a combination of reinforcement learning and deep learning. It involves using neural networks to approximate the value function or policy in a reinforcement learning problem. Deep Q-learning have started the realm of deep reinforcement learning, and it is realized by training Deep Q-Networks (DQN). DQN is trained in a way of approximating the optimal action-value function by a neural network.

In Section 2, we describe DQN and some ways to overcome possible challenges while training. The training algorithm is based on the original Q-learning with some modifications due to such challenges. In Section 3, we describe some variants of DQN to alleviate some possible problems from the original DQN.

2 Deep Q-Networks

Basically, the algorithm of training DQN is based on Q-learning in RL. Recall the update equation of Q-learning.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{2.1}$$

The update equation of Q-learning implies that the algorithm is aimed to learn the optimal action-value function motivated by the Bellman optimality equation:

$$q_*(s, a) = \mathbb{E}_\pi \big[R_{t+1} + \gamma \max_a q_*(S_{t+1}, a) \mid S_t = s, A_t = a\big] \tag{2.2}$$

One way to find the optimal action-value function is to approximate the action-value function by a neural network, denoted by Q_θ. The network is referred to *Deep Q-Networks* (DQN). A natural way to learn the function Q_θ is first run an episode of a game, obtain a trajectory $\tau = (s_0, a_0, r_1, s_1, a_1, \ldots, s_{T-1}, a_{T-1}, r_T, s_T)$, and then minimize the following loss $L(\theta)$ using gradient

descent algorithms:

$$L(\theta) = \frac{1}{T}\sum_{t=0}^{T-1}\Big[r_{t+1} + \gamma \max_a Q_\theta(s_{t+1}, a) - Q_\theta(s_t, a_t)\Big]^2 \tag{2.3}$$

However, this optimization encounters some challenges from a deep learning perspective. Most deep learning algorithms assume the data samples to be independent. However, it is impossible in reinforcement learning because an environment is assumed to have Markov property. That is, the samples $(s_t, a_t, r_{t+1}, s_{t+1})$ in the trajectory are highly correlated, which makes the optimization of Q_θ difficult. Furthermore, the data distribution is assumed to be fixed in deep learning, while the data distribution changes as the algorithm learns new behaviors in reinforcement learning. This discrepancy can be problematic when the original deep learning algorithms are applied to reinforcement learning algorithms.

To alleviate the problems of correlated data and non-stationary distributions, the authors of the paper [2, 3] utilized the technique known as *experience replay*. Moreover, another non-trainable network, called the *target network* is used to improve the stability of optimization. We will discuss these methods in the following subsections.

2.1 Experience Replay

An experience replay mechanism is employed to train the function Q_θ mitigating high temporal correlation between samples and non-stationary data distribution problems. We first introduce a *replay memory* (also known as a *replay buffer*) $\mathcal{D}$ to store the agent's experiences $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$ with a fixed capacity N. We notice that the experiences are sampled not only from one episode but also from many episodes that the agent experiences. Then, the network Q_θ is trained from the samples $e_j \sim \mathcal{D}$ that are uniformly sampled from the replay memory $\mathcal{D}$. Let $\mathcal{B}$ be the collected samples. Then the objective becomes the follows:

$$L(\theta) = \frac{1}{|\mathcal{B}|}\sum_{e_j \in \mathcal{B}}\Big[r_{j+1} + \gamma \max_a Q_\theta(s_{j+1}, a) - Q_\theta(s_j, a_j)\Big]^2. \tag{2.4}$$

There are several advantages to apply the mechanism in a deep learning perspective. First, each step of experience $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$ is potentially used for multiple times in weight updates. Moreover, this would let the network see rare experiences repeatedly, showing better data efficiency. Second, learning directly from consecutive samples in the same episode is inefficient. It is because there are strong temporal correlations between the samples, breaking the independency assumption on samples in deep learning. Hence, sampling randomized samples from the replay memory will break these correlations, making the weight update more efficient.

The authors [2, 3] mention that the uniform sampling method from the replay memory might have some limitations. First, uniform sampling does not differentiate important transitions and gives equal importance to them. Also, samples in the replay memory are always overwrited with recent

transitions in a finite capacity. The paper [6] introduces new type of experience replay mechanism that considers priorities on transition samples in the replay memory, alleviating the former problem.

2.2 Target Network

From Eq. 2.4, notice that the target of $Q_\theta(s_i, a_i)$ is $r_{i+1} + \gamma \max_a Q_\theta(s_{i+1}, a)$. However, since the target of our training object contains the same parameter θ, the target also changes while training, which is referred to *non-stationary target problem*. This problem would make the training unstable, and we need to freeze the network in the target. Hence, the authors [3] introduce another network called the *target network*. This network, denoted by $\hat{Q}$, is a separate network which is used only for generating the targets. The target network $\hat{Q}$ has the same architecture as the agent's network Q_θ. To freeze the target network, the parameters, say $\hat{\theta}$ in $\hat{Q}$ is fixed during training. Instead, every C updates, we copy the parameters θ from Q_θ and change the target networks parameter $\hat{\theta}$ into θ. With the target network $\hat{Q}$, the final objective will be the following:

$$L(\theta) = \frac{1}{|\mathcal{B}|} \sum_{e_j \in \mathcal{B}} \left[r_{j+1} + \gamma \max_a \hat{Q}_{\hat{\theta}}(s_{j+1}, a) - Q_\theta(s_j, a_j) \right]^2. \tag{2.5}$$

2.3 Deep Q-learning Algorithm

With the utilization of experience replay and the target network, we train the network Q_θ based on Q-learning and gradient descent algorithm. We call this algorithm *Deep Q-learning*, presented in Algorithm 2.1.

3 DQN variants

Following the great performance of DQN, some variants of DQN came out to alleviate some challenges that the original DQN has. In reinforcement learning, the algorithm called Double Q-learning was introduced to reduce maximization bias. In Section 3.1, we describe the algorithm, called *Double DQN*, which takes the idea of Double Q-learning. In Section 3.2, we describe a novel network architecture, called *dueling architecture*, whose main benefit is to generalize learning across actions. In Section 3.3, a novel technique for experience replay is described, called *prioritized experience replay*, alleviating some problems related to sampling experiences uniformly and first-in-first-out replay memory.

3.1 Double DQN

The idea of Double Q-learning is to reduce overestimation bias by decoupling the selection of actions from the target evaluation. In Double Q-learning algorithm, two action-value approximators Q_1 and

Algorithm 2.1 Deep Q-learning

Initialize replay memory $\mathcal{D}$ to capacity N

Initialize action-value function Q with random weights θ

Initialize target network $\hat{Q}$ with $\hat{\theta} = \theta$

Initialize step size η

for episode$= 1, \ldots, M$ **do**

  Start an episode at a state s_0

  for $t = 0, \ldots, T-1$ **do**

    With probability ϵ, select a random action a_t, otherwise select

$$a_t = \arg\max_a Q_\theta(s_t, a)$$

    Execute an action a_t and observe reward r_{t+1} and the next state s_{t+1}

    Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in $\mathcal{D}$

    Sample random minibatch $\mathcal{B}$ of transitions $e_j = (s_j, a_j, r_{j+1}, s_{j+1})$ from $\mathcal{D}$

    Set $y_j = \begin{cases} r_{j+1} & \text{for terminal } s_{j+1} \\ r_{j+1} + \gamma \max_a \hat{Q}_{\hat{\theta}}(s_{j+1}, a) & \text{for non-terminal } s_{j+1} \end{cases}$

    Compute the loss: $L(\theta) = \dfrac{1}{2|\mathcal{B}|} \displaystyle\sum_{e_j \in \mathcal{B}} \left[y_j - Q_\theta(s_j, a_j)\right]^2$

    Compute the gradient: $\nabla_\theta L(\theta) = -\dfrac{1}{|\mathcal{B}|} \displaystyle\sum_{e_j \in \mathcal{B}} \left[y_j - Q_\theta(s_j, a_j)\right] \nabla_\theta Q_\theta(s_j, a_j)$

    Perform a gradient descent step: $\theta \leftarrow \theta - \eta \nabla_\theta L(\theta)$

    Every C steps, update the target network: $\hat{\theta} = \theta$

  end for

end for

Q_2 are defined, and one of them are chosen randomly to update from experiences as follows:

$$\begin{aligned} Q_1(S_t, A_t) &\leftarrow Q_1(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q_2 \left(S_{t+1}, \arg\max_a Q_1(S_{t+1}, a)\right) - Q_1(S_t, A_t)\right] \\ Q_2(S_t, A_t) &\leftarrow Q_2(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q_1 \left(S_{t+1}, \arg\max_a Q_2(S_{t+1}, a)\right) - Q_2(S_t, A_t)\right] \end{aligned} \tag{3.1}$$

To imitate Double Q-learning in DQN algorithm, the target network $\hat{Q}$ would be a natural candidate for the second action-value function even though the target network is not learnable. We call the resulting algorithm *Double DQN*, whose loss function is the following:

$$L(\theta) = \frac{1}{|\mathcal{B}|} \sum_{e_t \in \mathcal{B}} \left[r_{t+1} + \gamma \hat{Q}_{\hat{\theta}} \left(s_{t+1}, \arg\max_a Q_\theta(s_{t+1}, a)\right) - Q_\theta(s_t, a_t)\right]^2 \tag{3.2}$$

where $\mathcal{B}$ is a set of batch samples $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$ from the replay memory.

3.2 Dueling DQN

The paper [5] proposes new architecture for DQN algorithm, called *dueling network architecture*. The original DQN architecture takes the current state s_t as input and outputs the action-value

function values for all finitely many actions, say $Q_\theta(s_t, a_1), Q_\theta(s_t, a_2), \ldots, Q_\theta(s_t, a_K)$. The dueling network architecture also takes the current state s_t as input. Instead, there are two streams in the dueling architecture, producing two outputs, $V(s_t)$ and $A(s_t, a_1), \ldots, A(s_t, a_K)$, the approximated values of state-value function and the advantage function, respectively. Naïvely, the action-value function is computed as $Q(s_t, a) = V(s_t) + A(s_t, a)$ for all actions $a \in \mathcal{A}$, allowing the utilization of pre-existing DQN algorithms, such as DQN and Double DQN. The comparison of the original DQN and the dueling network architectures are described in Figure 3.1.

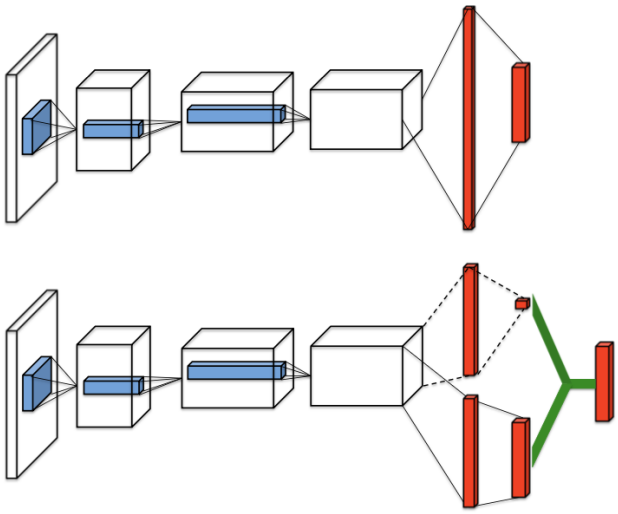

Figure 3.1: A single stream Q-network (**top**) and the dueling Q-network (**bottom**). The dueling network has two streams to separately estimate (scalar) state-value and the advantages for each action; the green output module combines them. Both networks output Q-values for each action.

The state-value output from the dueling architecture allows the model to learn which states are valuable or not, without having to learn the effect of each action for each state. This is particularly useful in states where its actions do not affect the environment in any relevant way. Consider Figure 3.2 from the paper [5]. The figure shows the state-value and advantage-value saliency maps for two different timesteps. From the saliency maps from state-value function, we can see that the state-value function focuses on the road and the horizon where new cars appear. Also, it pays attention to the score which is indicated in below. The advantage-value function, on the other hand, does not focus on the road and the horizon when there are no cars. This is because any action is not relevant to its current state. However, when there are cars in front, the advantage-value function pays attention to the cars, making its action choice very relevant.

3.2.1 Identifiability Issue

Consider the dueling network described in Figure 3.1. Let us denote the network outputs by $V(s; \theta, \beta) \in \mathbb{R}$ and $A(s, a; \theta, \alpha) \in \mathbb{R}^{|\mathcal{A}|}$. Here, θ is the parameters in the shared network, while α and β are the parameters of the two streams of fully-connected layers.

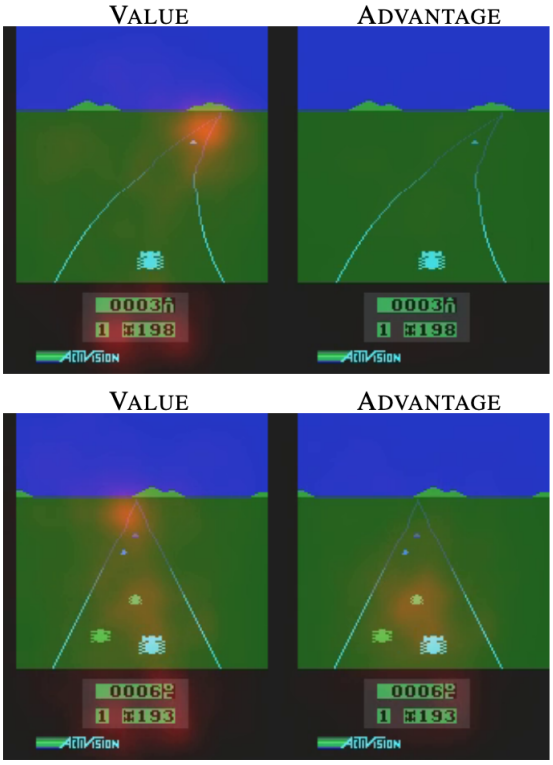

Figure 3.2: See, attend and drive: Value and advantage saliency maps (red-tinted overlay) on the Atari game Enduro, for a trained dueling architecture. The value stream learns to pay attention to the road. The advantage stream learns to pay attention only when there are cars immediately in front, so as to avoid collisions.

Now we find a way to obtain the action-value function from the dueling network. From the definition of advantage function, we might be tempted to construct the aggregating module as follows:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + A(s, a; \theta, \alpha). \tag{3.3}$$

However, the value $Q(s, a; \theta, \alpha, \beta)$ is just a parameterized estimate of true action-value function. Moreover, it would be wrong to conclude that the values $V(s; \theta, \beta)$ and $A(s, a; \theta, \alpha)$ are good estimators. This is because Eq. 3.3 is unidentifiable in the sense that the action-value $Q(s, a; \theta, \alpha, \beta)$ cannot recover the values $V(s; \theta, \beta)$ and $A(s, a; \theta, \alpha)$ uniquely.

To address this issue of identifiability, we may think of an aggregating module of state-value and advantage as follows:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + \left(A(s, a; \theta, \alpha) - \max_{a' \in \mathcal{A}} A(s, a'; \theta, \alpha) \right). \tag{3.4}$$

Notice that Eq. 3.4 is justified by the following proposition.

Proposition 1. *For the state-value function v_π and the action-value function q_π for some optimal policy π, let $\pi'(\cdot) = \arg\max_{a \in \mathcal{A}} q_\pi(\cdot, a)$ be a deterministic policy. Then π' is also an optimal policy,*

implying that $v_\pi = v_{\pi'}$ *and* $q_\pi = q_{\pi'}$*. Furthermore, we obtain that* $q_\pi(s, \pi'(s)) = v_\pi$ *and hence* $A_\pi(s, \pi'(s)) = 0$ *where* A_π *is the advantage function.*

That is, Eq. 3.4 forces the advantage function estimator to have zero advantage at the action with the best advantage value (and hence the best action-value). An alternative module replaces the max operator with an average:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + \left(A(s, a; \theta, \alpha) - \frac{1}{|\mathcal{A}|} \sum_{a' \in \mathcal{A}} A(s, a'; \theta, \alpha) \right). \tag{3.5}$$

This equation is justified by the simple relation between the state-value and action-value functions:

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) q_\pi(s, a) = \mathbb{E}_{a \sim \pi(\cdot|s)} \left[q_\pi(s, a) \right] \quad \Rightarrow \quad \mathbb{E}_{a \sim \pi(\cdot|s)} \left[A_\pi(s, a) \right] = 0. \tag{3.6}$$

Notice that the expectation is replaced by the numerical mean since we do not have access to the policy. Compared to Eq. 3.4, the aggregate module expressed in Eq. 3.5 is more stable in optimization. The stability comes from the average because the advantages only need to change as fast as the mean, instead of having to compensate any change to the optimal action's advantage in Eq. 3.4.

Notice that subtracting the mean in Eq. 3.5 does not change the relative rank of the A (and hence the action-values), preserving any greedy or ϵ-greedy policy based on the aggregated action-value function.

Moreover, it is important to note that Eq. 3.5 is viewed and implemented as part of the network and not as a separate algorithmic step. That is, the network output is the aggregated vector $Q(s, a; \theta, \alpha, \beta)$ through two streams, providing the action-value to the original DQN loss function. Hence, we do not describe the algorithm for dueling DQN because the only difference from Deep Q-learning is the way of producing the action-value function.

3.3 Prioritized Experience Replay

Let's come back to the discussion of the experience replay mechanism. It is used to address the following some challenges: strongly correlated sample updates that break the i.i.d. assumption of stochastic gradient descent algorithm, the rapid forgetting of possibly rare experiences that would be useful later on, and the change in data distribution as the agent learns from data. By using this technique, the Q-Network can be trained without having issues.

On the other hand, in order to design a replay memory, it leads to two choices: which transitions to store and which transitions to replay (and how to do so). The original experience replay method stores all transitions, and transitions are replayed by being sampled uniformly. However, sampling data from the replay memory uniformly does not differentiate important transitions, which would not be sufficiently efficient in training. The authors of [6] introduce the *prioritized experience replay* technique, which addresses the latter by introducing a way of selecting which transitions to replay.

Stochastic prioritization with TD-error Prioritized experience replay is a way to sample the ith transition with probability $P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$ measured by their TD-error. Here, the TD-error of a sample $(s_t, a_t, r_{t+1}, s_{t+1})$ is defined to be

$$\delta_i = r_{t+1} + \gamma \max_a \hat{Q}_{\hat{\theta}}(s_{t+1}, a) - Q_\theta(s_t, a_t), \tag{3.7}$$

and the priority p_i is measured in two variants. The first variant is the direct, proportional prioritization where $p_i = |\delta_i| + \epsilon$, where ϵ is a small positive number that prevents a non-zero probability of the sample with zero error (ϵ-soft). The second variant is an indirect, rank-based prioritization where $p_i = \frac{1}{rank(i)}$, where $rank(i)$ is the rank of the ith transition when the samples in the replay memory is sorted according to $|\delta_i|$. Both variants are monotonic in $|\delta|$, but the latter seems more robust as it is insensitive to outliers. Also, the heavy-tail property of the latter seems to guarantee that samples will be diverse. Furthermore, the exponent $\alpha \in [0, 1]$ in $P(i)$ determines how much prioritization is used. If the TD-error of a sample is large (so it is greater than one), then non-zero exponent α would give more priority to the sample. In contrast, if the TD-error of a sample is small (so it is less than one), then non-zero exponent α would reduce the number, giving less priority to the sample.

Annealing the bias When the sampling distribution changes, it introduces bias. Then, such a bias would change the solution that the estimates will converge to. In response, we correct this bias by using importance-sampling (IS) weights

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^\beta \tag{3.8}$$

that fully compensates for the non-uniform probabilities $P(i)$ if $\beta = 1$. These weights can be seamlessly used in Deep Q-learning algorithms, utilizing $w_i \delta_i$ instead of δ_i. For stability reasons, the IS-weights are normalized by $1/\max_i w_i$, which only downscales the update. Scaling the updates downwards reduces the gradient step for samples which have large TD-error, helping the network learn more stable. Moreover, to ensure the unbiasedness of updates at the end of training, we define a schedule on β that reaches 1 only at the end of learning. The algorithm of Double DQN with proportional prioritization is shown in Algorithm 3.1.

References

[1] Andrew Barto and Richard S. Sutton, *Reinforcement Learning: An Introduction* (2nd ed.), The MIT Press, 2018.

[2] V.Mnih, et. al, *Playing Atari with Deep Reinforcement Learning*, arXiv:1312.5602, 2013

[3] V. Mnih, et. al, *Human-level control through deep reinforcement learning*, Nature 518, 529-533, 2015.

[4] Hado van Hasselt, Arthur Guez, and David Silver, *Deep Reinforcement Learning with Double Q-learning*, AAAI 2016

Algorithm 3.1 Double DQN with proportional prioritization

Input: minibatch k, step-size η, replay period K and size N, exponents α and β.

Initialize replay memory $\mathcal{D} = \varnothing$, $\Delta = 0$, $p_1 = 1$

for episode= $1, \dots, M$ **do**
  Start an episode at a state s_0
  for $t = 0, \dots, T-1$ **do**
    With probability ϵ, select a random action A_t, otherwise select

$$a_t = \arg\max_a Q_\theta(s_t, a)$$

    Execute an action a_t and observe reward r_{t+1} and the next state s_{t+1}
    Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in $\mathcal{D}$ with maximal priority $p_t = \max_{i<t} p_i$
    if $t \equiv 0 \bmod K$ **then**
      for $j = 1, \dots, k$ **do**
        Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$
        Compute IS weight $w_j = (N \cdot P(j))^{-\beta} / \max_i w_i$
        Compute TD-error $\delta_j = r_{j+1} + \gamma \hat{Q}_{\hat{\theta}}\big(s_{j+1}, \arg\max_a Q_\theta(s_j, a)\big) - Q_\theta(s_j, a_j)$
        Update transition priority $p_j \leftarrow |\delta_j|$
        Accumulate weight-change $\Delta \leftarrow \Delta + w_j \, \delta_j \, \nabla_\theta Q_\theta(s_j, a_j)$
      end for
      Update weights $\theta \leftarrow \theta + \eta \, \Delta$, reset $\Delta = 0$
      Every C steps, update the target network: $\hat{\theta} = \theta$
    end if
  end for
end for

[5] Z. Wang, et. al, *Dueling Network Architectures for Deep Reinforcement Learning*, ICML 2016

[6] T. Schaul, et. al, *Prioritized Experience Replay*, ICLR 2016