Chapter 2 Value-based Methods

In the Sec. 1.3, we talked about Q-learning and SARSA, which are both value-based methods (i.e., try to learn the value function in the first place, and then derive a corresponding policy based on the value function). Why do we start a new chapter here on value-based methods?

Think about the game of go. There are about 10^{170} states. If we want to use Q-learning and SARSA, we have to maintain a Q-table for $|\mathcal{S}| \times |\mathcal{A}|$ size, which is impractical. In addition, in previous discussions, we have been using dynamic programming in tabular cases as an example, and have only dealt with scenarios where the state space and action space are discrete. What if we need to handle continuous cases? The solution here is instead of maintaining a table storing Q values, we want to approximate the value function $V^{\pi}(\mathbf{s})$ or state-action function $Q^{\pi}(\mathbf{s}, \mathbf{a})$. Then, to represent functions, neural networks come into play.

2.1 Value Function Approximation

Using a Q-table has two main limitations:

1. It requires that we visit every reachable state many times and apply every action many times to get a good estimate of $Q(\mathbf{s}, \mathbf{a})$.Thus, if we never visit a state $\mathbf{s}$, we have no estimate of $Q(\mathbf{s}, \mathbf{a})$, even if we have visited states that are very similar to $\mathbf{s}$.
2. It requires us to maintain a table of size $|\mathcal{S}| \times |\mathcal{A}|$ size, which is prohibitively large for any non-trivial problem.

To get around these we will look at how to use machine learning to approximate Q-functions. Instead of calculating an exact Q-function, we approximate it using simple methods that both eliminate the need for a large Q-table (therefore the methods scale better), and also allowing use to provide reasonable estimates of $Q(\mathbf{s}, \mathbf{a})$, even if we have not applied action $\mathbf{a}$ in state $\mathbf{s}$ previously.

There are a variety of methods other than neural nets to approximate a function, e.g. linear combinations of features, decision trees, nearest neighbor, Fourier / wavelet bases, etc.. They have their own unique properties, for example Fourier transformation would be particularly useful if you care about frequency information. But here we would like to care more about the differentiable ones. So in particular, we will look at linear function approximation and approximation using deep learning (deep Q-learning).

Linear Function Approximation

In linear Q-learning, we store features and weights, not states. What we need to learn is how important each feature is (its weight) for each action.

To represent this, we have two vectors:

1. A *feature vector* $f(s, a)$, which is a vector of $n \cdot |A|$ different functions, where n is the number of state features and $|A|$ the number of actions. Each function extracts the value of a feature for state-action pair (s, a). We say $f_i(s, a)$ extracts the i-th feature from the state-action pair (s, a) :
$$f(s, a) = \begin{pmatrix} f_1(s, a) \\ f_2(s, a) \\ \cdots \\ f_{n \times |A|}(s, a) \end{pmatrix}$$
2. A *weight vector* w of size n $\times|A|$: one weight for each feature-action pair. w_i^a defines the weight of a feature i for action a.

Defining State-Action Features

Often it is easier to just define features for states, rather than state-action pairs. The features are just a vector of n functions of the form $f_i(s)$. It is straightforward to construct $n \times |A|$ state-pair features from just n state features:

$$f_{i,k}(s,a) = \begin{cases} f_i(s) & \text{if } a = a_k \\ 0 & \text{otherwise } 1 \le i \le n, 1 \le k \le |A| \end{cases} \tag{2.1.1}$$

This effectively results in $|A|$ different weight vectors:

$$f(s,a_1) = \begin{pmatrix} f_{1,a_1}(s,a) \\ f_{2,a_1}(s,a) \\ 0 \\ 0 \\ 0 \\ 0 \\ \dots \end{pmatrix} \quad f(s,a_2) = \begin{pmatrix} 0 \\ 0 \\ f_{1,a_2}(s,a) \\ f_{2,a_2}(s,a) \\ 0 \\ 0 \\ \dots \end{pmatrix} \quad f(s,a_3) = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ f_{1,a_3}(s,a) \\ f_{2,a_3}(s,a) \\ \dots \end{pmatrix} \dots \tag{2.1.2}$$

Q-values from linear Q-functions

Give a feature vector f and a weight vector w, the Q-value of a state is a simple linear combination of features and weights:

$$\begin{aligned} Q(s,a) &= f_1(s,a) \cdot w_1^a + f_2(s,a) \cdot w_2^a + \dots + f_n(s,a) \cdot w_n^a \\ &= \sum_{i=0}^{n} f_i(s,a) w_i^a \end{aligned} \tag{2.1.3}$$

Linear Q-function Update

To use approximate Q-functions in reinforcement learning, there are two steps we need to change from the standard algorithsm: (1) initialisation; and (2) update.

For initialisation, initialise all weights to 0. Alternatively, you can try Q-function initialisation and assign weights that you think will be "good" weights.

For update, we now need to update the weights instead of the Q-table values. The update rule is now:

$$\text{For each state-action feature,} \quad w_i^a \leftarrow w_i^a + \alpha \cdot \delta \cdot f_i(s,a) \tag{2.1.4}$$

where δ depends on which algorithm we are using; e.g. Q-learning, SARSA. As this is linear, it is therefore convex, so the weights will converge. In the next section we will show how to merge this value approximation method into Q learning algorithm.

There might be something wrong with the algorithm here. What's the difference between Q iteration and Q learning? Besides, let's standardize the notation used in the algorithms (compared to the way used in Q learning and SARSA in previous chapter).

2.2 Value Functions in Theory

This part would be kind of aside the main discussion and a little bit *maths*. In previous discussions we mentioned that value functions does not have any convergence guarantees that it will ultimately find the optimal solution. Here we will find out more about this theoretically.

2.2.1 Value iteration theory

Let's begin with value iteration. In value iteration we have step 1, set $Q(\mathbf{s},\mathbf{a}) \leftarrow r(\mathbf{s},\mathbf{a}) + \gamma E[V(\mathbf{s}')]$ and step 2, set $V(\mathbf{s}) \leftarrow \max_{\mathbf{a}} Q(\mathbf{s},\mathbf{a})$. What we care about here is does this algorithm actually converge? And if so, what does it converge to?

Here we introduce an operator called Bellman operator $\mathcal{B}$ (in a more elegant matrix form):

$$\mathcal{B}V = \max_{\mathbf{a}} r_{\mathbf{a}} + \gamma \mathcal{T}_{\mathbf{a}} V \tag{2.2.1}$$

where V stands for the values of states, which is a vector that has the same dimension $\mathcal{S}$ as the state space, $r_{\mathbf{a}}$ is a stacked vector of reward at all states for an action $\mathbf{a}$ and $\mathcal{T}_{\mathbf{a}}$ is a $\mathcal{S} \times \mathcal{S}$ matrix, a matrix of transition for action $\mathbf{a}$ such that $\mathcal{T}_{\mathbf{a},i,j} = p\left(\mathbf{s}' = i \mid \mathbf{s} = j, \mathbf{a}\right)$.

One interesting property we can show is that $V^\star$ is *fixed point* of $\mathcal{B}$, which means that:

$$V^\star(\mathbf{s}) = \max_{\mathbf{a}} r(\mathbf{s}, \mathbf{a}) + \gamma E\left[V^\star\left(\mathbf{s}'\right)\right], \text{ so } V^\star = \mathcal{B}V^\star \tag{2.2.2}$$

Furthermore, $V^\star$ always exists, always unique, and it always corresponds to the optimal policy, the policy that maximize total rewards. The only problem is if we can actually reach this fixed point. Here we only show a high level sketch of proving why it converges here. We can prove that value iteration converges to $V^\star$ because $\mathcal{B}$ is a *contraction* (see the previous Sec 1.2.5 for proof). Recall that by contraction we mean that:

$$\text{for any } V \text{ and } \bar{V}, \text{ we have } \|\mathcal{B}V - \mathcal{B}\bar{V}\|_\infty \leq \gamma \|V - \bar{V}\|_\infty \tag{2.2.3}$$

So if we choose $V^\star$ as $\bar{V}$, given $\mathcal{B}V^\star = V^\star$ we have:

$$\|\mathcal{B}V - V^\star\|_\infty \leq \gamma \|V - V^\star\|_\infty \tag{2.2.4}$$

In the rest of the notes, we will use $V^\star$ to represent the value function under the optimal policy $\pi^\star$.

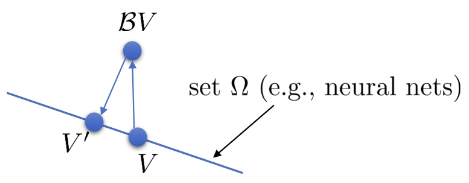
2.2.2 Fitted value iteration theory

In tabular value iteration we have one single step: $V \leftarrow \mathcal{B}V$, which represents setting $Q(\mathbf{s}, \mathbf{a}) \leftarrow r(\mathbf{s}, \mathbf{a}) + \gamma E\left[V\left(\mathbf{s}'\right)\right]$ and setting $V(\mathbf{s}) \leftarrow \max_{\mathbf{a}} Q(\mathbf{s}, \mathbf{a})$. Now we move on to non-tabular cases, which is the fitted value iteration. We will find out how to deal with the second step, which is the supervised learning part: $\phi \leftarrow \arg\min_\phi \frac{1}{2} \sum_i \left\|V_\phi\left(\mathbf{s}_i\right) - \mathbf{y}_i\right\|^2$

If we look at what step two actually does mathematically, one way to think of supervised learning is that: you have a set of value functions Ω by going through all possible weights of each neuron in the neural network. In supervised learning we sometimes refer to this as hypothesis set. Here the target value, or the updated value function is $\mathcal{B}V$ which we get from the first step. Therefore we can think of the entire fitted value iteration as:

$$V' \leftarrow \arg\min_{V' \in \Omega} \frac{1}{2} \sum \left\|V'(\mathbf{s}) - (\mathcal{B}V)(\mathbf{s})\right\|^2 \tag{2.2.5}$$

We can intuitively think of this procedure as projection of $\mathcal{B}V$, to Ω



To mathematically describe this operation we define a new operator Π, which is a projection onto Ω (in terms of ℓ_2 norm)

$$\Pi V = \arg\min_{V' \in \Omega} \frac{1}{2} \sum \left\|V'(\mathbf{s}) - V(\mathbf{s})\right\|^2 \tag{2.2.6}$$

It's easy to see that $\mathcal{B}$ is a contraction w.r.t. ∞-norm (max norm), and that Π is a contraction w.r.t. ℓ_2 norm (Euclidean distance):

$$\begin{aligned} \|\mathcal{B}V - \mathcal{B}\bar{V}\|_\infty &\leq \gamma \|V - \bar{V}\|_\infty \\ \|\Pi V - \Pi\bar{V}\|^2 &\leq \|V - \bar{V}\|^2 \end{aligned} \tag{2.2.7}$$

But unfortunately $\Pi\mathcal{B}$ is not a contraction of any kind.

This means that value iteration (tabular case) always converges, but in fitted value iteration it does not converge in general, and often not in practice.

2.2.3 Fitted Q-iteration theory

In Q-iteration it's actually the same. All we have to do is to make a tiny adjustment to operator $\mathcal{B}$.

$$\mathcal{B}Q = r + \gamma \mathcal{T} \max_{\mathbf{a}} Q \tag{2.2.8}$$

Note that this time max comes after the transition operator.

The following stuff are all the same: $\mathcal{B}$ is a contraction w.r.t. ∞-norm (max norm), Π is a contraction w.r.t. ℓ_2 norm (Euclidean distance), $\Pi\mathcal{B}$ is not a contraction of any kind.

This also applies to online Q-learning and basically any algorithms of this sort. Specifically, the actor-critic algorithm also does not ensures to converge for the same reason. We will try to solve the convergence problem in next chapter.

2.3 DQN: Deep Learning with Q-Functions

Before diving into Deep Q-Network (DQN), let's take a moment to look back on what we've done.

In Sec 1.3.3, we've shown that Q learning and SARSA both work under the same principle:

$$Q(s,a) = R(s,a) + \gamma\mathbb{E}[V(s')] = R(s) + \gamma \sum_{s'} T(s,a,s') V(s') \tag{2.3.1}$$

And in Q learning we use $\max_{a'} Q(s',a')$ to approximate $\mathbb{E}[V(s')]$, while in SARSA $\mathbb{E}[V(s')]$ is approximated online by $Q(s_{n+1}, a_{n+1})$, which is given by the *next actual state and action.*

2.3.1 Fitted Q-learning and online Q-learning

Now move on to Q-learning with value approximation, we are going to substitute Q by Q_ϕ, where we use a certain method with parameters ϕ to approximate the Q function, given by the following:

$$Q_{\phi'}(s,a) \leftarrow Q_\phi(s,a) + \alpha\left[R(s,a) + \gamma \max_{a'} Q(s',a') - Q_\phi(s,a)\right] \tag{2.3.2}$$

Or it can be rewritten as:

$$Q_{\phi'}(s,a) \leftarrow (1-\alpha)Q_\phi(s,a) + \alpha\underline{\left[R(s,a) + \gamma \max_{a'} Q(s',a')\right]} \tag{2.3.3}$$

And then we use a network that takes in s and a, and outputs $Q_\phi(s,a)$ to do the approximation: for a bunch of pre-collected data $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$, set $\mathbf{y}_i \leftarrow r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'_i} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$ and then fit the following:

$$\phi \leftarrow \arg\min_\phi \frac{1}{2}\sum_i \left\|Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i\right\|^2 \tag{2.3.4}$$

This is actually the fitted Q learning algorithm.

Fitted Q-learning

Algorithm 8 Fitted Q learning algorithm

```
1: repeat
2:     collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy
3:     repeat
4:         set $\mathbf{y}_i \leftarrow r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'_i} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
5:         set $\phi \leftarrow \arg\min_\phi \frac{1}{2}\sum_i \|Q_\phi(\mathbf{s}_i) - \mathbf{y}_i\|^2$
6:     until convergence
7: until convergence
```

Insight. Note that fitted Q-learning algorithm is an *off-policy* algorithm, which means that you could reuse samples from previous iterations. The policy affects the iteration step only in $\underline{\max_{\mathbf{a}'_i} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)}$. This is because intuitively $\max_{\mathbf{a}'_i} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i) = Q_\phi\left(\mathbf{s}'_i, \arg\max_{\mathbf{a}'_i} Q_\phi\right)$, and $\arg\max_{\mathbf{a}'_i} Q_\phi$ is actually the policy itself since we are following a greedy policy:

$$\pi'(\mathbf{a}_t \mid \mathbf{s}_t) = \begin{cases} 1 \text{ if } \mathbf{a}_t = \arg\max_{\mathbf{a}_t} Q^\pi(\mathbf{s}_t, \mathbf{a}_t) \\ 0 \text{ otherwise} \end{cases} \tag{2.3.5}$$

So one way to think of fitted Q learning is that you have a big bucket of different transitions $((\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i))$. And this is off-policy because transitions are independent of policy π.

The underlined part of taking max in line 3 of Alg. 8 is also why fitted Q learning can indeed improve the policy. While in step three it is actually the fitting error. So you could think of fitted Q learning as optimizing an error:

$$\mathcal{E} = \frac{1}{2} E_{(\mathbf{s},\mathbf{a})\sim\beta} \left[\left(Q_\phi(\mathbf{s},\mathbf{a}) - \left[r(\mathbf{s},\mathbf{a}) + \gamma \max_{\mathbf{a}'} Q_\phi\left(\mathbf{s}',\mathbf{a}'\right) \right] \right)^2 \right] \tag{2.3.6}$$

But of course the error itself does not reflect the goodness of your policy. It's just the accuracy with which we are able to copy the target values. If the error is zero, then

$$Q_\phi(\mathbf{s},\mathbf{a}) = r(\mathbf{s},\mathbf{a}) + \gamma \max_{\mathbf{a}'} Q_\phi\left(\mathbf{s}',\mathbf{a}'\right) \tag{2.3.7}$$

This is an optimal Q-function, corresponding to the optimal policy π'. However most guarantees of convergence are lost when we leave the tabular case (e.g., use neural networks).

Online Q-learning

We can make some adjustment and get a different version of Q-learning algorithm that looks more similar to what we've encountered in Sec. 1.3.3:

Algorithm 9 Online Q learning algorithm

```
1: repeat
2:     take one action $\mathbf{a}_i$ and observe one transition $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$
3:     set $\mathbf{y}_i = r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
4:     set $\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i)$
5: until convergence
```

Insight. This is *online* because it sample data while iterating. And it's also an *off-policy* algorithm since it does not rely on the latest policy to sample data.

Line 4 in Alg. 9 might look like policy gradient (to be explained later in Sec. 3.1), but actually if we substitute $\mathbf{y}_i$ with what it really means we would see the difference:

$$\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i) \left(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \left[r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i) \right] \right) \tag{2.3.8}$$

The problem is that Q_ϕ shows up in two places, and there's no gradient in the second term. So that online Q-learning algorithm *does not guarantee to converge.*

There is actually another problem with online Q-learning algorithm, which is the *correlated samples* (the underlined part). In reality, states in time step $t+1$ are quite similar to states at time step t because changes in states are generally continuous, which means that sequential states are strongly correlated. Besides, the target value is always changing, so that one sample is really hard to train the network.

2.3.2 Q-Learning with replay buffers

Now we will try to solve the aforementioned two problems (the sample correlation problem and the convergence problem), during which process we actually move from *fitted Q-learning* to *deep Q-learning.*

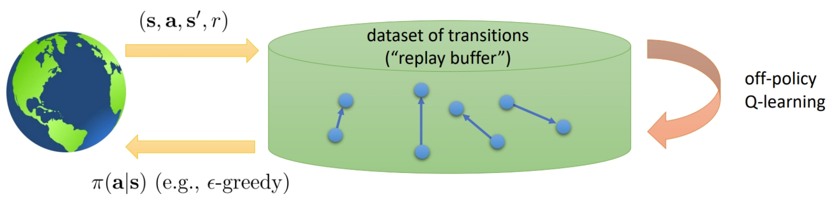
We'll start from the dataset problem first. Other than using parallel workers, another way of solving it is to use **replay buffers**. A replay buffer is literally a buffer that stores the dataset of transitions. Here is the algorithm of Q-learning with a replay buffer.

Algorithm 10 Q-learning with replay buffers

```
1: repeat
2:     collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy, add it to $\mathcal{B}$
3:     repeat
4:         sample a batch $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$ from $\mathcal{B}$
5:         set $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)])$
6:     until convergence
7: until convergence
```

Instead of collect dataset from real world, it simply samples a batch from the replay buffer $\mathcal{B}$. Therefore the samples are no longer correlated, but i.i.d.. We still do not have a real gradient yet, but at least we solved the correlated problem.

But where does the data come from? Actually what we need to do is to *periodically feed the replay buffer*. Otherwise if we use a fixed replay buffer produced by an initial policy, chances are that the policy is bad and does not cover any interesting regions of states. The entire workflow would look like the figure below.


$(\mathbf{s}, \mathbf{a}, \mathbf{s}', r)$
dataset of transitions
("replay buffer")
off-policy
Q-learning
$\pi(\mathbf{a}|\mathbf{s})$ (e.g., ϵ-greedy)

By adding a replay buffer, the samples are no longer correlated, and we also keep updating the replay buffer by adding new real-world transitions. Note that in this algorithm (Alg. 10), we would commonly repeat the inner loop for only one time, even though repeating for more times would be more efficient.

2.3.3 Q-Learning with target networks

Now solve the remaining problem: Q-learning is not gradient descent, and in particular, there's no gradient through target value, which make it hard to converge.

We can make a comparison between the iteration step of Q-learning with replay buffer and fitted Q-learning:

$$\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}\left(\mathbf{s}_i, \mathbf{a}_i\right)\left(Q_\phi\left(\mathbf{s}_i, \mathbf{a}_i\right) - \left[r\left(\mathbf{s}_i, \mathbf{a}_i\right) + \gamma \max_{\mathbf{a}'} Q_\phi\left(\mathbf{s}_i', \mathbf{a}_i'\right)\right]\right) \quad \text{Q-learning with replay buffer} \tag{2.3.9}$$

$$\phi \leftarrow \arg\min_\phi \frac{1}{2}\sum_i \left\| Q_\phi\left(\mathbf{s}_i, \mathbf{a}_i\right) - \left[r\left(\mathbf{s}_i, \mathbf{a}_i\right) + \gamma \max_{\mathbf{a}'} Q_\phi\left(\mathbf{s}_i', \mathbf{a}_i'\right)\right]\right\|^2 \quad \text{full fitted Q-learning} \tag{2.3.10}$$

In fitted Q-learning it performs what looks like supervised learning, which is a perfectly well-defined and stable regression. While in Q-learning with replay buffer it has a *moving target*. A target network would be something in between.

Algorithm 11 Q-learning with replay buffer and target network

```
1: repeat
2:   save target network parameters $\phi' \leftarrow \phi$
3:   for N times do
4:     collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}_i', r_i)\}$ using some policy, add it to $\mathcal{B}$
5:     for K times do
6:       sample a batch $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}_i', r_i)$ from $\mathcal{B}$
7:       set $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_{\phi'}(\mathbf{s}_i', \mathbf{a}_i')])$
8: until convergence
```

Note that *targets don't change in the inner loop!* This (from line 3 to line 9) is purely a supervised learning problem. Parameter K would typically be $1-4$, while N would be about $10,000$.

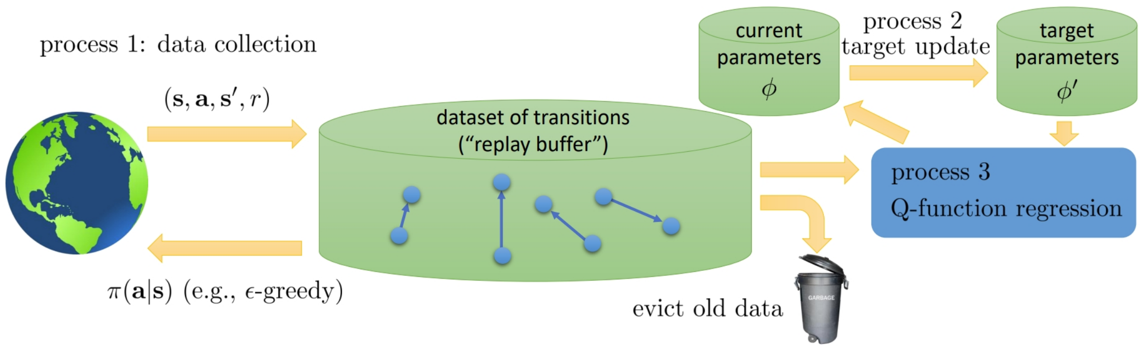
The classic Deep Q-Learning algorithm looks almost the same, simply choosing $K = 1$:

Algorithm 12 Deep Q-Learning

```
1: repeat
2:     take some action $\mathbf{a}_i$ and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$, add it to $\mathcal{B}$
3:     sample mini-batch $\{\mathbf{s}_j, \mathbf{a}_j, \mathbf{s}'_j, r_j\}$ from $\mathcal{B}$ uniformly
4:     compute $y_j = r_j + \gamma \max_{\mathbf{a}'_j} Q_{\phi'}(\mathbf{s}'_j, \mathbf{a}'_j)$ using target network $Q_{\phi'}$
5:     $\phi \leftarrow \phi - \alpha \sum_j \frac{dQ_\phi}{d\phi}(\mathbf{s}_j, \mathbf{a}_j)(Q_\phi(\mathbf{s}_j, \mathbf{a}_j) - y_j)$
6:     update $\phi'$: copy $\phi$ every $N(\approx 10000)$ steps
7: until Convergence
```

2.3.4 Comparison

We could view the three different Q-learning algorithms (fitted Q-iteration (the purely regression-based one), online Q-learning, deep Q-learning) in a more general way.


process 1: data collection
$(\mathbf{s}, \mathbf{a}, \mathbf{s}', r)$
$\pi(\mathbf{a}|\mathbf{s})$ (e.g., ϵ-greedy)
dataset of transitions
("replay buffer")
current parameters ϕ
process 2 target update
target parameters ϕ'
process 3
Q-function regression
evict old data

As is shown in the figure above, There are three different steps in the algorithm, and several important concepts:

- **replay buffer**: it is actually the dataset of transition, dipicting the transition model of the real world interaction. It is also the foundation of how we improve our policy.
- **data collection**: the data in replay buffer comes from data collection process, where we interact with the real world following some policy (e.g. $\epsilon-$greedy) and bring back one or more transtions. It can be viewed as a continuous process that is always running, and we call it **process 1**.
- **evict old data**: a replay buffer has a finite size, and we have to periodically discard old data. A simple and reasonable way is to structure the replay buffer as a circular buffer. When the buffer is full, the oldest data is overwritten with the newest data.
- **parameters** ϕ: these are the parameters you use to compute the target value. Current parameters ϕ are the parameters that you are going to give to process 1 to construct the $\epsilon-$greedy policy to collect more data.
- **parameters update**: parameters update is typically a very slow process which we call **process 2**. We typically run it very infrequently.
- **Q-function regression**: in this process we load a batch of transitions from the replay buffer, and the target parameters ϕ'. Then we use the target parameters to calculate target values for each of the transitions, regress onto the Q-function, and update the current parameters. We call this procedure **process 3**.

In fitted Q-iteration algorithm, process 3 is in the inner loop of process 2, which is in the inner loop of process 1. In online Q-learning, we evict the old transitions immediately, and process 1, 2, and 3 run at the same speed. In DQN, process 1 and 3 run at the same speed (they are fairly decoupled), but process 2 runs slower. And the replay buffer is quite big.

2.4 Improving DQN

2.4.1 Double DQN(DDQN)

In this section we are going to discuss some practical considerations we need to take while implementing Q-learning algorithm. To start with, as a prediction of total reward from taking $\mathbf{a}_t$ in $\mathbf{s}_t$, Q values are actually inaccurate. Specifically, the estimated Q value is systematically *larger* than the true value, which is often referred to as **overestimation** in Q-learning. The problem lies in the target value:

$$\text{target value:} \quad y_j = r_j + \underline{\gamma \max_{\mathbf{a}_j'} Q_{\phi'}\left(\mathbf{s}_j', \mathbf{a}_j'\right)} \tag{2.4.1}$$

The underlined part of taking max is the problem. Imagine we have two random variables: X_1 and X_2. Then $E\left[\max\left(X_1, X_2\right)\right] \geq \max\left(E\left[X_1\right], E\left[X_2\right]\right)$ (Jensen's inequality). This is because while taking max you are actually picking those with larger noise. Similarly $Q_{\phi'}\left(\mathbf{s}_j', \mathbf{a}_j'\right)$ can be viewed as a true value plus a noise, hence $\max_{\mathbf{a}_j'} Q_{\phi'}\left(\mathbf{s}_j', \mathbf{a}_j'\right)$ overestimates the next value by always selecting the noisy in positive direction.

One way of fixing this problem is called **double Q-learning**.

Note that:

$$\max_{\mathbf{a}'} Q_{\phi'}\left(\mathbf{s}', \mathbf{a}'\right) = Q_{\phi'}\left(\mathbf{s}', \arg\max_{\mathbf{a}'} Q_{\phi'}\left(\mathbf{s}', \mathbf{a}'\right)\right) \tag{2.4.2}$$

We are *selecting action* according to $Q_{\phi'}$, while also *calculating values* using the same function $Q_{\phi'}$. If the noise in these two procedures are de-correlated, then the problem could be alleviated (but not completely eliminated). The idea here is not to use the same network of choosing actions and estimating values. Double Q-learning uses two networks:

$$\begin{aligned} Q_{\phi_A}(\mathbf{s}, \mathbf{a}) &\leftarrow r + \gamma Q_{\phi_B}\left(\mathbf{s}', \arg\max_{\mathbf{a}'} Q_{\phi_A}\left(\mathbf{s}', \mathbf{a}'\right)\right) \\ Q_{\phi_B}(\mathbf{s}, \mathbf{a}) &\leftarrow r + \gamma Q_{\phi_A}\left(\mathbf{s}', \arg\max_{\mathbf{a}'} Q_{\phi_B}\left(\mathbf{s}', \mathbf{a}'\right)\right) \end{aligned} \tag{2.4.3}$$

In practice, we simply use the current and target network to implement these two networks. In standard Q-learning, we have $y = r + \gamma Q_{\phi'}\left(\mathbf{s}', \arg\max_{\mathbf{a}'} Q_{\phi'}\left(\mathbf{s}', \mathbf{a}'\right)\right)$. While in double Q-learning, we just use the *current* network (not the target network) to *select actions* and still use the *target* network to *evaluate values*, namely $y = r + \gamma Q_{\phi'}\left(\mathbf{s}', \arg\max_{\mathbf{a}'} Q_{\phi}\left(\mathbf{s}', \mathbf{a}'\right)\right)$. Here is the algorithm:

Algorithm 13 DDQN

```
Require: $Q_\phi$ is the current network, $Q_{\phi'}$ is the target network, $\mathcal{B}$ is the replay buffer
1: repeat
2:     take some action $\mathbf{a}_i$ and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}_i', r_i)$, add it to $\mathcal{B}$
3:     sample mini-batch $\left\{\mathbf{s}_j, \mathbf{a}_j, \mathbf{s}_j', r_j\right\}$ from $\mathcal{B}$
4:     compute $y_j = r_j + \gamma Q_{\phi'}\left(\mathbf{s}', \arg\max_{\mathbf{a}'} Q_\phi\left(\mathbf{s}', \mathbf{a}'\right)\right)$
5:     $\phi \leftarrow \phi - \alpha \sum_j \frac{dQ_\phi}{d\phi}\left(\mathbf{s}_j, \mathbf{a}_j\right)\left(Q_\phi\left(\mathbf{s}_j, \mathbf{a}_j\right) - y_j\right)$
6:     update $\phi'$: copy $\phi$ to $\phi'$ every $N(\approx 10000)$ steps
7: until Convergence
```

Take a look at Van Hasselt et al. (2016) for more details.

2.4.2 Using Multi-step Returns

There is another trick we can use called **Multi-step returns**, which is quite similar to N-step return when we first learn about time difference evaluation in section 1.3.2

The Q-learning target value consists of two parts, one is the reward of current time-step, the other is the estimated reward of next time-step given Q_ϕ'. At the beginning, Q_ϕ' is bad and the first term is the only values that matters (so we are not learning much about the Q-function), while when the policy becomes better, the second term would be more important. This is somehow similar to the actor-critic:

$$\text{Actor-critic:} \quad \nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta\left(\mathbf{a}_{i,t} \mid \mathbf{s}_{i,t}\right)\left(\underline{r\left(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}\right) + \gamma \hat{V}_\phi^\pi\left(\mathbf{s}_{i,t+1}\right)} - \hat{V}_\phi^\pi\left(\mathbf{s}_{i,t}\right)\right) \tag{2.4.4}$$

In actor-critic algorithm, to leverage the bias and variance trade-off in policy gradient, we can end the trajectory earlier, and only count the reward summed up to N steps from now. The way we construct multi-step return (N-step return estimator) is similar to that in actor-critic:

$$y_{j,t} = \sum_{t'=t}^{t+N-1} \gamma^{t-t'} r_{j,t'} + \gamma^N \max_{\mathbf{a}_{j,t+N}} Q_{\phi'}\left(\mathbf{s}_{j,t+N}, \mathbf{a}_{j,t+N}\right) \tag{2.4.5}$$

This adjustment would make the target value *less biased* when the Q function is inaccurate, and allows for typically faster learning especially at early on stage.

One subtle problem with this solution is that the learning process suddenly becomes *on-policy*, so we cannot efficiently make use of the off-policy data. Why is it on-policy? If we look at the summation of the rewards, we are collecting the rewards data using a certain trajectory, which is generated by a specific policy.

Add reference for *Safe and efficient off-policy reinforcement learning.*

To fix this problem, here are three possible solutions:

1. You can simply *ignore* this mismatch, which somehow works very well in practice.
2. You can cut the trace by dynamically adapting N to get only on-policy data. (This only works well when data is mostly on-policy, and that action space is small.)
3. You can use **importance sampling** (more on this later).

2.4.3 Prioritized Experience Replay

Note that the samples saved in the replay buffer are used randomly, and this is intuitively inefficient, since we don't want to waste computation on the samples that are not informative. A method called **prioritized experience replay (PER)** (Schaul et al., 2015) solves this issue by assigning weights to the samples so that "important" ones are drawn more frequently for training.

Based on this idea, we would sample with respect to some mysterious function $f\left((s_t, a_t, r_t, s_{t+1})\right)$ that exactly tells us the *"usefulness"* of samples, for fastest learning to get maximum reward. An ideal option for this magic function is TD error. As a recap, in Q learning the magnitude of the TD error (squared) is what we want to minimize according to the Bellman equation (See sec.1.3.2 if you forget what is TD error).

The TD error for vanilla DQN is:

$$\delta_i = r_t + \gamma \max_{a\in\mathcal{A}} Q_{\theta^-}\left(s_{t+1}, a\right) - Q_\theta\left(s_t, a_t\right) \tag{2.4.6}$$

And for Double DQN it would be:

$$\delta_i = r_t + \gamma Q_{\theta^-}\left(s_{t+1}, \operatorname{argmax}_{a\in\mathcal{A}} Q_\theta\left(s_{t+1}, a\right)\right) - Q_\theta\left(s_t, a_t\right) \tag{2.4.7}$$

Then naturally we link between TD error and priority:

$$p_i = |\delta_i| + \epsilon \tag{2.4.8}$$

where ϵ is a small constant ensuring that the sample has some non-zero probability of being drawn. Then the probability distribution would be:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{2.4.9}$$

where α determines the level of prioritization. If $\alpha \to 0$, then there is no prioritization, because all$p_i^\alpha = 1$. If $\alpha \to 1$, then we get to, in some sense, "full" prioritization, where sampling data points is more heavily dependent on the actual TD error values.

Note that there is one more technical issue to be solved: Prioritized replay introduce bias because it does not sample experiences uniformly at random due to the sampling proportion corresponding to TD error. We need to correct this bias by using importance sampling weights:

$$w_i = \left(\frac{1}{N}\frac{1}{P(i)}\right)^\beta \tag{2.4.10}$$

where N is the size of the replay buffer and β is a hyperparameter that controls the degree of importance sampling. w_i fully compensates for the non-uniform probability if $\beta = 1$. For stability, we always normalize weights by $max(w_i)$.

$$w_i = \frac{w_i}{\max(w_i)} \tag{2.4.11}$$

In practice, we don't update the priorities of the transitions in the entire replay buffer. Instead, we store a new transition to the transition buffer with the maximum priority and update the priorities of the sampled transitions in the replay buffer after each update.

2.4.4 Dueling Networks

The goal of Dueling DQN is to have a network that separately computes the advantage and value functions, and combines them back into a single Q-function only at the final layer:

- Action-independent value function: $V(s;\theta,\beta)$
- Action-dependent advantage function: $A(s,a;\theta,\alpha)$

Combining these two together, we have:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + A(s,a;\theta,\alpha) \tag{2.4.12}$$

However, naively adding these two values together can be problematic in two aspects:

1. It is problematic to assume that V and A gives reasonable estimates of the state-value and action advantages, respectively. The estimates given by these components, despite being parts of the parameterized Q-function, may not accurately represent the true state-value or advantage function.
2. The naive sum of the two is "unidentifiable," in that given the Q value, we cannot recover the V and A uniquely. It is empirically shown that this lack of identifiability leads to poor practical performance.

Therefore, the last module of the neural network implements forward mapping shown below:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \max_{a'\in|\mathcal{A}|} A(s,a;\theta,\alpha)\right) \tag{2.4.13}$$

which will force the Q value for the maximizing action to equal V, solving the identifiability issue. The network structure is visualized in figure 2.1:

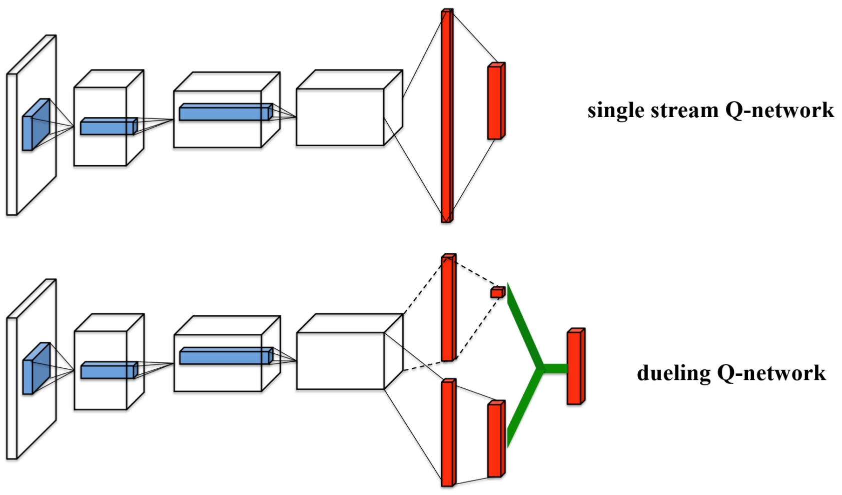


Figure 2.1: Dueling Networks

It may seem somewhat pointless to do this at first glance. Why decompose a function that we will just put back together?

Actually the advantage of this approach lies in the fact that it helps the network to generalize learning across different actions more effectively. By decoupling the value and advantage, the network can learn which states are valuable without being influenced by the action advantages, and at the same time, it can learn which actions are beneficial without being affected by the differences in value between states.

The more prosaic explanation is that the function decomposition is always technically correct (for an MDP). Coding the network like this incorporates *known structure of the problem* into the network, which otherwise it may have to spend resources on learning. So it's a way of injecting the designer's knowledge of reinforcement learning problems into the architecture of the network.

2.5 Q-Learning with Continuous Actions

In this section, we will introduce two methods for Q-learning with continuous action space.

2.5.1 Deep Deterministic Policy Gradients

To begin with, let's see why we need DDPG for continuous actions planning. Here is a concrete example: we are trying to solve the classic Inverted Pendulum control problem. In this setting, we can take only two actions: swing left or swing right. What make this problem challenging for Q-Learning Algorithms is that actions are continuous instead of being discrete. That is, instead of using two discrete actions like -1 or +1, we have to select from infinite actions in a given range. In DQN we have the greedy policy assumption, so that in target network you have to do the 'max-trick':

$$\pi\left(\mathbf{a}_t \mid \mathbf{s}_t\right)=\begin{cases}1 \text{ if } \mathbf{a}_t=\arg\max_{\mathbf{a}_t} Q_\phi\left(\mathbf{s}_t, \mathbf{a}_t\right)\\ 0 \text{ otherwise}\end{cases} \tag{2.5.1}$$

$$y_j=r_j+\gamma \max_{\mathbf{a}_j'} Q_{\phi'}\left(\mathbf{s}_j', \mathbf{a}_j'\right) \tag{2.5.2}$$

But turning to continuous action space, it's no longer possible to perform the $\arg\max$.

There are several solutions to performing the max:

Option 1: sample and optimize

The first option is to use optimization techniques. Gradient based optimization (e.g., SGD) involves multiple steps of optimization and would be a bit slow in the inner loop. It turns out that stochastic optimization would be a better option since action space here is typically low-dimensional.

In stead of giving a close form solution of optimization on continuous action space, a deadly simple way is to sample $(\mathbf{a}_1, \ldots, \mathbf{a}_N)$ from some distribution:

$$\max_{\mathbf{a}} Q(\mathbf{s}, \mathbf{a}) \approx \max\left\{Q\left(\mathbf{s}, \mathbf{a}_1\right), \ldots, Q\left(\mathbf{s}, \mathbf{a}_N\right)\right\} \tag{2.5.3}$$

However this does not work well in practice majorly because it is generally non-trivial to sample from an arbitrary distribution. Some more accurate solutions, like **cross-entropy method (CEM)** (simple iterative stochastic optimization) or **CMA-ES** (substantially less simple iterative stochastic optimization) works OK for up to about 40 dimensions.

Option 2: Better Q function structure

The second option is to use function class that is inherently *easy to optimize.* One example is to use **quadratic function**:

$$Q_\phi(\mathbf{s}, \mathbf{a})=-\frac{1}{2}\left(\mathbf{a}-\mu_\phi(\mathbf{s})\right)^T P_\phi(\mathbf{s})\left(\mathbf{a}-\mu_\phi(\mathbf{s})\right)+V_\phi(\mathbf{s}) \tag{2.5.4}$$

This approach is called **Normalized Advantage Functions (NAF)**. In this case the object to be maximized is given directly by the networks' output:

$$\arg\max_{\mathbf{a}} Q_\phi(\mathbf{s}, \mathbf{a})=\mu_\phi(\mathbf{s}) \quad \max_{\mathbf{a}} Q_\phi(\mathbf{s}, \mathbf{a})=V_\phi(\mathbf{s}) \tag{2.5.5}$$

Using NAF does not change the algorithm itself, so that it's just as efficient as Q-learning, but it sacrifices the representational power and is less expressive.