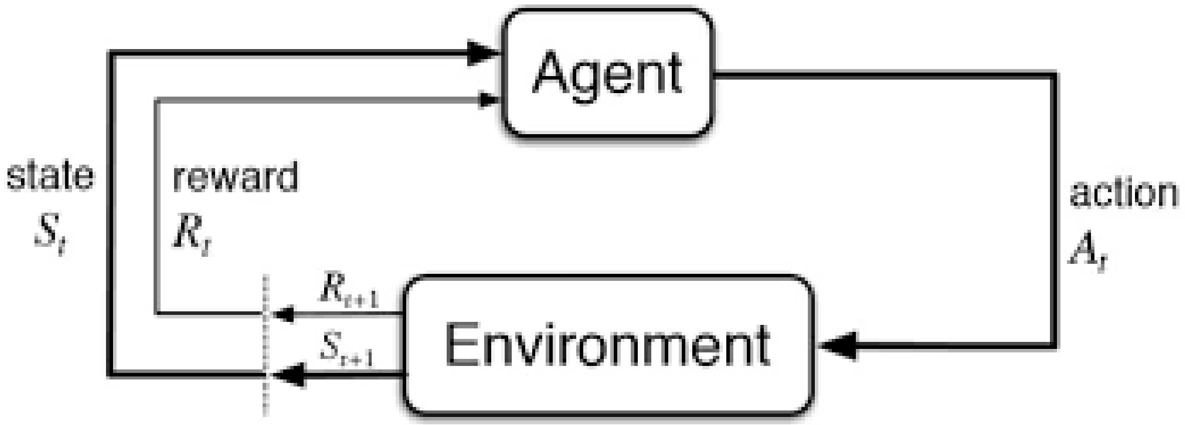
Offline Reinforcement learning / Conservative Q-learning

Idea of (online) RL

Setup of RL: We have an agent interacting with the environment. At time t, the agent observes a state s_t from the environment and it takes an action a_t; the environment transitions to the next state S_{t+1} and reward the agent with r_{t+1}. The objective is to train the agent that maximizes the total expected reward $E(\sum_{t=0}^{\infty} r_t)$. To factor in the probability that the agent might die and manifest our preference of immediate reward over long term reward and for some mathematical convenience down the line, we introduce a discount factor γ (<= 1) and adjust our objective to maximizing the discounted total reward $E(\sum_{t=0}^{\infty} \gamma^t r_t)$.

We always assume that the underlying environment is a Markov decision process in the sense that the current state S_t only depends on the previous state S_{t-1} and the action A_{t-1}.



The setup of RL makes it a fundamentally an online problem, i.e. actions are taken in real time and reward is given immediately.

All online learning algorithms are realization of one underlying principle: keep a balance between short term gain and long term gain (exploration vs exploitation) and iterate your policy through trial and error.

What makes the online learning possible is that you can engage with the environment, and environment tells you if you right or wrong, then you adjust your behavior by making good actions more likely and bad actions less likely.

Two styles of learning algorithms && Notations

We will review the on-policy and off-policy methods used in standard online algorithm.

POLICY GRADIENT

The goal of RL is to learn a policy, which is a conditional distribution over the action space $\pi(a_t|s_t)$ conditioned over the state.

Given a policy π and the transition probability of the MDP $T(s_{t+1}|s_t, a_t)$, we can write down expected discounted reward of this policy. then for any trajectory $\tau = (s_0, a_0, s_1, a_1, \dots s_h)$, we can write model its probability under π

$$p_\pi(\tau) = d_0(s_0) \prod_{t=0}^{H} \pi(a_t|s_t) T(s_{t+1}|a_t, s_t)$$

Therefore, the expected total reward of the policy is

$$J(\pi) = E_{\tau \sim p_\pi(\tau)}[\sum_{t=0}^{H} \gamma^t r(s_t, a_t)]$$

For any state s, we define the value of the state under the policy π to be

$$v_\pi(s) = E_{\tau \sim p_\pi(\tau|s_0=s)}[\sum_{t=0}^{H} \gamma^t r(s_t, a_t)]$$

For a pair of state and action s, a, we define its value to be

$$q_\pi(s, a) = E_{\tau \sim p_\pi(\tau|s_0=s, a_0=a)}[\sum_{t=0}^{H} \gamma^t r(s_t, a_t)]$$

If π is represented as a parametrized function π_θ (neural network), then we can directly optimize $J(\pi)$ through θ

Policy Gradient Theorem [Sutton && Barto 2020]

$$\nabla_\theta J(\theta) = \sum_s d(s) \sum_a \nabla_\theta \pi(a|s) q_\pi(s, a)$$

Through log-derivative trick, we can write the above equation as an expectation over the policy $\pi(a|s)$

$$\nabla_\theta J(\theta) = \sum_s d(s) \sum_a \nabla_\theta \pi(a|s) q_\pi(s, a) = \sum_s \sum_a \pi(a|s) \nabla_\theta \log \pi(a|s) q_\pi(s, a) = E_{\tau \sim p_\pi(\tau)} \nabla_\theta \log \pi(a|s) q_\pi(s, a)$$

This gives us a Monte-Carlo algorithm to sample the policy gradient by rolling out a trajectory.

Variants of policy gradient algorithms involving replacing $q_\pi(s, a)$ by something with less variance, for example the *advantage* $A(s, a) = q(s, a) - v(s)$

Algorithm 1 On-policy policy gradient with Monte Carlo estimator

```
1: initialize $\theta_0$
2: for iteration $k \in [0, \dots, K]$ do
3:     sample trajectories $\{\tau_i\}$ by running $\pi_{\theta_k}(\mathbf{a}_t|\mathbf{s}_t)$ ▷ each $\tau_i$ consists of $\mathbf{s}_{i,0}, \mathbf{a}_{i,0}, \dots, \mathbf{s}_{i,H}, \mathbf{a}_{i,H}$
4:     compute $\mathcal{R}_{i,t} = \sum_{t'=t}^{H} \gamma^{t'-t} r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t})$
5:     fit $b(\mathbf{s}_t)$ to $\{\mathcal{R}i, t\}$      ▷ use constant $b_t = \frac{1}{N} \sum_i \mathcal{R}i, t$, or fit $b(\mathbf{s}_t)$ to $\{\mathcal{R}i, t\}$
6:     compute $\hat{A}(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) = \mathcal{R}_{i,t} - b(\mathbf{s}_t)$
7:     estimate $\nabla_{\theta_k} J(\pi_{\theta_k}) \approx \sum_{i,t} \nabla_{\theta_k} \log \pi_{\theta_k}(\mathbf{a}_{i,t}|\mathbf{s}_{i,t}) \hat{A}(\mathbf{s}_{i,t}, \mathbf{a}_{i,t})$
8:     update parameters: $\theta_{k+1} \leftarrow \theta_k + \alpha \nabla_{\theta_k} J(\pi_{\theta_k})$
9: end for
```

Policy gradient method described in the above algorithm is an **on-policy** algorithm, this means the experience used to update the policy comes from the most up-to-date policy.

[Levine 2020]

APPROXIMATE DYNAMIC PROGRAMMING

The idea of q-learning is to iteratively solve Bellman optimality equation. If we can accurately compute

Rewrite $q_\pi(s,a) = E_{\tau \sim p_\pi(\tau | s_0 = s, a_0 = a)}[\sum_{t=0}^{H} \gamma^t r(s_t, a_t)]$
recursively, we have
$q_\pi(s,a) = r(s,a) + \gamma E_{s' \sim T(s'|s,a), a' \sim \pi(a'|s')} q_\pi(s', a')$

We can also view the above equation from the perspective of operator, i.e. mapping on the function space. Write

$$B^\pi q(s,a) = r(s,a) + \gamma E_{s' \sim T(s'|s,a), a' \sim \pi(a'|s')} q_\pi(s', a') = r(s,a) + P^\pi q(s,a)$$

Where P^π is the projection operator that “projects” the q-value to the next state-action pair.

Then, the above equation simply says that $q(s,a)$ is the fixed point of the Bellman operator B^π of the policy π.
Banach fixed point theorem gives an iterative way to compute q, by repeating the iteration $q_\pi^{k+1} = B^\pi q_\pi^k$ and it can be shown that $q_\pi = \lim_{k \to \infty} q_\pi^k$

Once we have accurately estimated the q-value of the current policy, we can do 1 step policy improvement by taking the greedy policy with respect to q, i.e $\pi_{new}(a|s) = \arg\max_a q_\pi(s,a)$

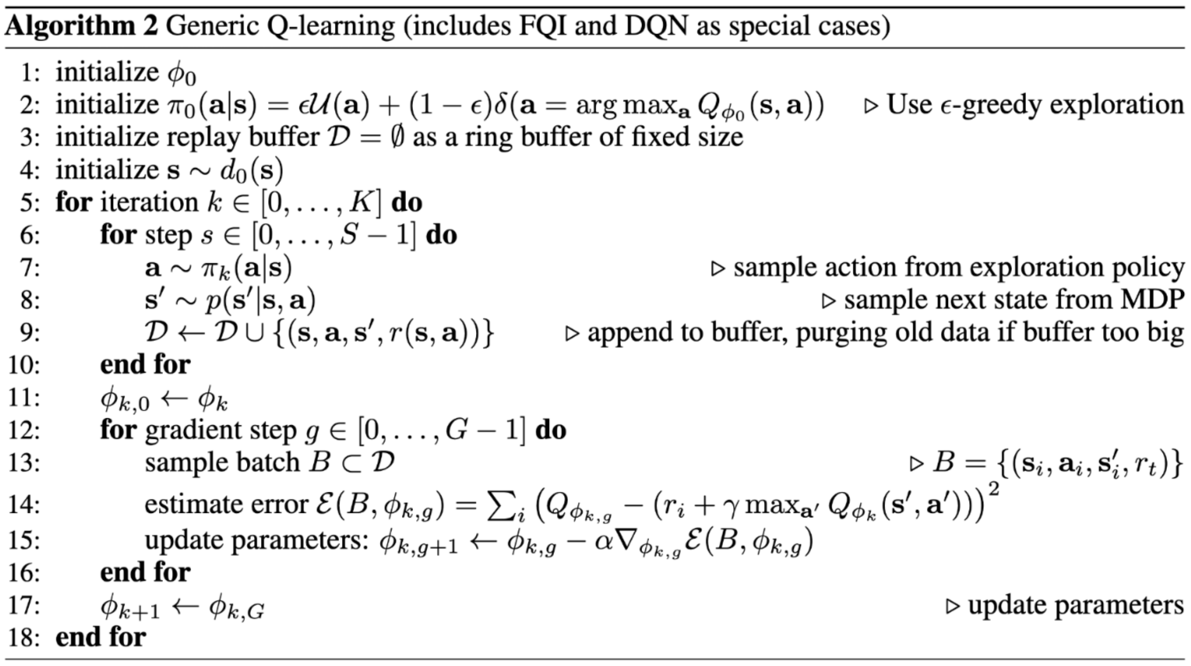
The interleaving of policy evaluation through Bellman operator and policy improvement through taking greedy policy gives us the paradigm of generalized policy iteration. At the limit, the policy converges, i.e. the greedy policy with respect to q_π is π itself. In this case, we get the Bellman equation becomes

$$q_\pi(s,a) = r(s,a) + \gamma E_{s' \sim T(s'|s,a)} \max_{a'} q_\pi(s', a')$$

We call this equation Bellman optimality equation. This equation only holds true for q-value of the optimal policy. By convention, we write q-value of the optimal policy as q^*. We can introduce Bellman optimality operator B^* such that

$$B^* q(s,a) = r(s,a) + \gamma E_{s' \sim T(s'|s,a)} \max_{a'} q(s', a')$$

The q-value of the optimal policy is the fixed point of the Bellman optimality operator, and by repeating the iteration $q^{k+1} = B^* q^k$, we get the standard q-learning algorithm.



Algorithm 2 Generic Q-learning (includes FQI and DQN as special cases)

```
1: initialize $\phi_0$
2: initialize $\pi_0(\mathbf{a}|\mathbf{s}) = \epsilon\mathcal{U}(\mathbf{a}) + (1-\epsilon)\delta(\mathbf{a} = \arg\max_{\mathbf{a}} Q_{\phi_0}(\mathbf{s},\mathbf{a}))$   ▷ Use $\epsilon$-greedy exploration
3: initialize replay buffer $\mathcal{D} = \emptyset$ as a ring buffer of fixed size
4: initialize $\mathbf{s} \sim d_0(\mathbf{s})$
5: for iteration $k \in [0,\dots,K]$ do
6:     for step $s \in [0,\dots,S-1]$ do
7:         $\mathbf{a} \sim \pi_k(\mathbf{a}|\mathbf{s})$   ▷ sample action from exploration policy
8:         $\mathbf{s}' \sim p(\mathbf{s}'|\mathbf{s},\mathbf{a})$   ▷ sample next state from MDP
9:         $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s},\mathbf{a},\mathbf{s}',r(\mathbf{s},\mathbf{a}))\}$   ▷ append to buffer, purging old data if buffer too big
10:    end for
11:    $\phi_{k,0} \leftarrow \phi_k$
12:    for gradient step $g \in [0,\dots,G-1]$ do
13:        sample batch $B \subset \mathcal{D}$   ▷ $B = \{(\mathbf{s}_i,\mathbf{a}_i,\mathbf{s}'_i,r_t)\}$
14:        estimate error $\mathcal{E}(B,\phi_{k,g}) = \sum_i \left(Q_{\phi_{k,g}} - (r_i + \gamma \max_{\mathbf{a}'} Q_{\phi_k}(\mathbf{s}',\mathbf{a}'))\right)^2$
15:        update parameters: $\phi_{k,g+1} \leftarrow \phi_{k,g} - \alpha\nabla_{\phi_{k,g}}\mathcal{E}(B,\phi_{k,g})$
16:    end for
17:    $\phi_{k+1} \leftarrow \phi_{k,G}$   ▷ update parameters
18: end for
```

[Levine 2020]

Q-learning as described above is an **off-policy** algorithm in the sense that the experience used to do Bellman backup (line 15) does not necessarily compute from the most up-to-date policy.

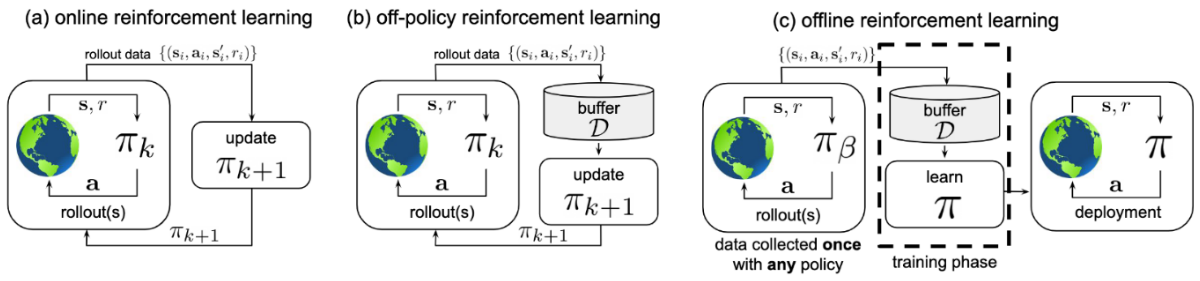
Idea and objective of offline RL

Use previously collected data (offline data) from an unknown behavior policy π_β to train a new policy that does better on the underlying MDP than π_β .

In contrast to online RL, offline RL does not need to interact with the underlying MDP which is often the bottleneck for the application of RL for real life problems, e.g. autonomous driving, robot doctors etc.

On the other hand, for real life use case of RL, we have abundant offline data generated by human. Therefore, a natural question is: can we leverage the offline data to train a an RL agent without it having to interact with the environment?

The picture below illustrate the difference between on-policy, off-policy, and offline RL//


(a) online reinforcement learning
rollout data $\{(\mathbf{s}_i,\mathbf{a}_i,\mathbf{s}'_i,r_i)\}$
$\mathbf{s}, r$
π_k
$\mathbf{a}$
rollout(s)
update π_{k+1}
π_{k+1}
(b) off-policy reinforcement learning
rollout data $\{(\mathbf{s}_i,\mathbf{a}_i,\mathbf{s}'_i,r_i)\}$
buffer $\mathcal{D}$
update π_{k+1}
π_k
rollout(s)
π_{k+1}
(c) offline reinforcement learning
$\{(\mathbf{s}_i,\mathbf{a}_i,\mathbf{s}'_i,r_i)\}$
π_β
rollout(s)
data collected **once** with **any** policy
buffer $\mathcal{D}$
learn π
training phase
π
deployment

Why offline RL is harder than online RL

The idea of RL is to learn by trial and error, but in an offline setting, you don't have "trials" any more, so you don't know if you made an "error".

It is because we cannot do counterfactual query and incorrect estimate of Q-value of the learnt policy cannot be corrected.

Approaches to offline RL

1. Do supervised learning on the offline data to learn the mapping s → a as initialization, then continue with online RL. This technique is called behavior cloning

[list problems with behavior cloning, noisy policies, performance of the resulting policy upper bounded by the behavior policy used to generate data]

2. Bootstrap / Approximate dynamic programming on offline data
Just like in the off-policy case, interleaving policy evaluation (trained with Bellman backup) and policy improvement by taking greedy policy with the current evaluation

$$\hat{Q}^{k+1} \leftarrow \arg\min_{Q} \mathbb{E}_{\mathbf{s},\mathbf{a},\mathbf{s}'\sim\mathcal{D}}\left[\left((r(\mathbf{s},\mathbf{a}) + \gamma\mathbb{E}_{\mathbf{a}'\sim\hat{\pi}^k(\mathbf{a}'|\mathbf{s}')}[\hat{Q}^k(\mathbf{s}',\mathbf{a}')]) - Q(\mathbf{s},\mathbf{a})\right)^2\right] \text{ (policy evaluation)}$$
$$\hat{\pi}^{k+1} \leftarrow \arg\max_{\pi} \mathbb{E}_{\mathbf{s}\sim\mathcal{D},\mathbf{a}\sim\pi^k(\mathbf{a}|\mathbf{s})}\left[\hat{Q}^{k+1}(\mathbf{s},\mathbf{a})\right] \quad \text{(policy improvement)}$$

The problem with this approach is out-of-distribution (OOD) state and action, i.e. state and action that do not appear in the offline dataset D.

At test time, you might get states that do not exist in the training data. More formally, state visitation frequency of learnt policy $d^{\pi}(s)$ differs from the state visitation frequency of the behavior policy $d^{\pi_\beta}(s)$. This means the policy network will output unpredictable actions. One way to mitigate this is to contrain how much the learnt policy differs from the behavior policy. But this undermines the purpose of offline RL which is to learn a policy that's potentially much better than the ones used to generate the offline data.

At training time, state distribution shifts would not occur by definition. However, the training process is affected by **action distribution shif**t. This is because in Bellman backup, the target value in the TD error depends on actions sampled from the current policy (e.g. a_{t+1} is outside the distribution the Q-function has seen).

$$Q^{\pi}(\mathbf{s}_t,\mathbf{a}_t) = r(\mathbf{s}_t,\mathbf{a}_t) + \gamma\mathbb{E}_{\mathbf{s}_{t+1}\sim T(\mathbf{s}_{t+1}|\mathbf{s}_t,\mathbf{a}_t),\mathbf{a}_{t+1}\sim\pi(\mathbf{a}_{t+1}|\mathbf{s}_{t+1})}\left[Q^{\pi}(\mathbf{s}_{t+1},\mathbf{a}_{t+1}))\right]$$

When the current policy $\pi(a|s)$ differs significantly from the behavior policy $\pi^{\beta}(a|s)$, we could get highly erroneous Q-value. Furthermore, this is exacerbated by we optimize the policy to produce actions that maximizes the q-value. This means the policy is biased towards OOD actions with erroneous Q-value.

An experimental result from from [Kumar et al. 2019]

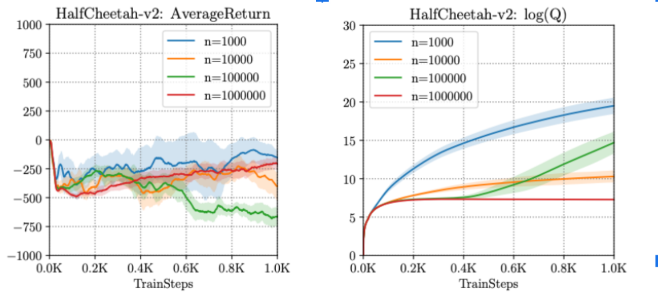


Figure 1: Performance of SAC on HalfCheetah-v2 (return (left) and log Q-values (right)) with off-policy expert data w.r.t. number of training samples (n). Note the large discrepancy between returns (which are negative) and log Q-values (which have large positive values), which is not solved with additional samples.

Conservative Q-learning [Kumar et al. 2020]

Conservative Q-learning (CQL) offers a fix for the over estimation of Q-value for OOD action. The idea is simple: push down the Q-value for state and action pair that's unlikely under the behavior policy π_β. i.e. when training the Q-function, we add a regularization term minimize the Q-value under certain distribution $\mu(s, a)$ along side the standard TD-error. Since the training does not query the state out side the distribution of the behavior policy, we restrict to $\mu(s, a)$ to the class of the distribution that have the same state-marginal as $\pi_\beta(s, a)$, i.e. $\int_A \mu(s, a)da = d^{\pi_\beta}(s)$;

The Q-function training objective becomes

$$\min_Q(\alpha \max_{\mu(a|s)} E_{s\sim D} E_{a\sim\mu(a|s)}[Q(s, a)] + \frac{1}{2} E_{s,a,s'\sim D}[(Q(s, a) - r(s, a) - \max_{a'} \gamma Q(s', a'))^2]$$

First part minimizes "big" Q-values, second part is the standard TD error.

We can formally show (Thm 3.1 in the paper) that for sufficiently large α, the resulting is a lower bound of Q^π for all s, a.

We can directly apply this Q-function to stadard actor-critic algorithm

The problem with this approach is that the learned Q-function can be "too conservative", a variant of the above CQL objective is to maximize the Q-value for state-action pairs that appear in the data set

$$\min_Q(\alpha \max_{\mu(a|s)} E_{s\sim D}(E_{a\sim\mu(a|s)}[Q(s, a)] - E_{s,a\sim D} Q[s, a]) + \frac{1}{2} E_{s,a,s'\sim D}[(Q(s, a) - r(s, a) - \max_{a'} \gamma Q(s', a'))^2]$$

The theoretical guarantee for this objective is that the expected Q-value over actions is a lower bound of $v(s)$ for all s. i.e. we no longer have point-wise lower bound, but we do lower-bound the true Q-value in expectation (over actions).

How well does CQL work

See the experimental evaluation section of the paper

Reference

Sutton && Barto 2020

Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems | Levine et al. 2020

Conservative Q-Learning for Offline Reinforcement Learning | Kumar et al. 2020

Stabilizing Off-Policy Q-Learning via Bootstrapping Error Reduction | Kumar et al. 2019