Report of AI3601:reinforcement learning

Xiangyun Rao, Hanlin Xu, Zhihao Xu
Shanghai Jiao Tong University
$520030910 + \{366\}, \{378\}, \{377\}$

Abstract

In this project, we invest some representative offline RL algorithms and reproduction 3 of them to apply in the **Hopper-v3** environment. In addition, we try some interesting trick in an unknown direction. Finally, we achieve a fairly good result.

1 Introduction

1.1 Why we chose this project:

Practically, offline RL is one of the most general case. In most areas of reinforcement learning applications, such as auto drive, the online interaction has an quite expensive cost. Therefore, we think using high quality offline dataset to train a good-enough agent is the megatrends of the future of the partical application of RL.

1.2 The environment

In this project, we are required to use the **Hopper-v3** from gym[mujoco].The hopper is a two-dimensional one-legged figure that consist of four main body parts - the torso at the top, the thigh in the middle, the leg in the bottom, and a single foot on which the entire body rests. The goal is to make hops that move in the forward (right) direction by applying torques on the three hinges connecting the four body parts.The action space is a Box(-1, 1, (3,), float32). Observations consist of positional values of different body parts of the hopper, followed by the velocities of those individual parts (their derivatives) with all the positions ordered before all the velocities.All observations start in state (0.0, 1.25, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0) with a uniform noise in the range of [-reset_noise_scale, reset_noise_scale] added to the values for stochasticity.

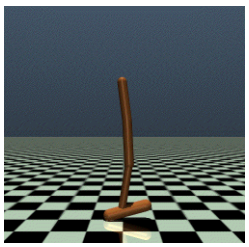

Figure 1: Hopper

Submitted to 37th Conference on Neural Information Processing Systems (NeurIPS 2023). Do not distribute.

2 Algorithm we've tried

In this part, we'd like to give a brief introduction and basic understanding of these three offline RL algorithms: BCQ, AWR, CQL, which we've implemented in Task 3.

2.1 BCQ[1]

BCQ(Batch-Constrained Reinforce Learning) is the baseline we choose. Because the published date of BCQ is earlier than BEAR and BEAR is seen as a developed version of BCQ for its MMD constrain, we choose BCQ as our baseline for our great motivation for a brand new direction.

One of the main problem of offline learning is extrapolation error, that is, how to act when the agent meets with the state that no exists in our offline dataset. The idea the author use is to take the policy all based on the origin dataset and no consideration of the no-seen-before state.

More accurately, BCQ use a VAE to make the input state more similar to origin dataset, and make some perturb to diversify the choice based on the Q-network(maybe this part is a tiny exploration). To avoid Q-Overestimate, BCQ uses the thought of double-dqn. The formula below are used to choose action, get the policy, and update the parameters respectively.

$$\phi \leftarrow \underset{\phi}{\operatorname{argmax}} \sum_{(s,a)\in\mathcal{B}} Q_\theta\left(s, a+\xi_\phi(s,a,\Phi)\right)$$

$$\pi(s) = \underset{a_i+\xi_\phi(s,a_i,\Phi)}{\operatorname{argmax}} Q_\theta\left(s, a_i+\xi_\phi\left(s,a_i,\Phi\right)\right), \{a_i \sim G_\omega(s)\}_{i=1}^n$$

$$\text{TD Target} = \gamma + \max_{a_i}\left[\lambda \min_{j=1,2} Q_{\theta'_j}\left(s', a_i\right) + (1-\lambda)\max_{j=1,2} Q_{\theta'_j}\left(s', a_i\right)\right]$$

In a word, BCQ using batch-constrain to solve the problem of extrapolation error. That is, no exploration but more the origin dataset.

Algorithm 1 BCQ

Input: Batch $\mathcal{B}$, horizon T, target network update rate τ, mini-batch size N, max perturbation Φ, number of sampled actions n, minimum weighting λ.
Initialize Q-networks $Q_{\theta_1}, Q_{\theta_2}$, perturbation network ξ_ϕ, and VAE $G_\omega = \{E_{\omega_1}, D_{\omega_2}\}$, with random parameters θ_1, θ_2, ϕ, ω, and target networks $Q_{\theta'_1}, Q_{\theta'_2}$, $\xi_{\phi'}$ with $\theta'_1 \leftarrow \theta_1, \theta'_2 \leftarrow \theta_2, \phi' \leftarrow \phi$.
for $t = 1$ **to** T **do**
  Sample mini-batch of N transitions (s, a, r, s') from $\mathcal{B}$
  $\mu, \sigma = E_{\omega_1}(s, a), \quad \tilde{a} = D_{\omega_2}(s, z), \quad z \sim \mathcal{N}(\mu, \sigma)$
  $\omega \leftarrow \operatorname{argmin}_\omega \sum (a - \tilde{a})^2 + D_{\mathrm{KL}}(\mathcal{N}(\mu, \sigma) || \mathcal{N}(0, 1))$
  Sample n actions: $\{a_i \sim G_\omega(s')\}_{i=1}^n$
  Perturb each action: $\{a_i = a_i + \xi_\phi(s', a_i, \Phi)\}_{i=1}^n$
  Set value target y (Eqn. 13)
  $\theta \leftarrow \operatorname{argmin}_\theta \sum (y - Q_\theta(s, a))^2$
  $\phi \leftarrow \operatorname{argmax}_\phi \sum Q_{\theta_1}(s, a + \xi_\phi(s, a, \Phi)), a \sim G_\omega(s)$
  Update target networks: $\theta'_i \leftarrow \tau\theta + (1-\tau)\theta'_i$
  $\phi' \leftarrow \tau\phi + (1-\tau)\phi'$
end for

2.2 AWR[2]

AWR(Adaptive-Weight Regression) is a PPO-like-structure algorithm that improved from RWR. It's quite easy and not a typical offline learning algorithm for its PPO-like-structure(use a buffer to training). It mainly based on an adaptive-weight policy improvement. The formulas below is the math derivation process of AWR.

The policy improvement:

$$\begin{aligned}\eta(\pi) &= J(\pi) - J(\mu) \\ &= J(\pi) - \sum_{i=1}^{k} w_i J\left(\pi_i\right) \\ &= \sum_{i=1}^{k} w_i \left(J(\pi) - J\left(\pi_i\right)\right) \\ &= \sum_{i=1}^{k} w_i \left(\mathbb{E}_{\mathbf{s}\sim d_\pi(\mathbf{s}), \mathbf{a}\sim\pi(\mathbf{a}|\mathbf{s})}\left[A^{\pi_i}(\mathbf{s}, \mathbf{a})\right]\right)\end{aligned}$$

Then we can list the Lagrange's equation. The result:

$$\pi(s) = \arg\max_{\pi} \sum_{i=;1}^{k} w_i \mathbb{E}_{\mathbf{s}\sim d_{\pi_i}(\mathbf{s})} \mathbb{E}_{\mathbf{a}\sim\pi_i(\mathbf{a}|\mathbf{s})} \left[\log \pi(\mathbf{a} \mid \mathbf{s}) \exp\left(\frac{1}{\beta}\left(\mathcal{R}_{\mathbf{s},\mathbf{a}}^{\pi_i} - \frac{\sum_j w_j d_{\pi_j}(\mathbf{s}) V^{\pi_j}(\mathbf{s})}{\sum_j w_j d_{\pi_j}(\mathbf{s})}\right)\right)\right]$$

The pseudo code of AWR. we can see that it use a buffer as the experience pool and take loss different from PPO.

Algorithm 1 Advantage-Weighted Regression

$\pi_1 \leftarrow$ random policy
$\mathcal{D} \leftarrow \emptyset$
for iteration $k = 1, ..., k_{\max}$ **do**
  add trajectories $\{\tau_i\}$ sampled via π_k to $\mathcal{D}$
  $V_k^{\mathcal{D}} \leftarrow \arg\min_V \mathbb{E}_{\mathbf{s},\mathbf{a}\sim\mathcal{D}}\left[\left|\left|\mathcal{R}_{\mathbf{s},\mathbf{a}}^{\mathcal{D}} - V(\mathbf{s})\right|\right|^2\right]$
  $\pi_{k+1} \leftarrow \arg\max_\pi \mathbb{E}_{\mathbf{s},\mathbf{a}\sim\mathcal{D}}\left[\log \pi(\mathbf{a}|\mathbf{s}) \exp\left(\frac{1}{\beta}\left(\mathcal{R}_{\mathbf{s},\mathbf{a}}^{\mathcal{D}} - V_k^{\mathcal{D}}(\mathbf{s})\right)\right)\right]$
end for

The PPO-like-structure cause some problems. For example, it's quite like IL so that its result is not good. And because its buffer cannot get more real time trajectory, it's performance varied violently with time passing by.

Novelty: We try a bootstrap aimed to get extra information from the learning process. And to avoid Q-overestimate, we use a partial-bootstrap, which will not cause Q-overestimate. Below is the specific implement. In addition, we choose a series of parameters to tune. Finally we choose 0.05 from $\{0, 0.05, 0.1, 0.2, 0.5\}$. The improved algorithm is name as bootstrap-AWR by us.

```
discounted_reward[-1] = reward[-1] + self.discount * cur_value[-1]
for t in reversed(range(0, len(reward)-1)):
    cur_reward = reward[t]
    next_return = discounted_reward[t+1]
    discounted_reward[t] = cur_reward + self.discount * ((0.5) * cur_value[t + 1] + 0.5 * next_return)
steps_per_shuffle = int(np.ceil(len(state) / batch_size))
```

Figure 2: The implement of bootstrap

2.3 CQL[3]

CQL(conservative Q-learning), aims to address over-estimation of Q-value by learning a conservative Q-function such that the expected value of a policy under this Q-function lower-bounds its true value. Moreover, CQL augments the standard Bellman error objective with a simple Q-value regularizer, which can be implemented on actor-critic structure. The specific formula is shown below. The $\mathcal{R}(\mu)$ stands for regularizer, the common format of which is KL-Divergence between current policy $\mu(a|s)$ and prior distribution $\varrho(a|s)$. And if the later one is set as uniform distribution, the formula can be rewritten as the second one below.

$$\min_Q \max_\mu \alpha \left(\mathbb{E}_{\mathbf{s}\sim\mathcal{D},\mathbf{a}\sim\mu(\mathbf{a}|\mathbf{s})}\left[Q(\mathbf{s},\mathbf{a})\right] - \mathbb{E}_{\mathbf{s}\sim\mathcal{D},\mathbf{a}\sim\hat{\pi}_\beta(\mathbf{a}|\mathbf{s})}[Q(\mathbf{s},\mathbf{a})]\right) + \frac{1}{2}\mathbb{E}_{\mathbf{s},\mathbf{a},\mathbf{s}'\sim\mathcal{D}}\left[\left(Q(\mathbf{s},\mathbf{a}) - \hat{\mathcal{B}}^{\pi_k}\hat{Q}^k(\mathbf{s},\mathbf{a})\right)^2\right] + \mathcal{R}(\mu) \quad (\mathrm{CQL}(\mathcal{R}))$$

$$\min_Q \alpha\mathbb{E}_{\mathbf{s}\sim\mathcal{D}}\left[\log\sum_{\mathbf{a}}\exp(Q(\mathbf{s},\mathbf{a})) - \mathbb{E}_{\mathbf{a}\sim\hat{\pi}_\beta(\mathbf{a}|\mathbf{s})}[Q(\mathbf{s},\mathbf{a})]\right] + \frac{1}{2}\mathbb{E}_{\mathbf{s},\mathbf{a},\mathbf{s}'\sim\mathcal{D}}\left[\left(Q - \hat{\mathcal{B}}^{\pi_k}\hat{Q}^k\right)^2\right]$$

3 Experiments and results

In this part, we will show how we conduct the experiment and make analysis towards the result.

3.1 Pre-consideration of the dataset

Before we launched the experiment, we've done careful examination about the dataset. We first sliced the dataset according to terminal points and divide it into independent tracks, as what they have been sampled. Then we counted the total reward of each track and show basic properties in table1. It's obvious that the reward value varies dramatically from one to another, which might introduce instability to our models. If we happen to use the data samples in tracks with low reward and simply do behavior cloning, the model will probably learn defective policy.

So we have removed some of them for a better efficiency of training.

Table 1: Properties of the dataset

Indicators:	Number	Max reward	Min reward	Average	Variance
Statistics	380	1306.22	7.25	619.39	305.72

3.2 Evaluations

We conducted our experiments using "Hopper-v3" mujoco environment. First we try to reproduce BCQ algorithm in this task. We used its original model structure and hyper-parameters, trained for 10000 epochs and evaluated the model by interacting with the environment for 32 times. Additionally, We didn't impose any restriction in path length during evaluations. Finally, the average return reported by the evaluation function is around 1600.

After that, we shifted our focus to other algorithms. We first reproduced AWR, which is not a typical offline RL method. Due to huge gap between original motivation of AWR and current task, We have invested a significant amount of time and effort to complete this code. But for its IL-like structure, we get a quite bad result(1500 ± 500). Therefore, we try to put bootstrap in our algorithm. The best return value that bootstrap-AWR can obtained is about 2880, which is much higher than basic BCQ.

However, during training process, we discovered that average return of AWR varied violently, the return value of which might be relevant to random factor.

Then we tried CQL. To be honest, we didn't fully understand the mechanism lying behind the algorithm, for the source code is obscure for us. Using default setting of CQL, we trained the model with given dataset. Similarly to AWR, the average return of CQL has reached 3000 in some training epochs, but quickly fell back to around 500. Even in the end, the model was completely deteriorating and wasn't able to recover. To get better results, we once dropped many tracks in dataset which have low return value, but it didn't change the circumstance. Figure3 below shows the return curve in 500 epochs training. As we can see, the curve varies frequently and even "converged" to 0 after 300 epochs.

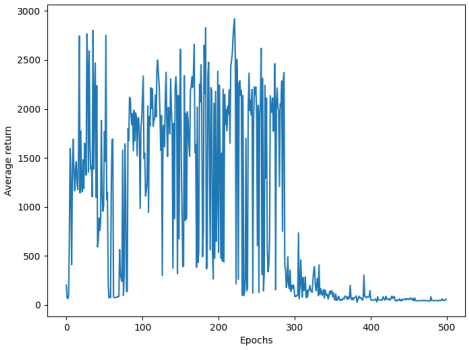

Figure 3: Return value curve of CQL

Subsequently, we revisited BCQ and made attempts to improve its performance. We noticed that the activation function chosen by BCQ was tanh, which might lead to gradient vanishing during training. So we replaced it by relu. On the other hand, we thought that training a value estimation network or action generation network from scratch is highly unstable, and repeatedly training can rarely outputs a better model. So we just loaded the parameters we trained before, used smaller learning rate to do fine-tuning[4]. With these optimization tricks, we achieved about 3250 in average return, ranking highest in our experiments. Figure4 depicts the training process of our new BCQ, which is more steady and finally converged.

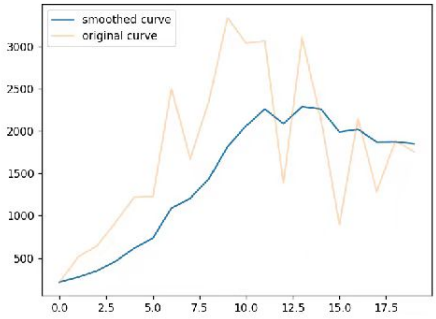

Figure 4: Return value curve of new BCQ

3.3 Final results

We conclude overall results during our experiments in table 2. Refined dataset means dropping bad tracks in the dataset.

Table 2: Overall results

Methods	Average return
CQL(refined dataset)	900±300
AWR(refined dataset)	1250±500
Baseline BCQ	1650±300
bootstrap-AWR(refined dataset)	2780±200
New BCQ	**3250±100**

4 Conclusion

In this project, we've found that we can design offline RL algorithm from the reasons that cause the poor performance of offline agent. And we found a possible direction to improve the performance of current algorithms(see our novelty in AWR). Ultimately we achieve a good result. In the future, we can explore how to designed a algorithm using partial-bootstrap with a reasonable method.

5 Labor division

Table 3: Labor division of our group

Name	Xiangyun Rao	Hanlin Xu	Zhihao Xu
Workload	AWR new BCQ Novelty	Baseline BCQ Presentation	CQL Report
Percentage	40%	30%	30%

6 Thanks

We are grateful to Professor Shuai Li, who present this amazing course, for her earnest teaching, reasonable course content, and the right amount of homework.

We are grateful to all the TAs of this course, for their patient of answering our problems and grading our homework carefully. Especially Ruofeng Yang's timely guide.

We are grateful to JidiAI network platform, for providing fair AI ranking, rich game environments, advanced open source algorithms, and fair AI competitions for this course.

We are grateful to our laptops and their CPU/GPUs, which are truly working hard to finish this project.(We think you can consider to provide calculate resource next year)

We are also grateful to Hopper, the agent we control in this project. Although it is just a virtual agent, hard work and dedication of an agent also matters, anyway.

At last, we want to show our respect to Xiangyun Rao, our leader, rescuer of our project, king of urgent, owner of herniated lumbar discs. Without his forward-looking urge, we couldn't start in midterm. Without his help, we couldn't finish other homework so rapidly, especially reinforcement learning.

7 References

[1] Fujimoto, S., Meger, D., & Precup, D. (2019) *What Matters In On-Policy Reinforcement Learning? A Large-Scale Empirical Study, In International conference on machine learning*

[2] Peng, X. B., Kumar, A., Zhang, G., & Levine, S. . (2019) *Off-policy deep reinforcement learning without exploration*

[3]Kumar, A., Zhou, A., Tucker, G., & Levine, S. ./ (2020). *Conservative q-learning for offline reinforcement learning. Advances in Neural Information Processing Systems*

[4]Andrychowicz, M., Raichuk, A., Stańczyk, P., Orsini, M., Girgin, S., Marinier, R., ... & Bachem, O. . (2020) *What matters in on-policy reinforcement learning? a large-scale empirical study*